Universidade do Minho
Escola de Engenharia

André Barbosa, PG50216
Carlos Soares, PG50280

8051 Microcontroller Design

Master’s in Industrial Electronics and Computers Engineering

Embedded Systems

Professor:
Adriano Tavares

September 22, 2022

“Always do your best, don’t let the pressure make you panic.”

Tupac Shakur

Contents

1 Introduction **1**

2 Analysis **1**
- 2.1 Requirements 1
- 2.2 Constraints 1
- 2.3 System Overview 1

3 Fundamental Theories **3**
- 3.1 8051 Microcontroller 3
  - 3.1.1 Peripherals 3
  - 3.1.2 Timer Operation and Programming 3
  - 3.1.3 Interrupt Controller 4
- 3.2 Zybo Z-7010 6

4 Design **8**
- 4.1 Instruction Set 8
- 4.2 RAM 8
- 4.3 ROM 9
- 4.4 Datapath 10
  - 4.4.1 Register 8-bit 10
  - 4.4.2 Instruction Register 10
  - 4.4.3 Arithmetic Logic Unit 11
  - 4.4.4 Program Counter 12
  - 4.4.5 General 13
- 4.5 Control Unit 13
- 4.6 Timer 14
- 4.7 Interrupt Controller 14
- 4.8 Tools 15

5 Implementation **17**
- 5.1 Global Constants 17
- 5.2 RAM 18
  - 5.2.1 Register bank 18
  - 5.2.2 Debounce 19
  - 5.2.3 RAM 20
- 5.3 ROM 23
- 5.4 Datapath 26
  - 5.4.1 Instruction Register 26
  - 5.4.2 8-bit Register 27

5.4.3 Program Counter 28
5.4.4 Arithmetic Logic Unit 31
5.4.5 Datapath 33
5.5 Control Unit 35
5.6 Timer 41
5.7 Interrupt Controller 44
5.8 I/O Layout 46

6 Verification 50

6.1 Tests 50
6.1.1 8-bit Register 50
6.1.2 RAM 51
6.1.3 ROM 53
6.1.4 ALU 55
6.1.5 Program Counter 57
6.1.6 Instruction Register 60
6.1.7 Timer 62
6.1.8 Interrupt Control 63
6.1.9 Debounce 66

7 Final Test Program 68

8 Gantt Diagram 70

9 References 71

List of Figures

1 System overview . . . 2
2 Timers and Operating Modes . . . 4
3 ZYBO Device Diagram . . . 7
4 RAM schematic . . . 9
5 ROM schematic . . . 10
6 Register 8-bit schematic . . . 10
7 Instruction Register schematic . . . 11
8 ALU schematic . . . 11
9 Program Counter schematic . . . 12
10 Datapath schematic . . . 13
11 Control Unit schematic . . . 14
12 Control Unit execute states schematic . . . 14
13 Timer schematic . . . 15
14 Interrupt Control schematic . . . 15
15 Output waveform 8-bit register testbench . . . 51
16 Output waveform RAM testbench. . . . 53
17 Output waveform ROM testbench. . . . 54
18 Output waveform ALU testbench (ADD acc, Immediate). . . . 56
19 Output waveform ALU testbench (SUBB acc, Immediate). . . 56
20 Output waveform ALU testbench (ANL acc, Immediate). . . . 56
21 Output waveform ALU testbench (ORL acc, Immediate). . . . 56
22 Output waveform ALU testbench (XRL acc, Immediate). . . . 57
23 Output waveform Program Counter testbench. . . . 58
24 Output waveform Program Counter testbench. . . . 59
25 Output waveform Program Counter testbench - Jump Not Zero. 59
26 Output waveform Program Counter testbench - Jump Zero. . 60
27 Output waveform Program Counter testbench - Jump If Not Carry. . . . 60
28 Output waveform Instruction Register testbench. . . . 62
29 Output waveform Timer mode 1 testbench. . . . 63
30 Output waveform Counter mode 1 testbench. . . . 64
31 Output waveform Interrupt Control testbench. . . . 65
32 Output waveform Debounce testbench. . . . 67
33 Output waveform Debounce testbench. . . . 67
34 Task Division: Gantt Diagram. . . . 70

1 Introduction

This document describes our 8051 design, the test code used to verify it and the physical hardware implementation. The main objective is to design an 8051 Microcontroller, using the Verilog code language, and finally to implement it in hardware using a Field-programmable gate array (FPGA). The procedure followed by the group will go through each step of the waterfall methodology.

2 Analysis

In this phase it is intended to present a brief introduction to the problem previously presented as well as to describe the steps for the realization of the project. There are several 8051 instruction sets, however the instruction set to be followed in this project will be the Intel MCS-51.

2.1 Requirements

- Two Interrupts;
- One Timer;
- Implement a subset of instructions.

2.2 Constraints

- Use Verilog;
- Use Xilinx Vivado;
- This project must be develop by a group of two students;

2.3 System Overview

In Figure 1 you can see a simple representation of the 8051 to be designed. It is then possible to separate the project into 4 major categories:

- Unit Control;
- Datapath;
- Peripherals;
- Interrupt Controller.

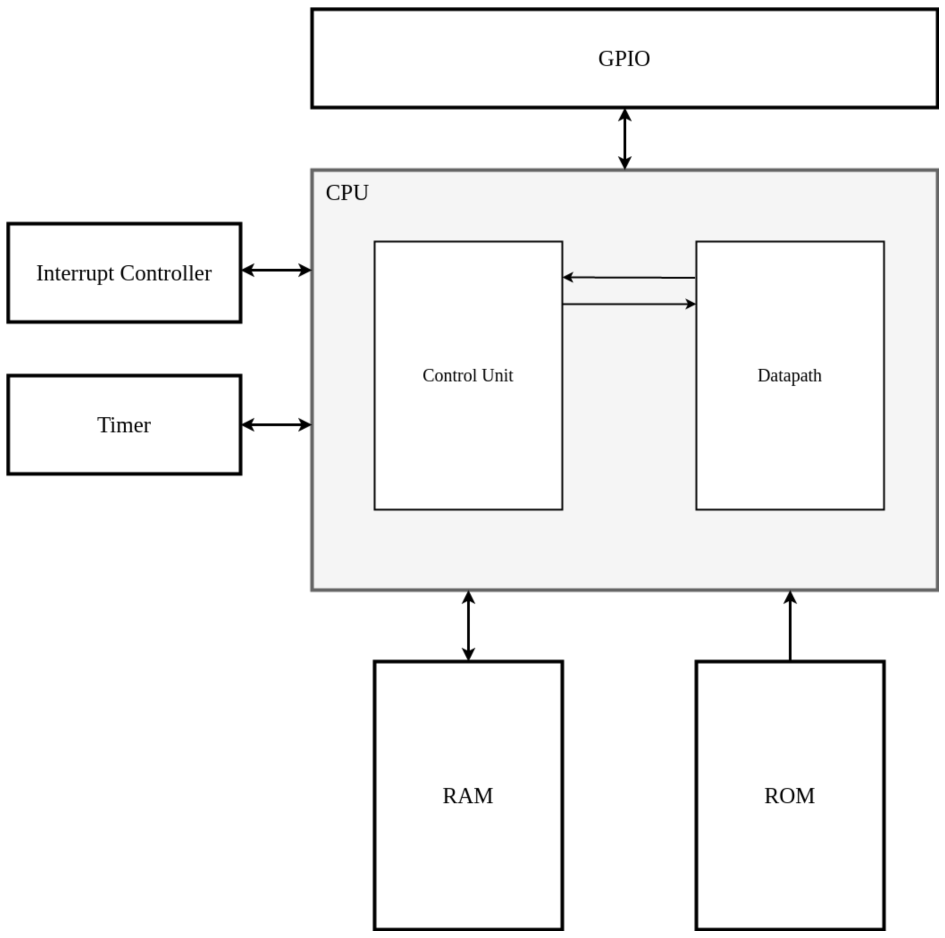


Figure 1: System overview

3 Fundamental Theories

3.1 8051 Microcontroller

With the knowledge acquired in course units attended [1], a brief analysis will now be presented that will allow the group to have a better interpretation of the problem and thus find the best possible solution during the design phase.

The 8051 was introduced in 1980 by Intel as one device in the MCS-51 family. This microcontrollers are all based on an 8-bit Complex Instruction Set Computing (CISC) core with Harvard architecture. In other words, the program storage and Random Acess Memory (RAM) are separeted. The instruction set of the MCU-51 family includes 255 Operation Codes (opcodes), and most instructions are executed within one or two machine cycles. All versions of the MCS-51 family have the following common properties:

- 8 bit Central Processing Unit (CPU);
- 128 Byte IRAM;
- 16 bit address bus (up to 64 kB memory);
- Separated External Data Memory (XDATA) and Program Storage Read Only Memory (PSROM);
- Two timers/counters;
- 32 input/output (I/O) pins;
- Integrated Universal Asynchronous Receiver Transmitter (UART) interface;
- Two external interrupts with two priority levels;
- Boolean processor for fast boolean operations;
- Internal clock generation.

3.1.1 Peripherals

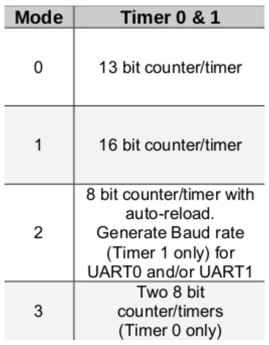
3.1.2 Timer Operation and Programming

One of the many tasks performed by a CPU is to time internal and external events [2]. So, a timer/counter is useful in the sense that is able to handle various timing requests independently.

Functional Description Timers are used for:

- Interval timing;
- Event counting;
- Baud rate generation.

As mentioned before, most of the MCS-51 only have two timers: timer0 and timer1. In the next Figure it is possible to see all their operating modes.



Mode	Timer 0 & 1
0	13 bit counter/timer
1	16 bit counter/timer
2	8 bit counter/timer with auto-reload. Generate Baud rate (Timer 1 only) for UART0 and/or UART1
3	Two 8 bit counter/timers (Timer 0 only)

Figure 2: Timers and Operating Modes

3.1.3 Interrupt Controller

The 8051 microcontroller has several interrupts sources, such as external interrupts, timer interrupts, and serial communication interrupts. These interrupts allow the microcontroller to perform tasks in a non-sequential order, allowing the microcontroller to respond to events in real-time. The basic theory behind interrupts is that they allow the microcontroller to temporarily halt its current task and execute a specific interrupt service routine (ISR) when an interrupt signal is received. This allows the microcontroller to perform multiple tasks simultaneously, making it more efficient and responsive.

He does have five types of interrupts, each one with its own vector address that the microcontroller uses to jump to the specific interrupt service routine (ISR), based on the interrupt source:

- External interrupts 0 and 1;
- Timer 0 and 1 interrupts:
- Serial communication interrupt (SCI).

These interrupts are organized in a priority hierarchy, with external interrupt 0 having the highest priority and the SCI having the lowest priority. When an interrupt occurs, the microcontroller saves the current state of the program by pushing the contents of the program counter and the status register onto the stack and then jumps to the appropriate interrupt service routine (ISR) to handle the interrupt. Once the interrupt service routine (ISR) is finished executing, the microcontroller retrieves the saved state of the program by popping the contents of the program counter and status register from the stack. This allows the microcontroller to return to the task it was performing before the interrupt occurred. The 8051 also has an interrupt enable (IE) and interrupt request (IR) registers that are used to enable or disable specific interrupts and to check if an interrupt has occurred. This allows the user to have a finer control over the interrupts and the interrupt service routine (ISR) handling.

To summarize this section, when it need it, the 8051 microcontroller processes interrupts in the following way:

- An interrupt signal is received by the microcontroller;
- The microcontroller checks the interrupt enable (IE) register to see if the interrupt is enabled. If the interrupt is not enabled, the microcontroller ignores the interrupt and continues with its current task;
- The microcontroller checks the interrupt request (IR) register to see if the interrupt has occurred. If the interrupt has not occurred, the microcontroller ignores the interrupt and continues with its current task;
- The microcontroller saves the current state of the program by pushing the contents of the program counter and the status register onto the stack;
- The microcontroller disables all interrupts to prevent nested interrupts;
- The microcontroller jumps to the appropriate interrupt service routine (ISR) to handle the interrupt;
- The ISR executes and performs the necessary actions to handle the interrupt;
- Once the ISR is finished executing, the instruction RETI is executed, which causes the microcontroller to retrieve the saved state of the program by popping the contents of the program counter and status register from the stack;

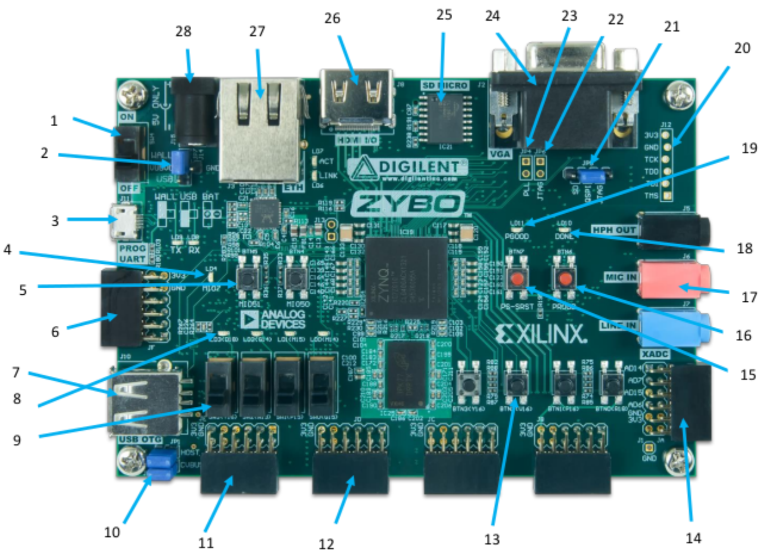
- The microcontroller re-enables interrupts, allowing them to occur again;
- The microcontroller returns to the task it was performing before the interrupt occurred.

3.2 Zybo Z-7010

The board to be used will be a **ZYBO Z-7010** which offers us the following features:

- 650Mhz dual-core Cortex-A9;
- DDR3 memory controller with 8 DMA channels;
- High-bandwidth peripheral controllers: 1G Ethernet, USB 2.0, SDIO;
- Low-bandwidth peripheral controller: SPI, UART, CAN, I2C
- Reprogrammable logic equivalent to Artix-7 FPGA:
  - 4,400 logic slices, each with four 6-input LUTs and 8 flip-flops;
  - o 240 KB of fast block RAM;
  - Two clock management tiles, each with a phase-locked loop (PLL) and mixed-mode clock manager (MMCM);
  - 80 DSP slices;
  - Internal clock speeds exceeding 450MHz;
  - On-chip analog-to-digital converter (XADC).
- GPIO: 6 pushbuttons, 4 slide switches, 5 LEDs;
- ZYNQ XC7Z010-1CLG400C;
- Six Pmod connectors (1 processor-dedicated, 1 dual analog/digital, 3 high-speed differential, 1 logic-dedicated);
- among others.

In the Figure 3 is illustrated the ZYBO Zynq-7000 development board and the respective components identification.



