

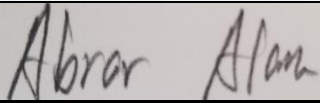


**Department of Electrical,  
Computer, & Biomedical Engineering**  
Faculty of Engineering & Architectural Science

<b>Course Title:</b>	Computer Organization and Architecture
<b>Course Number:</b>	COE 608
<b>Semester/Year (e.g.F2016)</b>	Winter 2021
<b>Instructor:</b>	Professor Nagi Mekhiel

<i>Assignment/Lab Number:</i>	5
<i>Assignment/Lab Title:</i>	CPU Control Unit Design

<i>Submission Date:</i>	March 22, 2021
<i>Due Date:</i>	March 20, 2021

<b>Student LAST Name</b>	<b>Student FIRST Name</b>	<b>Student Number</b>	<b>Section</b>	<b>Signature*</b>
ALAM	ABRAR	500725366	03	

\*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at:  
<http://www.ryerson.ca/senate/current/pol60.pdf>

## Table of Contents

Objective .....	2
Design and implementation.....	2
State Generator .....	3
Operational Decoder.....	4
Memory and State Generator .....	6
Results and Conclusions.....	6
LDAI .....	7
LDB .....	8
ADD .....	9
JMP .....	10
STB .....	11
Reference .....	13
Appendix .....	14

## Objective

The MIPS processor that is to be implemented in future lab, will consist of Arithmetic Logic Unit (ALU), registers, and data-memory, instruction memory blocks. However, to facilitate variety of MIPS operations, as seen in the previous lab (lab4b), we must need a programmable data-path that consists of numerous multiplexers (MUXs), reduces (RED), registers, write enable, and enable signals, which must be programmed according to the specific instruction being entered. To do these switchings, we must need a control signal generator. The objective of this lab is to implement a control signal block that accurately implements the instruction format specified in the *CPU specification* documentation (see **table 2**), and various control signal values implemented in the previous lab (see **table 1**). The block diagram of the control block is presented in **figure 1**.

## Design and implementation

**Figure 1** shows the Control Unit block diagram. Please note that although the entire 32-bit bus called **INST** has been shown as an input in **figure 1**, we only need **INST[31..28]** – *opcode* – and **INST[27..24]** – *function code* – are required (according to the **CPU Specification** document).

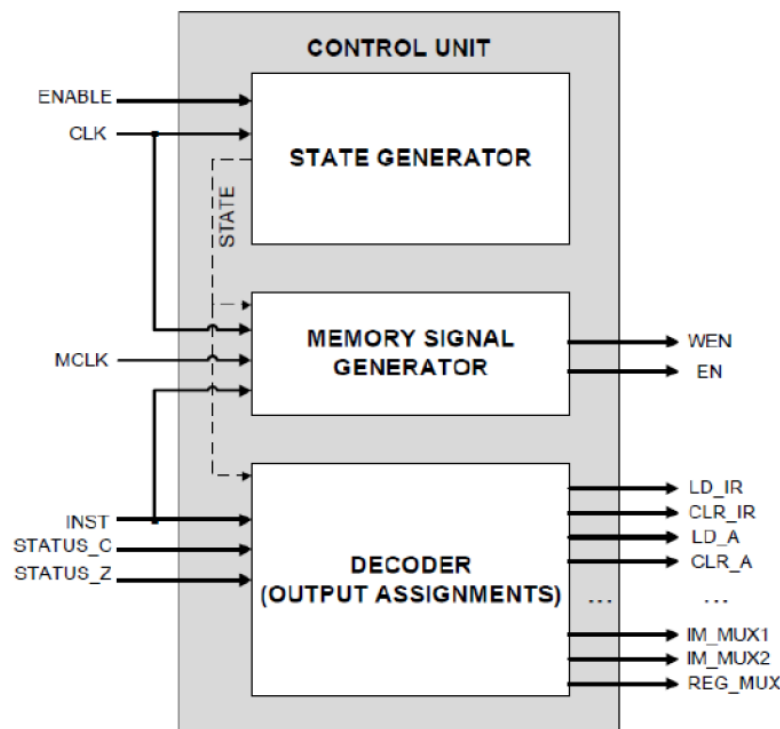


Figure 1: Implemented Control Unit block diagram.

As shown in **figure 1**, the control unit has been designed consists of three segments, such as, a sequential logic circuit to generate various states necessary to execute commands ( $T_0$ ,  $T_1$ , and  $T_2$ ), a memory signal generator that outputs **write enable (wen)** ,and **enable (en)** to be used as inputs to the data memory block to perform memory operations such as **LDA**, **LDAI**, etc. **Please note that we need a separate clock signal for memory called, mclk, which operates twice as fast as the main clock signal ( in this lab it was 25 ns), Clk (50 ns), to provide as low latency as possible for memory reading and**

writing operations, without being negatively affected by setup and hold times. According to **figure 1**, the very last part is for output assignments, which is a combinational circuit. A brief discussion on these 3 parts is presented as follows:

### State Generator

The state generator circuit is the sequential, and synchronous component of the control unit which generates appropriate instruction execution states (such as  $T_0$ ,  $T_1$ , and  $T_2$ ) based on the clock, Clk, and the current state of the Control Unit. **Figure 2** shows the functionalities of these states. **Please note that predecode state only useful during memory IO operations, such as LDA, LDAI, etc.**

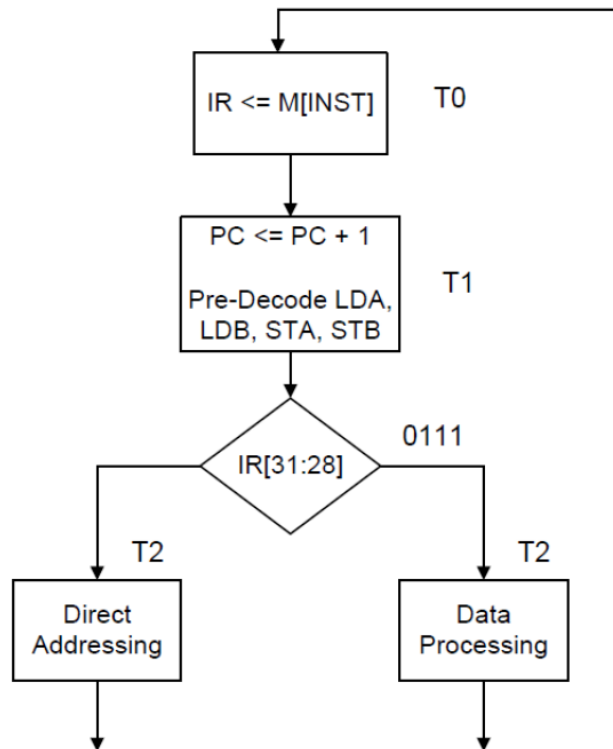


Figure 2: Instruction execution ASM state chart

Please also note that the **state generator** also generates a set of pulse signals **T** that describes the current state of the instruction being executed. The values of **T** can take the following form presented in **figure 3**.

