Ryerson University

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

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

Assignment/Lab Number:	4b
Assignment/Lab Title:	Data-Path Design

Submission Date:	March 13, 2021
Due Date:	March 12, 2021

Student LAST Name	**Student FIRST Name**	**Student Number**	**Section**	**Signature***
ALAM	ABRAR	500725366	03	Abrar Alam

Table of Contents

Objective

In this lab, a 32-bit processor's data-path was implemented primarily using already implemented and unit-tested components from previous labs such as, Program-Counter (PC), Arithmetic Logic Unit (ALU), 32-bit registers, 256x32 bit data memory modules, full adders of various bit size, and multiplexers (MUX). Besides, some new modules such as, Lower Zero Extenders (LZE), Upper Zero Extenders (UZE), Reducers (RED), and 4-to-1 MUX were implemented to be integrated within the data path to be devised. Upon finalizing all the components' implementations and testing, the entire data-path module was thoroughly tested for correctness of instruction execution, and result, according to the CPU Specification document. However, control signal, and Instruction Memory, M[INST] will be implemented in Lab 6. **Figure 1** shows the overall block diagram of the data-path that was implemented in this lab.

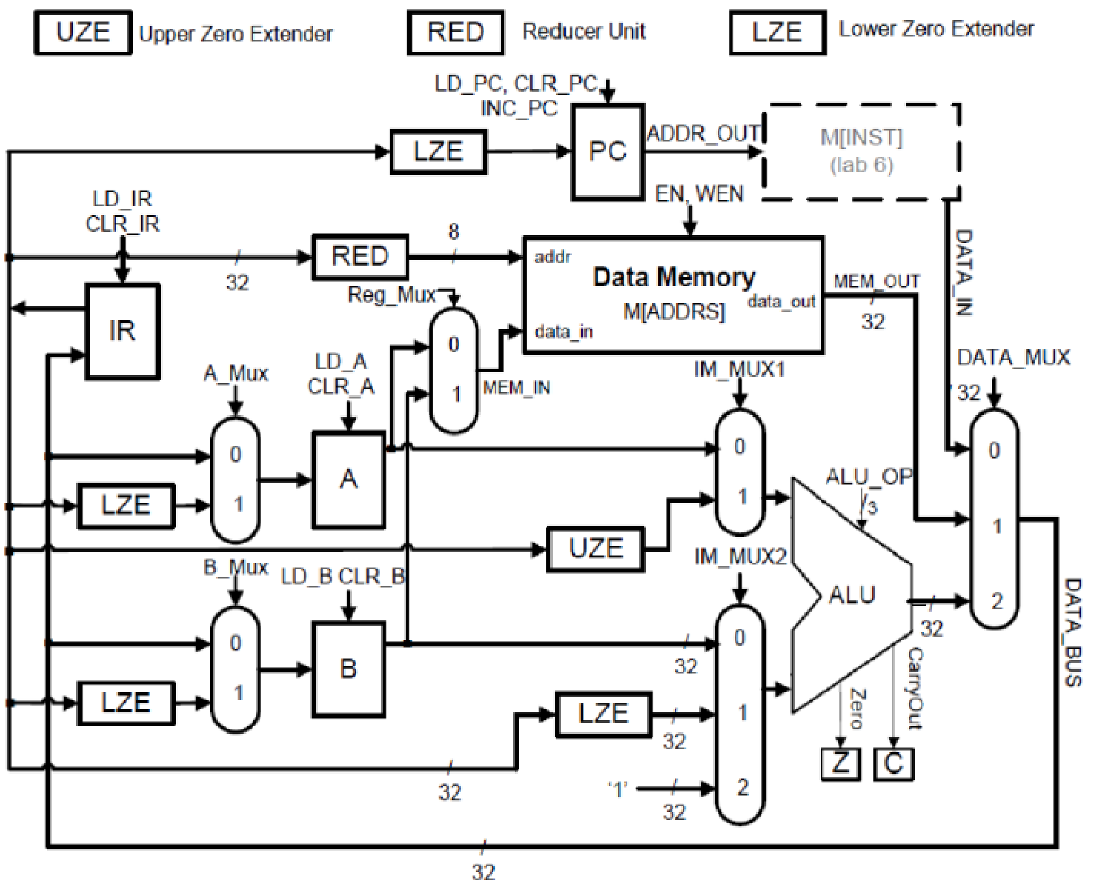


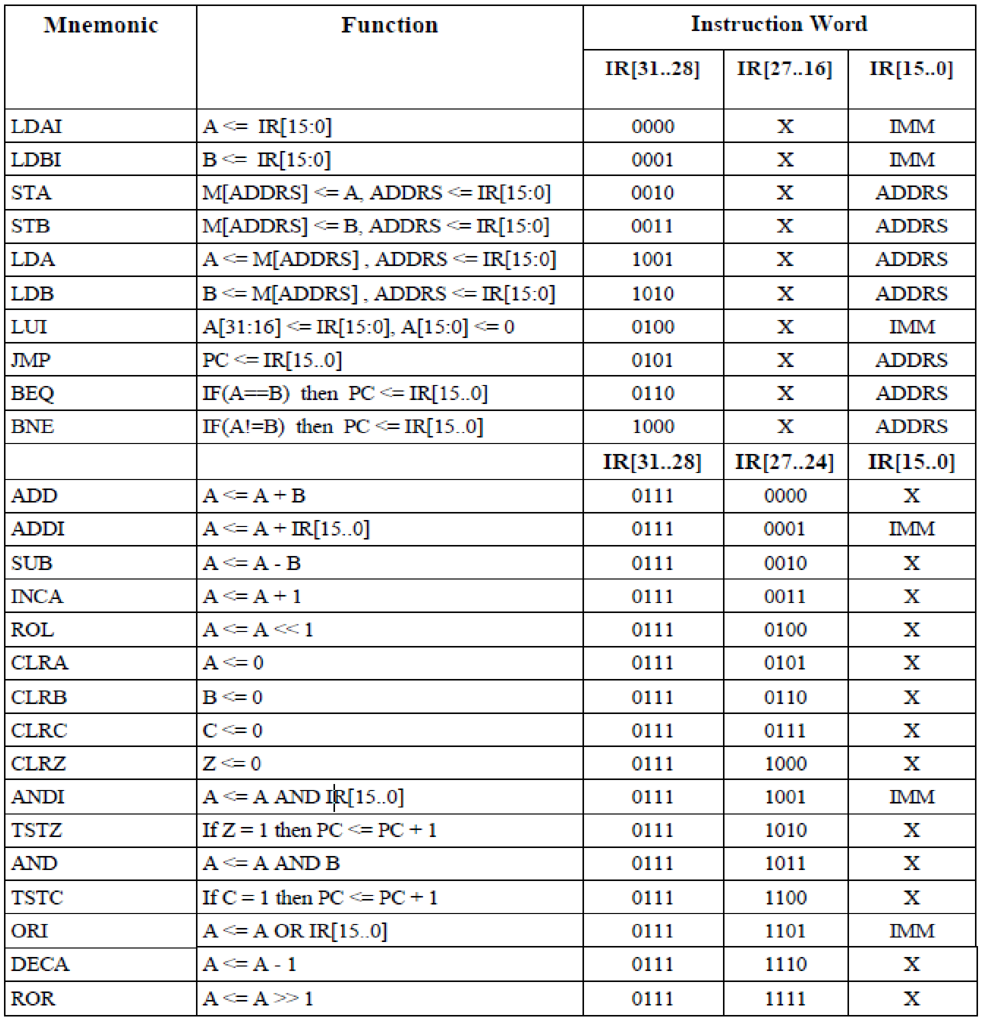
Figure 1: Block diagram of the data-path

Design and implementation

As seen in **figure 1**, to implement the complete data-path, extra components such as, LZE, UZE, RED, and a 4-to-1 MUX was needed constituent components, along with already tested components from previous labs. **Please note that M[Inst], or instruction memory was not implemented in this lab, rather its functionalities were simulate using dedicated *DATA_IN* input (see figure 1).** All the registers' outputs were declared as output signals to facilitate system testing, observation of internal register status.

As seen from **figure 1**, various control signals were used in the implemented data-path, such as, clear and load inputs for all Registers, PC, and ALU, selector signals for all MUXes. Besides, two different types of clock signals were used, one is being the data-path clock, Clk withclock period was 40 ns, while the other one being the data memory clock, mClk with a clock period of 20 ns (not shown in **figure 1**). In order to ensure minimal latency during the reading and writing operations, the frequency the mClk is double that of the data-path clock, Clk. This also ensures reading, and writing data within a single pipeline stage.

Table 1 shows the supported and implanted instructions of the data-path. Please also note that this table specifies details of specific instruction format, which is to be implemented in much detail in later labs. Please note that during the testing phase of this lab, the correctness of the data-path was tested based mainly on the **Function** column of **table 1**.



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

Table 1: Supported instructions, their format, and mechanism.

Please note that mapping data-in to the instruction memory input was not among the goals of this lab, thus only results of various operations were verified using functional and timing waveforms. For the LDA. LDB, STA, and STB operations, the IR must need to be latched (see **figure 1**). The latched IR[7..0] address can be sent to the data memory simultaneously with the data from/to register A or B. This also applies to other IMM and ADDRS based instructions. As for LDAI, and LDBI instructions, A and BH registers will be loaded respectively with immediate values found in IR passing through the LZE (see **figure 1**)

Extenders (Lower Zero, and Upper Zero Extenders)

AS for the extender type LZE, and UZE, these are used for loading addresses to the Instructions Register (IR), and immediate type operations such as ADDI, LDAI, etc. (see **table 1**). In the case of Load Upper Immediate (LUI), the lower 16-bits must be loaded into the upper 16-bits of the register A, and remining lower 16-bits of A will be cleared to 0. Upper Zero Extender (UZE) achieves this functionality. For some instruction such as, ADDI, ORI, etc. the lower 16 bits of the IR are used as the second operand, and thus make up the lower portion of the operand (bits 15..0). In all the cases, the remaining 16 bits are filled with 0's . This functionality was achieved by implementing Lower Zero Extenders (LZE). **Figure 2** shows the block diagram of the LZE implemented in this lab.

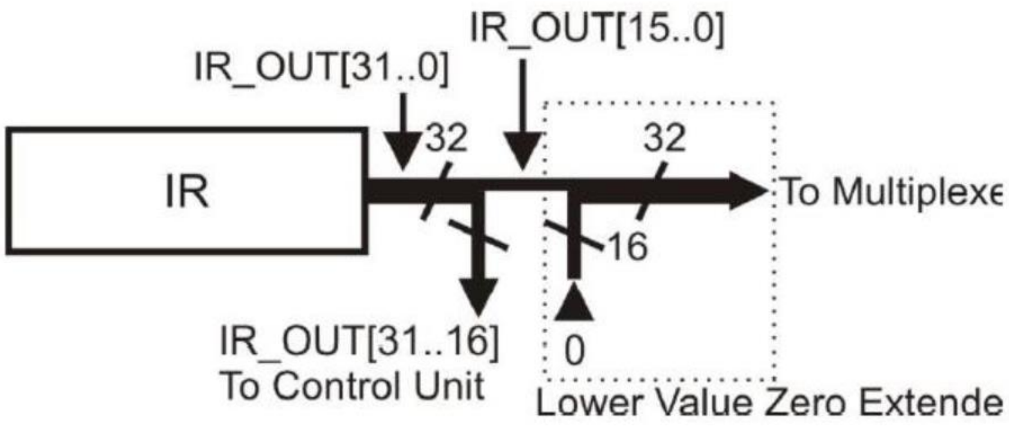


Figure 2: Block diagram of the Lower Zero Extender.

Reducer:

This unit simply takes 32-bits from the IR register and outputs the lower 8-bits of the IR. Next, these 8-bits then inputted into the address port of the data memory unit. Due to the addr_in port of the "data_mem" component is of type UNSIGNED, a conversion was made (since IR is of type STD_LOGIC_VECTOR) within its VHDL code.

Putting all these together: Data-Path Control Signal Table:

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 2: Data-Path Control Signals used to test correctness of various operations in this lab.

Results and Conclusions

Please refer to the *Appendix* section for the Data-Path module VHDL code.

Timing simulations of all the instructions

The correctness of the designed data-path was verified using both functional and timing simulation waveforms. Since the lab 4b manual only asks for submissions of the timing simulations, only timing simulations have been presented in this report. Please note that components reused from previous labs, were already unit tested thoroughly during those labs, and thus their functional and timing simulations are not presented again in this lab manual. Please also note that the actual VHDL code of the Data-Path is given in the *Appendix* section.