Callout	Component Description	Callout	Component Description
1	Power Switch	15	Processor Reset Pushbutton
2	Power Select Jumper and battery header	16	Logic configuration reset Pushbutton
3	Shared UART/JTAG USB port	17	Audio Codec Connectors
4	MIO LED	18	Logic Configuration Done LED
5	MIO Pushbuttons (2)	19	Board Power Good LED
6	MIO Pmod	20	JTAG Port for optional external cable
7	USB OTG Connectors	21	Programming Mode Jumper
8	Logic LEDs (4)	22	Independent JTAG Mode Enable Jumper
9	Logic Slide switches (4)	23	PLL Bypass Jumper
10	USB OTG Host/Device Select Jumpers	24	VGA connector
11	Standard Pmod	25	microSD connector (Reverse side)
12	High-speed Pmods (3)	26	HDMI Sink/Source Connector
13	Logic Pushbuttons (4)	27	Ethernet RJ45 Connector
14	XADC Pmod	28	Power Jack

Figure 3: ZYBO Device Diagram

4 Design

4.1 Instruction Set

As mentioned earlier, the instruction set to follow will be the Intel MCS-51, and using an approach where the instruction format is fixed and formed by 2 bytes, the following instructions have been selected:

- Arithmetic Operations:
  - Add:
  - Subb.
- Logical Operation:
  - And;
  - Or;
  - Xor.
- Boolean:
  - Jump Not Carry;
- Program Branching:
  - Jump Zero:
  - Jump Not Zero;
  - Short jump;
  - Reti.
- Data Transfer Operations:
  - 7 different Mov's:

4.2 RAM

The RAM will have 256 bytes, containing a bank of registers from position 0 to 7. The RAM has split in two virtual blocks, each one with 128 bytes, being the first block from position 0 to 127 and representing the internal data memory, and the second block from 128 to 255, that represents the SFR space. Still, it is necessary to include a register bank, with 8 registers, which will be positioned fixedly in positions 0 to 7.

Since the addressing mode for accessing registers is different from that used to access RAM elements, different control signals were used for reading and writing registers and RAM directs. A dual port memory will be used for this, allowing registers and direct RAM values to be modified simultaneously. In figure 4 it is shown the representation of the control signals for modifying the values in RAM.

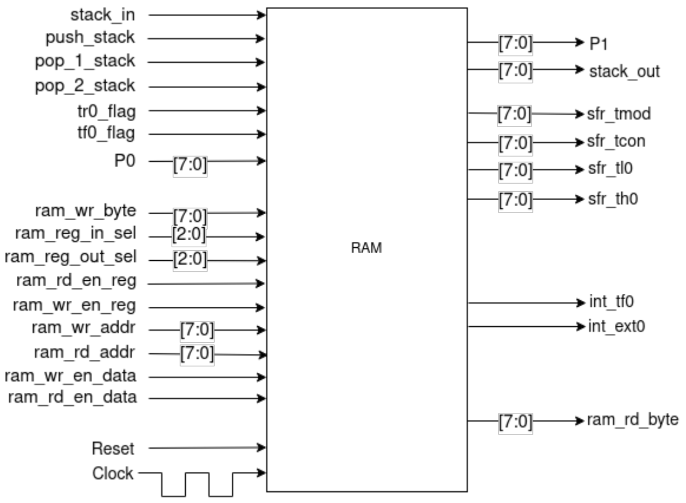


Figure 4: RAM schematic

In this scheme, it is also represented the control signals that allowed dealing with the SFRs. One of the fundamental SFRs for implementing ISR is the stack pointer. This register should be incremented when a push is identified and the opposite if a pop occurs. The stack pointer value always represents the address of the last element in the stack, and the reset value assigned is 8, i.e. the stack position cannot be changed, and its first value is always stored in RAM position 8.

4.3 ROM

The ROM to be designed can hold 1024 bytes, that is, each memory location will contain 8 bits. The ROM holds all the instructions that will be executed by the program. The next instruction to be executed is the memory position that corresponds to the value of the Program Counter. As Figure 5 shows, whenever you want to read the value of a certain position, it is necessary to enable the rom_en signal.

The first instruction to be executed will be at position 0x3B in order to avoid collision with the dedicated interrupt vectors.

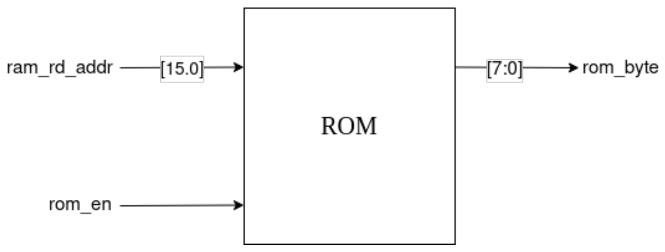


Figure 5: ROM schematic

4.4 Datapath

This section will explore the various elements that make up the datapath, identifying which signals will interfere with its behavior. Likewise, an overview of the datapath will be presented at the end.

4.4.1 Register 8-bit

This module will essentially update a value in the input to the register, synchronously, as shown in Figure 6.

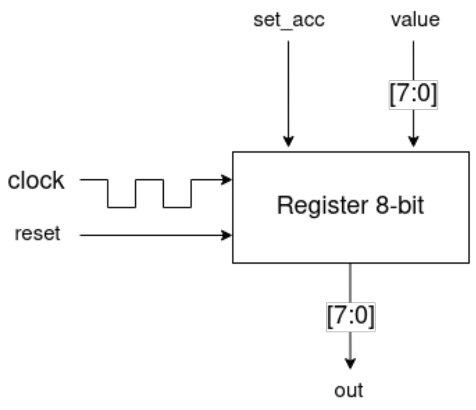


Figure 6: Register 8-bit schematic

The register input will be associated with *value*, and the *set_acc* signal, when positive, updates the internal value of the register with the value of the input. Signal *out* holds the internal value of the register.

4.4.2 Instruction Register

As shown in Figure 7, this module is quite similar to the previous one, however, the internal register has 16 bits.

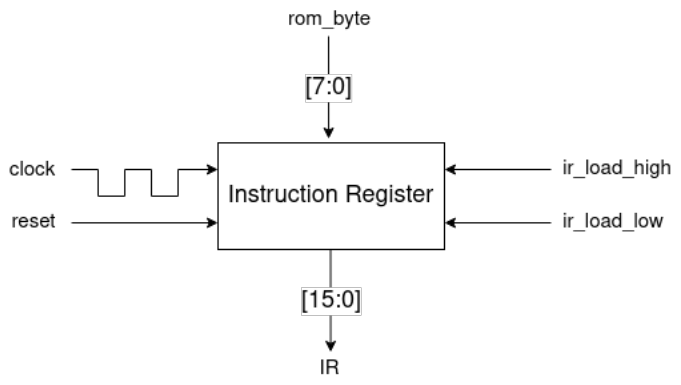


Figure 7: Instruction Register schematic

So, in the same way, when *ir_load_high* takes on value **1'b1**, the 8 least significant bits of the Instruction Register are updated. If *ir_load_low* takes on a positive value, it is the 8 most significant bits that are updated. The input is nothing more than the output of the ROM. The *IR* port passes the internal value of the Instruction Register to the output.

4.4.3 Arithmetic Logic Unit

Regarding the ALU, it is represented in the scheme of Figure 8.

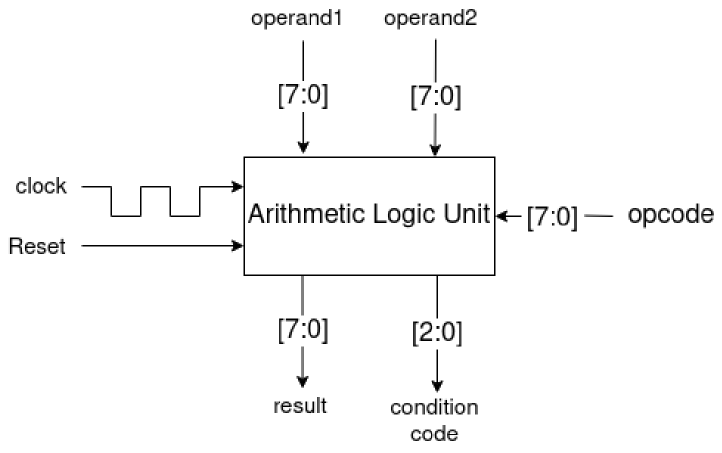


Figure 8: ALU schematic

It has three inputs, two corresponding to the operands to perform the operation and the other to indicate the instruction to be performed. In a simple way, the *opcode* will be used to identify which operation is to be performed. Finally, the output shows the value of the result, as well as the resulting condition flags, identified by *condition code*.

4.4.4 Program Counter

As for Program Counter, its design is more complex than the previous modules. The program counter will have to deal with program branching instructions, and with situations where interruptions occur. Figure 9 demonstrates the Program Counter schematic.

Starting from the principle, the *inc* flag will tell the counting register that it should be incremented by 1. Subsequently, the set of *inc_offset, jmp_z, jmp_nz, jmp_nc* flags differ only by the associated condition code flags. While the first one need not check any signals, the other three need to check the condition code flags to modify the Program Counter register. If any of these conditions are fully met the current value of the count register is incremented with the offset specified by *newValue_8bit*.

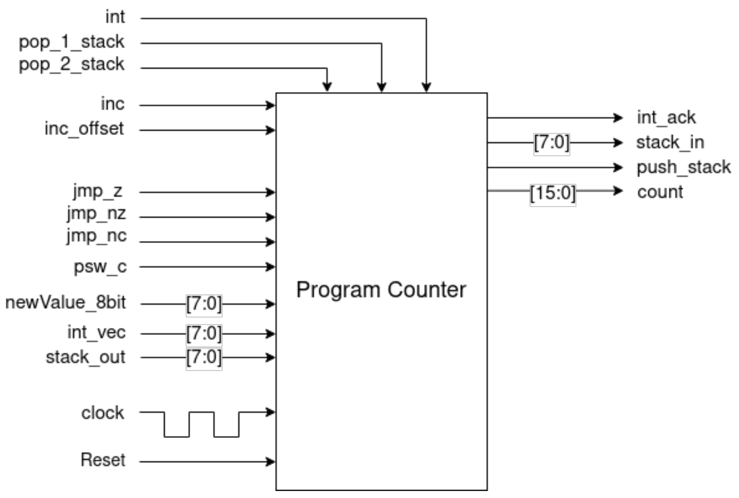


Figure 9: Program Counter schematic

Finally, in order to handle interrupts whenever they occur, the control signal *int* will have a positive value whenever one occurs. Then, the *push_stack* signal will be set to 1 in order to send the current count values to the stack, and afterwards, the *int_ack* flag is set indicating that an ISR is being executed. When an *pop_2_stack* at 1 is detected, the value of the *int_ack* signal is set to 0, and the program counter continues on its normal path.

4.4.5 General

The datapath, which describes the transformations of the contents of the registers, is represented in Figure 10. It is possible to see the representation of all the modules belonging.

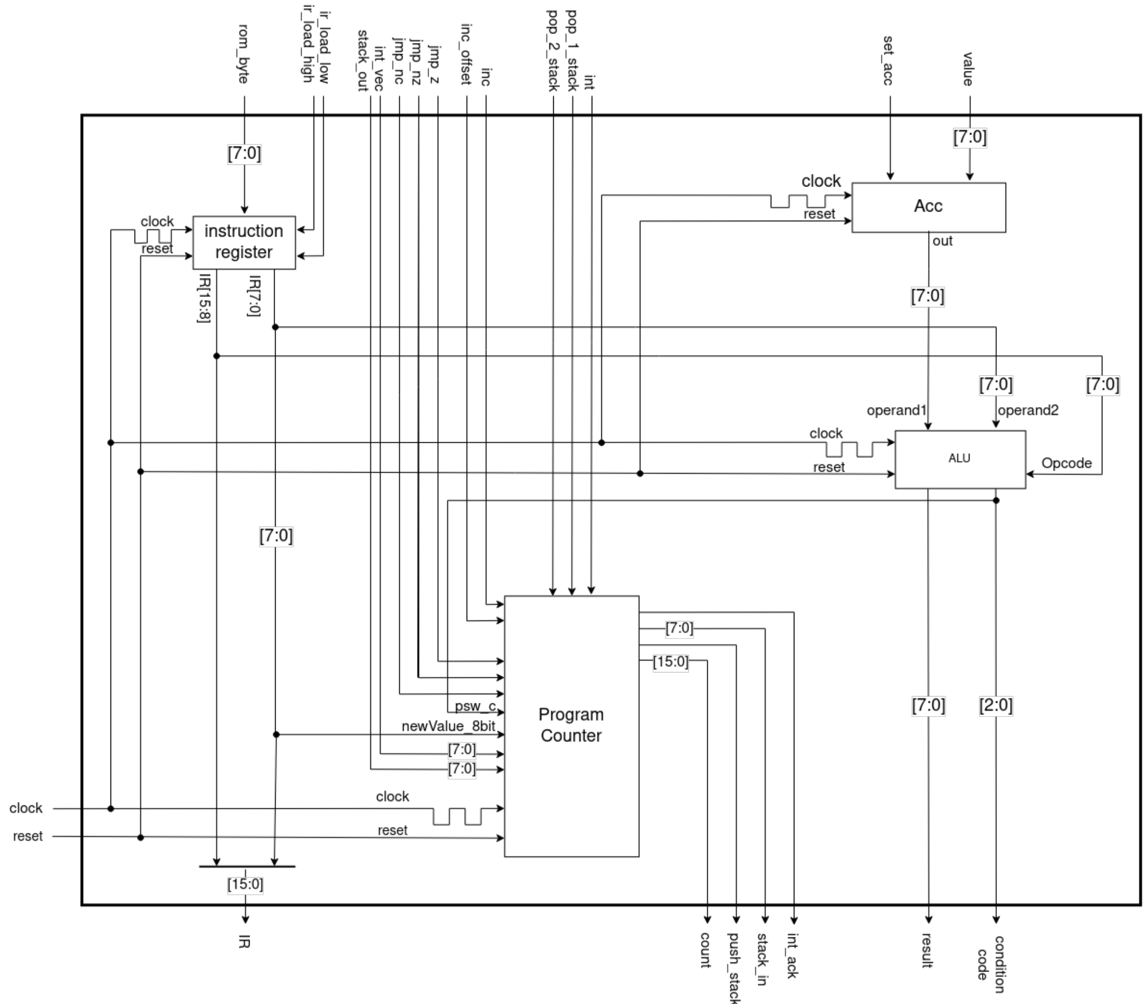


Figure 10: Datapath schematic

4.5 Control Unit

The state machine that is intended to be used to carry out the desired instructions is represented in Figure 11.

This state machine is composed fundamentally in 5 main states - start, fetch1, fetch2, decode and execute - having one more state for instructions

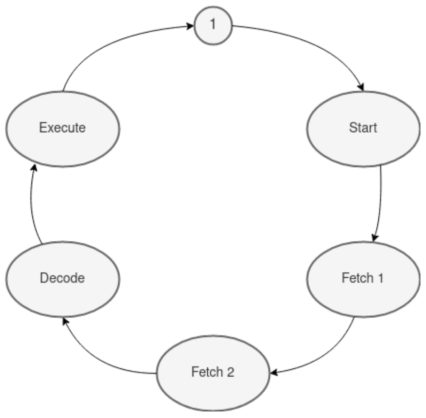


Figure 11: Control Unit schematic

that require one more clock cycle to execute. The execute state is further decomposed into several states that represent the various instructions, as shown in Figure 12.

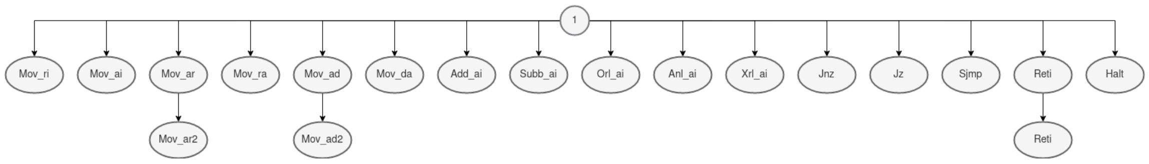


Figure 12: Control Unit execute states schematic

4.6 Timer

The timer to be drawn will be timer 0 and modes 0, 1 and 2. The registers associated with this timer will be the registers TCON0, TMOD0, TL0 and TH0. These registers are in the last 128 bytes of the RAM memory.

