# LAB8

**Student ID:** 39214
**Email:** Rismaili@al.uloyola.es
**Hand-in date:** 18/12/2022
**Comments:**

- This is the final LAB. in this LAB I have created the control logic of the processor and finalized some last remaining things.
- Since this is the last LAB I decided to make a more detailed report of the LAB.
- The board I used to implement the CPU is not the MiniZed board because it would not work on my laptop.
- I have removed and modified a LOT of the previous LABs so that this CPU can run properly (well, disregarding the time synchronization problems).

# INTRODUCTION

This is the first time I have finished a project in VHDL using VIVADO. I have had no prior experience with hardware description languages, but thanks to this project I now better understand the logic, syntax and flow of VHDL. Additionally I now have deeper and a more detailed understanding of how CPUs (simple ARM CPUs) function.

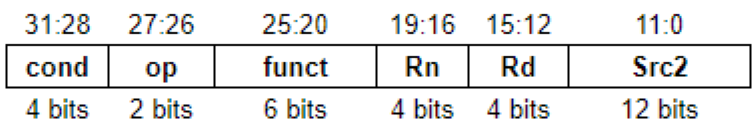
**Let’s get started.**

Since I modified (simplified) the processor (following professor James’ instructions) it has a different list of possible instructions it can run.

The following is the list of the 2 possible CPU instruction types:

- **Data-processing** instructions (DP instr)
- **Memory** (addressing) instructions (MEM instr)

Every instruction has 32 bits and the following structure:



| 31:28 | 27:26 | 25:20 | 19:16 | 15:12 | 11:0 |
|---|---|---|---|---|---|
| cond | op | funct | Rn | Rd | Src2 |
| 4 bits | 2 bits | 6 bits | 4 bits | 4 bits | 12 bits |

This is what each of the fields represents:

- **cond -** Serves no purpose in my processor (originally used for checking for certain conditions, i.e: check if two operands are equal, etc.)
- **op** - Defines the instruction type
- **funct** - Defines the control logic
- **Rn** - Defines the first operand
- **Rd** - Defines the destination register
- **Src2** - Defines the second operand

The main differences between the 2 types of instructions are the following fields:

- **op**
- **funct**
- **Src2**

The rest of the fields follow the same logic, regardless of instruction type.

Bits 31:20 (31 down to 20) are used for the **control logic** whereas bits 19:0 (19 down to 0) are used for the **addressing / destinations** of the operands and results.

## Data-processing instructions:

For data-processing instructions we have **2 main types**. Instructions which use immediate *(constant)* values for the second operand and instructions which use non-shifted *(more details down below)* registers for their second operand.

- **op** - Is always set to “00”. This sets the type of the instruction to data-processing.
- **funct** - is made up of 3 different sub-fields: **I**, **cmd** & **S**.
  - **I** - determines if we use an immediate value or not.
  - **cmd** - determines the type of data processing operation. *(ADD/SUB/AND/ORR)*
  - **S** - determines if we want to update the ALU flags. *(For our purposes it is redundant)*
- **Src2** - has 2 different variations depending on the **I** sub-field. *(More information in the pictures below)*

Data-processing instruction using an **immediate value** for the second operand:

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:8 | 7:0 |
|---|---|---|---|---|---|---|---|---|
| cond | op 00 | I 1 | cmd | S 0 | Rn | Rd | rot 0000 | imm8 |
| 4 bits | 2 bits | funct | | | 4 bits | 4 bits | Src2 | |
| | | 6 bits | | | | | 12 bits | |

For this type of data-processing instruction we set **I** to 1. **S** will always be set to 0 because we are not working with **ALU flags**. **Src2** has two sub-fields, **rot** and **imm8**. **Rot** is used for the rotation of the **imm8** value - effectively giving us more possible immediate values but for this CPU I haven’t implemented the necessary hardware. **Imm8** is an 8 bit immediate value which will be used as our second operand for operations.

Data-processing instruction using an **non-shifted register** for the second operand:

| 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:7 | 6:5 | 4 | 3:0 |
|---|---|---|---|---|---|---|---|---|---|---|
| cond | op 00 | I 0 | cmd | S 0 | Rn | Rd | shamt5 00000 | sh 00 | bit 4 0 | Rm |
| 4 bits | 2 bits | funct | | | 4 bits | 4 bits | Src2 | | | |
| | | 6 bits | | | | | 12 bits | | | |

For this type of data-processing instruction we set **I** to 0 and **S** will be set to 0 for the same reasons as stated above. Since we are using a non-shifted register for our second operand, we will set **shamt5**, **sh** and **bit 4** all to 0. This is because when I say non-shifted register I mean that the register found in **Rm** is not shifted by an amount that would be given in shamt5. *(Just to reiterate, this is the simplified version of the CPU, we have been instructed to not implement shifting operations)*

*As we can see in the above pictures, for both types of instructions there are orange and red colors. The red color means that this sub-field is redundant, whereas the orange color means that this is a fixed bit, which never changes - irrespective of what the instruction is.*

## Memory instructions:

*It's extremely late and I am really tired. I am sorry but I don't have the energy to go into much depth about memory instructions.*

The main difference between memory and data-processing instructions is their **Funct** and **Src2** field.

The **funct** field has 6 different sub-fields:

- **I** - This bit determines if we will use an immediate value or not.
- **P** - This bit and the W bit are used for advanced indexing modes (Redundant).
- **U** - Determines if we add/subtract the offset to the base. (Not implemented)
- **B** - Determines if the instruction is a STR or LDR, together with the L bit.
- **W** - Same as P.
- **L** - If L = 0 => STR, if L = 1 => LDR.

The **Src2** field has 2 different variations:
When using immediate values - **Imm12**
When using registers - **shamt5**, **sh**, **bit number 4**, **Rm**

## List of all supported instructions:

*In the beginning the **register file** and the **data memory** (RAM) have 0s in all positions.*

This is the list of all possible operations which my processor can run (as well as details on all of them):

**Data-processing instructions:**

**1. ADD R1, R1, #10** - *(**ADD** using an **immediate** (constant) value)*
<u>11100010100000010001000000001010</u>
Take the value found in R1 (which is 0) and add the constant 10 to it, store the result in register 1 of the register file (R1 = 10).

**2. SUB R2, R2, #5** - *(**SUB** using an **immediate** (constant) value)*
<u>11100010010000100010000000000101</u>
Take the value found in R2 (which is 0) and subtract the constant 5 from it, store the result in register 2 of the register file (R2 = -5).

**3. ADD R3, R1, R2** - *(**ADD** using a non-shifted second **register**)*
<u>11100000100000010011000000000010</u>
Take the value found in R1 (which is 10) and add the value found in R2 (which is -5) to it, store the result in register 3 of the register file (R3 = 5).

**4. SUB R4, R1, R2** - *(**SUB** using a non-shifted second **register**)*
<u>11100000010000010100000000000010</u>
Take the value found in R1 (which is 10) and subtract the value found in R2 (which is -5) from it, store the result in register 4 of the register file (R3 = 15).

**5. AND R5, R3, #5** - *(**AND** using an **immediate** (constant) value)*
<u>11100010000000110101000000000101</u>
Take the value found in R3 (which is 5) and perform the bitwise AND operation on the constant value 5, store the result in register 5 of the register file (R5 = 5).

**6. ORR R6, R4, #0** - *(**ORR** using an **immediate** (constant) value)*
<u>11100011100001000110000000000000</u>
Take the value found in R4 (which is 15) and perform the bitwise OR operation on the constant value 0, store the result in register 6 of the register file (R6 = 15).

**7. AND R7, R4, R6** - *(**AND** using a non-shifted second **register**)*
11100000000001000111000000000110
Take the value found in R4 (which is 15) and perform the bitwise AND operation on the value found in R6 (which is 15), store the result in register 7 of the register file (R7 = 15).

**8. ORR R8, R1, R6** - *(**ORR** using a non-shifted second **register**)*
11100001100000011000000000000110
Take the value found in R1 (which is 10) and perform the bitwise OR operation on the value found in R6 (which is 15), store the result in register 8 of the register file (R8 = 15).