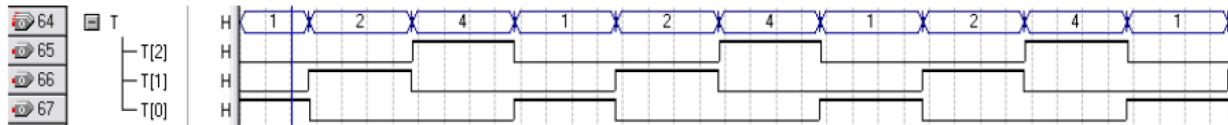


Figure 3: Format of the T pulse state descriptor signal.

As seen in **figure 1**, when the **ENABLE** signal is not high, the controller's state must be at **T0** and remain in this state until the **ENABLE** signal is high again. Whenever, this enabler is set to high, the control block resumes normal operation.

### Operational Decoder

This segment of the control block is solely a combinational logic circuit consisting of **case statements**, and some **if/else statements**, which sets correct control signals to the data-path during instruction execution stage **T2** (refer to **figure 4** for the data-path diagram for the CPU). In terms of VHDL code, this segment sets the required settings for the control signals, which was determined in the previous lab (lab 4b). This segment requires current state, status bits (**C** and **Z**), and the **INST** input to determine which instruction to execute, thus sets desired control signals for the data-path. **Table 1** shows the correct control signal values which we target to set in this segment.

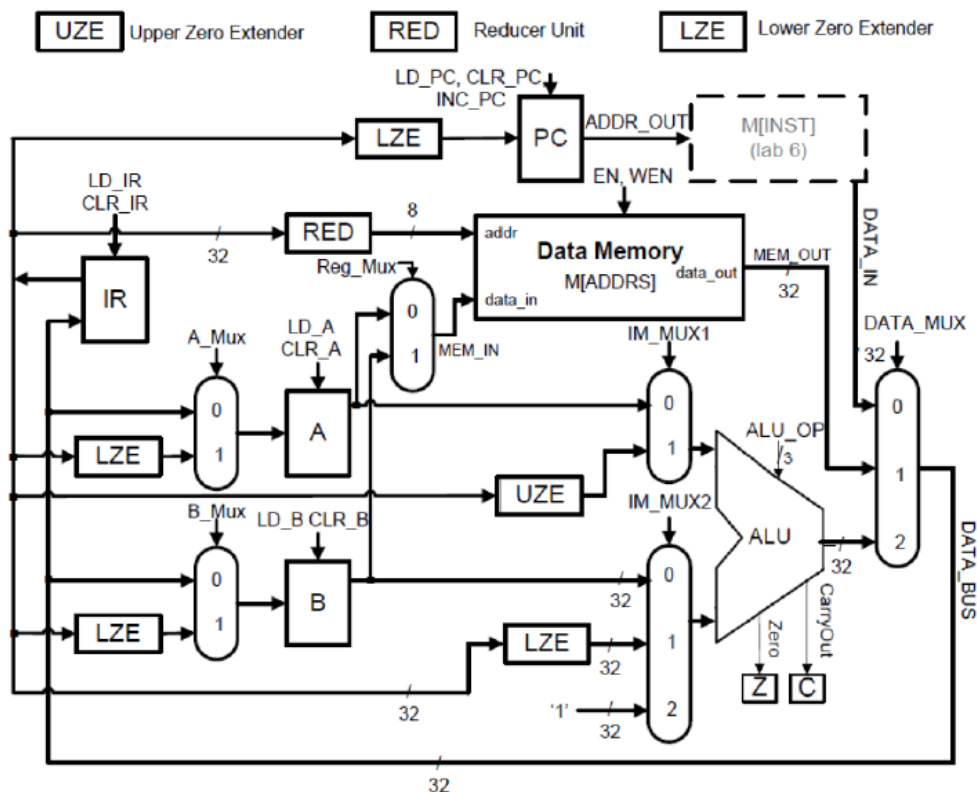


Figure 4: Data-path consists of setting up numerous MUXes and intermediate signal- a task necessary for the operational decoder segment

INST	CLR_IR LD_IR	LD_PC INC_PC	CLR_A LD_A	CLR_B LD_B	CLR_C LD_C	CLR_Z LD_Z	ALU OP	EN WEN	A/B MUX	REG MUX	Data MUX	IM_MUX1 IM_MUX_2
LDA	0/0	0/0	0/1	0/0	0/0	0/0	XXX	1/0	0/x	X	01	X
LDB	0/0	0/0	0/0	0/1	0/0	0/0	XXX	1/0	x/0	X	01	X
STA	0/0	0/0	0/0	0/0	0/0	0/0	XXX	1/1	X	0	X	X
STB	0/0	0/0	0/0	0/0	0/0	0/0	XXX	1/1	X	1	X	X
JMP	0/0	1/0	0/0	0/0	0/0	0/0	XXX	X	X	X	X	X
LDAI	0/0	0/0	0/1	0/0	0/0	0/0	XXX	X	1/x	X	X	X
LDBI	0/0	0/0	0/0	0/1	0/0	0/0	XXX	X	x/1	X	X	X
LUI	0/0	0/0	0/1	1/0	0/0	0/0	001	X	0/x	X	10	1/x
ANDI	0/0	0/0	0/1	0/0	0/1	0/1	000	X	0/x	X	10	0/01
DECA	0/0	0/0	0/1	0/0	0/1	0/1	110	X	0/x	X	10	0/10
ADD	0/0	0/0	0/1	0/0	0/1	0/1	010	X	0/x	X	10	0/00
SUB	0/0	0/0	0/1	0/0	0/1	0/1	110	X	0/x	X	10	0/00
INCA	0/0	0/0	0/1	0/0	0/1	0/1	010	X	0/x	X	10	0/10
AND	0/0	0/0	0/1	0/0	0/1	0/1	110	X	0/x	X	10	0/00
ADDI	0/0	0/0	0/1	0/0	0/1	0/1	010	X	0/x	X	10	0/01
ORI	0/0	0/0	0/1	0/0	0/1	0/1	001	X	0/x	X	10	0/01
ROL	0/0	0/0	0/1	0/0	0/1	0/1	100	X	0/x	X	10	0/x
ROR	0/0	0/0	0/1	0/0	0/1	0/1	101	X	0/x	X	10	0/x
CLRA	0/0	0/0	1/0	0/0	0/0	0/0	XXX	X	X	X	X	X
CLRB	0/0	0/0	0/0	1/0	0/0	0/0	XXX	X	X	X	X	X
CLRC	0/0	0/0	0/0	0/0	0/0	0/0	XXX	X	X	X	X	X
CLRZ	0/0	0/0	0/0	0/0	0/0	1/0	XXX	X	X	X	X	X
PC <= PC+4	0/0	1/1	0/0	0/0	0/0	0/0	XXX	X	X	X	X	X
IR <= M[INST]	0/1	0/0	0/0	0/0	0/0	0/0	XXX	X	X	X	00	X
PC <= IR[15..0]	0/0	1/0	0/0	0/0	0/0	0/0	XXX	X	X	X	X	X

**Table 1:** CPU control signals that was the basis of testing the Control Unit in this lab (we expect these values, except for the entries from the last three rows, during the execution state, **T2**).