The Timer's schematic is represented in Figure 13. Basically, the input ports are for updating the value of the SFRs present in RAM, and signal *P0_1* is the pin associated with the input that will function as the clock source if the counter mode is selected. The *tf0_flag* flag indicates that an overflow has occurred, while *tr0_flag* indicates when the count is running.

4.7 Interrupt Controller

The interrupt control will inform the processor that an interrupt has occurred and which interrupt vector it is associated with. The events responsible for

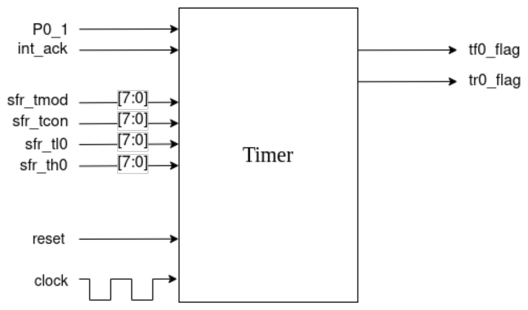


Figure 13: Timer schematic

causing an interrupt signal the processor. The events chosen for our adaptation to the 8051 will be: the external interrupt and the timer 0 overflow. The Interrupt Control schematic is shown in Figure 14.

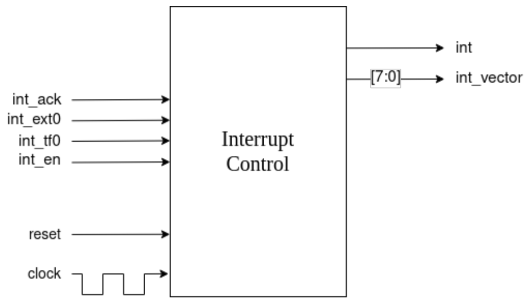


Figure 14: Interrupt Control schematic

The interrupt control system will be a vectored controller, that is, each interrupt has a special address in the interrupt vector. After the execution of the current instruction being executed, the *int_en* signal will have a positive value, and it will be checked if any events were recorded during the process. If so, the *int* flag will take on a positive value and the program counter value will be updated with the address of the vector associated with the interrupt, i.e, the value of *int_vector*.

4.8 Tools

All the tools used during the design phase which will be used in the implementation phase will be identified and described.

- **Xilinx Vivado**: for synthesis and analysis of hardware description language (HDL) designs;

- **GitHub**: provider of Internet hosting for software development and version;
- **Draw.io**: is a web-based, free and open-source diagramming software that allows users to create and edit a variety of diagrams.

5 Implementation

5.1 Global Constants

First, the file containing the opcodes of all the instructions that will be used in this project was written. It behaves like a verilog header that contains the instruction set.

```
`ifndef _OPCODES_V_
`define _OPCODES_V_

//Data transfer
`define MOV_RI   8'b0111_1xxx
`define MOV_AR   8'b1110_1xxx
`define MOV_RA   8'b1111_1xxx
`define MOV_AI   8'b0111_0100
`define MOV_AD   8'b1110_0101
`define MOV_DA   8'b1111_0101

//Arithemtic
`define ADD      8'b0010_01xx
`define ADD_AI   8'b0010_0100
`define SUBB     8'b1001_01xx
`define SUBB_AI  8'b1001_0100

//Logical
`define ANL      8'b0101_01xx
`define ANL_AI   8'b0101_0100
`define ORL      8'b0100_01xx
`define ORL_AI   8'b0100_0100
`define XRL      8'b0110_01xx
`define XRL_AI   8'b0110_0100

//Bolean
`define JNC      8'b1010_0000

//Program Branching
`define JZ       8'b0110_0000
`define JNZ      8'b0111_0000
`define SJMP     8'b1000_0000
`define RETI     8'b0011_0010

`define HALT     8'b1001_1000

`endif
```

Listing 1: Opcodes for all instructions.

5.2 RAM

Afterwards, the Random Access Memory (RAM) was developed. Before looking at the RAM module, two instantiated ram modules will be analyzed, Register bank and Debounce modules.

5.2.1 Register bank

In the Register bank module was set the input and output ports, line 2 to 9 in the Listing 2.

```
1  module register_bank(
2  input clock,
3  input reset,
4  input write_data,
5  input read_data,
6  input [2:0] reg_in_select,  //register to write in
7  input [2:0] reg_out_select, //register to read from
8  input [7:0] reg_in_data,
9  output [7:0] reg_out_data
10 );
```

Listing 2: Register Bank I/Os.

In the following Listing 3 is the register bank that uses a positive edge-triggered flip-flop to update a register bank *registerb* that consists of eight 8-bit registers. The register bank and an output wire *output_value* are declared as 8-bit vectors. The *reg[7:0]* and *registerb[7:0]* are 8-bit vectors. The code implements a sequential circuit that updates the values stored in the register bank based on two inputs, *reset* and *write_data*. If *reset* is high, the code block initializes all the 8 elements of the register bank to 0 using a for loop. If *write_data* is high, the code block updates the register in the register bank specified by the *reg_in_select* input with the value of *reg_in_data*. The value stored in the selected register will be updated on the next positive edge of the clock input.

```
reg[7:0] registerb [7:0];   //register bank
wire [7:0] output_value;
integer i;

always @(posedge clock)
begin
    if(reset)
    begin
        for(i = 0; i < 8; i = i+1) begin
            registerb[i] <= 8'd0;
        end
    end
    else if(write_data)
    begin
        registerb[reg_in_select] <= reg_in_data;
    end
end
```

Listing 3: Register Bank data and code block.

The only assign statement of the register bank, Listing 4, assigns the value of *reg_out_data* based on the value of *read_data*. The *reg_out_data* is assigned the value of *registerb[reg_out_select]* if *read_data* is **1'b1**. Otherwise, *reg_out_data* is assigned the value **8'hzz**.

```
assign reg_out_data = (read_data == 1'b1) ?  registerb[
  reg_out_select] : 8'hzz;
```

Listing 4: Register Bank assign statement.

5.2.2 Debounce

In order to remove the bouncing effect from switch inputs that can lead to incorrect readings and cause errors the Debounce module was written. In the Debounce module was set the input and output ports, takes an input *clock* and *button*, and produces an output *outDeb*, Listing 5.

```
module debounce(
input clock,
input button,
output outDeb
);
```

Listing 5: Debounce I/Os.

In the below code block of the Debounce module, Listing 6, it is implemented a simple, efficient debouncing circuit for a switch input. The module detects when the button has been pressed for a sufficient number of clock cycles, filters out any bouncing, and produces a clean output signal. The module has two 10-bit counters, *ctr_d* and *ctr_q*, and two 2-bit flip-flops, *sync_d* and *sync_q*, that are used to track the state of the input button. In the initial block the initial values of *ctr_d*, *ctr_q*,*sync_d*, and *sync_q* to 0. The first always block implements the state machine that counts the number of consecutive clock cycles during which the button is pressed. The state machine increments *ctr_d* by 1 on every clock cycle and resets it to 0 if the button is not pressed. The second always block updates the values of *ctr_q* and *sync_q* on every positive edge of the clock input.

```
reg [9:0] ctr_d, ctr_q;
reg [1:0] sync_d, sync_q;
```

```

initial begin
    ctr_d <= 0;
    ctr_q <= 0;
    sync_d <= 0;
    sync_q <= 0;
end

always @(*) begin
    sync_d[0] = button;
    sync_d[1] = sync_q[0];
    ctr_d = ctr_q + 1'b1;

    if (ctr_q == {10{1'b1}}) begin
        ctr_d = ctr_q;
    end

    if (!sync_q[1])
        ctr_d = 10'd0;
end

always @(posedge clock) begin
    ctr_q <= ctr_d;
    sync_q <= sync_d;
end
```

Listing 6: Debounce data and code block.

The output *outDeb* is assigned the value of 1 if *ctr_q* reaches its maximum value of 1023, which indicates that the button has been pressed for a sufficient number of clock cycles to be considered a valid input. Otherwise, *outDeb* is assigned the value of 0.

```
assign outDeb = ctr_q == {10{1'b1}};
```

Listing 7: Debounce assign statement.

5.2.3 RAM

The listing below shows the *RAM* module I/Os, it has multiple inputs and outputs defined between the 2 and 34 lines. The inputs include read and write enables for both the register bank and the RAM memory, address inputs for reading and writing, and the data input for writing. The outputs include the data read from the RAM, direct input to the stack, push and pop operations for the stack, flag inputs, inputs and outputs for IO ports (P0 and P1), and outputs for Special Function Registers (SFRs) related to timer and stack, Listing 8.

```
module ram(
input clock,
input reset,
input ram_rd_en_reg,
input ram_wr_en_reg,
input [2:0] ram_reg_in_sel,
input [2:0] ram_reg_out_sel,
input ram_rd_en_data,
input ram_wr_en_data,
input [7:0]ram_rd_addr,
input [7:0]ram_wr_addr,
input [7:0]ram_wr_byte,
output [7:0]ram_rd_byte,


input [7:0] stack_in,
input push_stack,
input pop_1_stack,
input pop_2_stack,

input tf0_flag,
input tr0_flag,
input [7:0]P0,

output [7:0]P1,
output int_tf0,
output int_ext0,

output [7:0]sfr_tmod,
output [7:0]sfr_tcon,
output [7:0]sfr_tl0,
output [7:0]sfr_th0,

output [7:0]stack_out
);
```

Listing 8: RAM I/Os.

Carrying the RAM module reaching the code block its implemented an 8-bit memory system with a stack, input and output ports, and a timer module. The memory system has 256 bytes of memory, and the stack pointer is initialized to **8'h08**, since the stack area actually begins from **8'h08**.

The first always block updates the previous state of the *tr0* and *tf0* flags from the timer, on the positive edge of the clock signal. The second always block handles various operations, including resetting the memory and stack, writing data to the RAM memory, updating the stack pointer, updating the status of the timer and updating the input and output ports, possible to verify in the Listing 9.

```
initial begin
    for(i=0; i < 256 ; i = i + 1)
        mem[i] = 8'b00000000;

    stack_pointer = 8'h08;
end

always @(posedge clock) begin
    last_tr0 <= tr0_flag;
    last_tf0 <= tf0_flag;
end

always @(posedge clock)
begin
    if(reset==1'b1) begin
        for(i=0; i < 256 ; i = i + 1)
            mem[i] = 8'b00000000;

        stack_pointer <= 8'h08;
    end
    else if(ram_wr_en_data)
    begin
        mem[ram_wr_addr] <= ram_wr_byte;
    end
    else if(push_stack == 1'b1) begin
        stack_pointer = stack_pointer + 1'b1;
        mem[stack_pointer] <= stack_in;
    end
    else if(pop_1_stack==1'b1 || pop_2_stack==1'b1) begin
        stack_pointer <= stack_pointer - 1'b1;
    end
    else begin
        if(pe_tf0)
            mem[tcon_addr][tf0_bit] <= 1'b1;
        if(pe_tr0)
            mem[tcon_addr][tr0_bit] <= 1'b1;

        mem[p0_addr] <= P0;
    end
end
```

Listing 9: RAM code block.

Ending the RAM module are described various assignments of signals and the behavior of the system. The first line assigns the *ram_rd_byte* signal, which is a value to be read from memory. It will be assigned the value stored in memory location specified by *ram_rd_addr* if *ram_rd_en_data* is 1 and *ram_rd_en_reg* is 0, otherwise it will be assigned to a don't care value, **8'hzz**. The next two lines assign the values of the *pe_tf0* and *pe_tr0* signals, which are positive edges of the *tf0_flag* and *tr0_flag* signals, respectively. The following four lines assign the values of the *sfr_tmod*, *sfr_tcon*, *sfr_tl0*, and *sfr_th0* signals to the values stored in memory at the specified addresses. The *int_ext0* signal is assigned 1 if either the external interrupt or all interrupt enable bit is set and *p0_0_deb* is equal to 1. The *int_tf0* signal is assigned 1 if either the timer0 interrupt or all interrupt enable bit is set and the *tf0_flag* is equal to 1. The *P1* signal is assigned the value stored in memory at the specified *p1_addr* location. The *stack_out* signal is assigned the value stored in memory at the current *stack_pointer* location.

```
assign ram_rd_byte = (ram_rd_en_data == 1'b1 && ram_rd_en_reg
    == 1'b0) ? mem[ram_rd_addr] : 8'hzz;

assign pe_tf0 = tf0_flag & ~last_tf0;
assign pe_tr0 = tr0_flag & ~last_tr0;

assign sfr_tmod = mem[tmod_addr];
assign sfr_tcon = mem[tcon_addr];
assign sfr_tl0  = mem[tl0_addr];
assign sfr_th0  = mem[th0_addr];

assign int_ext0 = ((mem[ie_addr][ex0_bit] == 1'b1 || mem[
   ie_addr][ae_bit] == 1'b1) && p0_0_deb == 1'b1) ? 1'b1 : 1'
   b0;
assign int_tf0  = ((mem[ie_addr][et0_bit] == 1'b1 || mem[
   ie_addr][ae_bit] == 1'b1) && tf0_flag == 1'b1) ? 1'b1 : 1'
   b0;

assign P1 = mem[p1_addr];
assign stack_out = mem[stack_pointer];
```

Listing 10: RAM assign statements.

5.3 ROM

Next, and following the same procedure as the RAM implementation, the ROM (read-omly-memory) memory was implemented, where the instructions executed by the microcontroller will be stored. The listing below shows the *ROM* module, where the ports to be used between the 2 and 4 lines are defined and subsequently the behavior adopted for reading the data stored in a given position. In essence, the ROM module will serve as the program memory of the microcontroller, where the instructions are stored and can be executed.

The module takes two inputs, *rom_en* that is a single-bit signal that enables reading from the memory and *rom_addr* a 16-bit address that specifies the location of the byte to be read. Also *rom_byte* a 8-bit data byte that is output from the memory.

```
module rom(
    input rom_en,
    input [15:0] rom_addr,
    output [7:0] rom_byte
);
```

Listing 11: Rom I/Os.