**Memory instructions:**

**9. STR R2, R0, #1** - *(**STR** using an **immediate** (constant) offset)*
11100100100000000010000000000001
Store the value found in R2 of the register file (R2 = -5) in the memory position which is found by adding the constant 1 to R0 (5 is stored to the memory position 1).

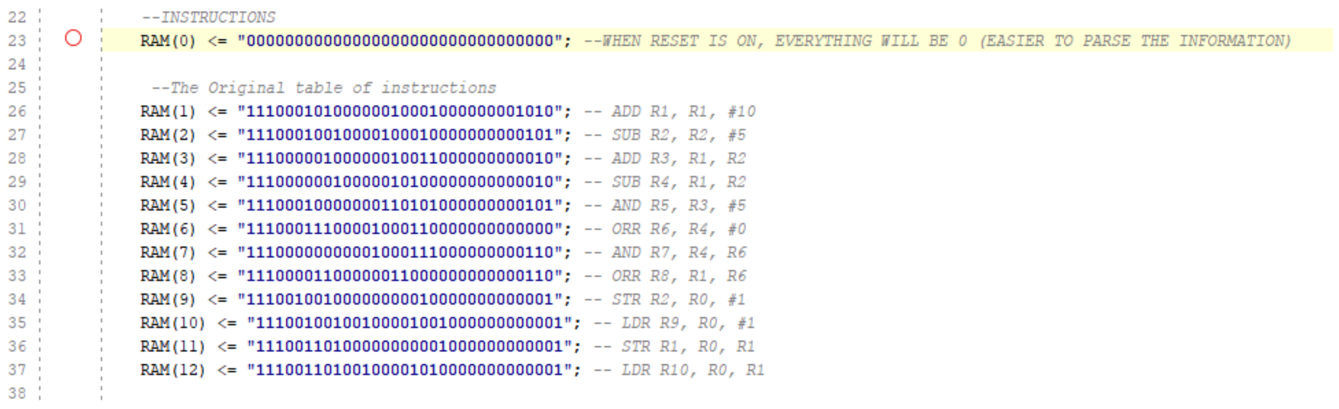
**10. LDR R9, R0, #1** - *(**LDR** using an **immediate** (constant) offset)*
11100100100100001001000000000001
Load the value found in memory position R0 (R0 = 0) + 1 into R9 of the register file (we load -5 into R9).

**11. STR R1, R0, R1** - *(**STR** using a non-shifted **register** offset)*
11100110100000000001000000000001
Store the value found in R1 of the register file (R1 = 10) in the memory position which is found by adding the value found in R0 and R1 (10 is stored to the memory position 10).

**12. LDR R10, R0, R1** - *(**LDR** using a non-shifted **register** offset)*
11100110100100001010000000000001
Load the value found in memory position R0 (R0 = 0) + R1 (R1 = 10) into R10 of the register file (we load 10 into R10).

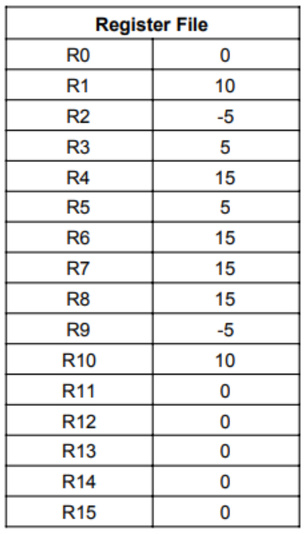
Giving us a grand total of **12 different possible operations** which can be performed using my CPU.

All of these instructions can be found in the instruction memory of the CPU (a picture of the instruction memory is given down below).

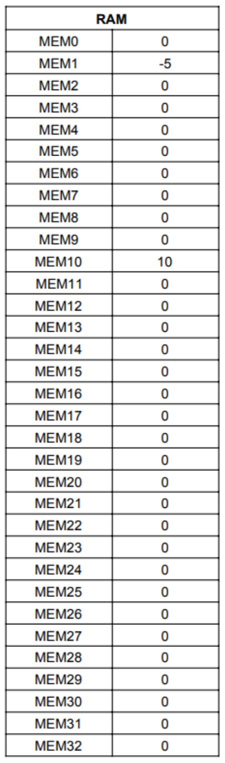


```
    --INSTRUCTIONS
    RAM(0) <= "00000000000000000000000000000000"; --WHEN RESET IS ON, EVERYTHING WILL BE 0 (EASIER TO PARSE THE INFORMATION)

     --The Original table of instructions
    RAM(1) <= "11100010100000010001000000001010"; -- ADD R1, R1, #10
    RAM(2) <= "11100010010000100010000000000101"; -- SUB R2, R2, #5
    RAM(3) <= "11100000100000010011000000000010"; -- ADD R3, R1, R2
    RAM(4) <= "11100000010000010100000000000010"; -- SUB R4, R1, R2
    RAM(5) <= "11100010000000110101000000000101"; -- AND R5, R3, #5
    RAM(6) <= "11100011100001000110000000000000"; -- ORR R6, R4, #0
    RAM(7) <= "11100000000001000111000000000110"; -- AND R7, R4, R6
    RAM(8) <= "11100001100000011000000000000110"; -- ORR R8, R1, R6
    RAM(9) <= "11100100100000000010000000000001"; -- STR R2, R0, #1
    RAM(10) <= "11100100100100001001000000000001"; -- LDR R9, R0, #1
    RAM(11) <= "11100110100000000001000000000001"; -- STR R1, R0, R1
    RAM(12) <= "11100110100100001010000000000001"; -- LDR R10, R0, R1
```

The image down below shows the final register file and RAM:



| Register File | |
|---|---|
| R0 | 0 |
| R1 | 10 |
| R2 | -5 |
| R3 | 5 |
| R4 | 15 |
| R5 | 5 |
| R6 | 15 |
| R7 | 15 |
| R8 | 15 |
| R9 | -5 |
| R10 | 10 |
| R11 | 0 |
| R12 | 0 |
| R13 | 0 |
| R14 | 0 |
| R15 | 0 |



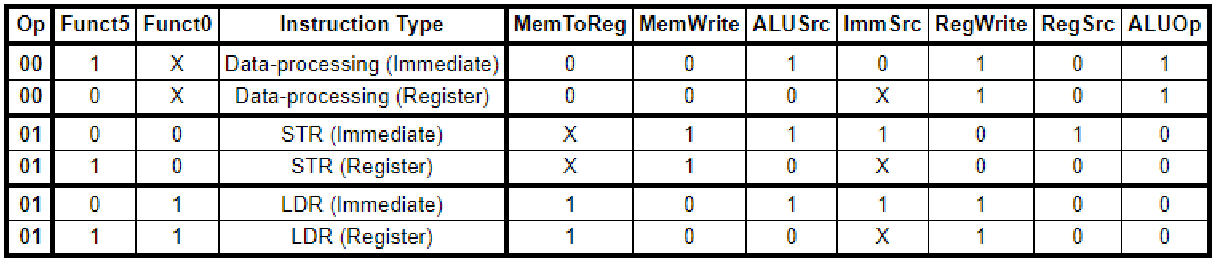
| RAM | |
|---|---|
| MEM0 | 0 |
| MEM1 | -5 |
| MEM2 | 0 |
| MEM3 | 0 |
| MEM4 | 0 |
| MEM5 | 0 |
| MEM6 | 0 |
| MEM7 | 0 |
| MEM8 | 0 |
| MEM9 | 0 |
| MEM10 | 10 |
| MEM11 | 0 |
| MEM12 | 0 |
| MEM13 | 0 |
| MEM14 | 0 |
| MEM15 | 0 |
| MEM16 | 0 |
| MEM17 | 0 |
| MEM18 | 0 |
| MEM19 | 0 |
| MEM20 | 0 |
| MEM21 | 0 |
| MEM22 | 0 |
| MEM23 | 0 |
| MEM24 | 0 |
| MEM25 | 0 |
| MEM26 | 0 |
| MEM27 | 0 |
| MEM28 | 0 |
| MEM29 | 0 |
| MEM30 | 0 |
| MEM31 | 0 |
| MEM32 | 0 |

*We leave register 0 to the value 0 so that we can use it to jump locations in RAM memory easier.*

## The control unit:

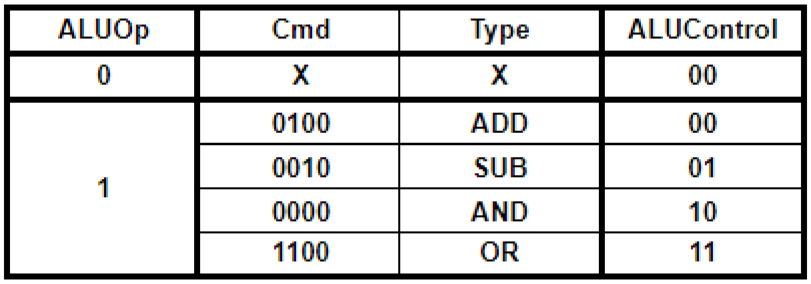
The control unit was the main part of this LAB. I have done some modifications to the main decoder so that it better suits the purposes of my CPU.