Mnemonic	Function	Instruction Word		
		IR[31..28]	IR[27..16]	IR[15..0]
LDAI	$A \leftarrow IR[15:0]$	0000	X	IMM
LDBI	$B \leftarrow IR[15:0]$	0001	X	IMM
STA	$M[ADDRS] \leftarrow A, ADDR \leftarrow IR[15:0]$	0010	X	ADDRS
STB	$M[ADDRS] \leftarrow B, ADDR \leftarrow IR[15:0]$	0011	X	ADDRS
LDA	$A \leftarrow M[ADDRS], ADDR \leftarrow IR[15:0]$	1001	X	ADDRS
LDB	$B \leftarrow M[ADDRS], ADDR \leftarrow IR[15:0]$	1010	X	ADDRS
LUI	$A[31:16] \leftarrow IR[15:0], A[15:0] \leftarrow 0$	0100	X	IMM
JMP	$PC \leftarrow IR[15:0]$	0101	X	ADDRS
BEQ	If $(A == B)$ then $PC \leftarrow IR[15:0]$	0110	X	ADDRS
BNE	If $(A \neq B)$ then $PC \leftarrow IR[15:0]$	1000	X	ADDRS
		IR[31..28]	IR[27..24]	IR[15..0]
ADD	$A \leftarrow A + B$	0111	0000	X
ADDI	$A \leftarrow A + IR[15:0]$	0111	0001	IMM
SUB	$A \leftarrow A - B$	0111	0010	X
INCA	$A \leftarrow A + 1$	0111	0011	X
ROL	$A \leftarrow A \ll 1$	0111	0100	X
CLRA	$A \leftarrow 0$	0111	0101	X
CLRB	$B \leftarrow 0$	0111	0110	X
CLRC	$C \leftarrow 0$	0111	0111	X
CLRZ	$Z \leftarrow 0$	0111	1000	X
ANDI	$A \leftarrow A \text{ AND } IR[15:0]$	0111	1001	IMM
TSTZ	If $Z = 1$ then $PC \leftarrow PC + 1$	0111	1010	X
AND	$A \leftarrow A \text{ AND } B$	0111	1011	X
TSTC	If $C = 1$ then $PC \leftarrow PC + 1$	0111	1100	X
ORI	$A \leftarrow A \text{ OR } IR[15:0]$	0111	1101	IMM
DECA	$A \leftarrow A - 1$	0111	1110	X
ROR	$A \leftarrow A \gg 1$	0111	1111	X

Table 2: CPU instruction format that was used to implement and test the designed Control Unit in this lab.

## Memory and State Generator

Since data-memory IO operations require fetching operands/storing results from/into the data-memory block, and the data-memory itself is a sequential circuit powered by clock signal, **mclk**, these operations (such as, LDA, LDAI, LDB, etc.) will require the generation/verification of more than one state (such as predecode, execution state). In **figure 1**, the inputs **wen**, **en**, **clk**, and **mclk** signals are intended for this purpose. Besides, these signals also accounts for mitigating the effects of setup, hold times, and data-memory access latency.

## Results and Conclusions

Please note that the student tested and verified all the supported instruction on Quartus. However, for the sake of brevity **five** various types of instructions and their simulations have been discussed in this report.

	Name	V	0 ps	100,0 ns	200,0 ns	300,0 ns	400,0 ns	500,0 ns	600,0 ns	700,0 ns	800,0 ns	900,0 ns	1.0 us	
in	clk	B 0												
in	mclk	B 0												
in	enable	B 0												
in	INST	H 000	00000000 0000AAAA 00000000											
out	clr_IR	B 0												
out	ld_IR	B X												
out	ld_PC	B X												
out	inc_PC	B X												
out	clr_A	B X												
out	ld_A	B X												
out	clr_B	B X												
out	ld_B	B X												
out	clr_C	B X												
out	ld_C	B X												
out	ld_Z	B X												
out	ALU_op	B XXX	XXX											
out	en	B X												
out	wen	B X												
out	A_Mux	B X												
out	B_Mux	B X												
out	REG_Mux	B X												
out	DATA_Mux	B XX	XX 00											
out	IM_MUX1	B X												
out	IM_MUX2	B XX	XX											
out	T	B XXX	001 010 100 001 010 100 001 010 100 001 010 100 001 010 100 001 010 100											