On following ROM implementation the memory is implemented as a register array *ROM* that is initialized in the initial block with a fixed set of instructions. The instructions are binary representations of assembly code operations for the microcontroller. For example, *ROM[3] = 8'b11100101;* sets the fourth location of the memory (indexed from 0) to 11100101 in binary, which is an instruction to move data directly to the accumulator register of the microcontroller.

```
reg [7:0] ROM[0:1040];  //1k

initial begin

ROM[3] = 8'b11100101;   //direct to acc
ROM[4] = 8'h90;         //direct is P1
ROM[5] = 8'b00100100;   //add imm to acc
ROM[6] = 8'h01;         //add one
ROM[7] = 8'b11110101;   //Mov Acc to direct 8ch
ROM[8] = 8'h90;
ROM[9] = 8'b00110010;
ROM[10] = 8'h00;

//Subtract 1 to the value of R1
ROM[11] = 8'b11101001;  //get the value of R1 to acc
ROM[12] = 8'h00;
ROM[13] = 8'b10010100;  //subtrat 1 to the value of acc
ROM[14] = 8'h01;
ROM[15] = 8'b11111001; //updat R0 with acc value
ROM[16] = 8'h00;

//Jump if R1 is 0
ROM[17] = 8'b01110000;  //jump if not zero
ROM[18] = 8'h08;

//If zero, increment the value of P0
ROM[19] = 8'b11100101;   //direct to acc
ROM[20] = 8'h90;         //direct is P1
ROM[21] = 8'b00100100;   //add imm to acc
ROM[22] = 8'h01;         //add one
```

```
ROM[23] = 8'b11110101;   //Mov Acc to direct 8ch
ROM[24] = 8'h90;

//Update again the value of R1 to ff
ROM[25] = 8'b01111001;   //update the value of R1 to ff
ROM[26] = 8'hff;

//Reti
ROM[27] = 8'b00110010;   //Mov Acc to direct 89h
ROM[28] = 8'h00;


//********** Turn on led and when timer on call isr
    ************//
ROM[59] = 8'b01110100;   //mov imm to acc
ROM[60] = 8'b00000001;   //value
ROM[61] = 8'b11110101;   //mov acc to dir P1
ROM[62] = 8'h90;        //position in memory
ROM[63] = 8'b01110100;   //mov imm to acc
ROM[64] = 8'h01;         //TMOD 0
ROM[65] = 8'b11110101;   //Mov Acc to direct 88h
ROM[66] = 8'h88;
ROM[67] = 8'b01110100;   //mov imm to acc
ROM[68] = 8'h00;         //load TL0
ROM[69] = 8'b11110101;   //Mov Acc to direct 8ch
ROM[70] = 8'h8a;
ROM[71] = 8'b01110100;   //mov imm to acc -----------------
ROM[72] = 8'h00;         //load TH0
ROM[73] = 8'b11110101;   //Mov Acc to direct 8ch
ROM[74] = 8'h8c;
ROM[75] = 8'b01110100;   //mov imm to acc
ROM[76] = 8'b10000011;   //enable EA
ROM[77] = 8'b11110101;   //Mov Acc to direct 89h EA
ROM[78] = 8'ha8;
ROM[79] = 8'b01110100;   //mov imm to acc
ROM[80] = 8'b00010000;   //enable TR0
ROM[81] = 8'b11110101;   //Mov Acc to direct 89h
ROM[82] = 8'h89;
ROM[83] = 8'b01111001;   //update the value of R1 to ff
ROM[84] = 8'hff;

//HALT
ROM[85]= 8'b10011000;   //halt
ROM[86]= 8'h00;
ROM[87]= 8'b10011000;   //halt
ROM[88]= 8'h00;
```

```
end
```

Listing 12: Rom data and code block.

The only assign statement of the ROM module, Listing 13, maps the output *rom_byte* to the corresponding byte in the memory if *rom_en* is high, otherwise it assigns the value **8'hzz** (high-impedance) to *rom_byte*

```
assign rom_byte = (rom_en == 1'b1) ? ROM[rom_addr] : 8'hzz;
```

Listing 13: Rom assign statement.

5.4 Datapath

First, before the general datapth implementation is demonstrated, the modules used to realize it will be presented. These modules improve the organization as well as the efficiency of the program, making it easier to maintain.

5.4.1 Instruction Register

For the Datapath implementation, initially was the creation of the module *instruction_register*, which can be seen in the Listings below, 14 and 15. It is a 16-bit register that stores the instruction to be executed. The *ir_load_high* is a signal to load the 8 most significant bits of the instruction into the register, the *ir_load_low* is a signal to load the 8 least significant bits of the instruction into the register, and *rom_byte* is the 8-bit input that provides the instruction data to be loaded into the register. The output of the circuit, represented by *IR*, is the 16-bit instruction stored in the register.

```
module instruction_register(
input clock,
input reset,
input ir_load_high,
input ir_load_low,
input [7:0] rom_byte,
output reg [15:0] IR
);
```

Listing 14: Instruction Register I/Os.

In the always block triggered by the posedge of the clock signal, and simply updates its value when one of its control signals is positive. Thus, this module takes as input the values that are put into the ROM output, and receives its value whenever *ir_load_high* and *ir_load_low* are set to one, Listing 15. If *ir_load_high* is high, the high 8 bits of the instruction register are set to the value of *rom_byte*. If *ir_load_low* is high, the low 8 bits of the instruction register are set to the value of *rom_byte*.

```
initial begin
    IR = 0;
end

always @(posedge clock)
begin
    if(reset)
    begin
        IR <= 0;
    end
    else if(ir_load_high)
    begin
        IR[15:8] <= rom_byte;
    end
    else if (ir_load_low)
    begin
        IR[7:0] <= rom_byte;
    end
 end
```

Listing 15: Instruction Register code block.

5.4.2 8-bit Register

It was decided to implement a module specifically for 8-bit registers, thus making the interpretation of the code easier and favoring a modular code structure. Thus, this module will have an input *set_acc* used to control when the value of the register should be set to *value*. The input *value* is an 8-bit input that specifies the value that will be written to the register when *set_acc* is high. Finnaly, *out* is the 8-bit output that holds the value of the register.

```
module register_8bit(
input clock,
input reset,
input set_acc,
input [7:0]value,
output reg [7:0] out
);
```

Listing 16: 8-bit Register I/Os.

Next the in the Listing 17, is defined the behavior of the register. If *reset* is high, the register is set to 0. If *reset* is low and *set_acc* is high, the value of value is written to the register. This means that the value stored in the register will change only when clock goes from high to low, and either *reset* is high or *set_acc* is high and *reset* is low.

```
initial begin
```

```
    out <= 8'b0;
end
always @(negedge clock)
begin
    if(reset)
        out <= 8'b0;
    else if (set_acc) begin
        out <= value;
    end
end
```

Listing 17: 8-bit Register code block.

5.4.3 Program Counter

Still in the Datapath, is now defined the Program Counter (PC) module, designated *program_counter*. This module contains the program counter register that holds the address of the current instruction being executed. In terms of inputs *inc*, *inc_offset*, *jmp_z*, *jmp_nz*, and *jmp_nc* are used to control the incrementing of the program counter and the *newValue_8bit* input is used in conjunction with the *inc_offset* input to specify the offset to add to the program counter. One can easily notice that the *jmp_z*, *jmp_nz*, and *jmp_nc* inputs are used for jump instructions. Additionally the program counter is updated based on the value of *accValue* input and the *psw_c* input (which represents the carry flag).

In the module is also implemented an interrupt handling feature. The input *int* signals an interrupt, and the *int_vec* input specifies the interrupt vector to jump to. When an interrupt occurs, the program counter's current value is pushed onto a stack and the program counter is updated with the interrupt vector. The *int_ack* output signals that the interrupt has been acknowledged, and the *stack_in* output is used to push the current value of the program counter onto the stack. The *pop_1_stack* and *pop_2_stack* inputs are used to pop values off the stack and update the program counter, Listing 18 and Listing 19.

```
module program_counter(
input clock,
input reset,
input inc,
input inc_offset,
input jmp_z,
input jmp_nz,
input jmp_nc,
input psw_c,
input [7:0] newValue_8bit,
input int,
input [7:0] int_vec,
input pop_1_stack,
input pop_2_stack,
input [7:0] accValue,
input [7:0] stack_out,
output reg int_ack,
output reg[7:0] stack_in,
output push_stack,
output reg[15:0] count
);
```

Listing 18: Program Counter I/Os.

Going through the module of the Program Counter is employed a finite state machine (FSM) to manage the interrupt handling process.

The FSM has three states:

- *s_idle* = idle state;
- *s_push1* = first push to stack;
- *s_push2* = second push to stack.

```
reg [1:0]state;
wire set_vec;

// idle -> if an extra cycle is required to execute the
   instruction
parameter s_idle  =  2'b00,
          s_push1 =  2'b01,
          s_push2 =  2'b10;

initial begin
    count <= 15'h003B;
    state <= s_idle;
    int_ack <= 1'b0;
    stack_in <= 8'hzz;
end

always @ (posedge clock)
begin
    if(reset)
        count <= 15'h003B;
    else if (inc)
        count <= count + 1;
    else if (inc_offset)
        count <= count + newValue_8bit;

    else if(jmp_z & accValue == 8'b0) begin
        count <= count + newValue_8bit;
    end
    else if(jmp_nz & accValue != 8'b0) begin
        count <= count + newValue_8bit;
    end
    else if(jmp_nc & (~psw_c)) begin
        count <= count + newValue_8bit;
    end
    else if(pop_1_stack) begin
        count[15:8] <= stack_out;
    end
    else if(pop_2_stack) begin
        count[7:0] <= stack_out;
        int_ack <= 1'b0;
    end
    else if(set_vec==1'b1) begin
        count <= {8'h00 , int_vec};
        int_ack <= 1'b1;
    end
end

always @ (posedge clock)
begin : FSM
    begin
        case(state)
            s_idle :
            begin
                if(int == 1'b1) begin
                    state <= s_push1;
                    stack_in <= count[7:0];
                end
            end
            s_push1:
            begin
                stack_in <= count[15:8];
                state <= s_push2;
            end
            s_push2:
            begin
                state <= s_idle;
            end
        default:
            state <= s_idle;
     endcase
    end
end
```

Listing 19: Program Counter data and code block.

The last part of the Program Counter module is the assigns part where the *push_stack* output signals when a value is being pushed onto the stack, and the *set_vec* output signals when the program counter is being updated with the interrupt vector, Listing 20.

```
assign push_stack = (state == s_push1 || state == s_push2) ?
   1'b1 : 1'b0;
assign set_vec = (state == s_push2) ? 1'b1 : 1'b0;
);
```

Listing 20: Program Counter assign statements.

5.4.4 Arithmetic Logic Unit

Finally, is now defined the module of the Arithmetic Logic Unit (ALU), demonstrated in the listings below, Listing 21 and 22, where its implemented various arithmetic and logic operations. It has inputs for two 8-bit operands *operand1* and *operand2*, and an *opcode* for specifying the operation to be performed. It also has outputs for the 9-bit result of the operation and three flag signals *psw_c*, *psw_ac*, and *psw_ov* which indicate the carry, auxiliary carry, and overflow flags respectively.

```
module arithmetic_logic_unit(
    input clock,
    input reset,
    input [7:0]operand1,
    input [7:0]operand2,
    input [7:0]opcode,
    output reg [8:0]result
    output reg psw_c,
);
```

Listing 21: Arithmetic Logic Unit I/Os.

To perform the desired behavior, it receives as input two 8-bit operands and the operations are specified using a case statement, *casex* type decision mode, that selects the operation based on the value of the opcode input. Some of the operations implemented are MOV, ADD, SUBB, ANL (and), ORL (or), and XRL (exclusive or). The result of the operation is stored in the output result register.

The flags are updated based on the result of each operation. The carry flag is set if there's a carry out of the most significant bit, the auxiliary carry flag is set if there's a carry out of bit 3, and the overflow flag is set if the result exceeds the range of an 8-bit number.

```
`include "opcodes.v"

```

```
reg psw_ac;
reg psw_ov;

initial begin
    result = 0;
    psw_c = 0;
    psw_ac = 0;
    psw_ov = 0;
    result = 0;
end

always @(posedge clock)
begin
    if(reset)   begin
       result = 0;
        psw_c = 0;
        psw_ac = 0;
        psw_ov = 0;
    end
    else    begin
        casex(opcode)

            `MOV_AI : begin   // MOV to the accumulator
                result = operand2;
                end
            `MOV_AR : begin   // MOV to the accumulator
                result = operand2;
                end
            `MOV_RI : begin
                result = operand2;
                end
            `ADD : begin
                result = operand1 + operand2;
                psw_c = result[8];
                psw_ac = result[4];
                psw_ov = result[8]&&~result[7] || result
    [7]&&~result[8];
                end
            `SUBB : begin
                result = operand1 - operand2;
                psw_c = result[8];
                psw_ac = result[4];
                psw_ov = result[8]&&~result[7] || result
    [7]&&~result[8];
                 end
            `ANL: begin
                result = operand1 & operand2;
                end
            `ORL:begin
```

```
                result = operand1 | operand2;
                end
            `XRL:begin
                result = operand1 ^ operand2;
                end
             default:
                result = result;
        endcase
    end
end
endmodule
```

Listing 22: Arithmetic Logic Unit data and code block.

5.4.5 Datapath

With all the modules presented above, it is now possible to demonstrate the implementation of the Datapath module for the 8051 microcontroller and is very explicit in the Listings, 23, 24, and 25. This module implements the data path of a computer system. It has inputs related to memory (read and write), program counter, interrupt, stack, and outputs related to opcode, stack, memory address and data.

```
module datapath(
input clock,
input reset,
input [7:0] ram_rd_byte,
input ram_rd_en_reg,
input ram_wr_en_reg,
input ram_rd_en_data,
input ram_wr_en_data,

input [7:0] rom_byte,

input pc_inc,
input pc_inc_offset,
input ir_load_high,
input ir_load_low,
input acc_load,

input pc_jmp_z,
input pc_jmp_nz,
input pc_jmp_nc,

input int,
input [7:0] int_vec,

input pop_1_stack,
```

```verilog
input pop_2_stack,
input [7:0] stack_out,

output [7:0] opcode,
output int_ack,

output [7:0]stack_in,
output push_stack,

output [7:0] ram_rd_addr,
output [7:0] ram_wr_addr,
output [7:0] ram_wr_byte,

output wire [15:0] rom_addr
);
```

Listing 23: Datapath I/Os.

Now it is easily realized that that the module contains five sub-modules, *instruction_register*, *arithmetic_logic_unit*, *register_8bit*, *program_counter*, and some logic to handle different operations. Keeping in mind the auxiliary variables for their instantiating are declared.

```verilog
reg  [7:0]  PSW;
wire [15:0] IR;
wire [8:0]  ALU_result;

wire [7:0]  acc_in = (ram_rd_en_reg == 1'b1 || ram_rd_en_data
    == 1'b1) ? ram_rd_byte : ALU_result[7:0];

wire [7:0]  acc_out;
wire psw_c;

instruction_register    ir(clock, reset, ir_load_high,
    ir_load_low, rom_byte, IR);

arithmetic_logic_unit   alu(clock, reset, acc_out, IR[7:0],
    opcode[7:0], ALU_result[8:0], psw_c);

register_8bit   acc(clock, reset, acc_load, acc_in, acc_out);

program_counter pc_dp(clock, reset, pc_inc, pc_inc_offset,
    pc_jmp_z, pc_jmp_nz, pc_jmp_nc, psw_c, IR[7:0], int,
    int_vec, pop_1_stack, pop_2_stack,acc_out, stack_out,
    int_ack, stack_in[7:0], push_stack, rom_addr);
```

Listing 24: Datapath data and modules instantiations.

Last but not least, concerning the behavior of the module, there are also some logic operations performed to determine the memory address and data for read and write operations. The *opcode* is extracted from the instruction stored in *IR*. The write address for memory and data to be written to memory are assigned based on the input signals and the read address for memory is also assigned based on the input signals.

```
assign opcode = IR[15:8];

assign ram_wr_addr = (ram_wr_en_data == 1'b1) ? IR[7:0] : 8'hzz;

assign ram_rd_addr = (ram_rd_en_data == 1'b1) ? IR[7:0] : 8'hzz;

assign ram_wr_byte = (ram_wr_en_reg == 1'b1) ?
                     ((opcode[7:3] == 5'b11111) ? acc_out : IR[7:0])
                     :(ram_wr_en_data == 1'b1) ? acc_out : 8'hzz;

```

Listing 25: Datapath assign statements.

5.5 Control Unit

Regarding the Control Unit module to be developed, it was decided to create a state machine capable of satisfying all the conditions for the instruction set selected. Firstly the *opcode* is the 8-bit input representing the instruction opcode, *loading_stack* is a signal for loading data onto the stack and *int* is the interrupt signal. Summing up all the ouputs, they include signals for controlling the read and write operations of two different memory blocks (RAM and ROM), program counter manipulation, instruction register loading, accumulator loading, interrupt enabling, and stack operations, Listing 26.

```
module control_unit(
    input clock,
    input reset,
    input [7:0] opcode,
    input loading_stack,
    input int,
    output ram_rd_en_reg,
    output ram_wr_en_reg,
    output [2:0] ram_reg_in_sel,
    output [2:0] ram_reg_out_sel,
    output ram_rd_en_data,
    output ram_wr_en_data,
    output rom_en,
    output pc_inc,
    output pc_inc_offset,
    output pc_jmp_z,
    output pc_jmp_nz,
    output pc_jmp_nc,
    output ir_load_high,
    output ir_load_low,
    output acc_load,
    output int_en,
    output pop_1_stack,
    output pop_2_stack
);
```

Listing 26: Control Unit I/Os.