The picture below shows the **main decoder** of the **control unit**:



| Op | Funct5 | Funct0 | Instruction Type | MemToReg | MemWrite | ALUSrc | ImmSrc | RegWrite | RegSrc | ALUOp |
|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 1 | X | Data-processing (Immediate) | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 00 | 0 | X | Data-processing (Register) | 0 | 0 | 0 | X | 1 | 0 | 1 |
| 01 | 0 | 0 | STR (Immediate) | X | 1 | 1 | 1 | 0 | 1 | 0 |
| 01 | 1 | 0 | STR (Register) | X | 1 | 0 | X | 0 | 0 | 0 |
| 01 | 0 | 1 | LDR (Immediate) | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| 01 | 1 | 1 | LDR (Register) | 1 | 0 | 0 | X | 1 | 0 | 0 |

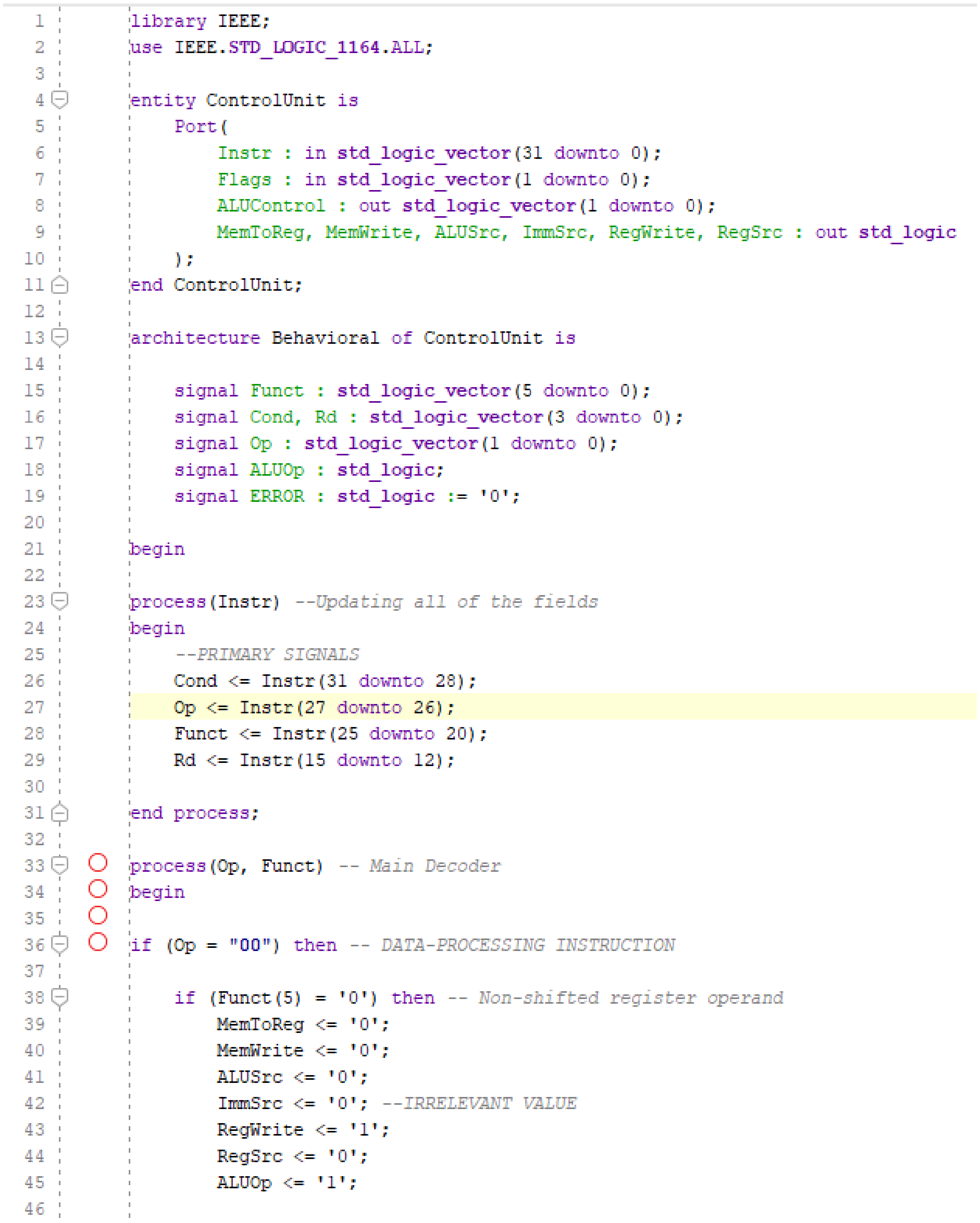
The picture below shows the **ALU decoder** of the **control unit**:



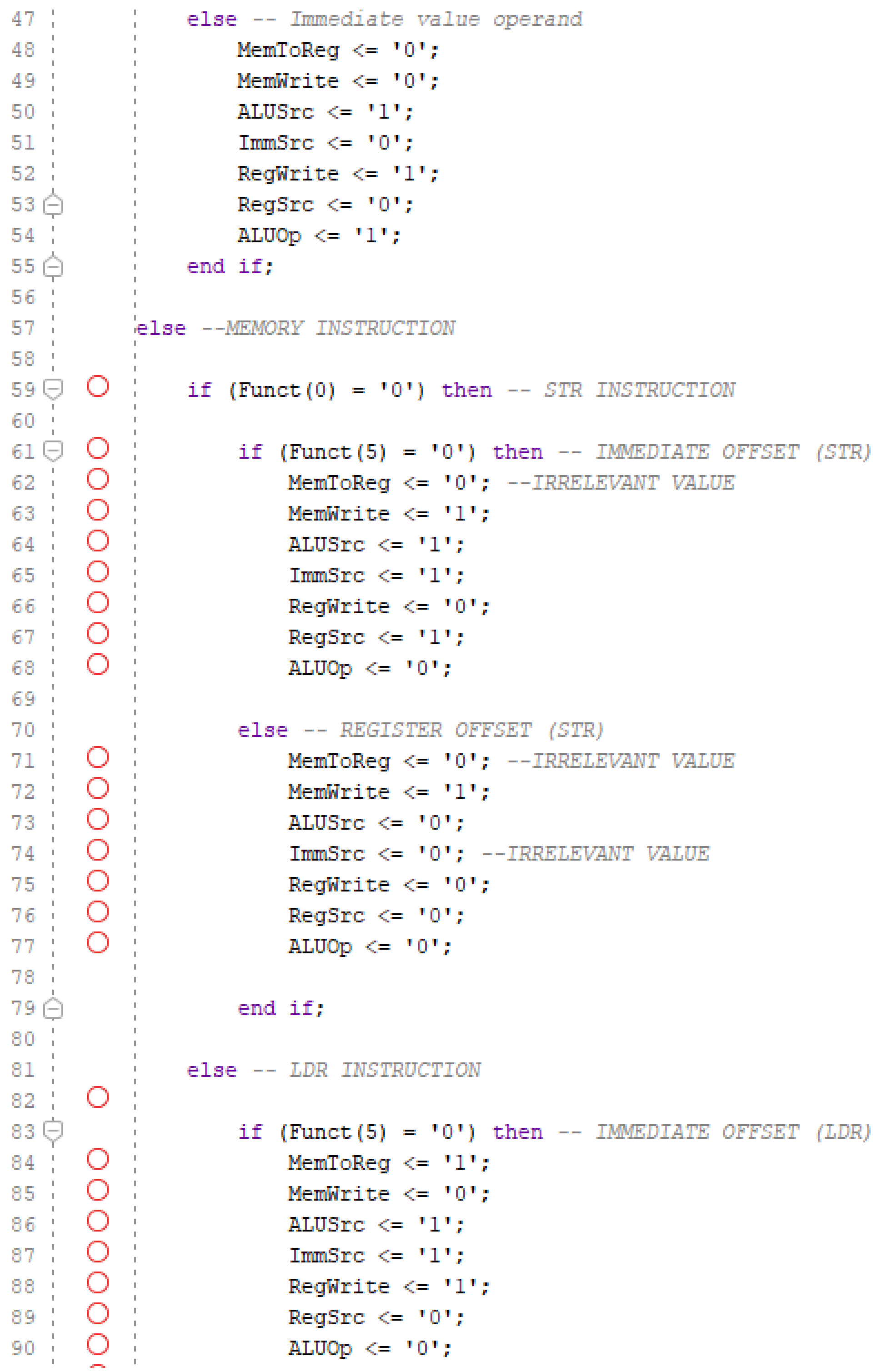
| ALUOp | Cmd | Type | ALUControl |
|---|---|---|---|
| 0 | X | X | 00 |
| 1 | 0100 | ADD | 00 |
|  | 0010 | SUB | 01 |
|  | 0000 | AND | 10 |
|  | 1100 | OR | 11 |