## LDB

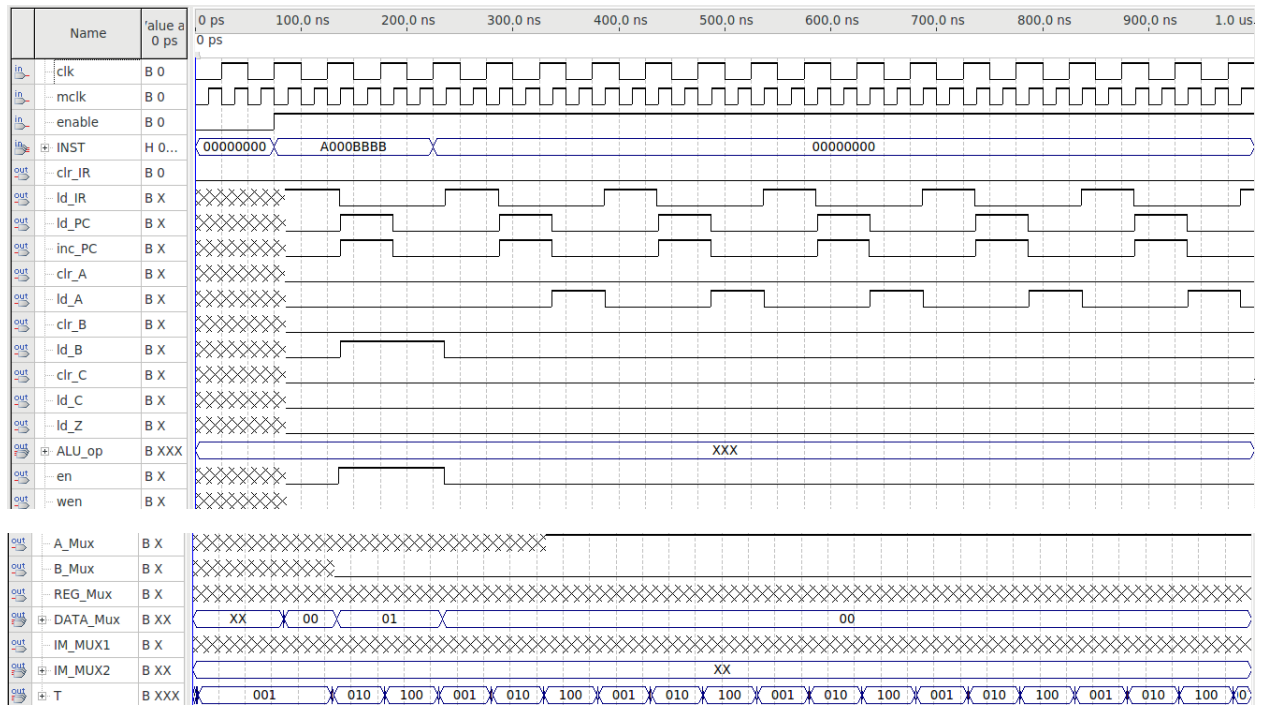


Figure 6: Timing simulation waveform of the Control Unit for LDB instruction.

From **figure 6**, we see that **T** signal starts from value **001**, that is, state **T0**. We know that during **T0**, and **T1** (corresponds to **T = 010**) the control signals will only account for loading Instruction Register (IR) with the value of the Instruction Memory (IM), which is pointed to by the current value of the Program Counter (PC); and subsequent increment of PC by 4, along with predecode operations applicable for memory IO operations. Thus, in our design, we expect the control signal values during **T0**, and **T1** states to correspond to the values that will successfully transfer Instruction Word from Instruction Memory to the Instruction Register, next, incrementing the current Program Counter value, and then perform the predecoding. In **figure 6**, we see that during **T=001** (state **T0**), our **ld\_IR** signal goes to high (means we loaded the Instruction Register from Instruction Memory), during **T=010**(state **T1**), we have both **ld\_PC**, and **inc\_PC** signal both to be high (indicating loading and incrementing the current Program Counter so that we can fetch the next instruction), and **en = 1** (means we will perform a reading operation on the data-memory), and **B\_Mux = 0** (meaning we will load Register **B** with the value we will read from the data-memory). Based on this observation we conclude that we have met the criteria for the fetch and predecode states (state **T0**, and **T1**), as illustrated in the ASM state diagram presented **figure 2**.

Now if we observe all the output values during **T = 100** (state **T2**, that is, the execution state) from **figure 6**, we can conclude that we indeed have achieved the correct control signal values stated in **table 1**, and thus have set correct control signal values for individual components illustrated in **figure 4**.

## ADD

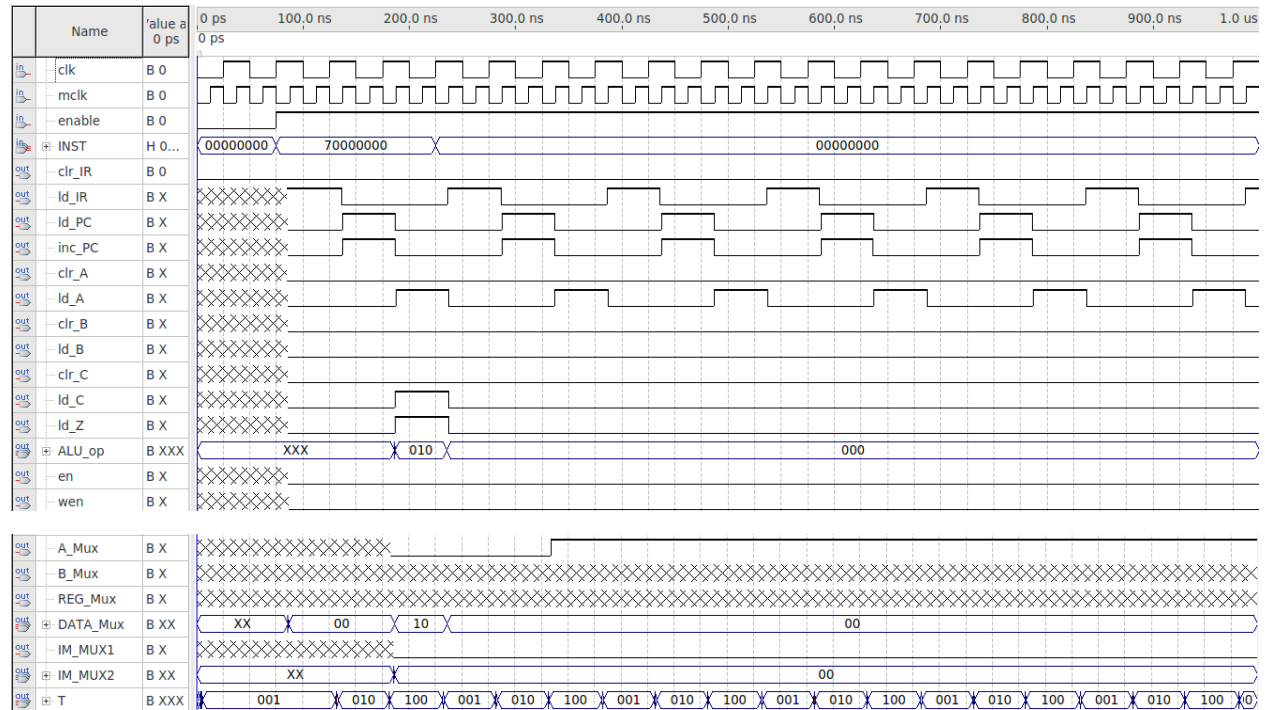


Figure 7: Timing simulation waveform of the Control Unit for **ADD** instruction.

From **figure 7**, we see that **T** signal starts from value **001**, that is, state **T0**. We know that during **T0**, and **T1** (corresponds to **T = 010**) the control signals will only account for loading Instruction Register (IR) with the value of the Instruction Memory (IM), which is pointed to by the current value of the Program Counter (PC); and subsequent increment of PC by 4, along with predecode operations applicable for memory IO operations. Thus, in our design, we expect the control signal values during **T0**, and **T1** states to correspond to the values that will successfully transfer Instruction Word from Instruction Memory to the Instruction Register, next, incrementing the current Program Counter value, and then perform the predecoding. In **figure 7**, we see that during **T=001** (state **T0**), our **ld\_IR** signal goes to high (means we loaded the Instruction Register from Instruction Memory), during **T=010**(state **T1**), we have both **ld\_PC**, and **inc\_PC** signal both to be high (indicating loading and incrementing the current Program Counter so that we can fetch the next instruction). Based on this observation we conclude that we have met the criteria for the fetch and predecode states (state **T0**, and **T1**) illustrated in **figure 2**. Please note that the predecoding step verification is not necessary in for **R** type instructions (because we are not performing memory IO operation, rather we are performing addition on two operands stored in registers).

