Lab 4B(Lab 4 Part-2): Data-Path Design:

Introduction:
The objective of this lab(lab 4 part 2) is to successfully construct a functional data-path scheme and test it for a processor, more specifically a 32-bit CPU unit. For this lab, previous lab components(PC, register, ALU, Data memory module) are taken side-by-side with some new data-path components, and they are assembled to create a sufficient CPU data-path. The data-path is then required to go through testing from different specifications and requirements in creating different waveforms for various operations. Note that the instruction memory is not filled out yet, this will be worked on for later labs(lab 6).

Please note that the following project files are the components from the previous labs that are required for this one:

In this lab we will build the and test the data-path for the 32-bit processor. Please copy the following VHDL files from the previous labs into your Lab4b folder, then add them to your project in the Add Files section of the New Project Wizard:

- From Lab 2:
  - "register32.vhd"
  - "add.vhd"
  - "mux2to1.vhd"
  - "pc.vhd"
- From Lab 3a:
  - "fulladd.vhd"
  - "adder4.vhd"
  - "adder16.vhd"
  - "adder32.vhd"
  - "alu.vhd"
- From Lab 4a:
  - "data_mem.vhd"

The following are the new components that are required for the data-path:

Implementation of the LZE(Lower Zero Extender) Unit:

```vhdl
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;
```

Figure 1: Code Implementation for the LZE(Lower Zero Extender) Unit Component

Implementation of the UZE(Upper Zero Extender) Unit:

```vhdl
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;
```

Figure 2: Code Implementation for the UZE(Upper Zero Extender) Unit Component

Implementation of the RED(Reducer Unit) Unit:

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity RED is
6   port(
7       RED_in   : in  std_logic_vector(31 downto 0);
8       RED_out  : out unsigned(7 downto 0)
9   );
10  end entity;
11
12  architecture Behavior of RED is
13  begin
14      RED_out <= unsigned (RED_in(7 downto 0));
15  end Behavior;
16
```

Figure 3: Code Implementation for the RED(Reducer Unit) Unit Component

Implementation of the mux4to1(4 IN-1 OUT MUX) Unit:

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity mux4to1 is
5   port(
6   s : in std_logic_vector(1 downto 0);
7   X1, X2, X3, X4 : in std_logic_vector(31 downto 0);
8   f: out std_logic_vector(31 downto 0)
9   );
10  end mux4to1;
11
12  architecture Behavior of mux4to1 is
13  begin
14      with s select
15          f <= X1 when "00",
16               X2 when "01",
17               X3 when "10",
18               X4 when "11";
19  end Behavior;
```

Figure 4: Code Implementation for the 4 Input and 1 Output MUX(mux4to1) Unit Component

Implementation of the 32-bit CPU Data Path Design:

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_arith.all;
4  use ieee.std_logic_unsigned.all;
5
6  entity data_path is
7  port(
8   --Clock Signals
9   Clk, mClk : in std_logic;
10
11  --Memory Signals
12  WEN, EN : in std_logic;
13
14  --Register Control Signals
15  Clr_A, Ld_A : in std_logic;
16  Clr_B, Ld_B : in std_logic;
17  Clr_C, Ld_C : in std_logic;
18  Clr_Z, Ld_Z : in std_logic;
19  ClrPC, Ld_PC : in std_logic;
20  ClrIR, Ld_IR : in std_logic;
21
22  --Register Outputs
23  Out_A : out std_logic_vector(31 downto 0);
24  Out_B : out std_logic_vector(31 downto 0);
25  Out_C : out std_logic;
26  Out_Z : out std_logic;
27  Out_PC : out std_logic_vector(31 downto 0);
28  Out_IR : out std_logic_vector(31 downto 0);
29
30  --Special PC Inputs
31  Inc_PC : in std_logic;
32
```

```
33  --Address and Data Bus Signals
34  ADDR_OUT : out std_logic_vector(31 downto 0);
35  DATA_IN : in std_logic_vector(31 downto 0);
36  DATA_BUS, MEM_OUT, MEM_IN : out std_logic_vector(31 downto 0);
37  MEM_ADDR : out unsigned(7 downto 0);
38
39  --MUX Controls
40  DATA_MUX : in std_logic_vector(1 downto 0);
41  REG_MUX : in std_logic;
42  A_MUX, B_MUX : in std_logic;
43  IM_MUX1 : in std_logic;
44  IM_MUX2 : in std_logic_vector(1 downto 0);
45
46  --ALU Operations
47  ALU_Op : in std_logic_vector(2 downto 0)
48 );
49 end entity;
50
51
52 architecture Behavior of data_path is
53
54  --Component Instantations Data Memory Module
55 component data_mem is
56 port(
57  clk : in std_logic;
58  addr : in unsigned(7 downto 0);
59  data_in : in std_logic_vector(31 downto 0);
60  wen : in std_logic;
61  en : in std_logic;
62  data_out : out std_logic_vector(31 downto 0)
63 );
64 end component;
65
```