It was decided to create a state machine capable of satisfying all the conditions for the instruction set selected. First, we started by defining the machine states as parameters. Obviously, and since it is a generic controller, the first states to be identified were the fetch, decode and execute states, as in Listing 27.

- **s_start**: the initial stage of the control unit;
- **s_fetch_1, s_fetch_2**: the stages for fetching the first and second byte of an instruction respectively;
- **s_decode**: the stage for decoding the instruction;
- **s_mov_ri, s_mov_ai, s_mov_ar, s_mov_ra, s_add_ai, s_subb_ai, s_orl_ai, s_anl_ai, s_xrl_ai**: stages for executing different types of instructions;
- **s_jnz, s_jz, s_sjmp, s_jnc**: stages for executing different types of jump instructions;
- **s_halt**: the stage for halting the execution of the processor;
- **s_reti1, s_reti2**: 2 pops, stages for returning from an interrupt;
- **s_mov_ar_2, s_mov_ad, s_mov_ad_2**: stages for executing certain types mov of instructions;
- **s_extra_wr**: an extra stage for writing the result of an instruction In this way a solution was arrived at, demonstrated in the following listing.

The FSM changes the state of the *state* register based on the value of *reset* and *int*. If the *reset* signal is high (1), the FSM sets the *state* register to *s_start*. If the *int* signal is low (0), the FSM updates the state of the *state* register based on the current value. The update of the *state* register is done in a case statement based on the current value of the *state* register.

If the current state is *s_decode*, the FSM looks at the value of *opcode* and changes the *state* register to the corresponding state for the instruction specified by the opcode.

```
`include "opcodes.v"

reg [4:0] state;

//=========== Internal Constants =====================
    parameter s_start    = 5'b00000,
              s_fetch_1  = 5'b00001,
              s_fetch_2  = 5'b00010,
              s_decode   = 5'b00011,
              s_mov_ri   = 5'b00100,
              s_mov_ai   = 5'b00101,
              s_mov_ar   = 5'b00110,
              s_mov_ra   = 5'b00111,
              s_add_ai   = 5'b01000,
              s_subb_ai  = 5'b01001,
              s_orl_ai   = 5'b01010,
              s_anl_ai   = 5'b01011,
              s_xrl_ai   = 5'b01100,
              s_jnz      = 5'b01101,
              s_jz       = 5'b11100,
              s_sjmp     = 5'b01111,
              s_jnc      = 5'b10000,
              s_halt     = 5'b10001,
              s_reti1    = 5'b10010,
              s_reti2    = 5'b10011,
              s_mov_ar_2  = 5'b10100,
              s_mov_ad    = 5'b10101,
              s_mov_da    = 5'b10110,
              s_extra_wr  = 5'b10111,
              s_mov_ad_2  = 5'b11000;
//====================================================


initial begin
    state <= s_start;
end

always @ (posedge clock)    //CU - start -fetch1-wait-fetch2-
    decode-execute
```

```
begin : FSM
  if (reset == 1'b1) begin
    state <=  s_start;
  end
  else if(int == 1'b0)begin
    case(state)
      s_start :
      begin
          if(reset != 1'b1)
          begin
            state <= s_fetch_1;
        end
        end
      s_fetch_1:
        state <= s_fetch_2;
        s_fetch_2:
            state <= s_decode;
      s_decode :
                casex(opcode)
            `MOV_RI:
                state <= s_mov_ri;
                    `SJMP:
                        state <= s_sjmp;
                    `MOV_AI:
                        state <= s_mov_ai;
                    `MOV_AR:
                        state <= s_mov_ar;
                    `MOV_RA:
                        state <= s_mov_ra;
                    `MOV_DA:
                        state <= s_mov_da;
                    `MOV_AD:
                        state <= s_mov_ad;
                    `ADD_AI:
                        state <= s_add_ai;
                    `SUBB_AI:
                        state <= s_subb_ai;
                    `ORL_AI:
                        state <= s_orl_ai;
                    `ANL_AI:
                        state <= s_anl_ai;
                    `XRL_AI:
                        state <= s_xrl_ai;
                    `JNZ:
                        state <= s_jnz;
                    `JZ:
                        state <= s_jz;
                    `JNC:
                        state <= s_jnc;
                    `RETI:
                        state <= s_reti1;
                    `HALT:
                        state <= s_halt;
                    default :
                  state <= s_halt;
                 endcase
            s_mov_ri:
                state <= s_extra_wr;
            s_mov_ai:
                state <= s_start;
            s_mov_ar:
                state <= s_mov_ar_2;
            s_mov_ra:
                state <= s_extra_wr;
            s_mov_ad:
                state <= s_mov_ad_2;
            s_mov_da:
                state <= s_start;
            s_add_ai:
                state <= s_start;
            s_subb_ai:
                state <= s_start;
            s_orl_ai:
                state <= s_start;
            s_anl_ai:
                state <= s_start;
            s_xrl_ai:
                state <= s_start;
            s_jnz:
                state <= s_start;
            s_jz:
                state <= s_start;
            s_sjmp:
                state <= s_start;
            s_jnc:
                state <= s_start;
            s_mov_ar_2:
                state <= s_start;
            s_mov_ad_2:
                state <= s_start;
            s_reti1:
                state <= s_reti2;
            s_reti2:
                state <= s_start;
            s_extra_wr:
                state <= s_start;
            s_halt:
                state <= s_halt;
```

```
          default :
            state <= s_halt;
    endcase
  end
  else begin
     if(state == s_halt)
         state <= s_start;
  end
end
```

Listing 27: Control Unit data and code block.

Completing the control unit module in the subsequent Listing 28, the code describes the assignment of various control signals based on the current state of the processor. All the control signals will be used to control the operations of the processor and various components of it such as the Datapath, ALU (Arithmetic Logic Unit), ROM (Read Only Memory), RAM (Random Access Memory), and PC (Program Counter).

```
assign rom_en = (state == s_fetch_1 || state == s_fetch_2) ?
    1'b1 : 1'b0;
assign pc_inc = ((state == s_fetch_1 || state == s_fetch_2)
    && loading_stack == 1'b0) ? 1'b1 : 1'b0;

assign ir_load_high = (state == s_fetch_1) ? 1'b1 : 1'b0;
assign ir_load_low = (state == s_fetch_2) ? 1'b1 : 1'b0;

//Ram storage in register
assign ram_wr_en_reg = (state == s_mov_ri || state ==
    s_mov_ra || state == s_extra_wr) ? 1'b1 : 1'b0;
//Get for which register should be stored
assign ram_reg_in_sel = (state == s_mov_ri || state ==
    s_mov_ra || state == s_extra_wr) ? opcode[2:0] : 3'bzzz;

//Ram load to register
assign ram_rd_en_reg = (state == s_mov_ar || state ==
    s_mov_ar_2) ? 1'b1 : 1'b0;
assign ram_reg_out_sel = (state == s_mov_ar || state ==
    s_mov_ar_2) ? opcode[2:0] : 3'bzzz;

//Ram storage in direct
assign ram_wr_en_data = (state == s_mov_da) ? 1'b1 : 1'b0;
assign ram_rd_en_data = (state == s_mov_ad || state ==
    s_mov_ad_2) ? 1'b1 : 1'b0;

//acc_load control signal
assign acc_load = (state == s_mov_ai || state == s_add_ai ||
    state == s_subb_ai || state == s_mov_ar_2 || state ==
    s_mov_ad_2 || state == s_xrl_ai) ? 1'b1 : 1'b0;
```

```
assign alu_en = (state == s_decode) ? 1'b1 : 1'b0;

//Short jump with offset
assign pc_inc_offset = (state == s_sjmp) ? 1'b1 : 1'b0;
//Jump if not zero
assign pc_jmp_z = (state == s_jz) ? 1'b1 : 1'b0;
//Jump if zero
assign pc_jmp_nz = (state == s_jnz) ? 1'b1 : 1'b0;
//Jump if not cary
assign pc_jmp_nc = (state == s_jnc) ? 1'b1 : 1'b0;

assign int_en = (state == s_start || state == s_halt) ? 1'b1
    : 1'b0;

// stack
assign pop_1_stack = (state == s_reti1) ? 1'b1 : 1'b0;
assign pop_2_stack = (state == s_reti2) ? 1'b1 : 1'b0;
```

Listing 28: Control Unit assign statements.

5.6 Timer

One of the most important modules of the 8051 is the Timer module that was developed as far as you can only see from the Listings 29, ??, and 31.

How is it possible to analyze, Listing 29, *sfr_tmod* is the value stored in the register TMOD, *sfr_tcon* is the value stored in the register TCON, *sfr_tl0* is the value stored in the register TL0, *sfr_th0* is the value stored in the register TH0, *P0_1* an input from P0, and *int_ack* a signal indicating if an interrupt has been handled, which is used to clear the overflow flag. When it comes to the outputs, *tf0_flag* is the flag that indicates the timer overflow and *tr0_flag* is the flag that indicates that the timer is running.

```
module timer_8051(
input clock,
input reset,
input [7:0]sfr_tmod,
input [7:0]sfr_tcon,
input [7:0]sfr_tl0,
input [7:0]sfr_th0,
input  P0_1,
input int_ack,
output tf0_flag,
output tr0_flag
);
```

Listing 29: Timer I/Os.

Looking at the listing 30, the module implements a timer using the input values stored in the registers, and it updates the timer value based on the system clock. The timer can operate in three modes (0, 1, and 2) depending on the value stored in the *sfr_tmod* register. Each mode has different rules for updating the timer values. The timer can also operate in either timer or counter mode, depending on the value stored in the *sfr_tcon* register. The *P0_1* input is used in counter mode.

```
reg [7:0]tmod;
reg [7:0]tcon;
reg [7:0]tl0;
reg [7:0]th0;

reg last_P0_1;
reg p0_edge;
reg tr0_edge;
reg last_tr0;
wire pe_P0_1;

parameter tr0_bit = 3'b100, tf0_bit = 3'b101;

initial begin
    tmod = 8'h00;
    tcon = 8'h00;
    tl0 = 8'h00;
    th0 = 8'h00;
    last_P0_1 = 1'b0;
    p0_edge = 1'b0;
end

always @(posedge clock)
begin
    if(reset == 1'b1) begin
        tmod <= 8'h00;
        tcon <= 8'h00;
        th0 <= 8'h00;
        tl0 <= 8'h00;
    end
    else
    if(sfr_tcon[4]== 1'b0) begin
        tmod <= sfr_tmod;
        tcon <= sfr_tcon;
        th0  <= sfr_th0;
        if(sfr_tmod[1:0] == 2'h2)
            tl0 <= sfr_th0;
        else
            tl0  <= sfr_tl0;
    end
    else if(tmod[2] == 1'b0) begin
        tcon[4] <= 1'b1;
        case(tmod[1:0])
            0:  begin
                    if(tl0 == 8'h1f) begin
                        if(th0 == 8'hff)
                            tcon[5] <= 1'b1;

                        th0 <= th0 + 1;
                    end
                    tl0 <= (tl0 + 1) & 8'b00011111;
                end
            1:  begin
                    if(tl0 == 8'hff) begin
                        if(th0 == 8'hff)
                            tcon[5] <= 1'b1;

                        th0 <= th0 + 1;
                    end

                    tl0 <= tl0 + 1;
                end

            2:  begin
                    tl0 <= tl0 + 1;
                    if(tl0==8'hff) begin
                        tcon[5] <= 1'b1;
                    end
                end
        endcase
    end
    else if(pe_P0_1 == 1'b1) begin
        tcon[4] <= 1'b1;
        if(tcon[4]== 1'b1 && tmod[2] == 1'b1) begin
            case(tmod[1:0])
                0:  begin
                        if(tl0 == 8'h1f) begin
                            if(th0 == 8'hff)
                                tcon[5] <= 1'b1;

                            th0 <= th0 + 1;
                        end
                        tl0 <= (tl0 + 1) & 8'b00011111;
                    end
                1:  begin
                        if(tl0 == 8'hff) begin
                            if(th0 == 8'hff)
                                tcon[5] <= 1'b1;

                            th0 <= th0 + 1;
                        end
                        tl0 <= tl0 + 1;
                    end
                2:  begin
                        tl0 <= tl0 + 1;
                        if(tl0==8'hff) begin
                            tcon[5] <= 1'b1;
                        end
                    end
            endcase
        end
end

if(tcon[5] == 1'b1) begin
    case(tmod[1:0])
        0:  ;
        1:  ;
        2:  begin
            tl0 <= th0;
            end
    endcase
    if(int_ack)
        tcon[5] <= 1'b0;
    end
end

always @(posedge clock) begin
    last_P0_1 <= P0_1;
end
```

Listing 30: Timer data and code block.

The overflow flag (TF0) is set when the timer overflows, and the *int_ack* input is used to clear the flag. The *tr0_flag* is set when the timer is running.

```
assign pe_P0_1 = P0_1 & ~last_P0_1;
assign tf0_flag = tcon[5];
assign tr0_flag = tcon[4];
```

Listing 31: Timer assign statements.

5.7 Interrupt Controller

Reaching the Interrupt Control module it is possible to see the implementation in the next listings, 32, 33 and 34. Will be used to control the interrupt handling in an 8051-based microcontroller.

The *int_en* its the a signal that enables or disables the processing of interrupts, *int_tf0* the signal that indicates a timer interrupt request, *int_ext0* the signal that indicates an external interrupt request, and *int_ack* the signal that acknowledges the interrupt request. The two outputs *int* and *int_vec* are an interrupt signal that requests an interrupt from the processor and an 8-bit interrupt vector that specifies the type of interrupt, respectively.

```
module interrupt_control(
input clock,
input reset,
input int_en,
input int_tf0,
input int_ext0,
input int_ack,
output int,
output [7:0] int_vec
);
```

Listing 32: Interrupt Controler I/Os.

Passing to the code block, Listing 33 the signal *done_int_ext0* is used to keep track of the previous state of the external interrupt request. It is initially set to 0 and updated in the always block triggered by the posedge of the clock signal. The *done_int_ext0* will be low whenever *int_ext0* is low, and high whenever if *int_ext0* is high, *int_ack* is high and *int_tf0* is low.

```
reg done_int_ext0;

initial begin
done_int_ext0 <= 1'b0;
end

always @(posedge clock) begin
    if(reset == 1'b1)
        done_int_ext0 <= 1'b0;
    else if(int_ext0 == 1'b1 && int_ack == 1'b1 && int_tf0
    == 1'b0)
        done_int_ext0 <= 1'b1;
    else if(int_ext0 == 1'b0)
        done_int_ext0 <= 1'b0;
end
```

Listing 33: Interrupt Controler data and code block.

Regarding the assign statements of the output's of the interrupt controller module, *int_vec* is based on the value of *int_tf0* and *int_ext0*. If *int_tf0* is high, *int_vec* is set to **8'h0B**, which means timer interrupt has higher priority. If *int_ext0* is high, *int_vec* is set to **8'h03**, which means external interrupt has higher priority. If *int* is high indicates to the processor a request for interruption. This only happens if *int_en* is high, either *int_tf0* is high or (*int_ext0* is high and *done_int_ext0* is 0) and *int_ack* is low, int is set to 1. Otherwise, int is set to 0.

```
assign int_vec = (int_tf0  == 1'b1) ? 8'h0B :
                 (int_ext0 == 1'b1) ? 8'h03 : 8'h00;

assign int = (int_en == 1'b1 && (int_tf0  == 1'b1 || (
    int_ext0 == 1'b1 && done_int_ext0 == 1'b0)) && int_ack ==
    1'b0) ? 1'b1 : 1'b0;
```

Listing 34: Interrupt Controler assign statements.