Now if we observe all the output values during **T = 100** (state **T2**, that is, the execution state) from **figure 7**, we can conclude that we indeed have achieved the correct control signal values stated in **table 1**, and thus have set correct control signal values for individual components illustrated in **figure 4**.

JMP

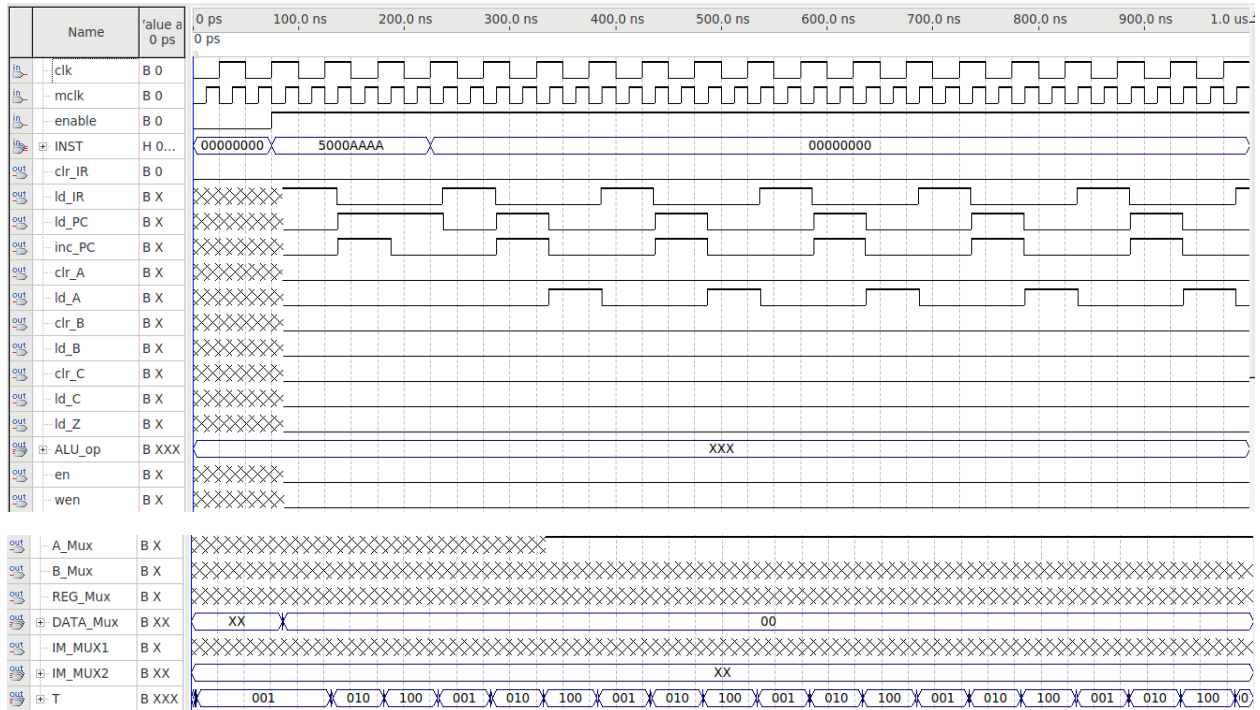


Figure 8: Timing simulation waveform of the Control Unit for **JMP** instruction.

From **figure 8**, we see that **T** signal starts from value **001**, that is, state **T0**. We know that during **T0**, and **T1** (corresponds to **T = 010**) the control signals will only account for loading Instruction Register (IR) with the value of the Instruction Memory (IM), which is pointed to by the current value of the Program Counter (PC); and subsequent increment of PC by 4, along with predecode operations only applicable for memory IO operations (JMP is not an memory IO operation). Thus, in our design, we expect the control signal values during **T0**, and **T1** states to correspond to the values that will successfully transfer Instruction Word from Instruction Memory to the Instruction Register, next, incrementing the current Program Counter value, and then perform the predecoding. In **figure 8**, we see that during **T=001** (state **T0**), our **ld\_IR** signal goes to high (means we loaded the Instruction Register from Instruction Memory), during **T=010**(state **T1**), we have both **ld\_PC**, and **inc\_PC** signal both to be high (indicating loading and incrementing the current Program Counter so that we can fetch the next instruction). Based on this observation we conclude that we have met the criteria for states **T0**, and **T1**, as illustrated in **figure 2**. Please note that the predecoding step verification is not necessary in for **J** type instructions (because we are not performing memory IO operation, rather we are about to load our desired address to the Program Counter (PC)).

Now if we observe all the output values during **T = 100** (state **T2**, that is, the execution state) from **figure 8**, we can conclude that we indeed have achieved the correct control signal values stated in **table 1**, and thus have set correct control signal values for individual components illustrated in **figure 4**. It is important to point out that we indeed have loaded our Program Counter (PC) by setting/keeping **ld\_PC** signal to high during the execution state, **T2** (Corresponds to **T = 100**), thus successfully achieved the requirement of **JMP** instruction.