LDAI (Timing simulation)

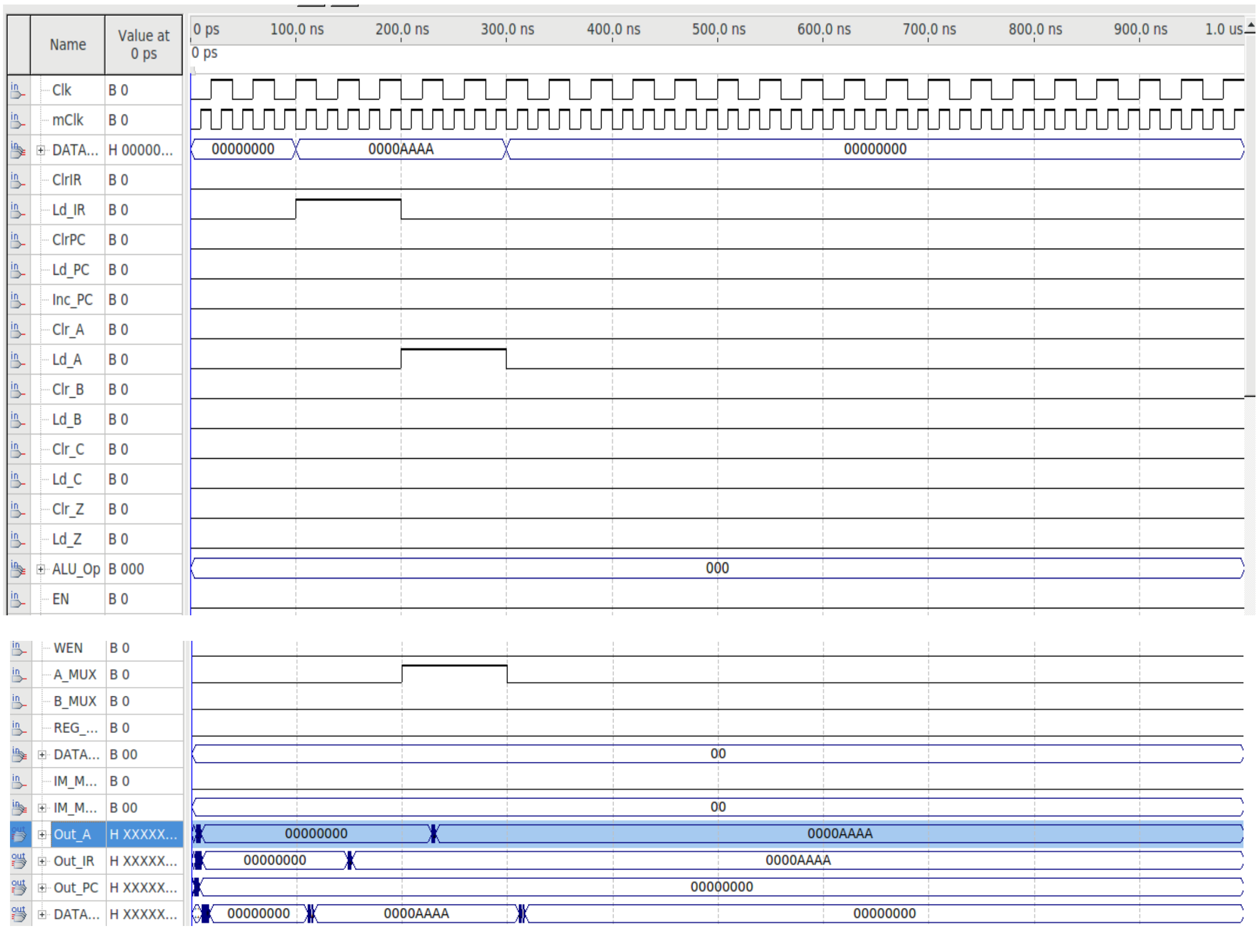


Figure 3: Timing waveform showing the correctness of the implementation of LDAI instruction

From **figure 3**, we see that our data input value's lower 16 bit is loaded into the Register A (with upper 16 bit of A has been padded with 0s), which is what was expected.

LDBI (Timing simulation)

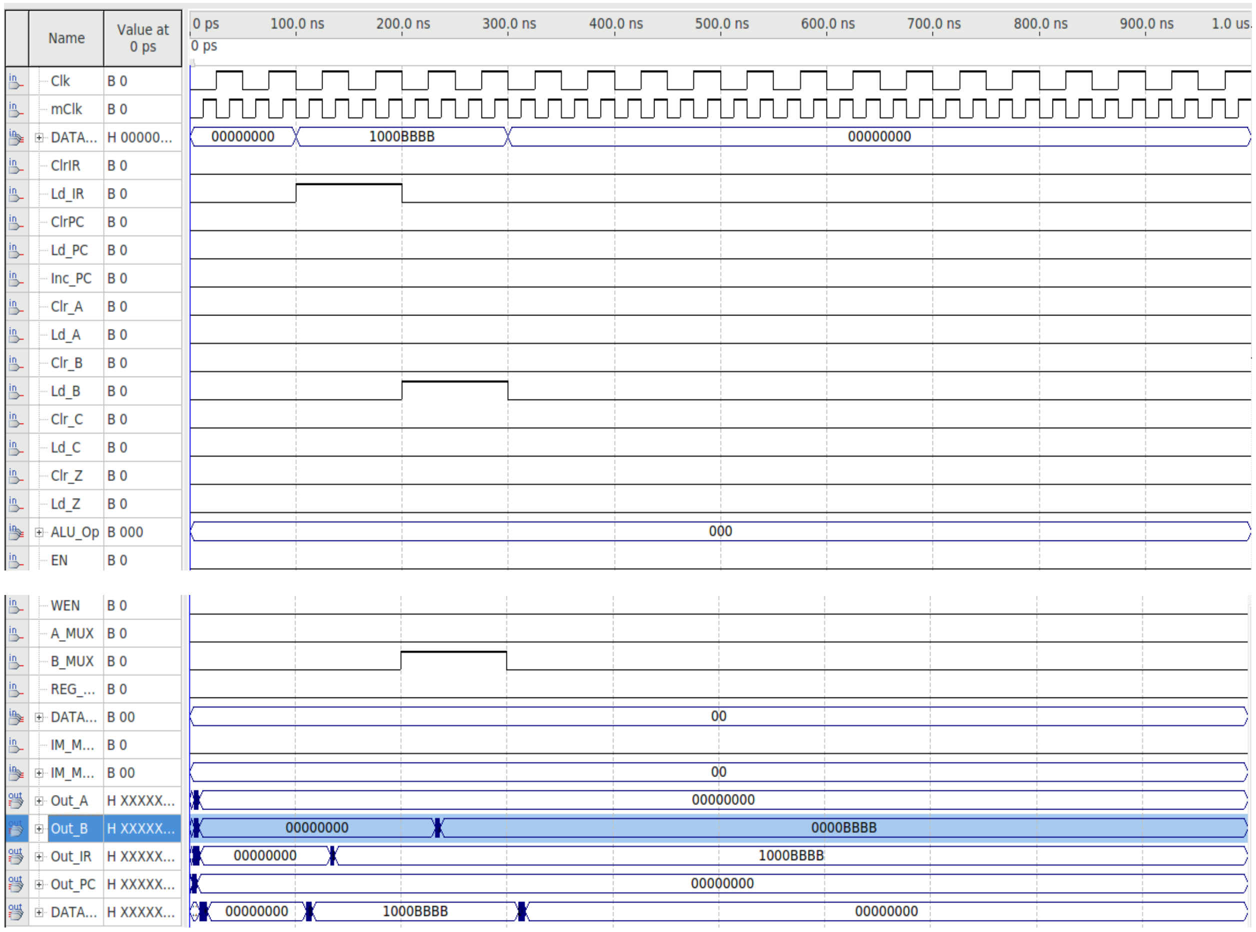


Figure 4: Timing waveform showing the correctness of the implementation of LDBI instruction

From **figure 4**, we see our data-in values lower 16-bits have been loaded into the lower 16-bits of the register B, while its upper 16-bits were set to 0s, which is what we expect.

STA (Timing simulation)

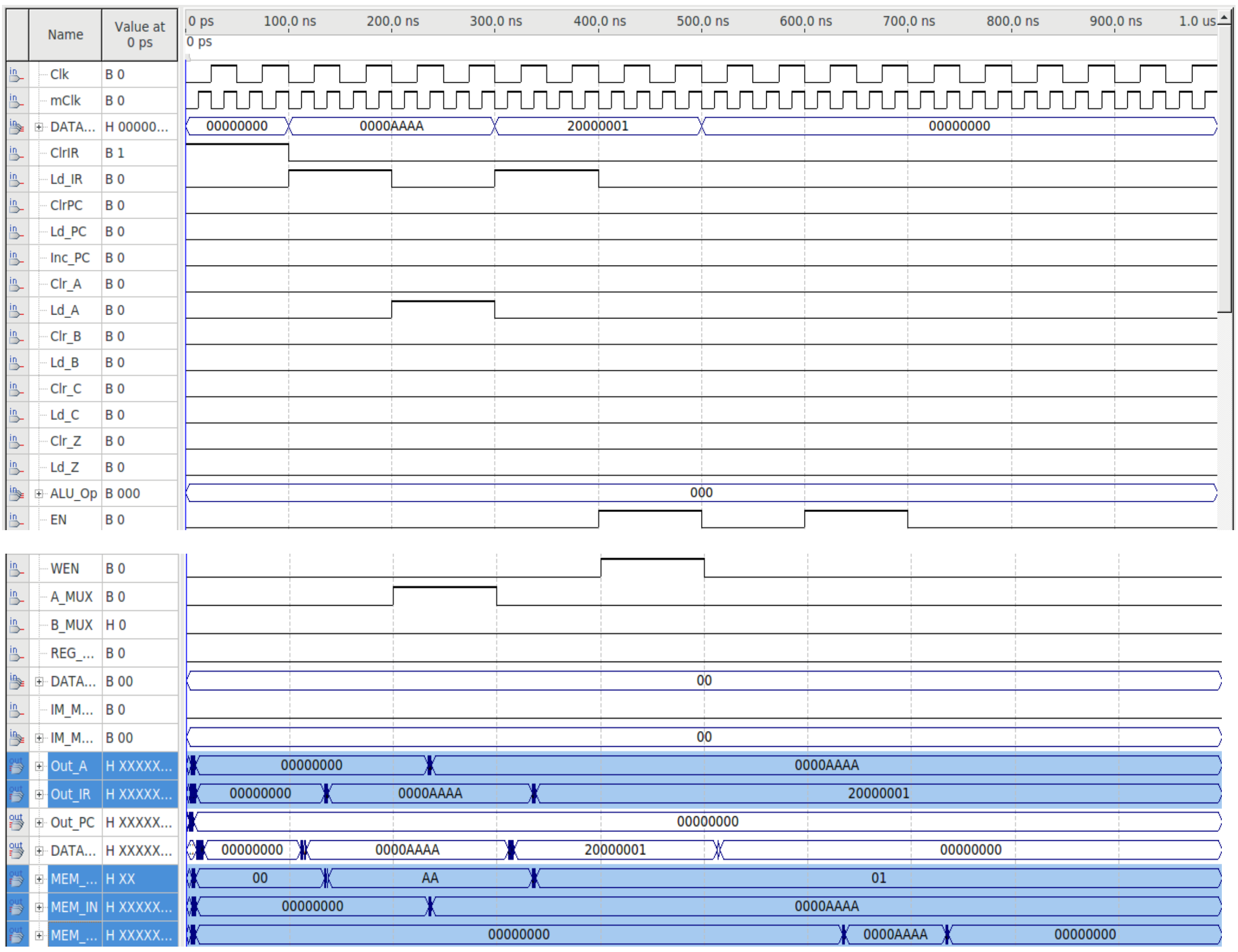


Figure 5: Timing waveform showing the correctness of the implementation of STA instruction

From **figure 5**, we see that first, IR's (Data_in) lower 16 bits are loaded into the 16-bit input called "MEM_ADDR" (3rd row from the bottom). We expect to store our register A content in this address of memory. The waveform above shows that we successfully achieved this behavior (for example, 0000AAAA is seen in the "MEM_OUT" signal). See **Table 1** for more details.