These tables are used for decoding the instructions and activating the correct control signals for each of the possible instructions.

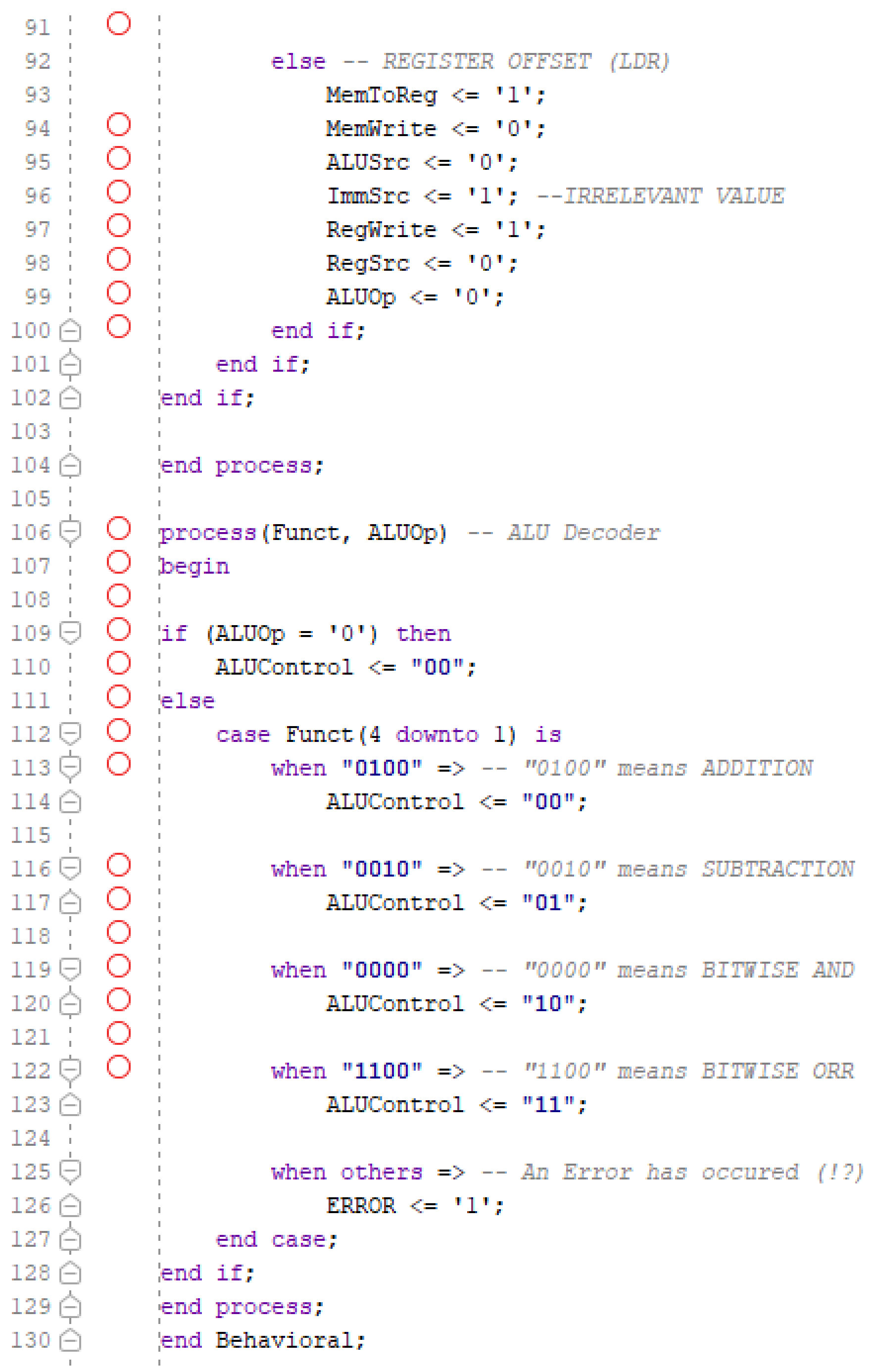
# DESIGN CODE



```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ControlUnit is
    Port(
        Instr : in std_logic_vector(31 downto 0);
        Flags : in std_logic_vector(1 downto 0);
        ALUControl : out std_logic_vector(1 downto 0);
        MemToReg, MemWrite, ALUSrc, ImmSrc, RegWrite, RegSrc : out std_logic
    );
end ControlUnit;

architecture Behavioral of ControlUnit is

    signal Funct : std_logic_vector(5 downto 0);
    signal Cond, Rd : std_logic_vector(3 downto 0);
    signal Op : std_logic_vector(1 downto 0);
    signal ALUOp : std_logic;
    signal ERROR : std_logic := '0';

begin

process(Instr) --Updating all of the fields
begin
    --PRIMARY SIGNALS
    Cond <= Instr(31 downto 28);
    Op <= Instr(27 downto 26);
    Funct <= Instr(25 downto 20);
    Rd <= Instr(15 downto 12);

end process;

process(Op, Funct) -- Main Decoder
begin

if (Op = "00") then -- DATA-PROCESSING INSTRUCTION

    if (Funct(5) = '0') then -- Non-shifted register operand
        MemToReg <= '0';
        MemWrite <= '0';
        ALUSrc <= '0';
        ImmSrc <= '0'; --IRRELEVANT VALUE
        RegWrite <= '1';
        RegSrc <= '0';
        ALUOp <= '1';