## STB

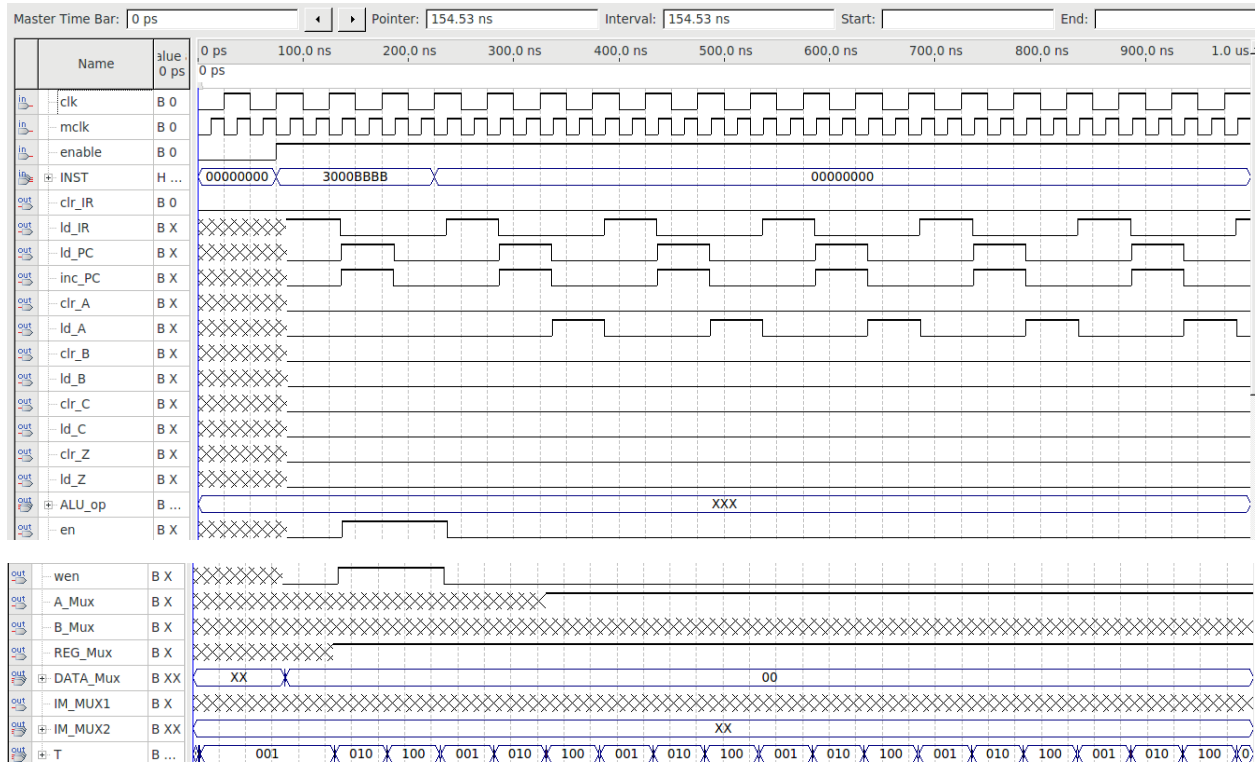


Figure 9: Timing simulation waveform of the Control Unit for **STB** instruction.

From **figure 9**, we see that **T** signal starts from value **001**, that is, state **T0**. We know that during **T0**, and **T1** (corresponds to **T = 010**) the control signals will only account for loading Instruction Register (IR) with the value of the Instruction Memory (IM), which is pointed to by the current value of the Program Counter (PC); and subsequent increment of PC by 4, along with predecode operations applicable for memory IO operations (**STB** is a memory IO operation because we want to store the content of register **B** into some data-memory address, thus, we must ensure that **wen** and **en** signals are set to high before the execution state, **T2**, or when **T=100**). Thus, in our design, we expect the control signal values during **T0**, and **T1** states to correspond to the values that will successfully transfer Instruction Word from Instruction Memory to the Instruction Register, next, incrementing the current Program Counter value, and then perform the predecoding (to account for the ). In **figure 9**, we see that during **T=001** (state **T0**), our **ld\_IR** signal goes to high (means we loaded the Instruction Register from Instruction Memory), during **T=010**(state **T1**), we have both **ld\_PC**, and **inc\_PC** signal both to be high (indicating loading and incrementing the current Program Counter so that we can fetch the next instruction), and both **en**, and **wen** = 1 (means we will perform a writing operation on the data-memory), and **Reg\_Mux** = 1 (meaning we will store Register **B**'s content in a specified address of the data-memory). Based on this observation we conclude that we have met the criteria for state **T0**, and **T1** illustrated in the ASM state diagram presented **figure 2**.

Now if we observe all the output values during  $T = 100$  (state **T2**, that is, the execution state) from **figure 9**, we can conclude that we indeed have achieved the correct control signal values stated in **table 1**, and thus have set correct control signal values for individual components illustrated in **figure 4**.

## Reference

1. *COE 608 Lab 5 – CPU Control Unit Design*. D2L.
2. *COE 608 Lab 5 Tutorial – CPU Control Unit Design*. D2L.
3. “Setup and hold time definition”,  
[https://www.idconline.com/technical\\_references/pdfs/electronic\\_engineering/Setup\\_and\\_hold\\_time\\_definition.pdf](https://www.idconline.com/technical_references/pdfs/electronic_engineering/Setup_and_hold_time_definition.pdf)
4. *COE 608- CPU Specifications*. D2L

## Appendix

The complete VHDL code for the designed Control Unit is presented below:

```
library ieee;
use ieee.std_logic_1164.ALL;

ENTITY Control IS
    PORT(
        clk, mclk : IN STD_LOGIC;
        enable : IN STD_LOGIC;
        statusC, statusZ : IN STD_LOGIC;
        INST: IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        A_Mux, B_Mux : OUT STD_LOGIC;
        IM_MUX1, REG_Mux : OUT STD_LOGIC;
        IM_MUX2, DATA_Mux : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        ALU_op : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
        inc_PC, ld_PC : OUT STD_LOGIC;
        clr_IR: OUT STD_LOGIC;
        ld_IR : OUT STD_LOGIC;
        clr_A,clr_B, clr_C, clr_Z: OUT STD_LOGIC;
        ld_A, ld_B, ld_C, ld_Z : OUT STD_LOGIC;
        T : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
        wen, en : OUT STD_LOGIC
    );
END Control;

ARCHITECTURE description OF Control IS
    TYPE STATETYPE IS (state_0, state_1, state_2);
    SIGNAL present_state: STATETYPE;
    SIGNAL Instruction_sig: STD_LOGIC_VECTOR(31 DOWNTO 0);
    SIGNAL Instruction_sig2: STD_LOGIC_VECTOR(31 DOWNTO 24);
BEGIN
    Instruction_sig <= INST(31 DOWNTO 28);
    Instruction_sig2 <= INST(31 DOWNTO 24);

    -----OPERATION DECODER-----

    PROCESS(present_state, INST, statusC, statusZ, enable, Instruction_sig, Instruction_sig2)
```

```

BEGIN

    if enable = '1' then

        if present_state = state_0 then

            DATA_Mux <= "00"; --Fetch address of next instruction

            clr_IR <= '0';

            ld_IR <= '1';

            ld_PC <= '0';

            inc_PC <= '0';

            clr_A <= '0';

            ld_A <= '0';

            ld_B <= '0';

            clr_B <= '0';

            clr_C <= '0';

            ld_C <= '0';

            clr_Z <= '0';

            ld_Z <= '0';

            en <= '0';

            wen <= '0';


        elsif present_state = state_1 then

            clr_IR <= '0'; --INCREMENT PC COUNTER But how??

            ld_IR <= '0';

            ld_PC <= '1';

            inc_PC <= '1';

            clr_A <= '0';

            ld_A <= '0';

            ld_B <= '0';

            clr_B <= '0';

            clr_C <= '0';

            ld_C <= '0';

            clr_Z <= '0';

            ld_Z <= '0';

            en <= '0';

            wen <= '0';


            if Instruction_sig = "0010" then --STA