STB (Timing simulation)

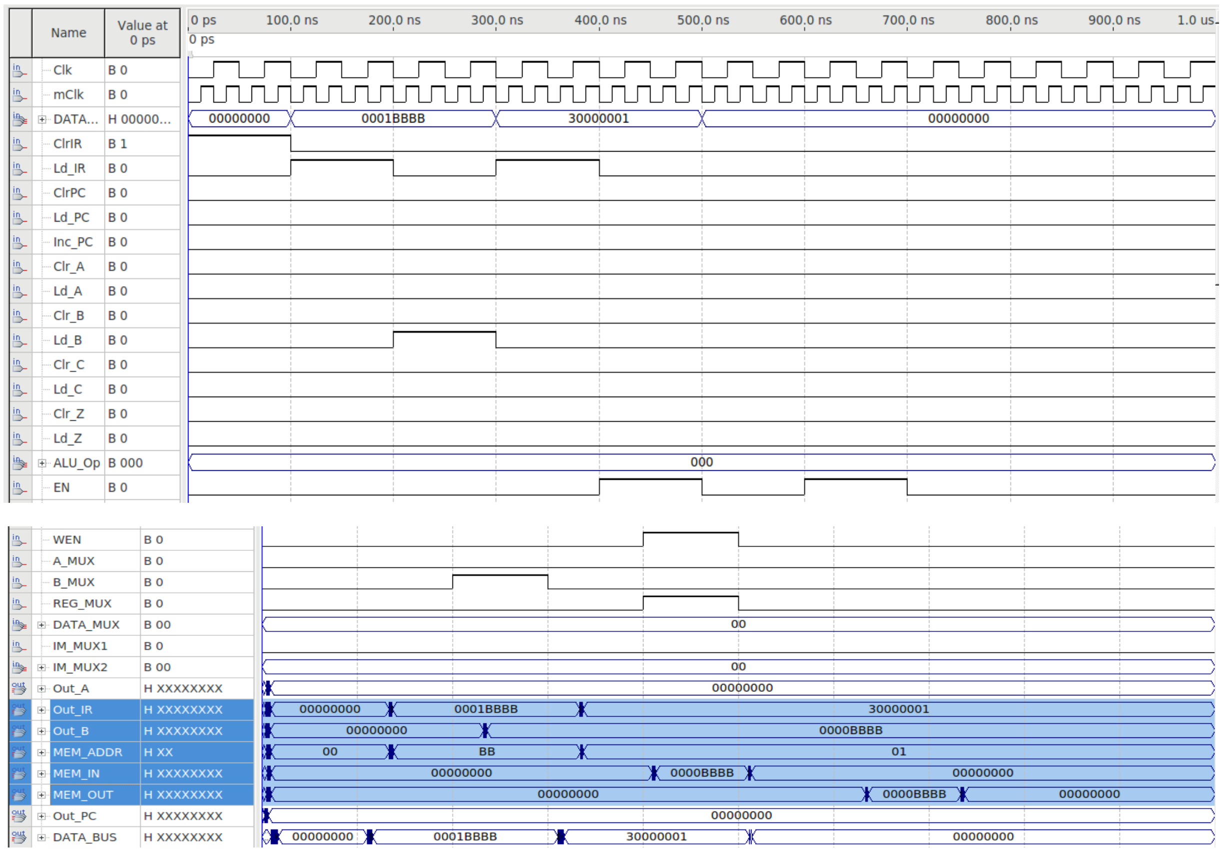


Figure 6: Timing waveform showing the correctness of the implementation of STB instruction

Figure 6 showing the correctness of STB instruction implementation (refer to **table** 1 for details). This instruction almost similar as STA in the previous page, only difference is that our source is register B rather than A.

LDA (Timing simulation)

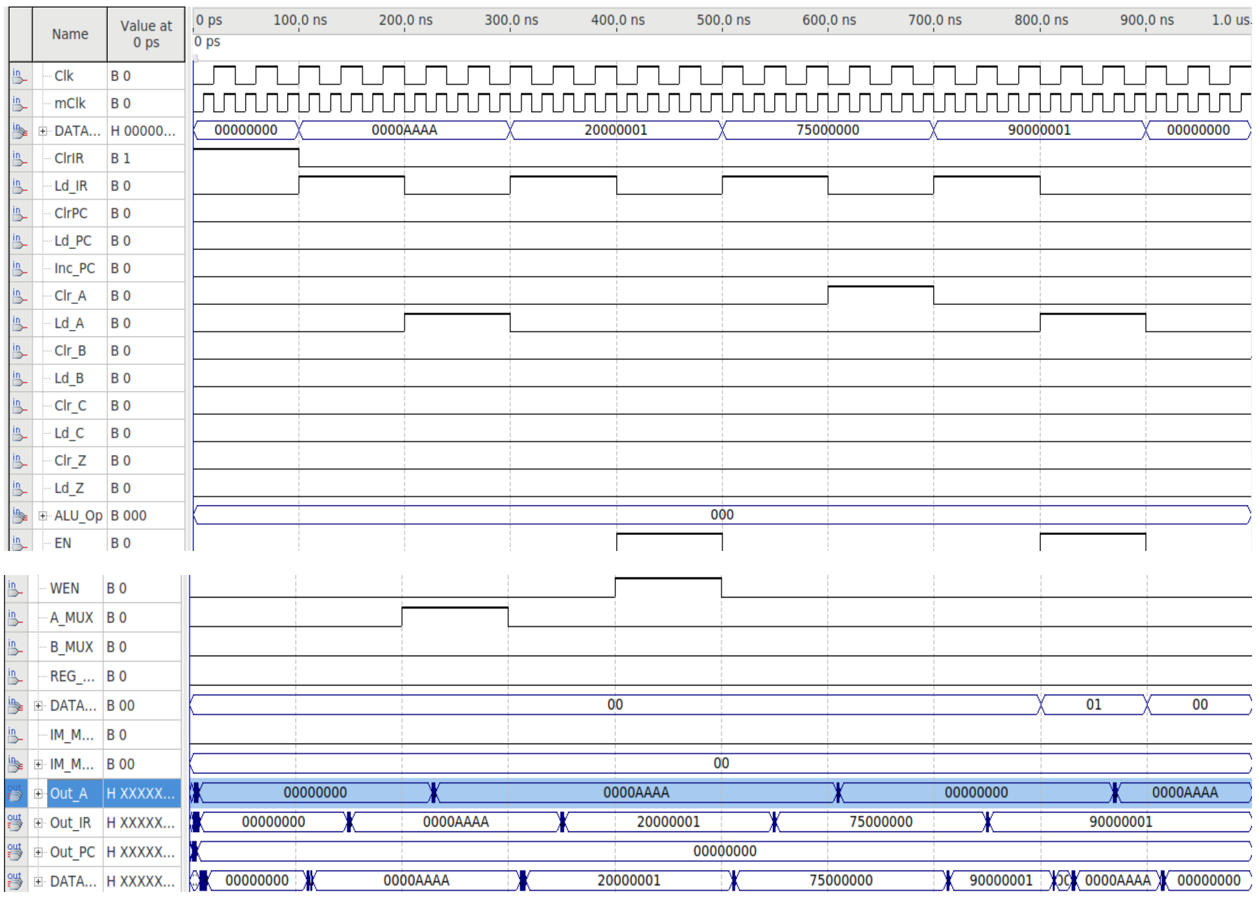


Figure 7: Timing waveform showing the correctness of the implementation of LDA instruction

Figure 7 shows the correctness of the implementation of this instruction. We observe, first we wrote 00000000, and 0000AAAA into some memory address, then we successfully load these values into register A successfully, which is what STA is supposed to achieve.

LDB (Timing simulation)

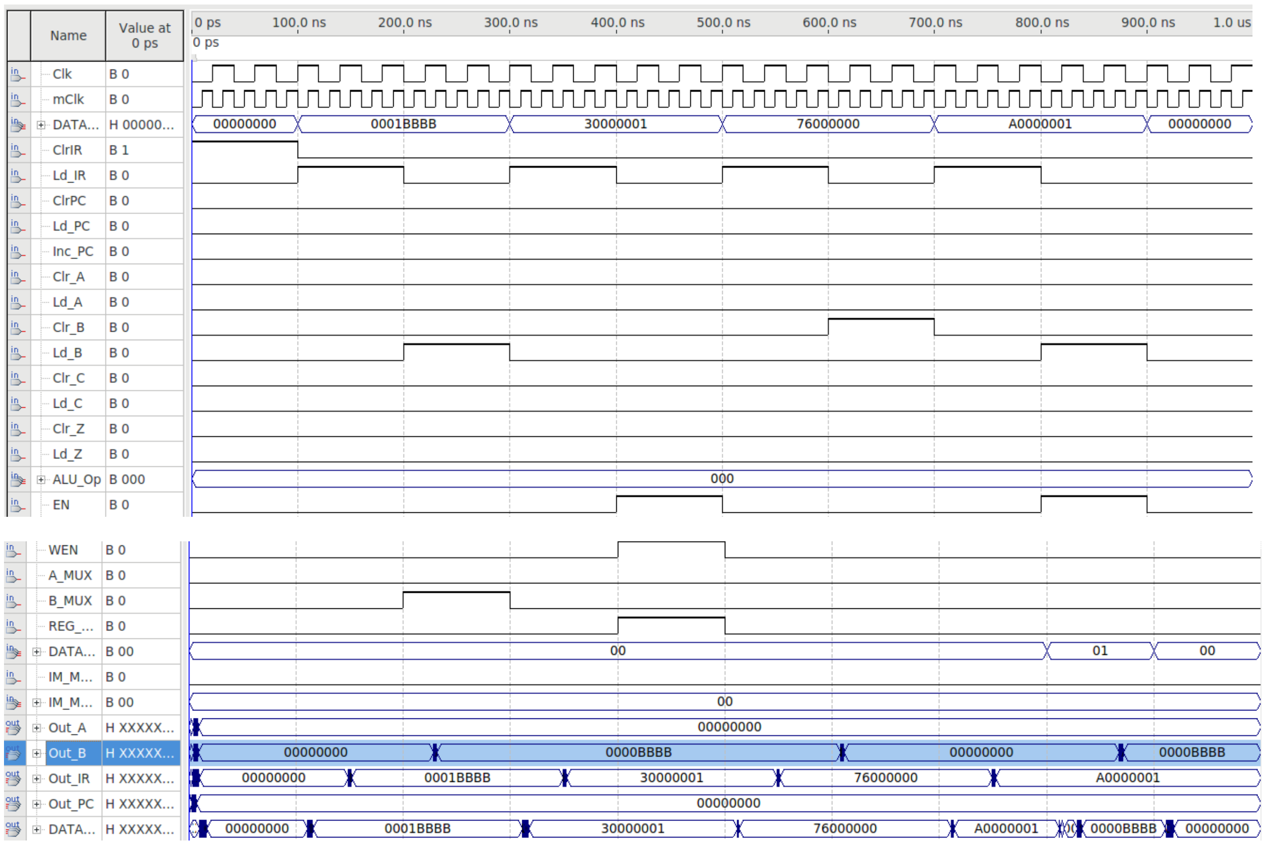


Figure 8: Timing waveform showing the correctness of the implementation of LDB instruction

Figure 8 proves the correctness of LDB instruction implementation. We notice that we successfully loaded our previously written values into register B.