```



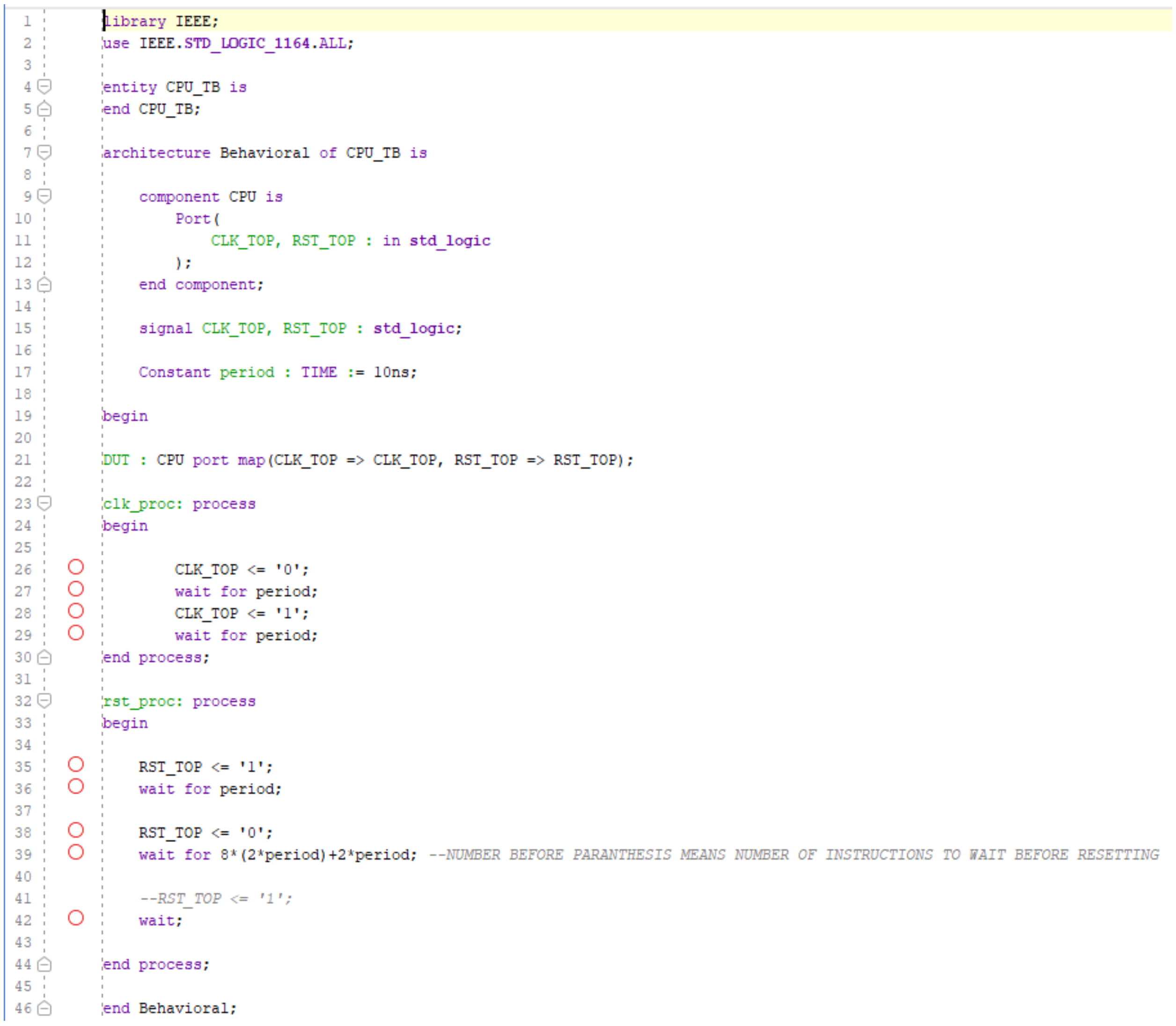
```vhdl
    else -- Immediate value operand
        MemToReg <= '0';
        MemWrite <= '0';
        ALUSrc <= '1';
        ImmSrc <= '0';
        RegWrite <= '1';
        RegSrc <= '0';
        ALUOp <= '1';
    end if;

else --MEMORY INSTRUCTION

    if (Funct(0) = '0') then -- STR INSTRUCTION

        if (Funct(5) = '0') then -- IMMEDIATE OFFSET (STR)
            MemToReg <= '0'; --IRRELEVANT VALUE
            MemWrite <= '1';
            ALUSrc <= '1';
            ImmSrc <= '1';
            RegWrite <= '0';
            RegSrc <= '1';
            ALUOp <= '0';

        else -- REGISTER OFFSET (STR)
            MemToReg <= '0'; --IRRELEVANT VALUE
            MemWrite <= '1';
            ALUSrc <= '0';
            ImmSrc <= '0'; --IRRELEVANT VALUE
            RegWrite <= '0';
            RegSrc <= '0';
            ALUOp <= '0';

        end if;

    else -- LDR INSTRUCTION

        if (Funct(5) = '0') then -- IMMEDIATE OFFSET (LDR)
            MemToReg <= '1';
            MemWrite <= '0';
            ALUSrc <= '1';
            ImmSrc <= '1';
            RegWrite <= '1';
            RegSrc <= '0';
            ALUOp <= '0';
```



```
        else -- REGISTER OFFSET (LDR)
            MemToReg <= '1';
            MemWrite <= '0';
            ALUSrc <= '0';
            ImmSrc <= '1'; --IRRELEVANT VALUE
            RegWrite <= '1';
            RegSrc <= '0';
            ALUOp <= '0';
        end if;
    end if;
end if;

end process;

process(Funct, ALUOp) -- ALU Decoder
begin

if (ALUOp = '0') then
    ALUControl <= "00";
else
    case Funct(4 downto 1) is
        when "0100" => -- "0100" means ADDITION
            ALUControl <= "00";

        when "0010" => -- "0010" means SUBTRACTION
            ALUControl <= "01";

        when "0000" => -- "0000" means BITWISE AND
            ALUControl <= "10";

        when "1100" => -- "1100" means BITWISE ORR
            ALUControl <= "11";

        when others => -- An Error has occured (!?)
            ERROR <= '1';
    end case;
end if;
end process;
end Behavioral;
```

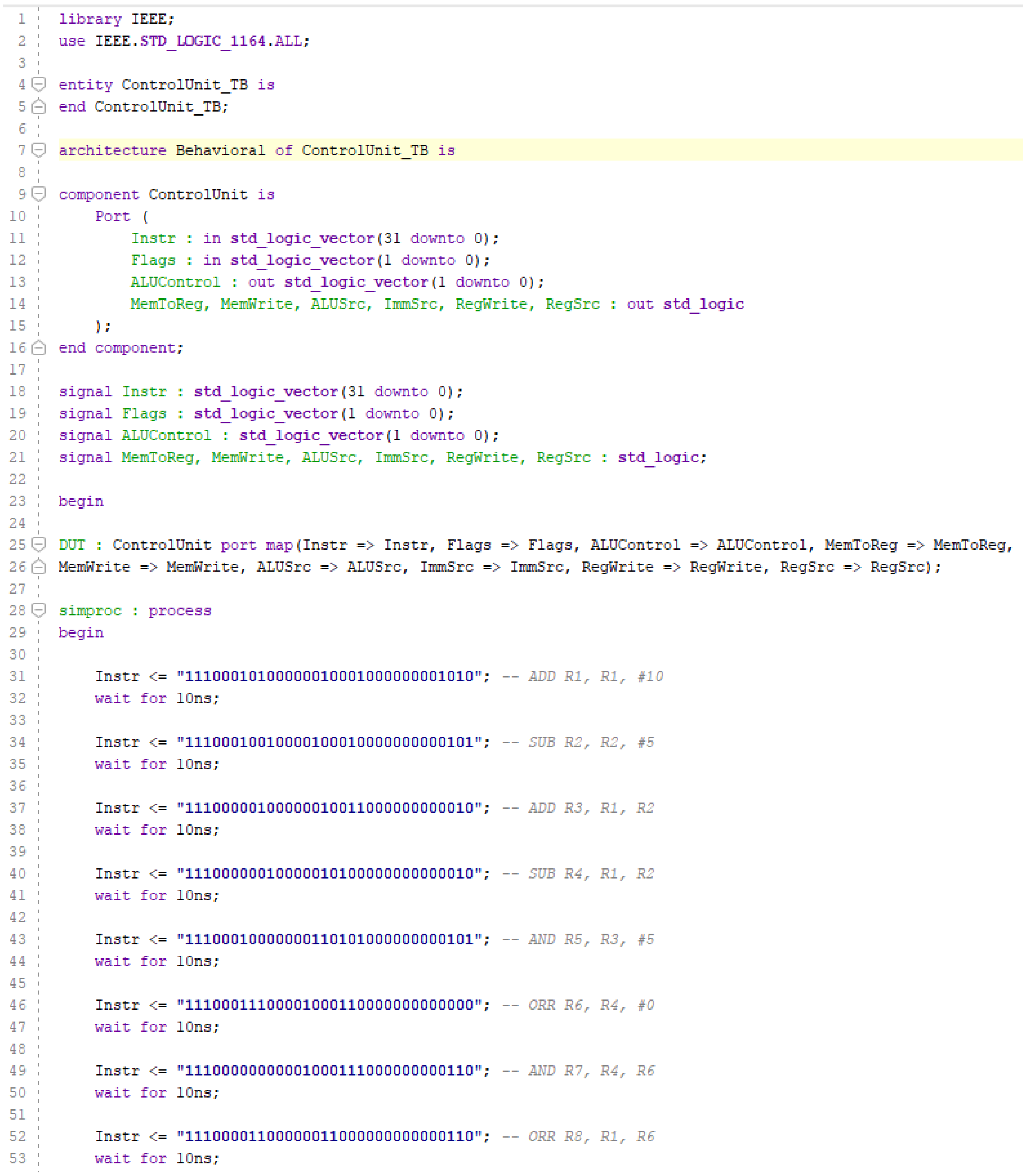
In the pictures above we can see the design file of the control unit. It is fairly simple once you have followed all of the possible data paths and laid out a table with all of the control signals (as I have done above).