```



```

    clr_IR <= '0';
    ld_IR <= '0';
    ld_PC <= '1';
    inc_PC <= '1';
    clr_A <= '0';
    ld_A <= '0';
    ld_B <= '0';
    clr_B <= '0';
    clr_C <= '0';
    ld_C <= '0';
    clr_Z <= '0';
    ld_Z <= '0';
    REG_Mux <= '0';
    DATA_Mux <= "00";
    en <= '1';
    wen <= '1';

elsif Instruction_sig = "0011" then --STB
    clr_IR <= '0';
    ld_Z <= '0';
    ld_IR <= '0';
    ld_PC <= '1';
    inc_PC <= '1';
    CLR_A <= '0';
    ld_A <= '0';
    ld_B <= '0';
    clr_B <= '0';
    clr_C <= '0';
    ld_C <= '0';
    clr_Z <= '0';
    ld_Z <= '0';
    REG_Mux <= '1';
    DATA_Mux <= "00";
    en <= '1';
    wen <= '1';

```

```

elsif Instruction_sig = "1001" then --LDA
    clr_IR <= '0';
    ld_IR <= '0';
    ld_PC <= '1';
    inc_PC <= '1';
    clr_A <= '0';
    ld_A <= '1';
    ld_B <= '0';
    clr_B <= '0';
    clr_C <= '0';
    ld_C <= '0';
    clr_Z <= '0';
    ld_Z <= '0';
    A_Mux <= '0';
    DATA_Mux <= "01";
    en <= '1';
    wen <= '0';

elsif Instruction_sig = "1010" then --LDB
    clr_IR <= '0';
    ld_IR <= '0';
    ld_PC <= '1';
    inc_PC <= '1';
    clr_A <= '0';
    ld_A <= '0';
    ld_B <= '1';
    clr_B <= '0';
    clr_C <= '0';
    ld_C <= '0';
    clr_Z <= '0';
    ld_Z <= '0';
    B_Mux <= '0';
    DATA_Mux <= "01";
    en <= '1';
    wen <= '0';

end if; --END IF FOR LOAD STORE IN STAGE 1

```

```

elsif present_state = state_2 then

    if Instruction_sig = "0101" then --JUMP

        clr_IR <= '0';

        ld_IR <= '0';

        ld_PC <= '1';

        inc_PC <= '0';

        clr_A <= '0';

        ld_A <= '0';

        ld_B <= '0';

        clr_B <= '0';

        clr_C <= '0';

        ld_C <= '0';

        clr_Z <= '0';

        ld_Z <= '0';

    elsif Instruction_sig = "0110" then --BEQ

        clr_IR <= '0';

        ld_IR <= '0';

        ld_PC <= '1';

        inc_PC <= '0';

        clr_A <= '0';

        ld_A <= '0';

        ld_B <= '0';

        clr_B <= '0';

        clr_C <= '0';

        ld_C <= '0';

        clr_Z <= '0';

        ld_Z <= '0';

    elsif Instruction_sig = "1000" then --BNE

        clr_IR <= '0';

        ld_IR <= '0';

        ld_PC <= '1';

        inc_PC <= '0';

```

```

        clr_A <= '0';
        ld_A <= '0';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '0';
        clr_Z <= '0';
        ld_Z <= '0';

elsif Instruction_sig = "1001" then --LDA
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '0';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '0';
        clr_Z <= '0';
        ld_Z <= '0';
        A_Mux <= '0';
        DATA_Mux <= "01";
        EN <= '1';
        WEN <= '0';

elsif Instruction_sig = "1010" then --LDB
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '0';
        ld_B <= '0';
        clr_B <= '0';

```

```

        clr_C <= '0';
        ld_C <= '0';
        clr_Z <= '0';
        ld_Z <= '0';
        B_Mux <= '0';
        DATA_Mux <= "01";
        EN <= '1';
        WEN <= '0';

    elsif Instruction_sig = "0010" then --STA
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '0';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '0';
        clr_Z <= '0';
        ld_Z <= '0';
        REG_Mux <= '0';
        DATA_Mux <= "00";
        EN <= '1';
        WEN <= '1';

    elsif Instruction_sig = "0011" then --STB
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '0';
        ld_B <= '0';
        clr_B <= '0';

```

```

        clr_C <= '0';
        ld_C <= '0';
        clr_Z <= '0';
        ld_Z <= '0';
        REG_Mux <= '1';
        DATA_Mux <= "00";
        EN <= '1';
        WEN <= '1';

    elsif Instruction_sig = "0000" then --LDAI
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '1';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '0';
        clr_Z <= '0';
        ld_Z <= '0';
        A_Mux <= '1';

    elsif Instruction_sig = "0001" then --LDBI
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '0';
        ld_B <= '1';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '0';
        clr_Z <= '0';
        ld_Z <= '0';
        B_Mux <= '1';

```

```

elsif Instruction_sig = "0100" then --LUI

    clr_IR <= '0';
    ld_IR <= '0';
    ld_PC <= '0';
    inc_PC <= '0';
    clr_A <= '0';
    ld_A <= '1';
    ld_B <= '0';
    clr_B <= '1';
    clr_C <= '0';
    ld_C <= '0';
    clr_Z <= '0';
    ld_Z <= '0';
    ALU_op <= "001";
    A_Mux <= '0';
    DATA_Mux <= "10";
    IM_MUX1 <= '1';

elsif Instruction_sig2 = "01111001" then --ANDI

    clr_IR <= '0';
    ld_IR <= '0';
    ld_PC <= '0';
    inc_PC <= '0';
    clr_A <= '0';
    ld_A <= '1';
    ld_B <= '0';
    clr_B <= '0';
    clr_C <= '0';
    ld_C <= '1';
    clr_Z <= '0';
    ld_Z <= '1';
    ALU_op <= "000";
    A_Mux <= '0';
    DATA_Mux <= "10";
    IM_MUX1 <= '0';
    IM_MUX2 <= "01";

```

```

elsif Instruction_sig2 = "01111110" then --DECA

    clr_IR <= '0';

    ld_IR <= '0';

    ld_PC <= '0';

    inc_PC <= '0';

    clr_A <= '0';

    ld_A <= '1';

    ld_B <= '0';

    clr_B <= '0';

    clr_C <= '0';

    ld_C <= '1';

    clr_Z <= '0';

    ld_Z <= '1';

    ALU_op <= "110";

    A_Mux <= '0';

    DATA_Mux <= "10";

    IM_MUX1 <= '0';

    IM_MUX2 <= "10";

elsif Instruction_sig2 = "01110000" then --ADD

    clr_IR <= '0';

    ld_IR <= '0';

    ld_PC <= '0';

    inc_PC <= '0';

    clr_A <= '0';

    ld_A <= '1';

    ld_B <= '0';

    clr_B <= '0';

    clr_C <= '0';

    ld_C <= '1';

    clr_Z <= '0';

    ld_Z <= '1';

    ALU_op <= "010";

    A_Mux <= '0';

    DATA_Mux <= "10";

    IM_MUX1 <= '0';

    IM_MUX2 <= "00";