LUI (Timing simulation)

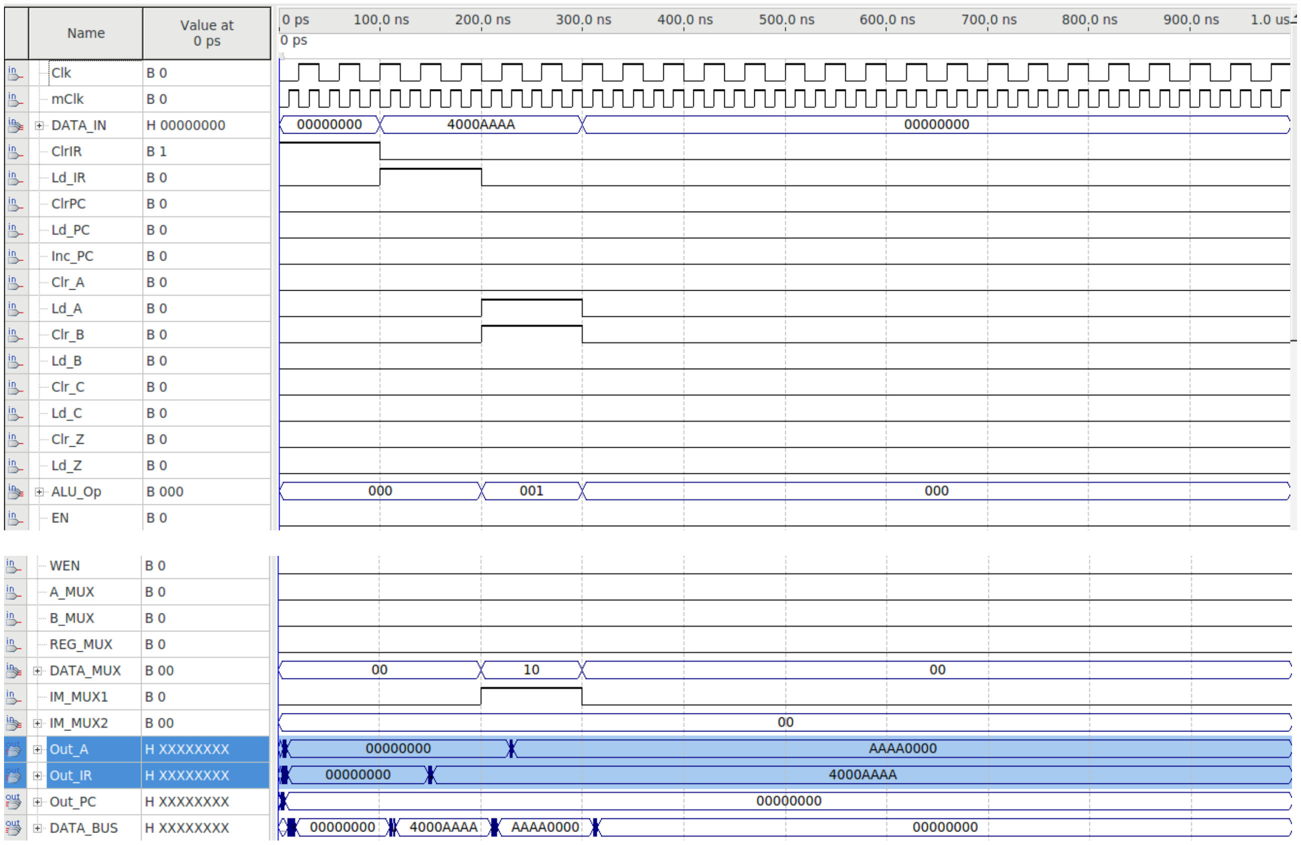


Figure 9: Timing waveform showing the correctness of the implementation of LUI instruction

Figure 9 shows that Out_IR's lower 16 bits are inserted into the upper 16 bits of register A, and the lower 16 bit of A is cleared to 0s, which is what we want to achieve with LUI instruction.

JMP (Timing simulation)

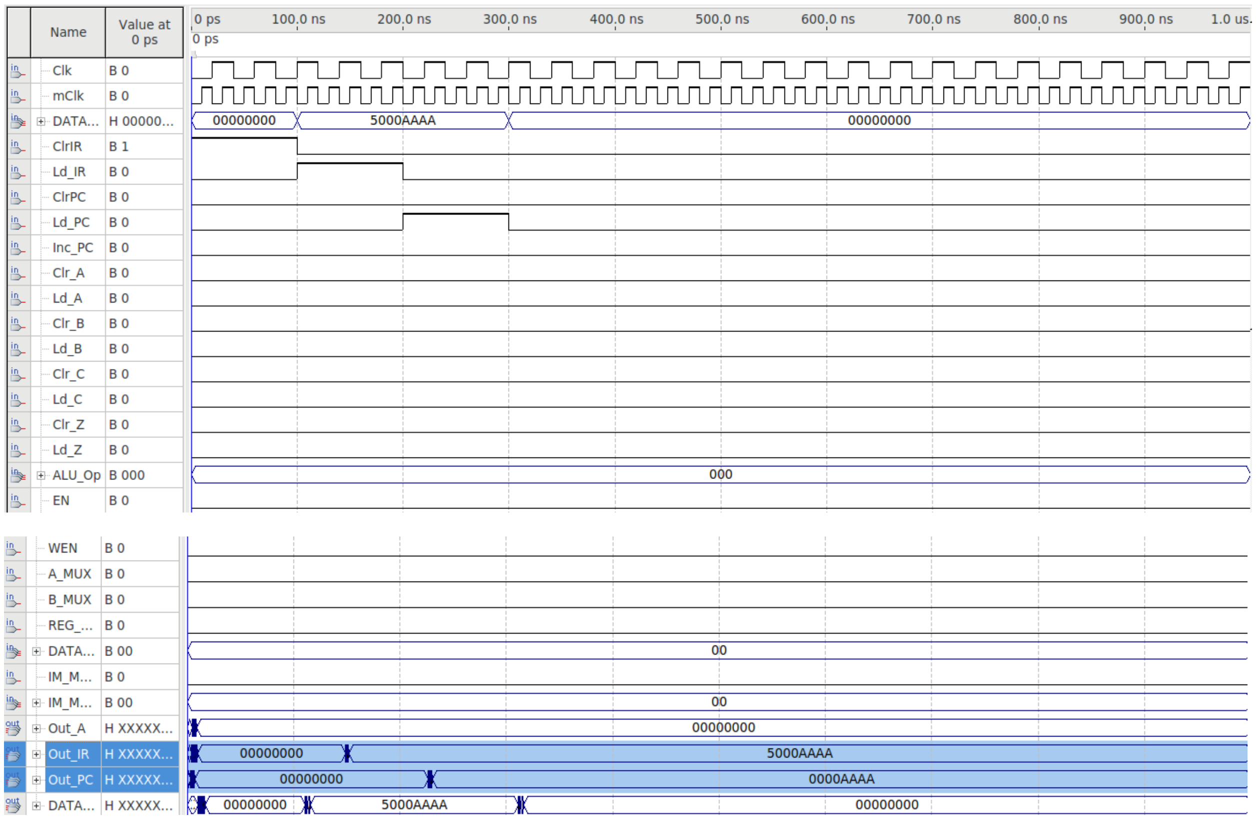


Figure 10: Timing waveform showing the correctness of the implementation of JMP instruction

In **figure 10**, we observe that the lower 16 bits of the IR is inserted into the lower 16 bits of the PC (Out_PC signal), while the upper 16 bits are cleared to 0s, which is what we expect from JMP instruction.

ANDI (Timing simulation)

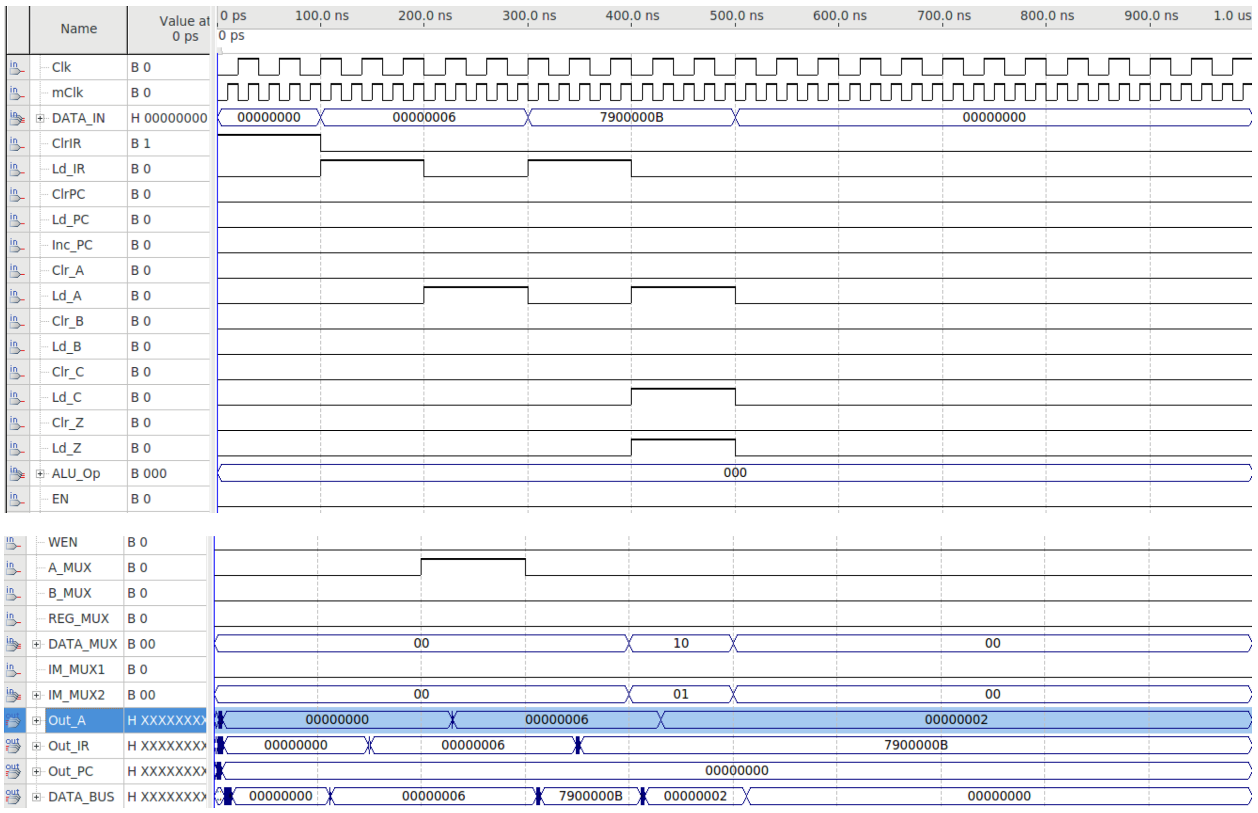


Figure 11: Timing simulation showing correctness of ANDI instruction

ADDI (Timing simulation)

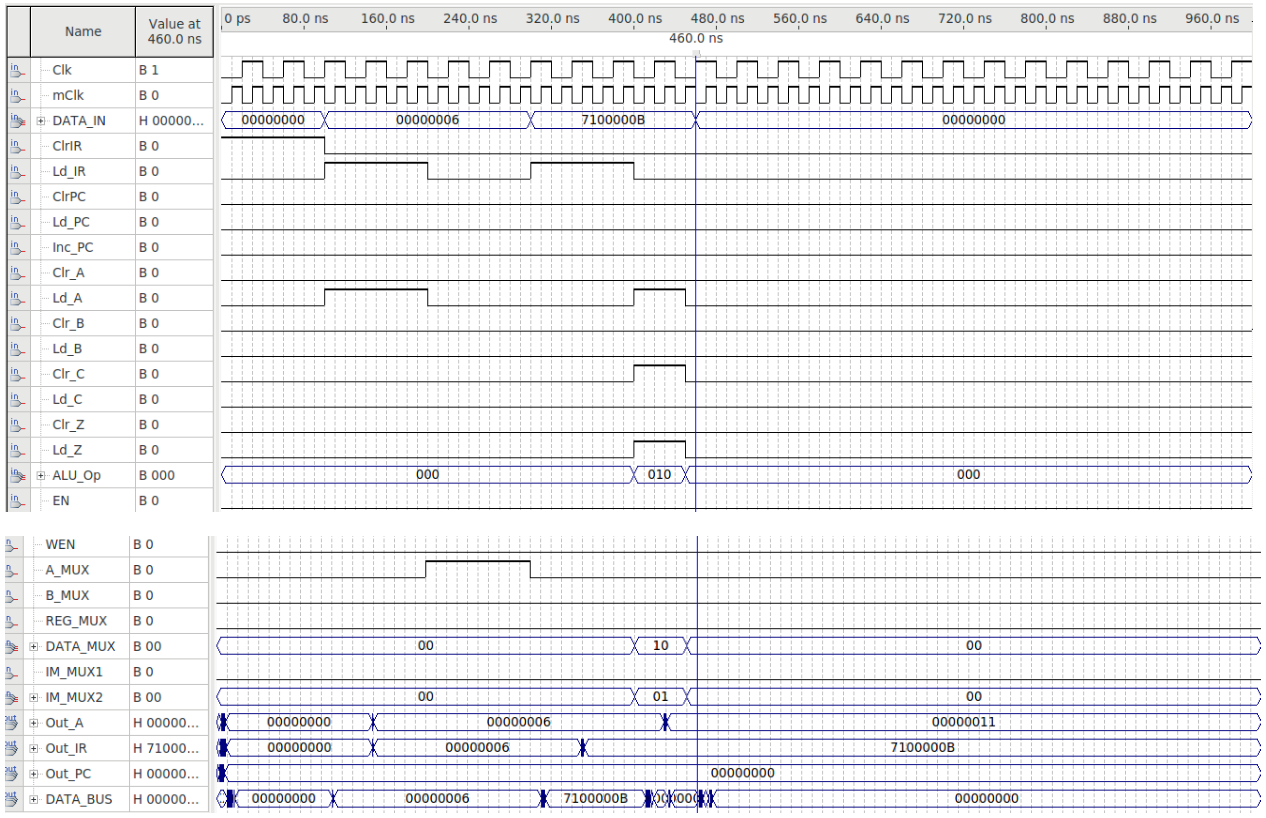


Figure 12: Timing waveform showing the correctness of the implementation of ADDI instruction

From **Figure 12**, we see we successfully added current value with a stored value in register A (only the lower 16 bits; see the **out_A** signal output). This shows the correctness of ADDI instruction.