# TESTBENCH CODE

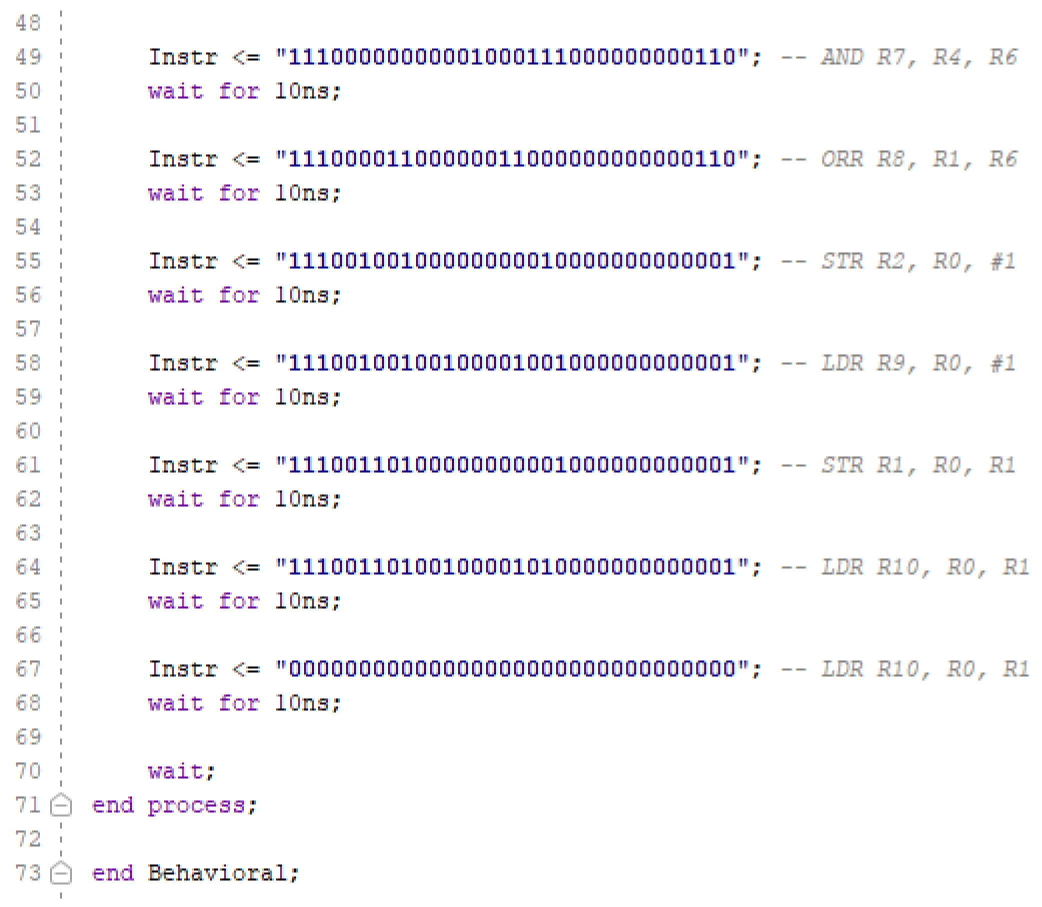


```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity CPU_TB is
end CPU_TB;

architecture Behavioral of CPU_TB is

    component CPU is
        Port(
            CLK_TOP, RST_TOP : in std_logic
        );
    end component;

    signal CLK_TOP, RST_TOP : std_logic;

    Constant period : TIME := 10ns;

begin

DUT : CPU port map(CLK_TOP => CLK_TOP, RST_TOP => RST_TOP);

clk_proc: process
begin

        CLK_TOP <= '0';
        wait for period;
        CLK_TOP <= '1';
        wait for period;
end process;

rst_proc: process
begin

    RST_TOP <= '1';
    wait for period;

    RST_TOP <= '0';
    wait for 8*(2*period)+2*period; --NUMBER BEFORE PARANTHESIS MEANS NUMBER OF INSTRUCTIONS TO WAIT BEFORE RESETTING

    --RST_TOP <= '1';
    wait;

end process;

end Behavioral;
```

This is the testbench of the CPU (top file) of the system. The thing to note here is that I have made a little formula to flip on the reset after X instructions where X is the number of instructions + 1 (because of the timing issues the instructions are always one behind). But I have decided in the end to never flip the reset back on after the X instructions get executed (mostly because of sleep deprivation).



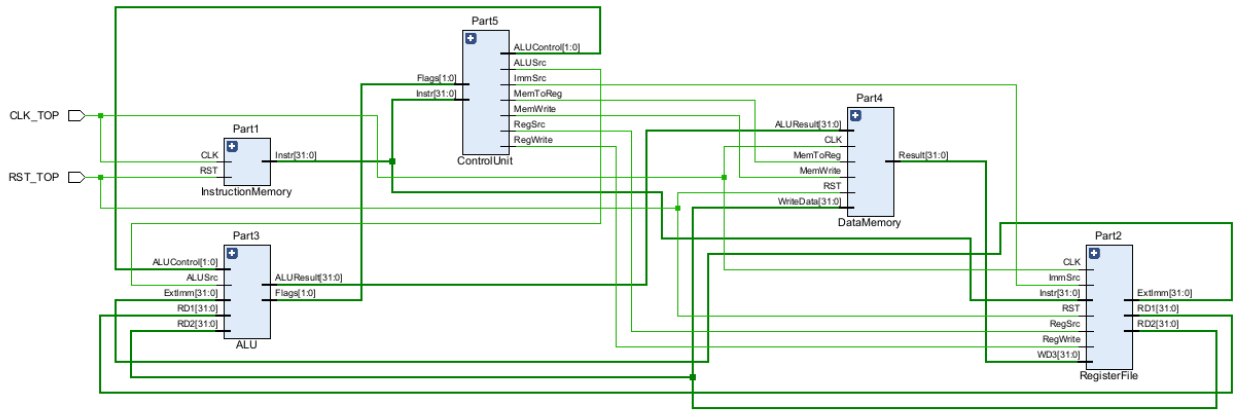
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ControlUnit_TB is
end ControlUnit_TB;

architecture Behavioral of ControlUnit_TB is

component ControlUnit is
    Port (
        Instr : in std_logic_vector(31 downto 0);
        Flags : in std_logic_vector(1 downto 0);
        ALUControl : out std_logic_vector(1 downto 0);
        MemToReg, MemWrite, ALUSrc, ImmSrc, RegWrite, RegSrc : out std_logic
    );
end component;

signal Instr : std_logic_vector(31 downto 0);
signal Flags : std_logic_vector(1 downto 0);
signal ALUControl : std_logic_vector(1 downto 0);
signal MemToReg, MemWrite, ALUSrc, ImmSrc, RegWrite, RegSrc : std_logic;

begin

DUT : ControlUnit port map(Instr => Instr, Flags => Flags, ALUControl => ALUControl, MemToReg => MemToReg,
MemWrite => MemWrite, ALUSrc => ALUSrc, ImmSrc => ImmSrc, RegWrite => RegWrite, RegSrc => RegSrc);

simproc : process
begin

    Instr <= "11100010100000010001000000001010"; -- ADD R1, R1, #10
    wait for 10ns;