5.8 I/O Layout

Finishing the implementation, in the Listing below 35, you will find all the necessary constraints written. The code sets up the I/O standard and physical pin location for the clock signal and several other digital signals (P0[0:7], P1[0:7], and reset).

The clock signal is specified with a frequency of 32.000 MHz and a waveform with a low time of 0.000ns and a high time of 16.000ns. The I/O standard for all the signals is set to LVCMOS33 since its low power consumption and and has ability to operate at low voltage levels, making it a good choice for applications that require low power and high speed operation. By using LVCMOS33 as the I/O standard for the inputs and outputs, is ensured compatibility and reliability in the design.

Because our development board is the zybo 7000, the clock pin is L16. The reset pin chosen is **Y16**, for the external interrupt **V16** (P0[0]) and for the LED's **M14**(P1[0]), **M15**(P1[1]), **G14**(P1[2]), and **D18**(P1[3]). All the other pins are were chosen as Pmod ports allowing to interface with a variety of sensors, actuators, and other peripheral devices to expand its capabilities and perform a wider range of tasks if necessary. The Pmods chosen were the Pmod JB and the Pmod JC since they are high speed.

The rest of the constraints code was used to configure the debugging system of the integrated circuit design in the context of hardware design and verification. The main focus was on monitoring various signals within the FPGA design, such as the clock, state signals, and internal signals, for debugging and analysis purposes.

It creates an instance of the Integrated Logic Analyzer (ILA) core and sets its properties such as data depth, input pipeline stages, and trigger options. Then, the code connects various signals of the design to the probes of the ILA core to enable monitoring and analysis of these signals. The code also sets the frequency of the clock input to the debugging system, enables or disables clock dividers, and sets the user scan chain. These settings allow the ILA core to capture and display the signals of interest in the context of the overall system operation. Basically this portion of the code becomes useful for the chip scope on analyzing and debugging the FPGA design.

```
#Clock signal
set_property -dict {PACKAGE_PIN L16 IOSTANDARD LVCMOS33} [
    get_ports clock]
create_clock -period 32.000 -name sys_clk_pin -waveform
    {0.000 16.000} -add [get_ports clock]

set_property IOSTANDARD LVCMOS33 [get_ports {P0[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {P0[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {P0[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {P0[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {P0[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {P0[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {P0[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {P0[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {P1[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {P1[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {P1[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {P1[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {P1[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {P1[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {P1[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {P1[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports reset]

#Reset
set_property PACKAGE_PIN Y16 [get_ports reset]

#Leds
set_property PACKAGE_PIN M14 [get_ports {P1[0]}]
set_property PACKAGE_PIN M15 [get_ports {P1[1]}]
set_property PACKAGE_PIN G14 [get_ports {P1[2]}]
set_property PACKAGE_PIN D18 [get_ports {P1[3]}]

#Pmod JC (high-speed)
set_property PACKAGE_PIN V15 [get_ports {P1[4]}]
set_property PACKAGE_PIN W15 [get_ports {P1[5]}]
set_property PACKAGE_PIN T11 [get_ports {P1[6]}]
set_property PACKAGE_PIN T10 [get_ports {P1[7]}]

#external int
set_property PACKAGE_PIN V16 [get_ports {P0[0]}]

#Pmod JB (high-speed)
set_property PACKAGE_PIN T20 [get_ports {P0[1]}]
set_property PACKAGE_PIN U20 [get_ports {P0[2]}]
set_property PACKAGE_PIN V20 [get_ports {P0[3]}]
set_property PACKAGE_PIN W20 [get_ports {P0[4]}]
set_property PACKAGE_PIN Y18 [get_ports {P0[5]}]
set_property PACKAGE_PIN Y19 [get_ports {P0[6]}]
set_property PACKAGE_PIN W18 [get_ports {P0[7]}]

connect_debug_port u_ila_0/probe3 [get_nets [list {state[0]}
   {state[1]}]]
connect_debug_port u_ila_0/probe4 [get_nets [list {ctrl_unit/
   state[2]} {ctrl_unit/state[3]} {ctrl_unit/state[4]}]]
connect_debug_port u_ila_0/probe7 [get_nets [list int_ext0]]


create_debug_core u_ila_0 ila
set_property ALL_PROBE_SAME_MU true [get_debug_cores u_ila_0]
set_property ALL_PROBE_SAME_MU_CNT 4 [get_debug_cores u_ila_0
   ]
set_property C_ADV_TRIGGER true [get_debug_cores u_ila_0]
set_property C_DATA_DEPTH 4096 [get_debug_cores u_ila_0]
set_property C_EN_STRG_QUAL true [get_debug_cores u_ila_0]
set_property C_INPUT_PIPE_STAGES 1 [get_debug_cores u_ila_0]
set_property C_TRIGIN_EN false [get_debug_cores u_ila_0]
set_property C_TRIGOUT_EN false [get_debug_cores u_ila_0]
set_property port_width 1 [get_debug_ports u_ila_0/clk]
connect_debug_port u_ila_0/clk [get_nets [list
   clock_IBUF_BUFG]]
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports
   u_ila_0/probe0]
set_property port_width 1 [get_debug_ports u_ila_0/probe0]
connect_debug_port u_ila_0/probe0 [get_nets [list int_ack]]
create_debug_port u_ila_0 probe
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports
   u_ila_0/probe1]
set_property port_width 1 [get_debug_ports u_ila_0/probe1]
connect_debug_port u_ila_0/probe1 [get_nets [list tf0_flag]]
create_debug_port u_ila_0 probe
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports
   u_ila_0/probe2]
set_property port_width 1 [get_debug_ports u_ila_0/probe2]
connect_debug_port u_ila_0/probe2 [get_nets [list {ctrl_unit/
   state[0]_i_1_n_0}]]
create_debug_port u_ila_0 probe
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports
   u_ila_0/probe3]
set_property port_width 1 [get_debug_ports u_ila_0/probe3]
connect_debug_port u_ila_0/probe3 [get_nets [list {ctrl_unit/
   state[1]_i_1_n_0}]]
create_debug_port u_ila_0 probe
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports
```

```
    u_ila_0/probe4]
set_property port_width 1 [get_debug_ports u_ila_0/probe4]
connect_debug_port u_ila_0/probe4 [get_nets [list {ctrl_unit/
    state[2]_i_1_n_0}]]
create_debug_port u_ila_0 probe
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports
    u_ila_0/probe5]
set_property port_width 1 [get_debug_ports u_ila_0/probe5]
connect_debug_port u_ila_0/probe5 [get_nets [list {ctrl_unit/
    state[3]_i_1_n_0}]]
create_debug_port u_ila_0 probe
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports
    u_ila_0/probe6]
set_property port_width 1 [get_debug_ports u_ila_0/probe6]
connect_debug_port u_ila_0/probe6 [get_nets [list {ctrl_unit/
    state[4]_i_2_n_0}]]
set_property C_CLK_INPUT_FREQ_HZ 300000000 [get_debug_cores
    dbg_hub]
set_property C_ENABLE_CLK_DIVIDER false [get_debug_cores
    dbg_hub]
set_property C_USER_SCAN_CHAIN 1 [get_debug_cores dbg_hub]
connect_debug_port dbg_hub/clk [get_nets clock_IBUF_BUFG]
```

Listing 35: Constraints.

6 Verification

6.1 Tests

In order to avoid future difficulties and the appearance of unexpected errors when grouping the modules, it was decided to carry out tests on each of the modules. This tests are designated by **Testbench** and are used for simulation purpose only.

6.1.1 8-bit Register

Next is represented the testbench code used to perform the simulation for the 8-bit register module and the respective output waveform in Figure 15.

To test this module an input value is assigned, regIn of **8'hff**. The *register_8bit* module was instantiated and after several clock cycles it is intended to check if when the set input becomes **1'b1**, the output value, regOut, becomes the input value regIn.

```
`timescale 1ns / 1ps

module register_8bit_tb();

reg reset = 1'b1;
reg set = 1'b0;
reg[7:0] regIn = 8'hff;
wire[7:0] regOut;

reg clock = 1'b0;
always #50 clock = !clock; // 100 time ticks per cycle = 10
    MHz clock

register_8bit register(clock,reset,set,regIn,regOut);

initial begin

@(negedge clock) begin
reset = 1'b1;
end

@(negedge clock) begin
reset = 1'b0;
end

@(negedge clock) begin
set = 1'b1;
end

```

```
@(negedge clock) begin
set = 1'b0;
end

@(negedge clock) begin
end

$finish;
end
endmodule
```

Listing 36: 8-bit register testbench.

As it can be observed by the waveform of Figure 15 everything went according to what was intended.

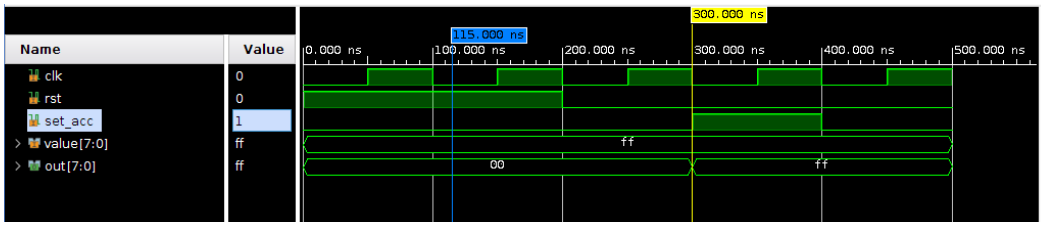


Figure 15: Output waveform 8-bit register testbench

6.1.2 RAM

Next is represented the testbench code used to perform the simulation for the RAM module and the respective output waveforms. The purpose of this module is to verify that the values are being correctly stored both in the internal RAM memory and in the register bank.

```
`timescale 1ns / 1ps

module ram_tb();

reg reset = 1'b0;
reg ram_rd_en_reg= 1'b0;
reg ram_wr_en_reg= 1'b0;
reg [2:0] ram_reg_in_sel= 3'b0;
reg [2:0] ram_reg_out_sel=3'b0;
reg ram_rd_en_data= 1'b0;
reg ram_rd_en_sfr= 1'b0;
reg ram_wr_en_data= 1'b0;
reg [7:0]ram_rd_addr= 8'b0;
reg [7:0]ram_wr_addr= 8'b0;
reg [7:0]ram_wr_byte= 8'b0;
```

```verilog
wire [7:0]ram_rd_byte;

//stack
reg [7:0]stack_in= 8'b0;
reg push_stack= 1'b0;
reg pop_1_stack= 1'b0;
reg pop_2_stack= 1'b0;

wire [7:0]stack_out;

reg clock = 1'b0;
always #50 clock = !clock;

ram ram(clock,reset,ram_rd_en_reg,ram_wr_en_reg,
    ram_reg_in_sel,ram_reg_out_sel,ram_rd_en_data,
    ram_rd_en_sfr,ram_wr_en_data,ram_rd_addr,ram_wr_addr,
    ram_wr_byte,ram_rd_byte,enable_stack,
         stack_in,push_1_stack,push_2_stack,pop_1_stack,
    pop_2_stack,stack_out,int_pend,int_req);

initial begin
@(negedge clock) begin
    reset = 1'b1;
end

@(negedge clock) begin
    reset = 1'b0;
end

@(negedge clock) begin
    ram_wr_byte = 8'h03;
    ram_wr_addr = 8'h74;
    ram_wr_en_data = 8'h01;
end

@(negedge clock) begin
    ram_wr_en_data = 8'h00;
    ram_rd_addr = 8'h74;
    ram_rd_en_data = 8'h01;
end

@(negedge clock) begin
    ram_rd_en_data = 8'h00;
    ram_wr_byte = 8'h04;
    ram_reg_in_sel = 3'b010;
    ram_wr_en_reg = 8'h01;
end

@(negedge clock) begin
```

```
61      ram_wr_en_reg = 8'h00;
62      ram_reg_out_sel = 3'b010;
63      ram_rd_en_reg = 8'h01;
64  end
65
66  @(negedge clock) begin
67      ram_rd_en_reg = 8'h00;
68  end
69
70  @(negedge clock) begin
71  end
72
73  $finish;
74  end
75  endmodule
```

Listing 37: RAM testbench.

First, a random value was loaded into a selected position, in this case position **0x74**, as shown in line 44, and the control signal *ram_wr_en_data* was enabled for writing. In the following clock cycle the same position was read, and the result was as expected **0x03**, as shown in Figure 16.

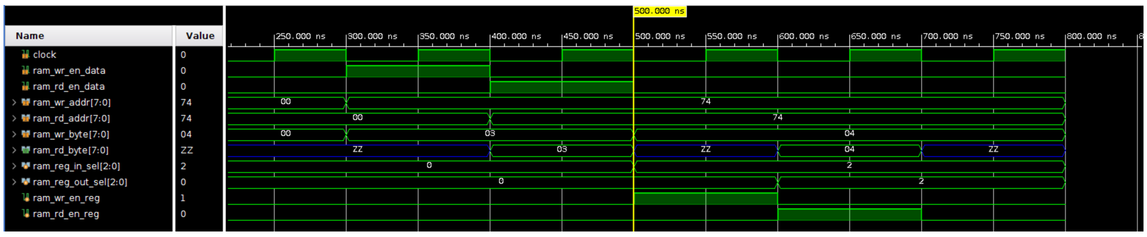


Figure 16: Output waveform RAM testbench.

Finally, to test the loading of values into registers, a similar process was used. The value **0x04** was assigned to register 2 and in Figure 16 it was verified that the value returned by the RAM is as expected.

6.1.3 ROM

Next is represented the testbench code used to perform the simulation for the ROM module and the respective output waveform in Figure 17.

To test this module different input values were assigned for the rom_addr. The module *rom* was instantiated and after several clock cycles, as soon as rom_en becomes **1'b1** the rom should be able to read value from position given by the values of the rom_addr. In the rom module, it has been previously defined, in **position 0x003B** the value is **8'h74** and in in **position 0x003C** the value is **8'h03**.

```verilog
`timescale 1ns / 1ps

module rom_tb();

reg rom_en = 1'b0;
reg [15:0] rom_addr = 15'h0000;
wire [7:0] rom_byte;


reg clock = 1'b0;
always #50 clock = !clock; // 100 time ticks per cycle = 10
    MHz clock

rom romTB(rom_en,rom_addr,rom_byte);

initial begin

@(negedge clock) begin
rom_addr = 15'h003B;

rom_en = 1'b1;
end

@(negedge clock) begin
rom_addr = 15'h003C;
end

@(negedge clock) begin
//extra cycle
end

$finish;
end
endmodule
```

Listing 38: ROM testbench.

As can be seen from the waveform in Figure 17, everything went according to what was intended.

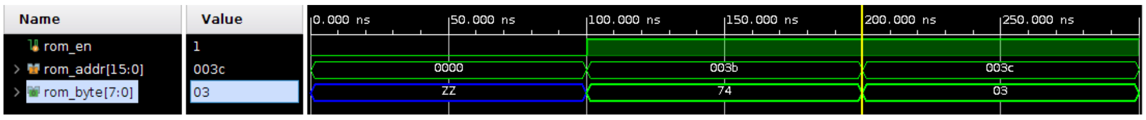


Figure 17: Output waveform ROM testbench.

6.1.4 ALU

Next is represented the testbench code used to perform the simulation for the ALU module and the respective outputs waveforms in Figures 18, 19, 20, 21 and 22.