ORI (Timing simulation)

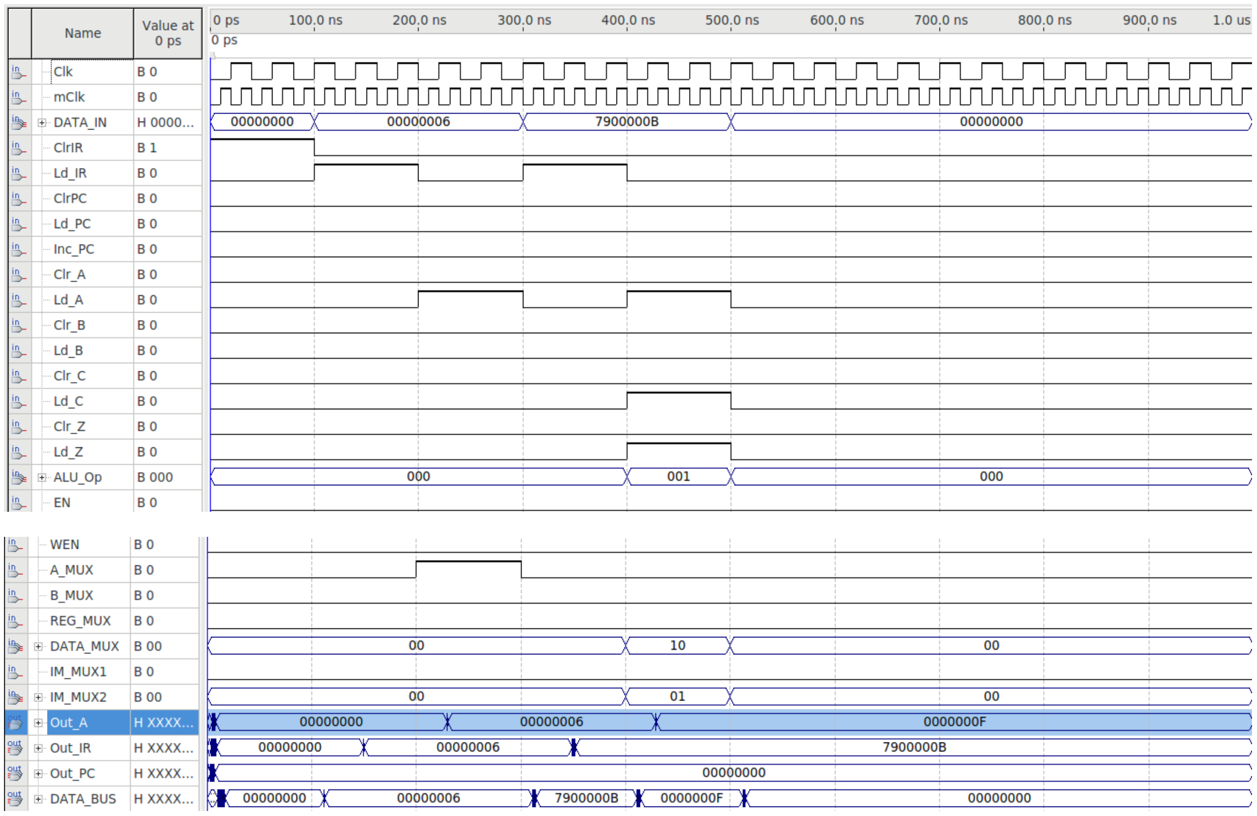


Figure 13: Timing waveform showing the correctness of the implementation of ORI instruction

ADD (Timing simulation)

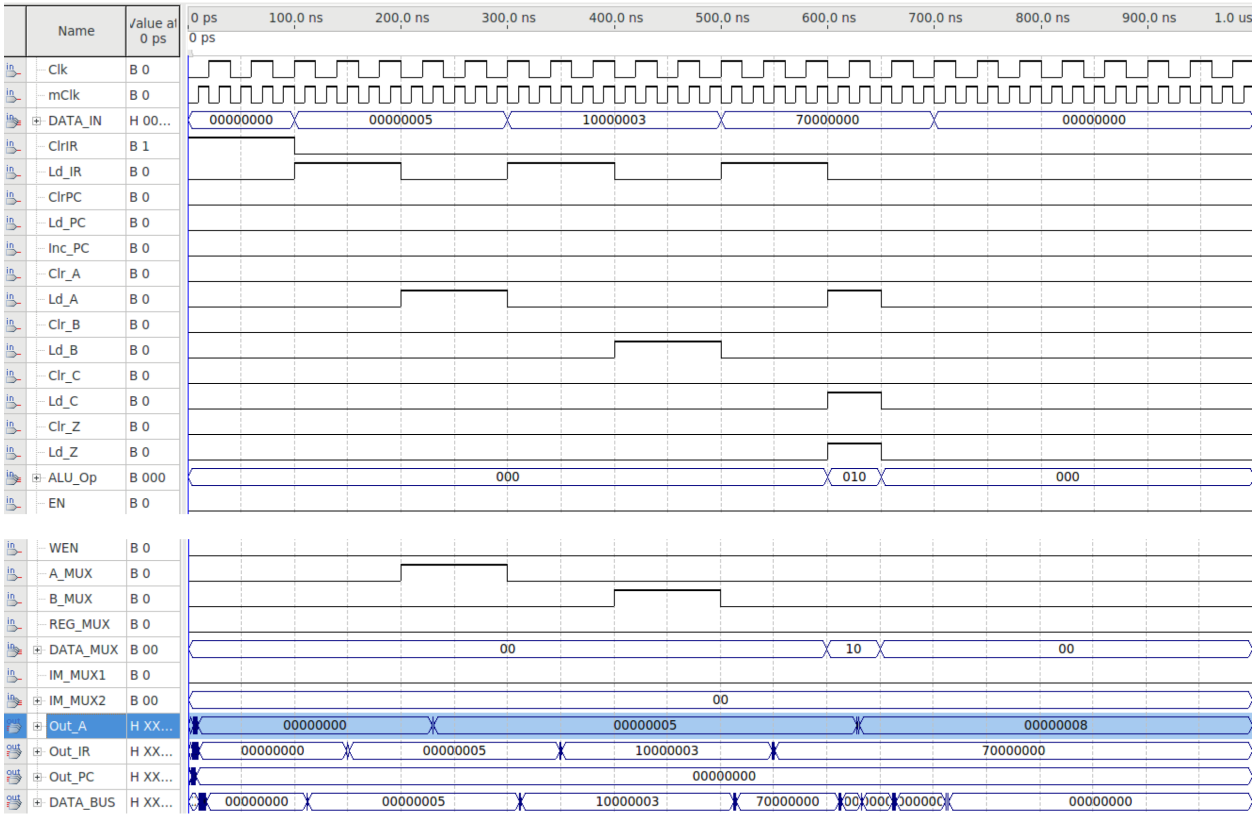


Figure 14: Timing waveform showing the correctness of the implementation of ADD instruction

SUB (Timing simulation)

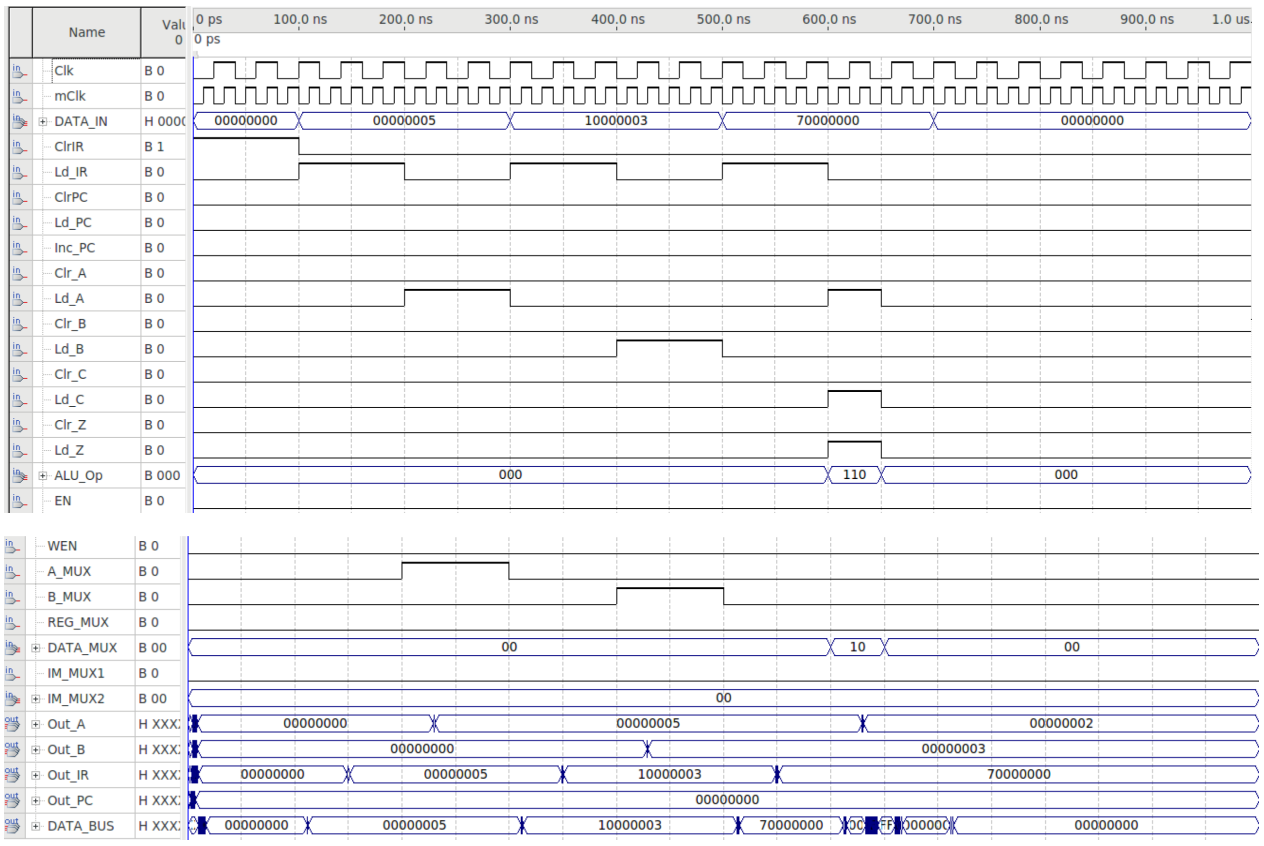


Figure 15: Timing waveform showing the correctness of the implementation of SUB instruction

DECA (Timing simulation)