    Instr <= "11100010010000100010000000000101"; -- SUB R2, R2, #5
    wait for 10ns;

    Instr <= "11100000100000010011000000000010"; -- ADD R3, R1, R2
    wait for 10ns;

    Instr <= "11100000010000010100000000000010"; -- SUB R4, R1, R2
    wait for 10ns;

    Instr <= "11100010000000110101000000000101"; -- AND R5, R3, #5
    wait for 10ns;

    Instr <= "11100011100001000110000000000000"; -- ORR R6, R4, #0
    wait for 10ns;

    Instr <= "11100000000001000111000000000110"; -- AND R7, R4, R6
    wait for 10ns;

    Instr <= "11100001100000011000000000000110"; -- ORR R8, R1, R6
    wait for 10ns;
```



```vhdl

    Instr <= "11100000000000100011100000000110"; -- AND R7, R4, R6
    wait for 10ns;

    Instr <= "11100001100000011000000000000110"; -- ORR R8, R1, R6
    wait for 10ns;

    Instr <= "11100100100000000010000000000001"; -- STR R2, R0, #1
    wait for 10ns;

    Instr <= "11100100100100001001000000000001"; -- LDR R9, R0, #1
    wait for 10ns;

    Instr <= "11100110100000000001000000000001"; -- STR R1, R0, R1
    wait for 10ns;

    Instr <= "11100110100100001010000000000001"; -- LDR R10, R0, R1
    wait for 10ns;

    Instr <= "00000000000000000000000000000000"; -- LDR R10, R0, R1
    wait for 10ns;

    wait;
end process;