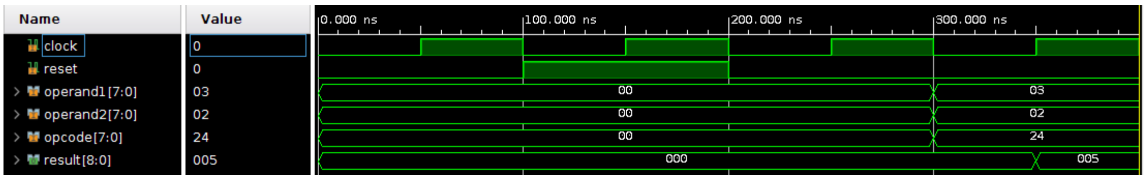
To test this module different input values were assigned for both operand 1 and operand2. The module *arithmetic_logic_unit* was instantiated and after several clock cycles, depending on the value assigned to the opcode, it is intended to verify that the operations, add, subb, anl, orl and xrl return the correct output value, aluResult.

```verilog
`timescale 1ns / 1ps

module alu_tb();

reg reset = 1'b0;

reg [7:0]operand1 = 1'b0;
reg [7:0]operand2 = 1'b0;
reg [7:0]opcode = 1'b0;
wire [8:0]ALU_result;

reg clock = 1'b0;
always #50 clock = !clock; // 100 time ticks per cycle = 10
    MHz clock

arithmetic_logic_unit alu(clock, reset, operand1, operand2
    [7:0], opcode[7:0], ALU_result[8:0]);

initial begin

@(negedge clock) begin
reset = 1'b1;
end

@(negedge clock) begin
reset = 1'b0;
end

@(negedge clock) begin
opcode = 8'b00100100;
operand1 = 8'b00000011;
operand2 = 8'b00000010;
end

@(negedge clock) begin
end

```

```
$finish;
end
endmodule
```

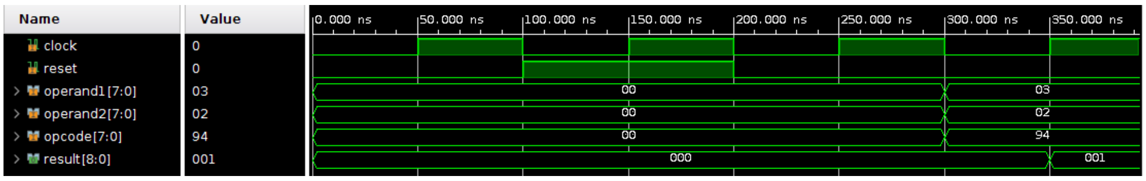
Listing 39: ALU testbench.

As can be seen from the waveforms in Figures 18, 19, 20, 21 and 22, everything went according to what was intended.



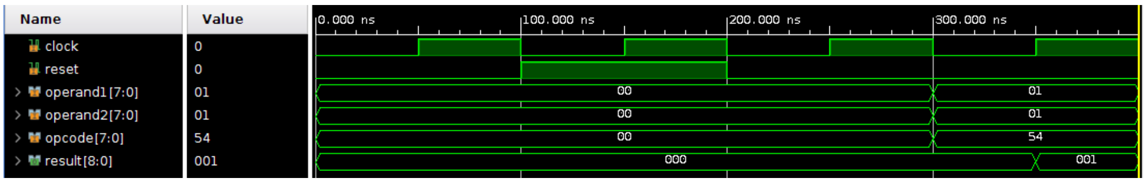
Name	Value
clock	0
reset	0
operand1[7:0]	03
operand2[7:0]	02
opcode[7:0]	24
result[8:0]	005

Figure 18: Output waveform ALU testbench (ADD acc, Immediate).



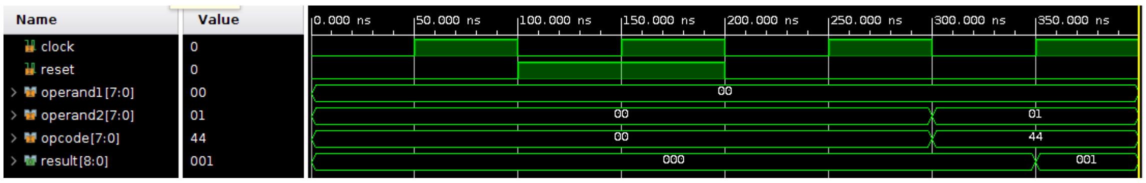
Name	Value
clock	0
reset	0
operand1[7:0]	03
operand2[7:0]	02
opcode[7:0]	94
result[8:0]	001

Figure 19: Output waveform ALU testbench (SUBB acc, Immediate).



Name	Value
clock	0
reset	0
operand1[7:0]	01
operand2[7:0]	01
opcode[7:0]	54
result[8:0]	001

Figure 20: Output waveform ALU testbench (ANL acc, Immediate).



Name	Value
clock	0
reset	0
operand1[7:0]	00
operand2[7:0]	01
opcode[7:0]	44
result[8:0]	001

Figure 21: Output waveform ALU testbench (ORL acc, Immediate).

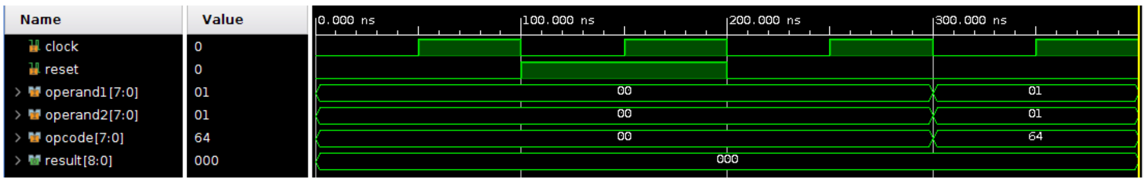


Figure 22: Output waveform ALU testbench (XRL acc, Immediate).

6.1.5 Program Counter

The next module to be tested is the Program Counter module. One of the testbench used is represented in the listing 40. In this testbench it is tested how the program counter reacts when it is notified that an interrupt has occurred, i.e. the control signal int is positive.

```
`timescale 1ns / 1ps

module tb_timer();

reg clock    = 1'b0;
reg reset    = 1'b0;
reg [7:0] value_8bit = 8'h00;
reg [7:0] int_vec = 8'h00;
reg [7:0] acc   = 8'h00;
reg [7:0] stack_out   = 8'h00;


reg pc_inc     = 1'b0;
reg pc_inc_offset = 1'b0;
reg pc_jmp_z     = 1'b0;
reg pc_jmp_nz = 1'b0;
reg int     = 1'b0;
reg pop_1_stack = 1'b0;
reg pop_2_stack = 1'b0;


wire int_ack;
wire push_stack;
wire [7:0] stack_in;
wire [15:0] count;

program_counter pc_dp(clock, reset, pc_inc, pc_inc_offset,
    pc_jmp_z,pc_jmp_nz, value_8bit, int, int_vec,
pop_1_stack, pop_2_stack,acc, stack_out, int_ack, stack_in,
    push_stack, count);

always #50 clock = ~clock;

```

```
initial begin
@(negedge clock) begin
    reset = 1'b1;
end

@(negedge clock) begin
    reset = 1'b0;
    int = 1'b1;
    int_vec = 8'h0b;
end
@(negedge clock) begin
    int = 1'b0;
end

$finish;
end
endmodule
```

Listing 40: Program Counter testbench

For testing purposes it was considered that the interrupt that occurred was the timer 0 overflow, so the *int_vec* register was assigned a value of **0x0b**. As can be seen in figure 23 the value of the program counter is updated to the value associated with the interrupt address at instant , right after the stack is pushed.

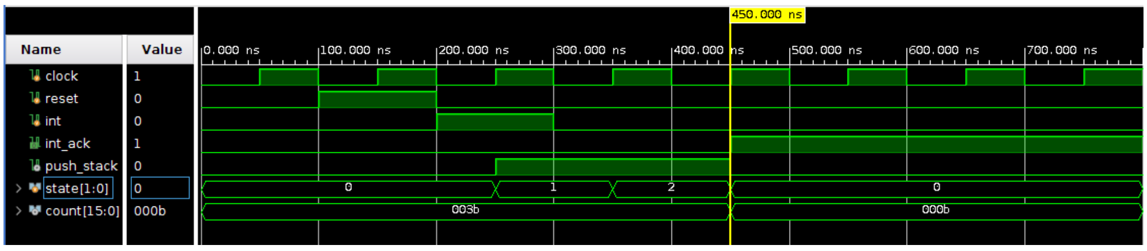


Figure 23: Output waveform Program Counter testbench.

Other tests were also performed on the program counter, the testbench implemented always follows the same logic as the previous one, so it will not be presented. To check in detail the behavior of the program counter for different jump instructions, the output waves of the remaining tests performed will also be presented.

First, it was checked whether the counting increment was being performed correctly, and Figure 24 checks it. Also in the same simulation at time 350ns, it was verified if the count value is valid when an offset is added, i.e., a short jump instruction occurs. As can be seen, all the values obtained are valid.

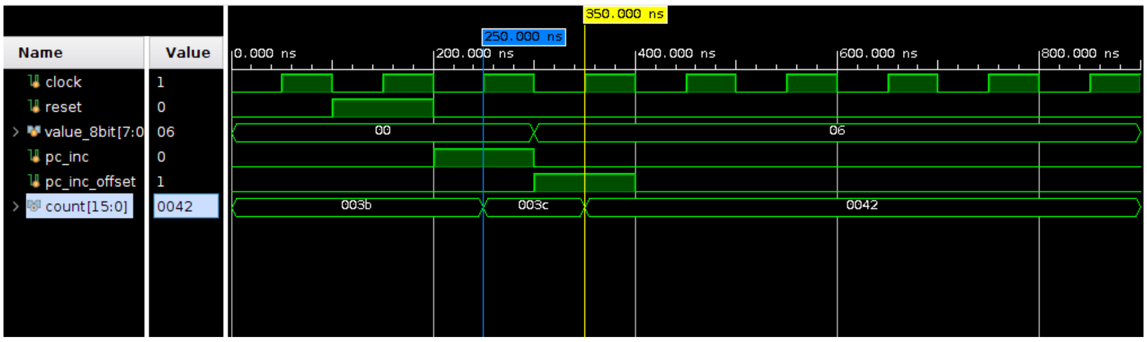


Figure 24: Output waveform Program Counter testbench.

Next the behavior of the program counters was analyzed through simulations performed for the Jump Zero and Jump not Zero instructions. As can be seen in Figure 25, when the control signal *jmp_nz* becomes 1, and the accumulator value is non-zero, there is a jump associated with the offset in the instruction, in this case the offset is represented by the port *newValue_8bit*.

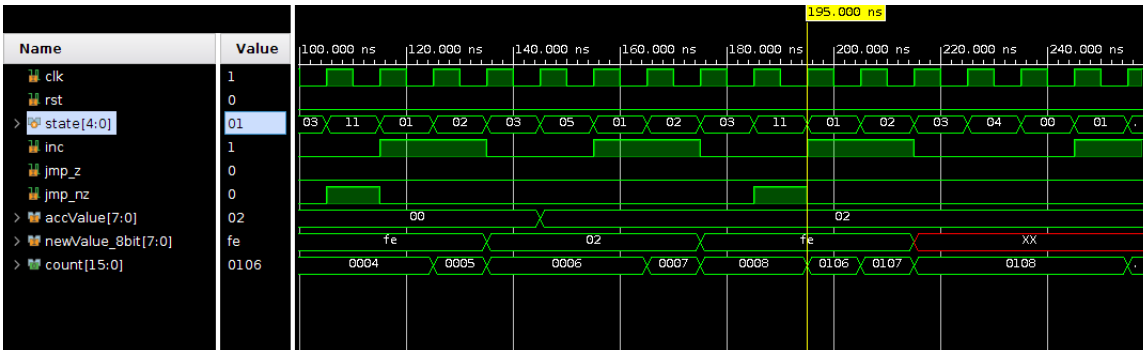


Figure 25: Output waveform Program Counter testbench - Jump Not Zero.

Similarly, regarding the Jump Zero instruction, in Figure 26, it can be seen that when the value of the control signal *jmp_z* is 1, and the accumulator value is zero, a jump to the expected position occurs.

Furthermore, tests were also performed to verify the behavior of the program counter when the Jump If Not Carry instruction occurs. As shown in figure 27, at the occurrence of the instruction, and if the carry has a positive value, the program counter continues its normal execution, that is, no jump occurs.

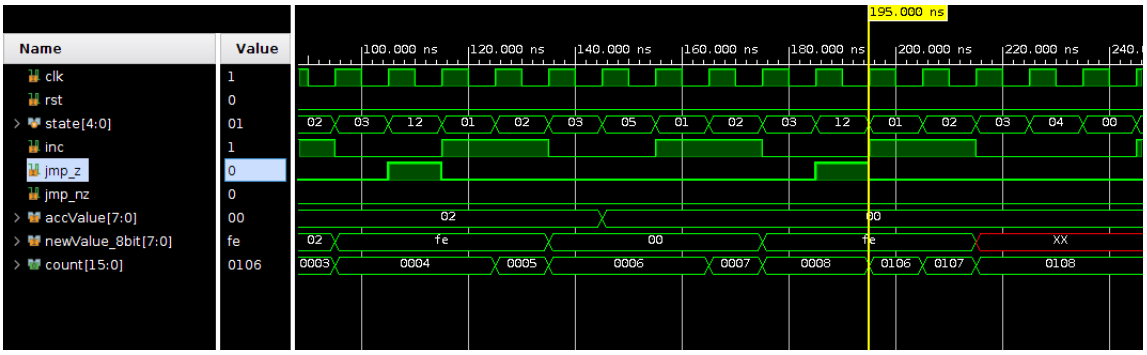


Figure 26: Output waveform Program Counter testbench - Jump Zero.

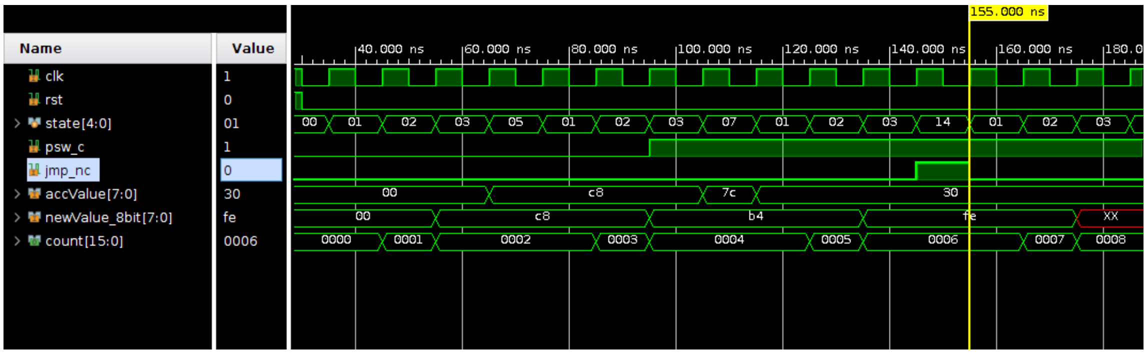


Figure 27: Output waveform Program Counter testbench - Jump If Not Carry.

6.1.6 Instruction Register

Next is represented the testbench code used to perform the simulation for the Instruction Register module and the respective output waveform in Figure 28.

To test this module a input value *rom_byte*, **8'hff** has been assigned to the instruction register. The module *instruction_register* was instantiated and after several clock cycles, depending on the value assigned to the *ir_load_high*, 8 MSB of instruction register, and *ir_load_low*, 8 LSB of instruction register, it is intended to verify that the output, *IR* is correct. When *ir_load_high* is **1'b1** it should load the value from the input *rom_byte* to the 8 MSB of instruction register and when *ir_load_low* is **1'b1** it should load the value from the input *rom_byte* to the 8 LSB of the instruction register.

```
`timescale 1ns / 1ps

module ir_tb();

```

```
reg reset = 1'b1;
reg ir_load_high = 1'b0;
reg ir_load_low = 1'b0;
reg[7:0] rom_byte = 8'hff;
wire[15:0] IR;

reg clock = 1'b0;
always #50 clock = !clock; // 100 time ticks per cycle = 10
    MHz clock

instruction_register ir(clock, reset, ir_load_high,
    ir_load_low, rom_byte, IR);

initial begin

@(negedge clock) begin
reset = 1'b1;
end

@(negedge clock) begin
reset = 1'b0;
end

@(negedge clock) begin
ir_load_high = 1'b1;
end

@(negedge clock) begin
ir_load_high = 1'b0;
end

@(negedge clock) begin
ir_load_low = 1'b1;
end

@(negedge clock) begin
ir_load_low = 1'b0;
end

@(negedge clock) begin
end

$finish;
end
endmodule
```

Listing 41: Instruction Register testbench.