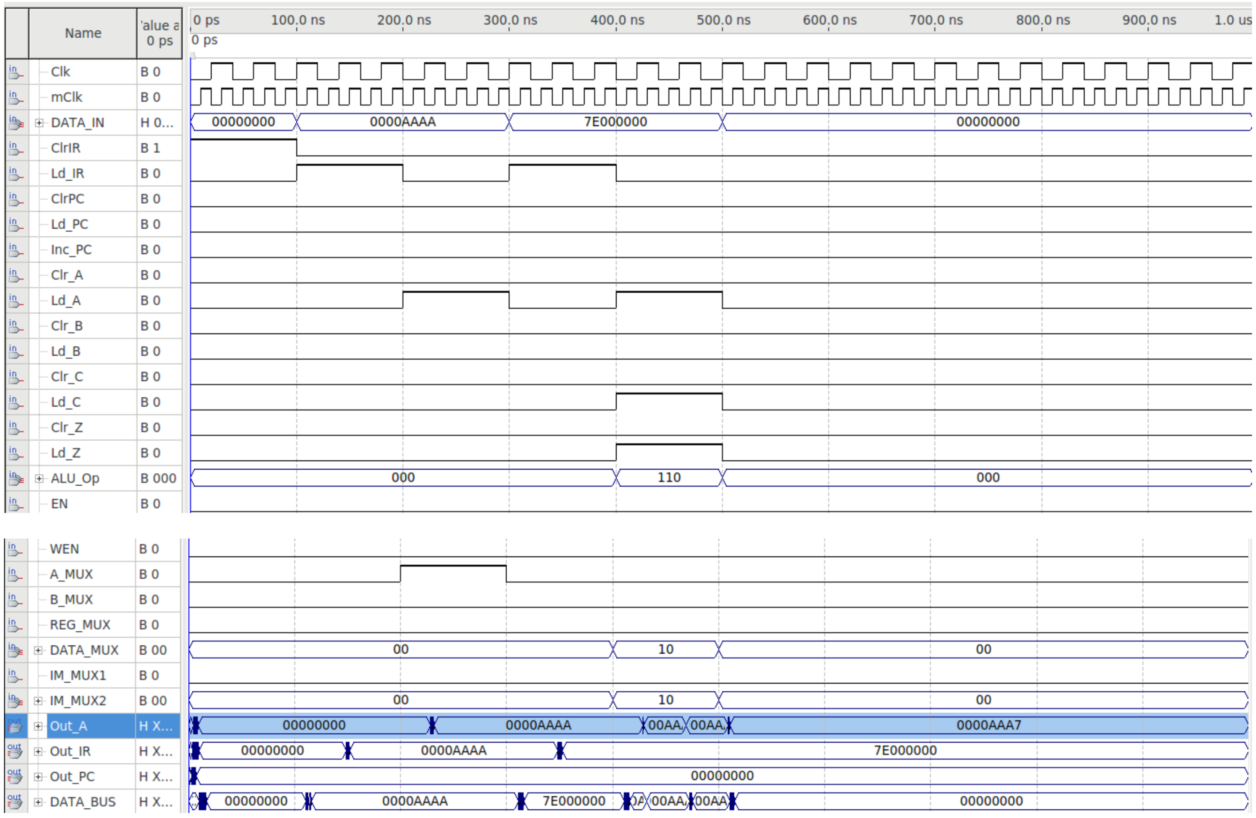


Figure 16: Timing waveform showing the correctness of the implementation of DECA instruction

INCA (Timing simulation)

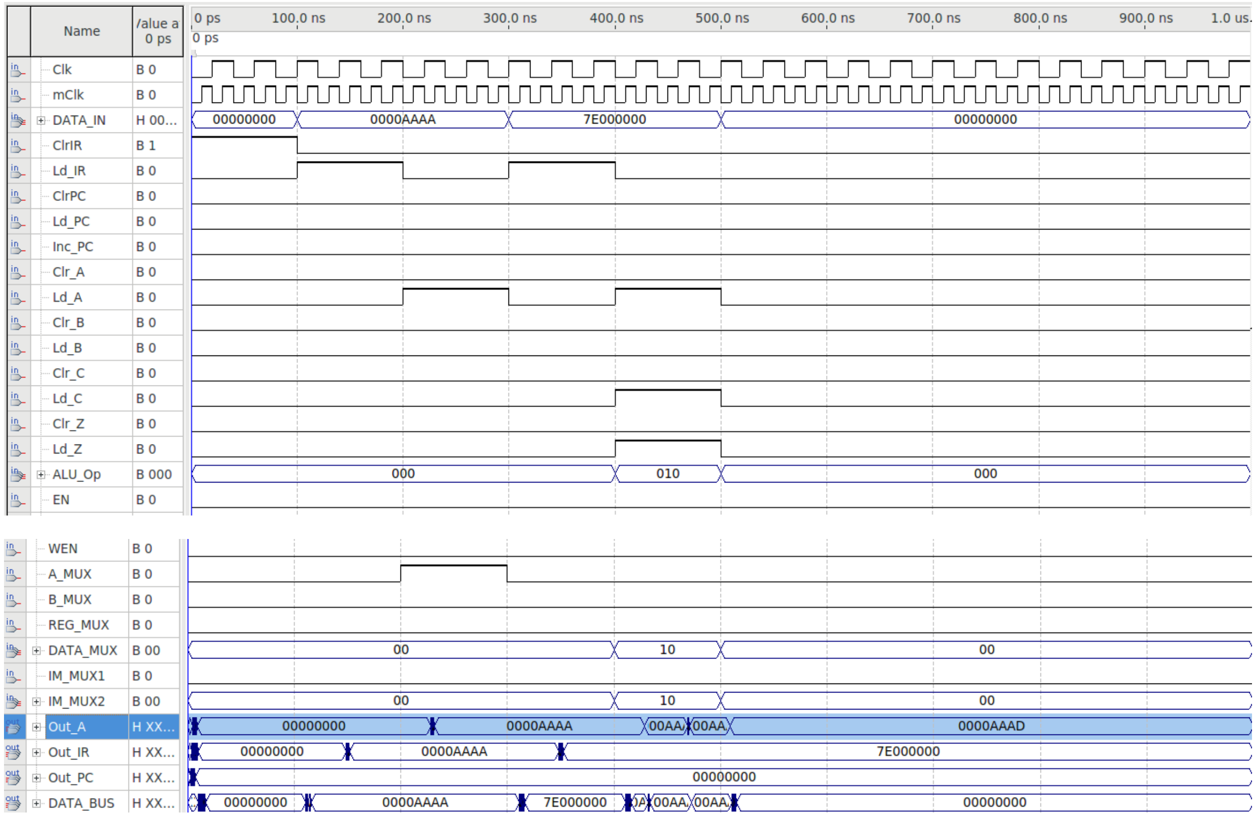


Figure 17: Timing waveform showing the correctness of the implementation of INCA instruction

ROL (Timing simulation)

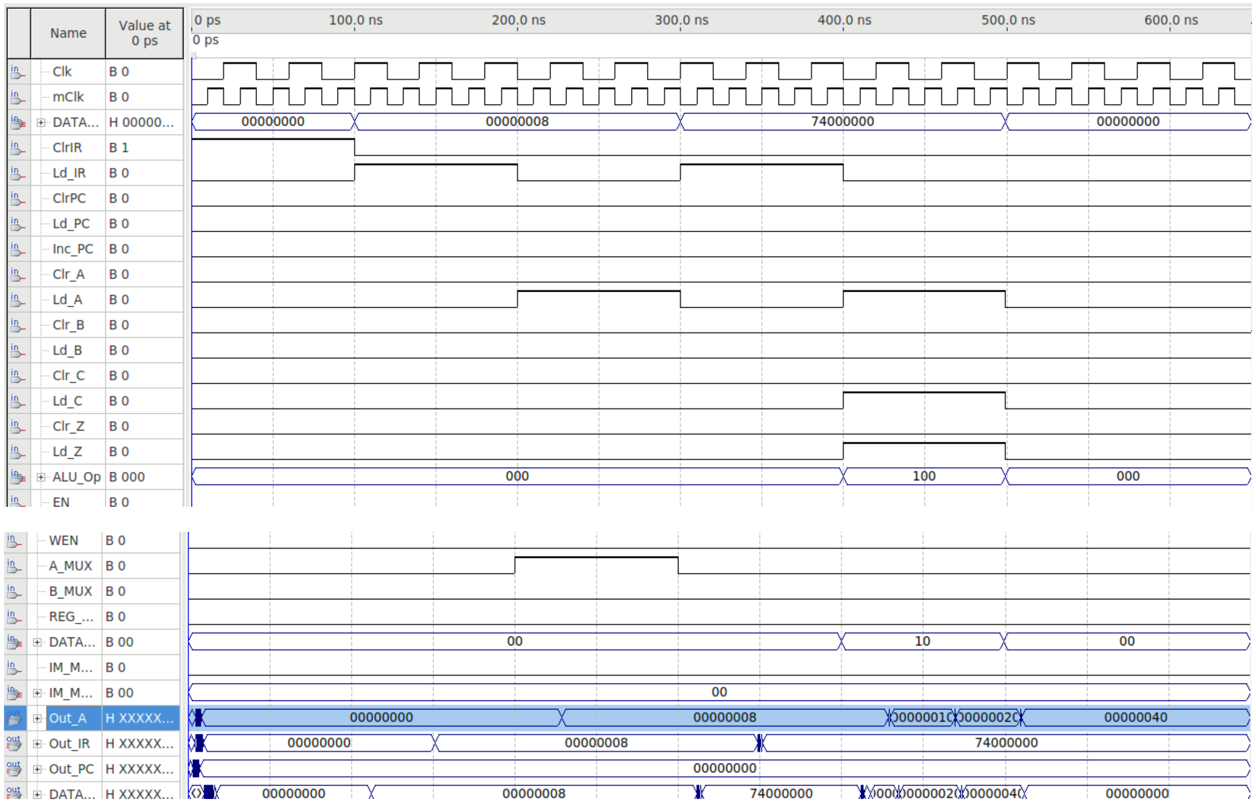


Figure 18: Timing waveform showing the correctness of the implementation of ROL instruction

ROR (Timing simulation)

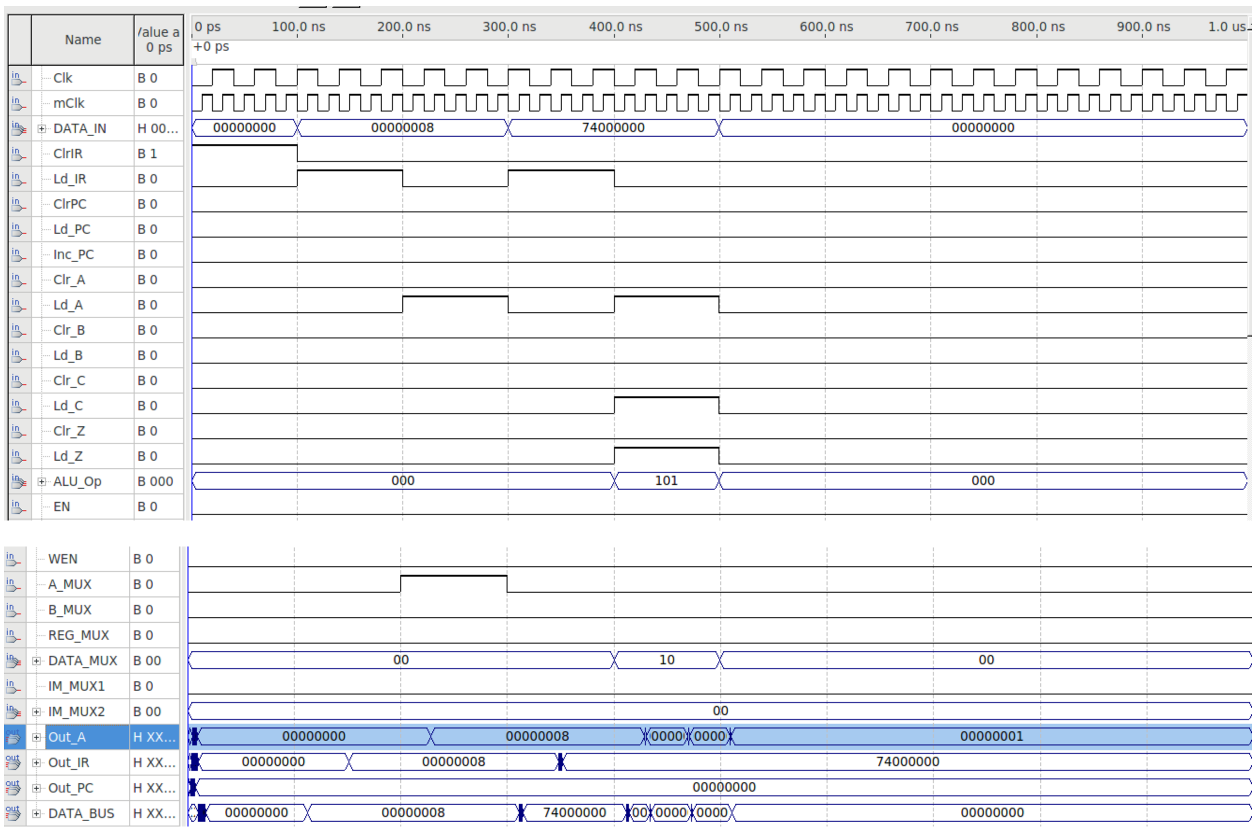


Figure 19: Timing waveform showing the correctness of the implementation of ROR instruction

CLRA (Timing simulation)