end Behavioral;
```

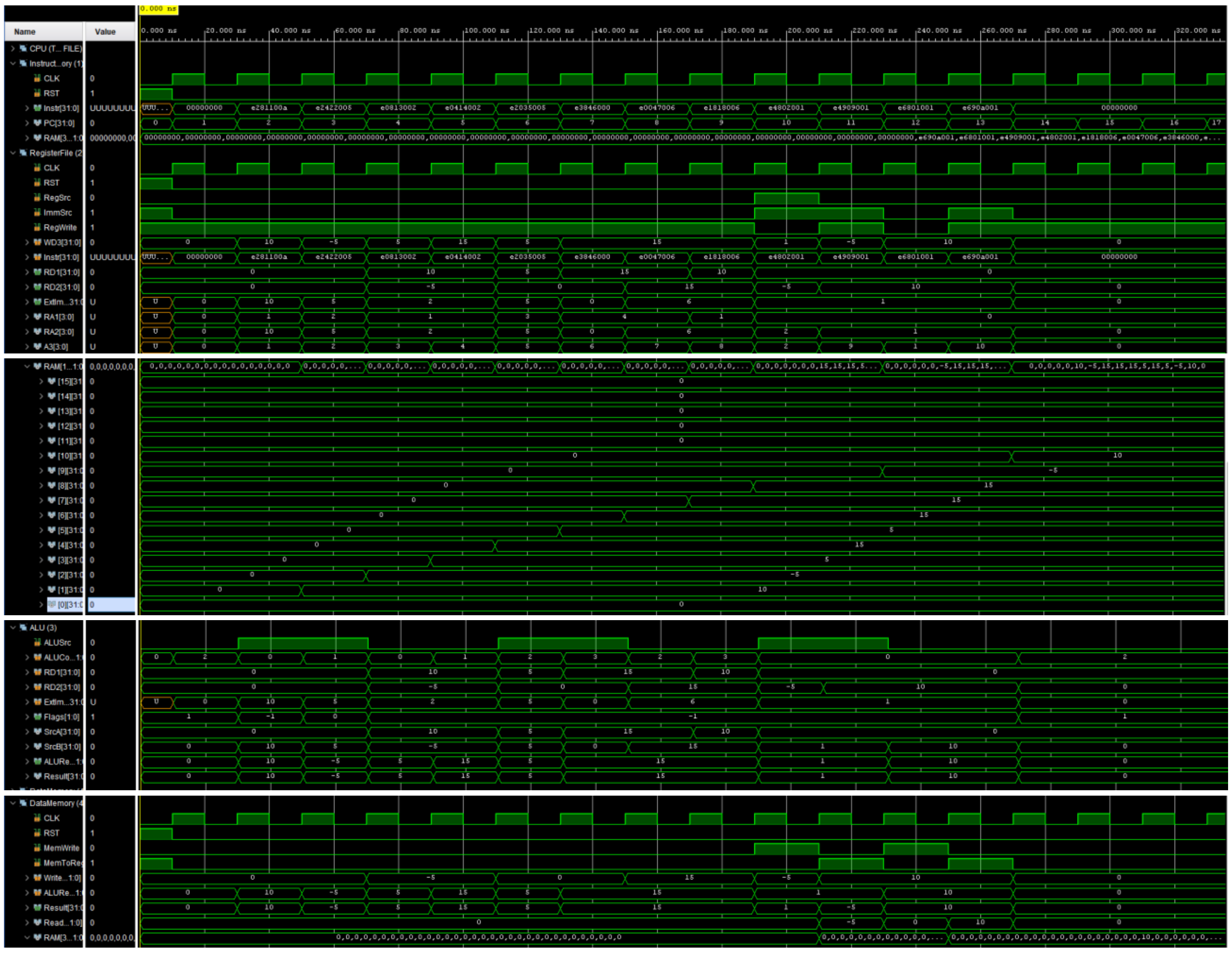
In the pictures above we can see the test bench of the control unit. I have tested all 12 instructions, which I have provided earlier, to see if they all activate the correct control signals. They do, I just forgot to add the simulation pictures of this testbench (I have already zipped the project files and prepared them for upload).

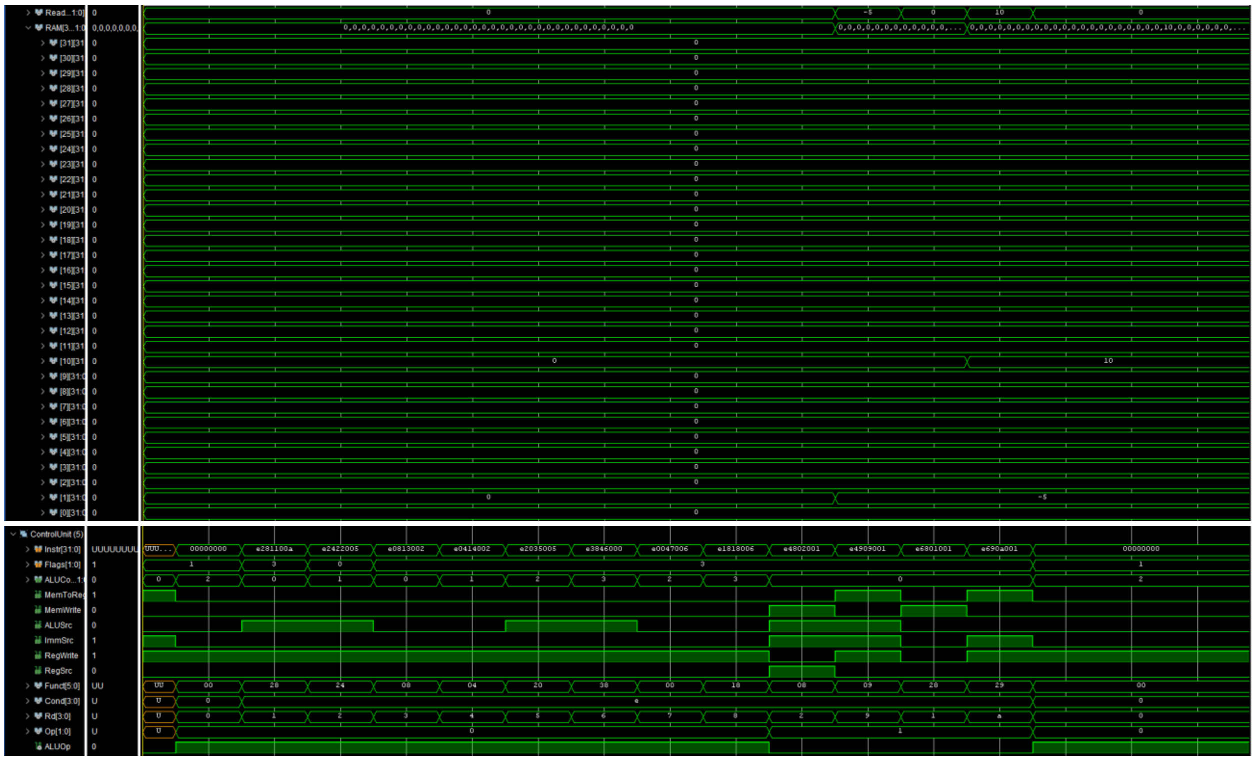
# ELABORATED DESIGN



Since I have made a LOT of changes not only in the top file (CPU) but also in all of the other components (more readable and efficient as well as fix some issues and remove some redundant parts) I thought it would be a good idea to take another look at the elaborated design of my CPU. And as I was expecting, the end elaborated design does look quite a bit more different compared to the previous LAB (LAB7).

# BEHAVIORAL SIMULATION





In the above pictures we can see the processor doing its thing.

As we can see, all of the registers and ram positions have all of the values that we expected them to have after running the 12 instructions (feel free to give it a good look and compare the expected results with the actual results). This means that the processor is fully functional.

One thing I want to note is that the instruction memory is a bit wonky. What I mean by that is that the instruction that the CPU is working with is always one behind of the PC. This means that if the PC has the value 5, the processor is working with the instruction which is found in the RAM(4) of the instruction memory and not RAM(5). The reason why I did this is because when I used the asynchronous approach (which I showed you on Friday) the instructions were perfectly in line with the clock (and PC) but for some reason, the register file didn’t work as it should. Once I made the instruction memory fully synchronous to the clock the system started to work better.

One more thing, all of the values get written 1 clock cycle behind. There are some wonky issues with timing but I couldn’t fix it. It would probably be better to rewrite the entire CPU than to try and find the problem which is causing this timing issue.

Now, I will explain the data path and my reasoning for the first instruction (how it all works).

*(I wanted to do this for all of the instructions and give detailed pictures and analysis for all of them but I don't have the energy to do it tonight anymore)*

Okay, let's get started!

The first instruction for this CPU is the following instruction:

**ADD R1, R1, #10** - this is the assembly instruction

11100010100000010001000000001010 - this is the machine code instruction

And what it basically does is that it takes the value found in R1 (which is 0) and adds the constant 10 to it. It then stores the result in register 1 of the register file (meaning R1 will be set to 10).

Now lets decode the machine code instruction:

11100010100000010001000000001010

The bits that are colored with red are "fixed" bits. What does that mean?
It means that for the purposes of this CPU they have no real purpose and for all possible ADD (immediate value) instructions, they will remain the same.

11100010100000010001000000001010

This is the op code, "00" means that this is a data-processing instruction.

11100010100000010001000000001010

This is the funct field of the instruction, here we determine whether we use an immediate value or non-shifted register (violet color), the command which we will be using (yellow color). For our purposes I = '1' and cmd = "0100" which means we will be using an immediate value and the ADD command. We ignore the red part because it is flag related, meaning it is redundant.

11100010100000010001000000001010

And lastly we have the address part of the instruction. Yellow stands for the first operand, green stands for the destination register and violet stands for the second operand. In our case Rn = 1, Rd = 1 and #10 as our second operand.

Now that we have fully decoded our instruction, we have to look at the control signals which we need to flip for it to work. But before that, we have to look at what path the data has to take for it to function properly.

We start from the instruction memory where we get to RAM(1) in our case this means that we load the first proper instruction into our system. The instruction then gets “split” into multiple sections/lines. First we will go to the control unit, and see the logic behind the entire CPU.

As we can see on the control unit block, there are 2 main inputs: Instr & ALUFlags.

The instruction gets separated into more sections inside of the control unit where it goes through 2 main blocks: The main decoder and the ALU decoder.

After going through those 2 blocks, we get all of our control signals. We will take a look at all of them, starting with MemToReg.

**MemToReg** will be set to 0 because we do not want to read from the data memory and we want to immediately get the value from the ALU and store it in the file register.

**MemWrite** will be set to 0 because we do not want to write to the data memory.

**ALUControl** will be set to 00 because that is what we defined the addition to be. (This is handled by the ALU decoder). (For memory instructions its always 0 because implementing the subtraction of offsets would add a bit more complexity to the CPU and I didn’t have the time to deal with that)

**ALUSrc** will be set to 1 because we want to get our second operand from the extension block where we have our immediate values.

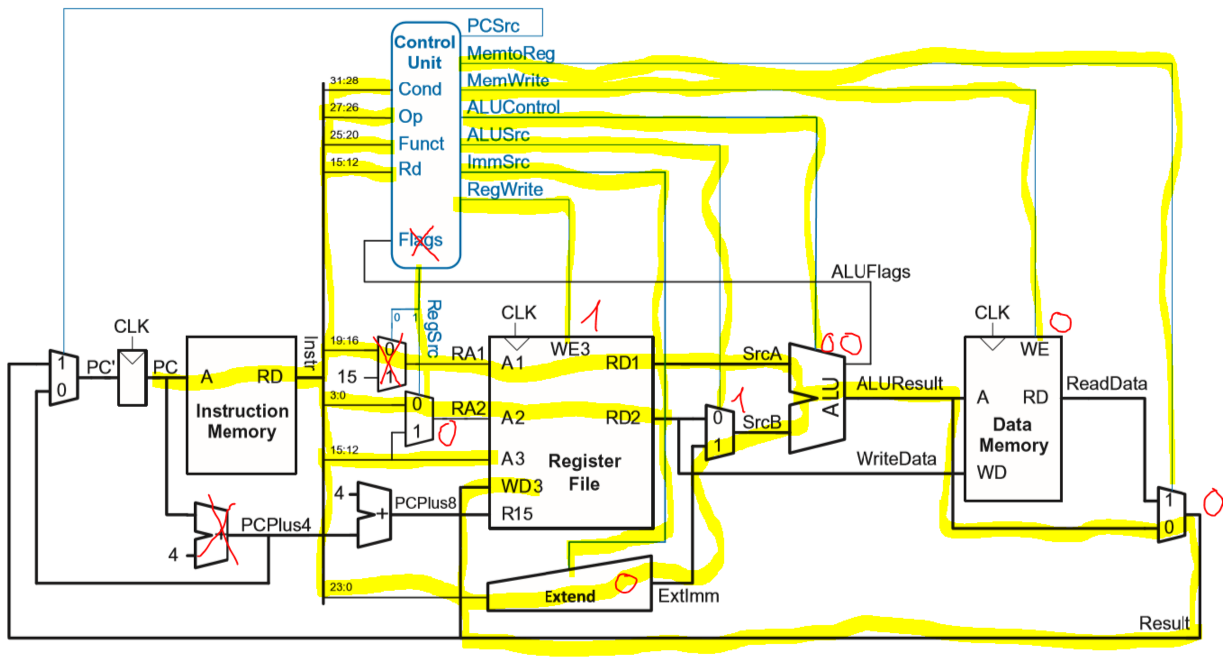
**ImmSrc** will be set to 0 because it is a data-processing instruction, meaning 8 bits of information and not 12. (technically speaking, because we haven’t implemented the rotations).

**RegWrite** will be set to 1, because we want to write to the register.

**RegSrc** will be set to 0, not that it really matters because we are using an immediate value and not a second register.

After activating/deactivating all of the control signals, the data now can flow freely.

*(We ignore the PCSrc, Second RegSrc mux and ALUFlags because they serve no purpose for our CPU)*



In the above picture we can see how the data flows through the CPU after all of the control signals have been decided.

The data goes through the register file into the ALU (where SrcA comes from the register file and SrcB comes from the extension block) where SrcA gets added with SrcB and the ALUResult gets directly written onto the Register file with address A3.

**Ta da! We have our first instruction of the CPU!**

Thank you for your time.