```



```

elsif Instruction_sig2 = "01110010" then --SUB
    clr_IR <= '0';
    ld_IR <= '0';
    ld_PC <= '0';
    inc_PC <= '0';
    clr_A <= '0';
    ld_A <= '1';
    ld_B <= '0';
    clr_B <= '0';
    clr_C <= '0';
    ld_C <= '1';
    clr_Z <= '0';
    ld_Z <= '1';
    ALU_op <= "110";
    A_Mux <= '0';
    DATA_Mux <= "10";
    IM_MUX1 <= '0';
    IM_MUX2 <= "00";
elsif Instruction_sig2 = "01110011" then --INCA
    clr_IR <= '0';
    ld_IR <= '0';
    ld_PC <= '0';
    inc_PC <= '0';
    clr_A <= '0';
    ld_A <= '1';
    ld_B <= '0';
    clr_B <= '0';
    clr_C <= '0';
    ld_C <= '1';
    clr_Z <= '0';
    ld_Z <= '1';
    ALU_op <= "010";
    A_Mux <= '0';
    DATA_Mux <= "10";
    IM_MUX1 <= '0';
    IM_MUX2 <= "10";

```

```

elsif Instruction_sig2 = "01111011" then --AND

    clr_IR <= '0';

    ld_IR <= '0';

    ld_PC <= '0';

    inc_PC <= '0';

    clr_A <= '0';

    ld_A <= '1';

    ld_B <= '0';

    clr_B <= '0';

    clr_C <= '0';

    ld_C <= '1';

    clr_Z <= '0';

    ld_Z <= '1';

    ALU_op <= "000";

    A_Mux <= '0';

    DATA_Mux <= "10";

    IM_MUX1 <= '0';

    IM_MUX2 <= "00";

elsif Instruction_sig2 = "01110001" then --ADDI

    clr_IR <= '0';

    ld_IR <= '0';

    ld_PC <= '0';

    inc_PC <= '0';

    clr_A <= '0';

    ld_A <= '1';

    ld_B <= '0';

    clr_B <= '0';

    clr_C <= '0';

    ld_C <= '1';

    clr_Z <= '0';

    ld_Z <= '1';

    ALU_op <= "010";

    A_Mux <= '0';

    DATA_Mux <= "10";

    IM_MUX1 <= '0';

    IM_MUX2 <= "01";

```

```

elsif Instruction_sig2 = "01111101" then --ORI
    clr_IR <= '0';
    ld_IR <= '0';
    ld_PC <= '0';
    inc_PC <= '0';
    clr_A <= '0';
    ld_A <= '1';
    ld_B <= '0';
    clr_B <= '0';
    clr_C <= '0';
    ld_C <= '1';
    clr_Z <= '0';
    ld_Z <= '1';
    ALU_op <= "001";
    A_Mux <= '0';
    DATA_Mux <= "10";
    IM_MUX1 <= '0';
    IM_MUX2 <= "01";
elsif Instruction_sig2 = "01110100" then --ROL
    clr_IR <= '0';
    ld_IR <= '0';
    ld_PC <= '0';
    inc_PC <= '0';
    clr_A <= '0';
    ld_A <= '1';
    ld_B <= '0';
    clr_B <= '0';
    clr_C <= '0';
    ld_C <= '1';
    clr_Z <= '0';
    ld_Z <= '1';
    ALU_op <= "100";
    A_Mux <= '0';
    DATA_Mux <= "10";
    IM_MUX1 <= '0';
elsif Instruction_sig2 = "01111111" then --ROR

```

```

        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '1';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '1';
        clr_Z <= '0';
        ld_Z <= '1';
        ALU_op <= "101";
        A_Mux <= '0';
        DATA_Mux <= "10";
        IM_MUX1 <= '0';
    elsif Instruction_sig2 = "01110101" then --CLR_A
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '1';
        ld_A <= '0';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '0';
        clr_Z <= '0';
        ld_Z <= '0';
    elsif Instruction_sig2 = "01110110" then --CLR_B
        clr_IR <= '0';
        ld_IR <= '0';
        ld_PC <= '0';
        inc_PC <= '0';
        clr_A <= '0';
        ld_A <= '0';

```

```

ld_B <= '0';
clr_B <= '1';
clr_C <= '0';
ld_C <= '0';
clr_Z <= '0';
ld_Z <= '0';
elsif Instruction_sig2 = "01110111" then --CLR_C
    clr_IR <= '0';
    ld_IR <= '0';
    ld_PC <= '0';
    inc_PC <= '0';
    clr_A <= '0';
    ld_A <= '0';
    ld_B <= '0';
    clr_B <= '0';
    clr_C <= '1';
    ld_C <= '0';
    clr_Z <= '0';
    ld_Z <= '0';
elsif Instruction_sig2 = "01111000" then --CLR_z
    clr_IR <= '0';
    ld_IR <= '0';
    ld_PC <= '0';
    inc_PC <= '0';
    clr_A <= '0';
    ld_A <= '0';
    ld_B <= '0';
    clr_B <= '0';
    clr_C <= '0';
    ld_C <= '0';
    clr_Z <= '1';
    ld_Z <= '0';
elsif Instruction_sig2 = "01111010" then --TSTZ
    if(statusZ = '1') then
        clr_IR <= '0';--INCREMENT PC COUNTER
        ld_IR <= '0';

```

```

        ld_PC <= '1';
        inc_PC <= '1';
        clr_A <= '0';
        ld_A <= '0';
        ld_B <= '0';
        clr_B <= '0';
        clr_C <= '0';
        ld_C <= '0';
        clr_Z <= '0';
        ld_Z <= '0';
    end if;

    elsif Instruction_sig2 = "01111100" then --TSTC
        if(statusC = '1') then
            clr_IR <= '0';--INCREMENT PC COUNTER
            ld_IR <= '0';
            ld_PC <= '1';
            inc_PC <= '1';
            clr_A <= '0';
            ld_A <= '0';
            ld_B <= '0';
            clr_B <= '0';
            clr_C <= '0';
            ld_C <= '0';
            clr_Z <= '0';
            ld_Z <= '0';
        end if;
    end if; --For stage 2 Ops
end if;

end if;--For Enable
END PROCESS;

-----STATE MACHINE-----

PROCESS (clk, enable)
begin
    if enable = '1' then
        if rising_edge (clk) then
            if present_state = state_0 then present_state <= state_1;

```

```

        elsif present_state = state_1 then present_state <= state_2;
        else present_state <= state_0;
        end if;
    end if;

    else present_state <= state_0;
    end if;
END PROCESS;

WITH present_state select
T <= "001" when state_0,
    "010" when state_1,
    "100" when state_2,
    "001" when others;
END description;

```