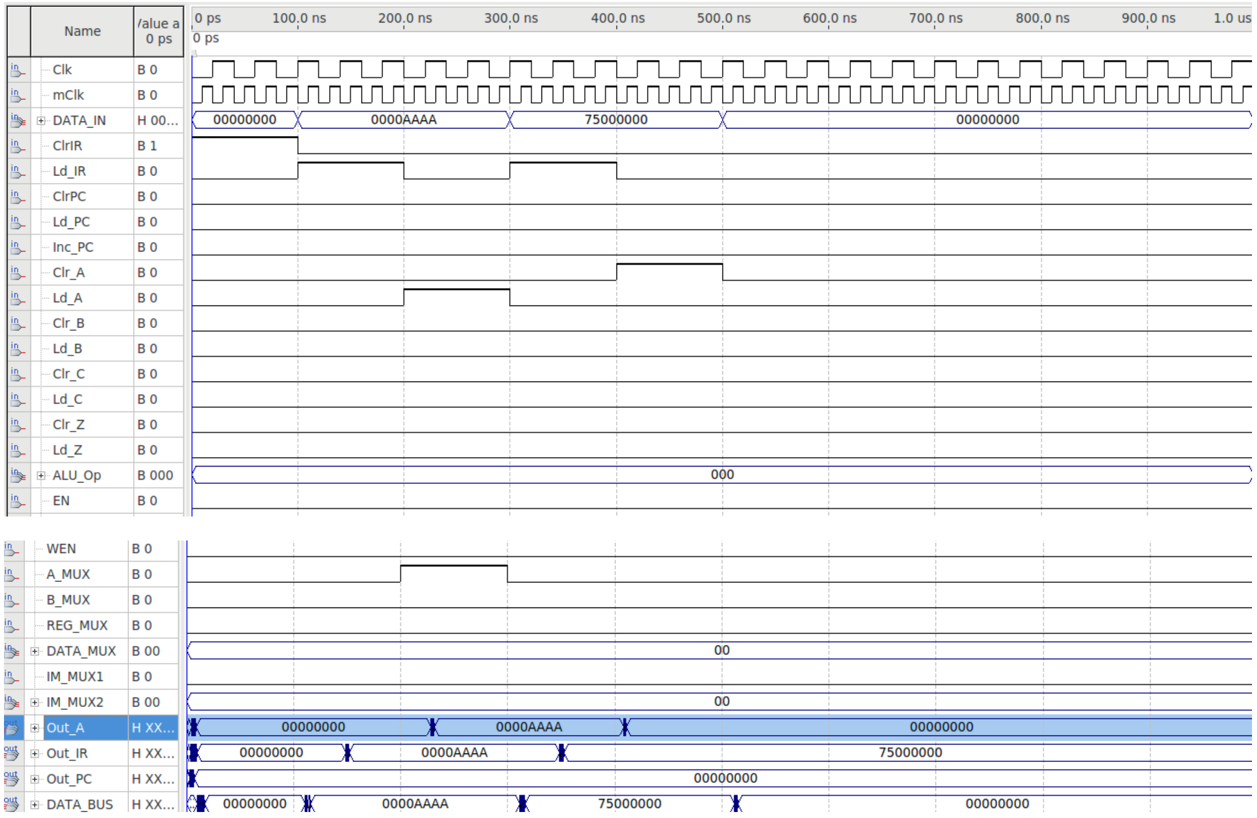


Figure 20: Timing waveform showing the correctness of the implementation of CLRA instruction

CLRB (Timing simulation)

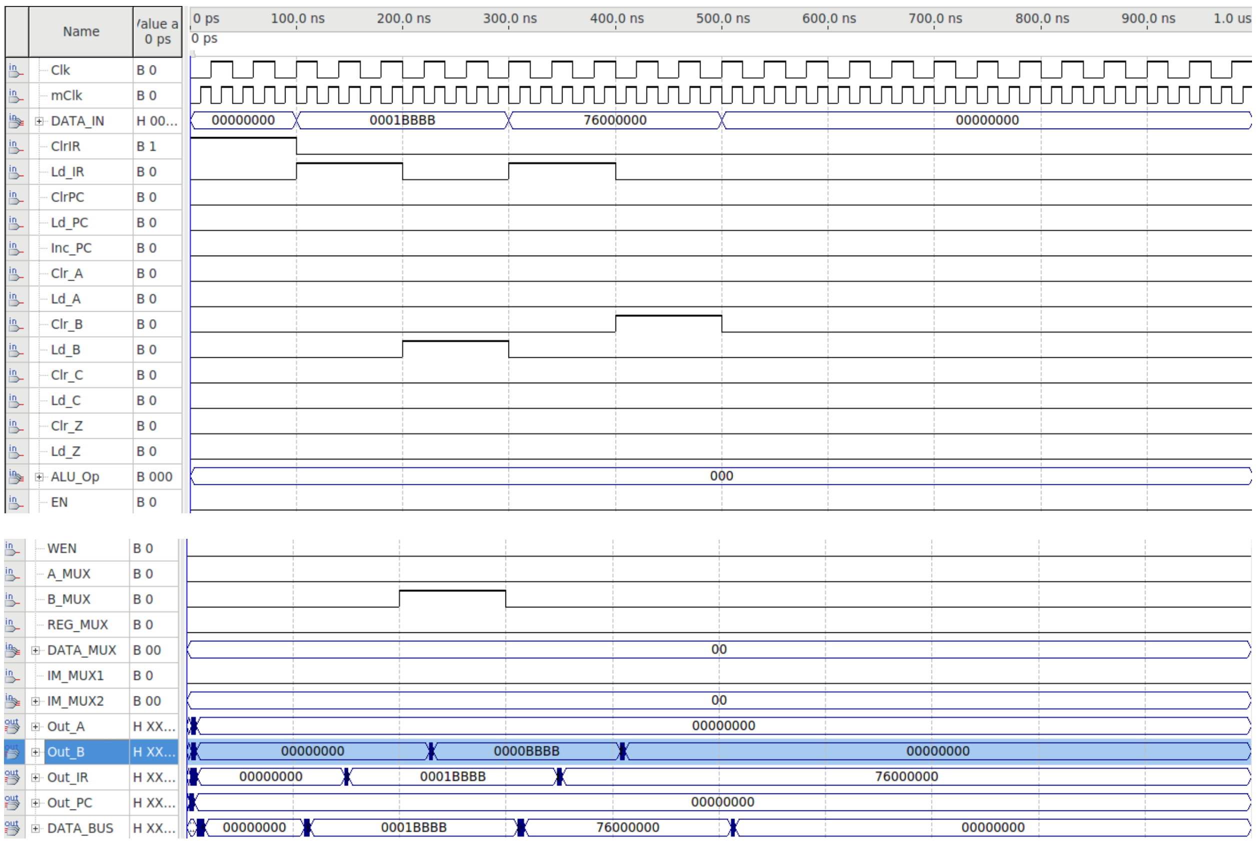


Figure 21: Timing waveform showing the correctness of the implementation of CLRB instruction

Control-Signal Table

Please refer to **table 2** in **page 5** for the completed table.

Answers to the post-lab questions

I) Based on **table 2**,

INCA : requires **CLR_IR** ,and **LD_IR** both to be cleared to 0, **LD_PC**, and **INC_PC** both to be 0, **CLR_A** to be 0, and **LD_A** to be 1, **CLR_B,** and **LD_B** both to be 0, **CLR_C** to be 0, and **LD_C** to be 1, **CLR_Z** to be 0, **LD_Z** to be 1, **ALU OP** to be 010 (since this is basically an addition operation), **EN,** and **WEN** values don't matter, **A_MUX** control to be 0, and **B_MUX** doesn't matter, **REG_MUX** control signal doesn't matter, **Data_MUX** control signal to be 10, **IM_MUX1** control to be 0, and **IM_MUX2** control to be 10.

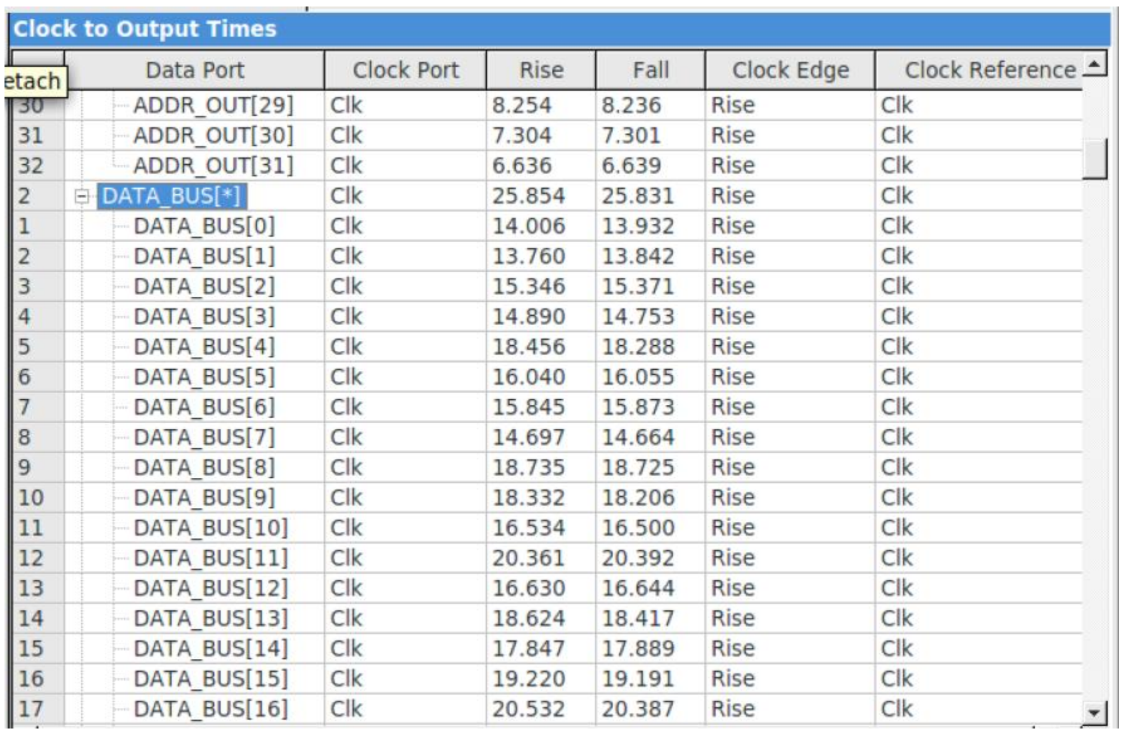
ADDI**:** requires every input of the data-path to be exactly same as **INCA** , except for **IM_MUX_2** control signal, where we require 01, instead.

LDBI**:** requires **CLR_IR** ,and **LD_IR** both to be cleared to 0, **LD_PC**, and **INC_PC** both to be 0, both **CLR_A** and **LD_A** to be 0, **CLR_B** to be 0 **,**and **LD_B** both to be 1, both **CLR_C** and **LD_C** to be 0, both **CLR_Z** and **LD_Z** to be 0, **ALU OP** doesn't matter, **EN,** and **WEN** values don't matter, **A_MUX** control doesn't matter, and **B_MUX** control signal to be 1, **REG_MUX** control signal doesn't matter, **Data_MUX** control signal doesn't matter, and both **IM_MUX1** and **IM_MUX2** control don't matter.

LDA: requires **CLR_IR** ,and **LD_IR** both to be cleared to 0, **LD_PC**, and **INC_PC** both to be 0, **CLR_A** to be 0 and **LD_A** to be 1, **CLR_B** to be 0 **,**and **LD_B** both to be 0, both **CLR_C** and **LD_C** to be 0, both **CLR_Z** and **LD_Z** to be 0, **ALU OP** doesn't matter, **EN** to be 1**,** and **WEN** to be 0, **A_MUX** control to be 0, and **B_MUX** control signal doesn't matter, **REG_MUX** control signal doesn't matter, **Data_MUX** control signal to be 01, and both **IM_MUX1** and **IM_MUX2** control don't matter.

ii) We know any digital logic circuit has both setup times, and hold-times requirements to function properly. If we do not allow the setup time, that is, if we make the clock signal frequency so high that we do not even allow the input signal to be stable for long enough to be latched properly before the active clock edge, our output will be very likely to be wrong. On the other hand, if clock frequency becomes so high that we do not allow the minimum amount of time for the data to stabilize after the active edge of the clock signal, our input data will inevitably be wrong, so will our output. **Therefore, we require our operating speed or Clk's period to be : $T_{Clk} \geq MAX(Setup\ time, Hold\ time)$ for reliable operations.**

iii) We need to analyze maximum propagation delay from main clock signal, Clk to any output. To do that, we use Quartus software's Time Quest Analyzer **clock to output times** table. By using this feature, we conclude that the maximum clock to output delay happens for the **DATA_BUS** output (refer to **figure 1**). **Figure 22** shows the average **DATA_BUS** output delay for the active clock edge.