As can be seen from the waveform in Figure 28 everything went according to what was intended.

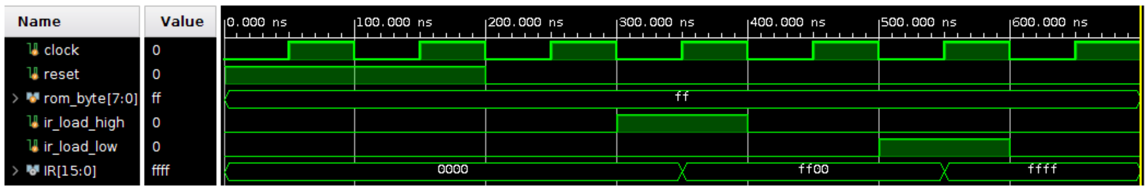


Figure 28: Output waveform Instruction Register testbench.

6.1.7 Timer

It was decided to implement a testbench that allows one to verify the correct operation of the timer created, as shown in listing 42. Two tests have been created, however only one testbench will be presented, as they are very similar. The idea of this testbench, is just to check that by assigning values to the respective SFR of the timer and starting the count, the overflow flag shows a positive value after some time. Therefore, the values shown in line 28 to 30 are assigned, so that the count won't be long, and the selected mode is timer mode 1. The count is started on line 34.

```
1  `timescale 1ns / 1ps
2
3  module tb_timer();
4
5  reg clock    = 1'b0;
6  reg reset    = 1'b0;
7  reg [7:0]sfr_tmod = 8'h00;
8  reg [7:0]sfr_tcon = 8'h00;
9  reg [7:0]sfr_th0  = 8'h00;
10 reg [7:0]sfr_tl0  = 8'h00;
11 reg P0_1     = 1'b0;
12 reg int_ack  = 1'b0;
13
14 wire tf0_flag;
15 wire tr0_flag;
16
17 timer_8051 timer(clock,reset,sfr_tmod,sfr_tcon,sfr_tl0,
       sfr_th0,P0_1,int_ack,tf0_flag,tr0_flag);
18
19 always #50 clock = ~clock;
20
21 initial begin
22 @(negedge clock) begin
23 reset = 1'b1;
24 end
25
26 @(negedge clock) begin
27     reset = 1'b0;
```

```verilog
    sfr_tmod = 8'h01;
    sfr_tl0 =  8'hfa;
    sfr_th0 =  8'hff;
end

@(negedge clock) begin
    sfr_tcon = 8'h10;
end

end
endmodule
```

Listing 42: Timer testbench.

The output waveform is represented in Figure 29. As you can see, after the start of counting, and following a few clock cycles, the tf0 flag, identified in the simulation as *tf0_flag*, has a positive value.

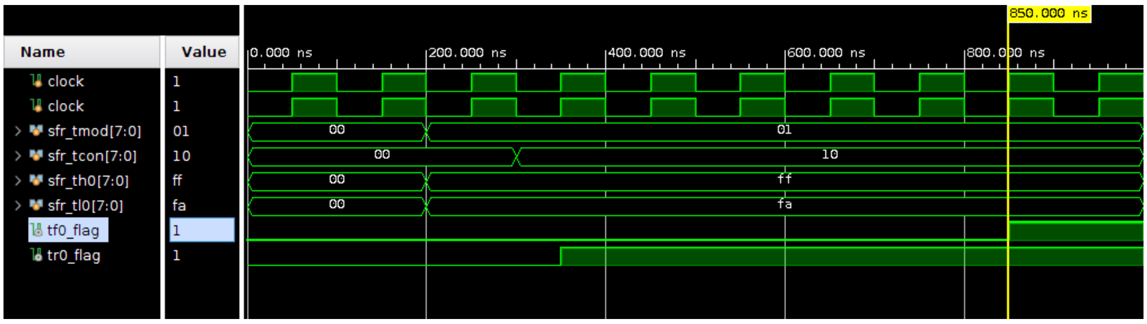


Figure 29: Output waveform Timer mode 1 testbench.

As mentioned above, another test was performed, this time in counter mode. For this the *P0_1* pin serves as a clock source to the counter, and at each positive transition the timer registers are incremented. As demonstrated in Figure 30, at time 1050ns, the upward transition of the timer flag occurs, thus verifying the correct operation of the Timer module.

6.1.8 Interrupt Control

Next is represented the testbench code used to perform the simulation for the Instruction Control module and the respective output waveform in Figure 33.

To test this module different input values were assigned for *int_en*, *int_tf0*, *int_ext0* and *int_ack*. The module *interrupt_control* was instantiated. After several clock cycles it is intended to verify that the outputs, *int* and *int_vec* are correct. When *int_tf0* is **1'b1** and *int_ext0* is **1'b0**, timer has higher priority and *int_vec* should remain with the value **8'h0B**, otherwise when

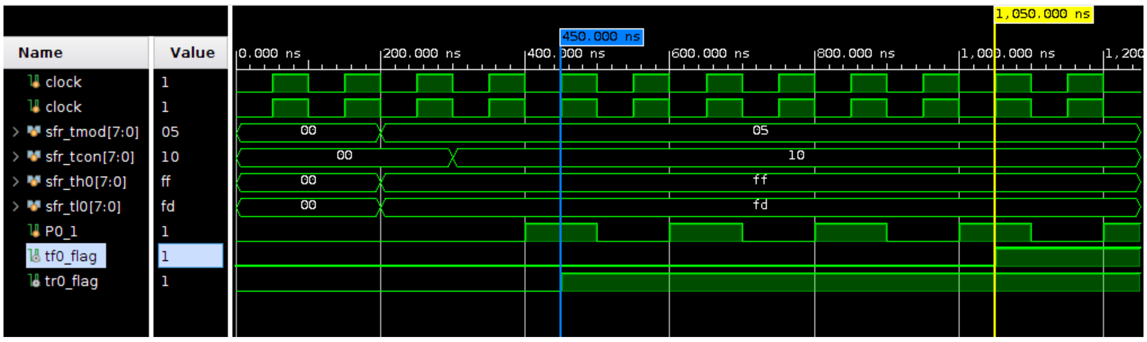


Figure 30: Output waveform Counter mode 1 testbench.

int_tf0 is **1'b0** and int_ext0 is **1'b1**, the interrupt is external and *int_vec* should remain with the value **8'h03**. Whenever there is an *int_tf0* at **1'b1** or *int_ext0* at **1'b1**, the interrupt enable, *int_en* at **1'b1** and *int_ack* at **1'b0**, *int* should be active at **1'b1**.

This way the interrupt only occurs when *int_ack* is at **1'b0**.

```
`timescale 1ns / 1ps

module tb_intCtrl();

reg clock    = 1'b0;
reg reset    = 1'b0;
reg int_en   = 1'b0;
reg int_tf0  = 1'b0;
reg int_ext0= 1'b0;
reg int_ack  = 1'b0;

wire  int;
wire  [7:0] int_vec;

interrupt_control  IC(clock,reset,int_en,int_tf0,int_ext0,int,
    int_vec,int_ack);

always #50 clock = ~clock;

initial begin

@(negedge clock) begin
reset = 1'b1;
end

@(negedge clock) begin
    reset = 1'b0;
    int_en  = 1'b1;
    int_tf0 = 1'b1;
end

@(negedge clock) begin
    int_ack = 1'b1;
end

@(negedge clock) begin
    int_ack = 1'b0;
    int_tf0 = 1'b0;
end

@(negedge clock) begin
end

@(negedge clock) begin
    int_en   = 1'b1;
    int_ext0 = 1'b1;
end

@(negedge clock) begin
    int_ack = 1'b1;
end

@(negedge clock) begin
end

$finish;
end

endmodule
```

Listing 43: Interrupt Control testbench.

As can be seen from the waveform in Figure 33 everything went according to what was intended.

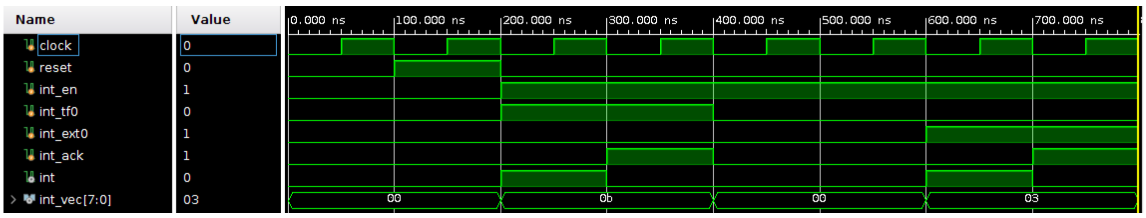


Figure 31: Output waveform Interrupt Control testbench.

6.1.9 Debounce

Next is represented the testbench code used to perform the simulation for the Debounce module and the respective output waveform in Figure 28.

To test this module an input value is assigned, **button**. The module *debounce* was instantiated and after several clock cycles, the value of button becomes **1'b1**.

```
`timescale 1ns / 1ps

module tb_debounce();

reg clock = 1'b0;
reg reset = 1'b0;
reg button = 1'b0;
wire outDeb;

debounce deb(clock,button,outDeb);

always #50 clock = ~clock;

initial begin
@(negedge clock) begin
    button = 1'b0;
end

@(negedge clock) begin
    button = 1'b1;
end

end
endmodule
```

Listing 44: Debounce testbench.

As can be seen from the waveform in Figure 33 everything went according to what was intended. That is, the button was pressed at time instant, 300.0ns and was set to 1, but the output, outDeb was only set to 1 at time instant 102,750.0ns, thus preventing the rapid triggering of an action from a single event. It ensures that the digital circuit only reacts to a single instance of a rapidly changing input signal and eliminates any unwanted triggering that may occur due to electrical noise or switch bounce.

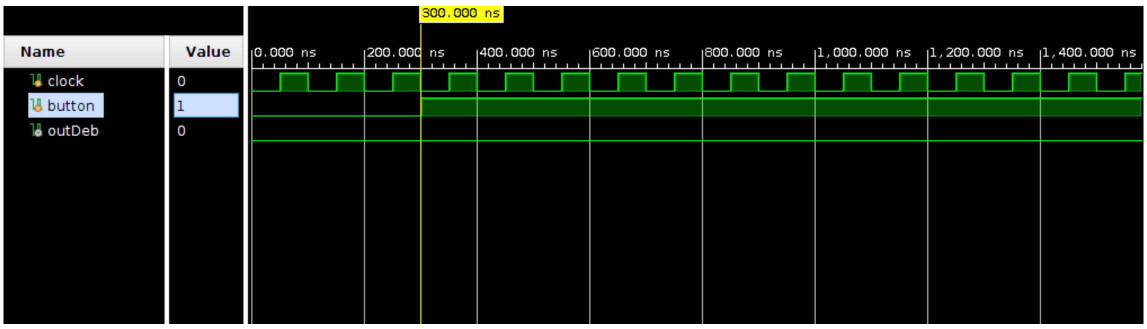


Figure 32: Output waveform Debounce testbench.

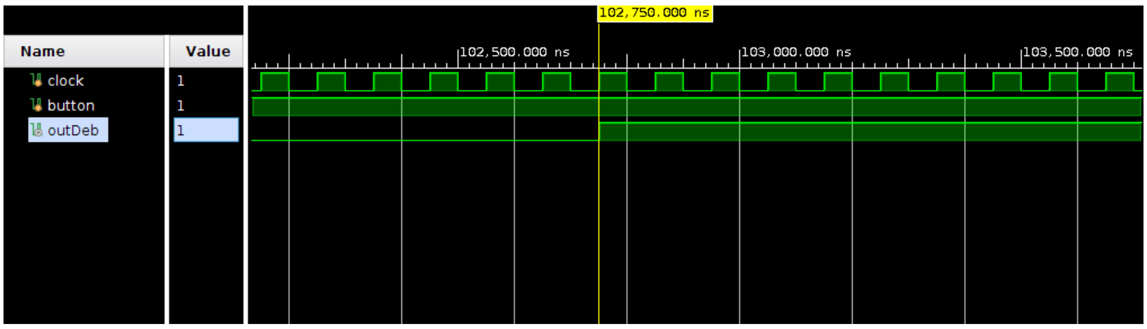


Figure 33: Output waveform Debounce testbench.

7 Final Test Program

The code on the following Listing 45 is based on the instructions that are stored in the ROM. The final test code is below and it's in the assembly language for a better interpretation. Also to add here is a link for demonstration, (`https://youtu.be/p-Q8BfgXRZk`) and to the source code (`https://github.com/andrebarbosaa/8051.git`).

```
ORG 0x0003
EXT0_ISR:
    ; put all leds to on
     MOV A,#0x0f
     MOV P1,A
RETI

ORG 0x003B
TIMER0_ISR:
    ;polling of R1 until it's zero
    MOV A,R1
    SUBB A,#0x01
    MOV R1,A

    JNZ NOT_INC_P1
    ;increment 1 to the value of P1
    MOV A,P1
    MOV A,#0x01
    MOV P1,A

    ;update the value of R1 again
    MOV R1,#0xFF

    NOT_INC_P1: ;if R1 not zero
RETI


ORG 0x003B
;first instruction to be executed
MOV A,#0x01 ;send value 1 to P1
MOV P1,A  ;MOV 0x90,A

MOV A,#0x01 ;Timer mode 1
MOV TMOD,A  ;MOV 0x88,A

MOV A,#0x00
MOV TL0,A ;MOV 0x88,A

MOV A,#0x00
MOV TH0,A ;MOV 0x88,A

MOV A,#0x83 ;enable EA
MOV IE,A  ;MOV 0xA8,A

MOV R1,#0xFF  ;update the value of R1 for polling

MOV A,#0x10 ;enable TR0
MOV TCON,A  ;MOV 0x89,A

JUMP, $
```
Listing 45: Final assembly code.

8 Gantt Diagram

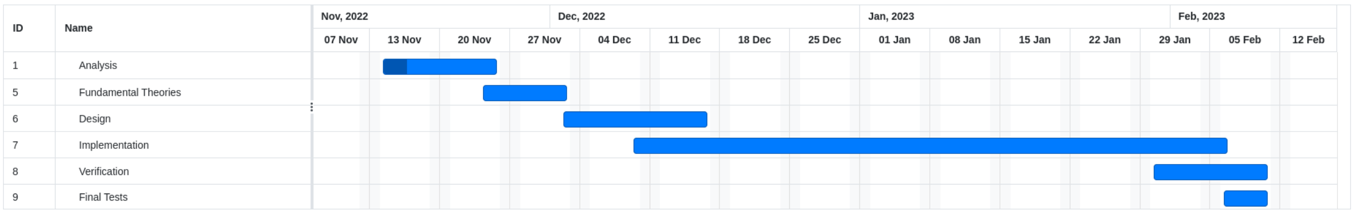


Figure 34: Task Division: Gantt Diagram.

9 References

[1] TAVARES Adriano, LIMA Carlos, CARDOSO Paulo, MENDES José, CABRAL Jorge, "PROGRAMAÇÃO DE MICROCONTROLADORES", 2012.

[2] (`https://www.silabs.com/documents/public/data-sheets/C8051F388-B.pdf`).