Clock to Output Times

	Data Port	Clock Port	Rise	Fall	Clock Edge	Clock Reference
30	ADDR_OUT[29]	Clk	8.254	8.236	Rise	Clk
31	ADDR_OUT[30]	Clk	7.304	7.301	Rise	Clk
32	ADDR_OUT[31]	Clk	6.636	6.639	Rise	Clk
2	DATA_BUS[*]	Clk	25.854	25.831	Rise	Clk
1	DATA_BUS[0]	Clk	14.006	13.932	Rise	Clk
2	DATA_BUS[1]	Clk	13.760	13.842	Rise	Clk
3	DATA_BUS[2]	Clk	15.346	15.371	Rise	Clk
4	DATA_BUS[3]	Clk	14.890	14.753	Rise	Clk
5	DATA_BUS[4]	Clk	18.456	18.288	Rise	Clk
6	DATA_BUS[5]	Clk	16.040	16.055	Rise	Clk
7	DATA_BUS[6]	Clk	15.845	15.873	Rise	Clk
8	DATA_BUS[7]	Clk	14.697	14.664	Rise	Clk
9	DATA_BUS[8]	Clk	18.735	18.725	Rise	Clk
10	DATA_BUS[9]	Clk	18.332	18.206	Rise	Clk
11	DATA_BUS[10]	Clk	16.534	16.500	Rise	Clk
12	DATA_BUS[11]	Clk	20.361	20.392	Rise	Clk
13	DATA_BUS[12]	Clk	16.630	16.644	Rise	Clk
14	DATA_BUS[13]	Clk	18.624	18.417	Rise	Clk
15	DATA_BUS[14]	Clk	17.847	17.889	Rise	Clk
16	DATA_BUS[15]	Clk	19.220	19.191	Rise	Clk
17	DATA_BUS[16]	Clk	20.532	20.387	Rise	Clk

Figure 22: Time Quest analyzer console showing clock to output delays.

So, average delay, $T_{delay} = \frac{25.854+25.831}{2} ns = 25.8425\ ns$

However, the reliable limit for our data-path clock, Clk must be $T_{Clk} \geq MAX(25.854, 25.831) = 25.854\ ns$

Reference

1. *COE 608 Lab 4b – Data-Path Design*. D2L.
2. *COE 608 Lab 4b Tutorial – Data-Path Design*. D2L.
3. “Setup and hold time definition”, https://www.idconline.com/technical_references/pdfs/electronic_engineering/Setup_and_hold_time_definition.pdf
4. *COE 608- CPU Specifications.* D2L

Appendix

This section presents VHDL codes of all the components designed in this lab. Please VHDL codes for the components which have been reused have not been presented here again, since these were already unit tested before.

Upper Zero Extender (UZE)

```
library ieee;

use ieee.std_logic_1164.all;

use ieee.numeric_std.all;


entity UZE is

port(

    UZE_in : in std_logic_vector(31 downto 0);

    UZE_out : out std_logic_vector(31 downto 0)

    );

end entity;


architecture Behavior of UZE is

    signal zeros : std_logic_vector(15 downto 0) := (others => '0');

begin

    UZE_out <= UZE_in(15 downto 0) & zeros;

end Behavior;
```

Lower Zero Extender

```
library ieee;

use ieee.std_logic_1164.all;

use ieee.numeric_std.all;


entity LZE is

port ( LZE_in : in std_logic_vector(31 downto 0);

      LZE_out : out std_logic_vector(31 downto 0)

      );

end entity;
```

```
architecture Behavior of LZE is
signal zeros : std_logic_vector(15 downto 0) := (others => '0');
begin
    LZE_out <= zeros & LZE_in(15 downto 0);
end Behavior;
```

Reducer (RED)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity RED is
port(
    RED_in : in std_logic_vector(31 downto 0);
    RED_out : out unsigned(7 downto 0)
    );
end entity;

architecture Behavior of RED is
begin
    RED_out <= unsigned (RED_in(7 downto 0));
end Behavior;
```

MUX 4-to-1

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity RED is
port(
    RED_in : in std_logic_vector(31 downto 0);
    RED_out : out unsigned(7 downto 0)
    );
end entity;

architecture Behavior of RED is
begin
    RED_out <= unsigned (RED_in(7 downto 0));
end Behavior;
```

Data-Path module

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity data_path is
    port(
        -- Clock signal
        Clk, mClk : in std_logic; -- m clk is memory clock

        --Memory Signals
        WEN, EN : in std_logic;

        --Register Control Signals (CLR and LD)
        Clr_A, Ld_A : in std_logic;
        Clr_B, Ld_B : in std_logic;
        Clr_C, Ld_C : in std_logic; --Carry register
        Clr_Z, Ld_Z : in std_logic; --Zero flag register
        ClrPC, Ld_PC : in std_logic;
        ClrIR, Ld_IR : in std_logic; -- Clearing and loading instruction register

        --Register Outputs
        Out_A : out std_logic_vector(31 downto 0);
        Out_B   : out std_logic_vector(31 downto 0);
        Out_C   : out std_logic;
        Out_Z   : out std_logic;
        Out_PC   : out std_logic_vector(31 downto 0);
        Out_IR   : out std_logic_vector(31 downto 0);

        --Special Inputs to PC
        Inc_PC : in std_logic;

        --Address and Data Bus signals for debugging
        ADDR_OUT : out std_logic_vector(31 downto 0);
        DATA_IN : in std_logic_vector(31 downto 0);
        DATA_BUS,
        MEM_OUT,
        MEM_IN  : out std_logic_vector(31 downto 0);
        MEM_ADDR : out unsigned(7 downto 0);

        --Various MUX controls
        DATA_MUX : in std_logic_vector(1 downto 0);
        REG_MUX : in std_logic;
        A_MUX,
        B_MUX : in std_logic;
        IM_MUX1 : in std_logic;
        IM_MUX2 : in std_logic_vector(1 downto 0);

        --ALU operations
        ALU_Op : in std_logic_vector(2 downto 0)
    );
end entity;

architecture Behavior of Data_Path is
    --Component Instantiations
    --Data Memory Module
    component data_mem is
        port(
            clk : in std_logic;
            addr    :   in unsigned(7 downto 0);
            data_in :   in std_logic_vector(31 downto 0);
            wen : in std_logic;
            en  : in std_logic;
            data_out    : out std_logic_vector(31 downto 0)
        );
    end component;

    --Register 32
    component register32 is
        port(
            d : in std_logic_vector(31 downto 0); --input 32 bit
            ld: in std_logic; --load/enable
            clr: in std_logic; --clear
            clk: in std_logic;
            Q: out std_logic_vector(31 downto 0)
        );
    end component;

    --Program Counter
    component pc is
        port(
            clr : in std_logic;
            clk : in std_logic;
            ld : in std_logic;
            inc : in std_logic;
            d : in std_logic_vector(31 downto 0);
            q : out std_logic_vector(31 downto 0)
            );
    end component;

    --LZE
```

```
component LZE is
    port(
        LZE_in : in std_logic_vector(31 downto 0);
        LZE_out : out std_logic_vector(31 downto 0)
    );
end component;

--UZE
component UZE is
    port(
        UZE_in : in std_logic_vector(31 downto 0);
        UZE_out : out std_logic_vector(31 downto 0)
        );
end component;

--RED
component RED is
    port(
        RED_in : in std_logic_vector(31 downto 0);
        RED_out : out unsigned(7 downto 0)
    );
end component;

--Mux 2 to 1
component mux2to1 is
    port( s : in std_logic;
        w0, w1 : in std_logic_vector(31 downto 0);
        f : out std_logic_vector (31 downto 0)
        );
end component;

--mux4to1
component mux4to1 is
    port( s :   in std_logic_vector (1 downto 0);
    X1, X2, X3, X4 : in std_logic_vector (31 downto 0);
    f : out std_logic_vector (31 downto 0));
end component;

--ALU
component ALU is --Only this component name was custom.
    port(
          a : in std_logic_vector(31 downto 0);
          b : in std_logic_vector(31 downto 0);
          op : in std_logic_vector(2 downto 0);
        result : out std_logic_vector(31 downto 0);
        zero : out std_logic;
        cout : out std_logic
        );
end component;

--Signal Instantiations
signal IR_OUT : std_logic_vector(31 downto 0);
signal data_bus_s : std_logic_vector(31 downto 0);
signal LZE_out_PC : std_logic_vector(31 downto 0);
signal LZE_out_A_Mux : std_logic_vector(31 downto 0);
signal LZE_out_B_Mux : std_logic_vector(31 downto 0);
signal RED_out_Data_Mem : unsigned(7 downto 0);
signal A_Mux_out : std_logic_vector(31 downto 0);
signal B_Mux_out : std_logic_vector(31 downto 0);
signal reg_A_out : std_logic_vector(31 downto 0);
signal reg_B_out : std_logic_vector(31 downto 0);
signal reg_Mux_out : std_logic_vector(31 downto 0);
signal data_mem_out : std_logic_vector(31 downto 0);
signal UZE_IM_MUX1_out : std_logic_vector(31 downto 0);
signal IM_MUX1_out : std_logic_vector(31 downto 0);
signal LZE_IM_MUX2_out : std_logic_vector(31 downto 0);
signal IM_MUX2_out : std_logic_vector(31 downto 0);
signal ALU_out : std_logic_vector(31 downto 0);
```

```
    signal zero_flag : std_logic;
    signal carry_flag : std_logic;
    signal temp : std_logic_vector(30 downto 0) := (others => '0');
    signal out_pc_sig : std_logic_vector(31 downto 0);
begin
    IR: register32 port map(
        data_bus_s,
        Ld_IR,
        ClrIR,
        Clk,
        IR_OUT
        );
    LZE_PC: LZE port map(
        IR_OUT,
        LZE_out_PC
        );

    PC0: pc port map(
        ClrPC,
        Clk,
        Ld_PC,
        Inc_PC,
        LZE_out_PC,
        -- adder out
        out_pc_sig
        );
    LZE_A_Mux: LZE port map(
            IR_OUT,
            LZE_out_A_Mux
        );
    A_Mux0: mux2to1 port map(
        A_MUX,
        data_bus_s, LZE_out_A_Mux,
        A_Mux_out
        );
    Reg_A: register32 port map(
        A_Mux_out,
        Ld_A,
        Clr_A,
        Clk,
        reg_A_out
        );
    LZE_B_Mux: LZE port map(
        IR_OUT,
        LZE_out_B_Mux
        );

    B_Mux0: mux2to1 port map(
        B_MUX,
        data_bus_s, LZE_out_B_Mux,
        B_Mux_out
        );

    Reg_B: register32 port map(
        B_Mux_out,
        Ld_B,
        Clr_B,
        Clk,
        reg_B_out
        );

    Reg_Mux0: mux2to1 port map(
            REG_MUX,
            reg_A_out, reg_B_out,
            reg_Mux_out
        );

    RED_data_Mem: RED port map(
        IR_OUT,
        RED_out_Data_Mem
        );

    Data_Mem0: data_mem port map(
        mClk,
        RED_out_Data_Mem,
        reg_Mux_out,
        WEN,
        EN,
        data_mem_out
        );

    UZE_IM_MUX1: UZE port map(
        IR_OUT,
        UZE_IM_MUX1_out
        );

    IM_MUX1a: mux2to1 port map(
        IM_MUX1,
        reg_A_out, UZE_IM_MUX1_out,
        IM_MUX1_out
        );

    LZE_IM_MUX2: LZE port map(
        IR_OUT,
        LZE_IM_MUX2_out
        );

    IM_MUX2a:   mux4to1 port map(
        IM_MUX2,
        reg_B_out, LZE_IM_MUX2_out, (temp & '1'), (others => '0'),
        IM_MUX2_out
        );
        ALU0: ALU port map(
            IM_MUX1_out,
            IM_MUX2_out,
            ALU_Op,
            ALU_out,
            zero_flag,
            carry_flag
            );

        DATA_MUX0: mux4to1 port map(
            DATA_MUX,
            DATA_IN, data_mem_out, ALU_out, (others => '0'),
            data_bus_s
            );

        DATA_BUS <= data_bus_s;
        Out_A <= reg_A_out;
        Out_B <= reg_B_out;
        Out_IR <= IR_OUT;
        ADDR_OUT <= out_pc_sig;
        Out_PC <= out_pc_sig;

        MEM_ADDR <= RED_out_Data_Mem;
        MEM_IN <= reg_Mux_out;
        MEM_OUT <= data_mem_out;

    end Behavior;
```