```
66  --Register32
67  component register32 is
68  port (
69   d : in std_logic_vector(31 downto 0) ;
70   ld : in std_logic ;
71   clr : in std_logic ;
72   clk : in std_logic ;
73   Q : out std_logic_vector(31 downto 0)) ;
74   end component;
75
76   --Program Counter
77  component pc is
78  port (
79   clr : in std_logic ;
80   clk : in std_logic ;
81   ld : in std_logic ;
82   inc : in std_logic ;
83   d : in std_logic_vector(31 downto 0) ;
84   q : out std_logic_vector(31 downto 0) ) ;
85   end component;
86
87   --LZE
88  component LZE is
89  port(
90   LZE_in : in std_logic_vector(31 downto 0);
91   LZE_out : out std_logic_vector(31 downto 0)
92   );
93   end component;
94
95   --UZE
96  component UZE is
97  port(
98   UZE_in : in std_logic_vector(31 downto 0);
99   UZE_out : out std_logic_vector(31 downto 0)
100  );
101  end component;
```

```
102
103  --RED
104 component RED is
105 port(
106      RED_in   : in  std_logic_vector(31 downto 0);
107      RED_out  : out unsigned(7 downto 0)
108 );
109  end component;
110
111  --mux2to1
112 component mux2to1 is
113 port (
114  s : in std_logic ;
115  w0, w1 : in std_logic_vector(31 downto 0) ;
116  f : out std_logic_vector(31 downto 0) ) ;
117  end component;
118
119  --mux4to1
120 component mux4to1 is
121 port(
122  s : in std_logic_vector(1 downto 0);
123  X1, X2, X3, X4 : in std_logic_vector(31 downto 0);
124  f: out std_logic_vector(31 downto 0)
125  );
126  end component;
127
128  --ALU
129 component ALU32 is
130 port(
131  a, b : in std_logic_vector(31 downto 0);
132  op : in std_logic_vector(2 downto 0);
133  result : out std_logic_vector(31 downto 0);
134  zero, cout : out std_logic );
135  end component;
136
```

```
--Signal Instantations
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,
   --ADDR 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
   );
```

```
209     B_Mux0: mux2to1 port map(
210     B_MUX,
211     data_bus_s,
212     LZE_out_B_mux,
213     B_mux_out
214     );
215
216     Reg_B: register32 port map(
217     B_mux_out,
218     Ld_B,
219     Clr_B,
220     Clk,
221     reg_B_out
222     );
223
224     Reg_Mux0: mux2to1 port map(
225     REG_MUX,
226     Reg_A_out,
227     Reg_B_out,
228     Reg_Mux_out
229     );
230
231     RED_Data_Mem: RED port map(
232     IR_OUT,
233     RED_out_data_mem
234     );
235
236     Data_Mem0: data_mem port map(
237     mClk,
238     RED_out_data_mem,
239     Reg_Mux_out,
240     WEN,
241     EN,
242     data_mem_out
243     );
244
245     UZE_IM_MUX1: UZE port map(
246     IR_OUT,
247     UZE_IM_MUX1_out
248     );
249
250     IM_MUX1a: mux2to1 port map(
251     IM_MUX1,
252     reg_A_out,
253     UZE_IM_MUX1_out,
254     IM_MUX1_out
255     );
256
257     LZE_IM_MUX2: LZE port map(
258     IR_OUT,
259     LZE_IM_MUX2_out
260     );
261
262     IM_MUX2a: mux4to1 port map(
263     IM_MUX2,
264     reg_B_out,
265     LZE_IM_MUX2_out, (temp &'1'), (others =>'0'),
266     IM_MUX2_out
267     );
268
269     ALU0: ALU32 port map(
270     IM_MUX1_out,
271     IM_MUX2_out,
272     ALU_Op,
273     ALU_out,
274     zero_flag,
275     carry_flag
276     );
277
```

```
278     DATA_MUX0: mux4to1 port map(
279     DATA_MUX,
280     DATA_IN,
281     data_mem_out,
282     ALU_out,
283     (others => '0'),
284     data_bus_s
285     );
286
287     DATA_BUS <= data_bus_s;
288     OUT_A <= reg_A_out;
289     OUT_B <= reg_B_out;
290     OUT_IR <= IR_OUT;
291     ADDR_OUT <= out_pc_sig;
292     OUT_PC <= out_pc_sig;
293
294     MEM_ADDR <= RED_out_data_mem;
295     MEM_IN <= Reg_Mux_out;
296     MEM_OUT <= data_mem_out;
297
298 end Behavior;
```

Figure 5 – 1: Code Implementation for the 32-bit CPU Data Path Design Component

Data Path Functional Simulation Waveforms - Testing Portion:

1)

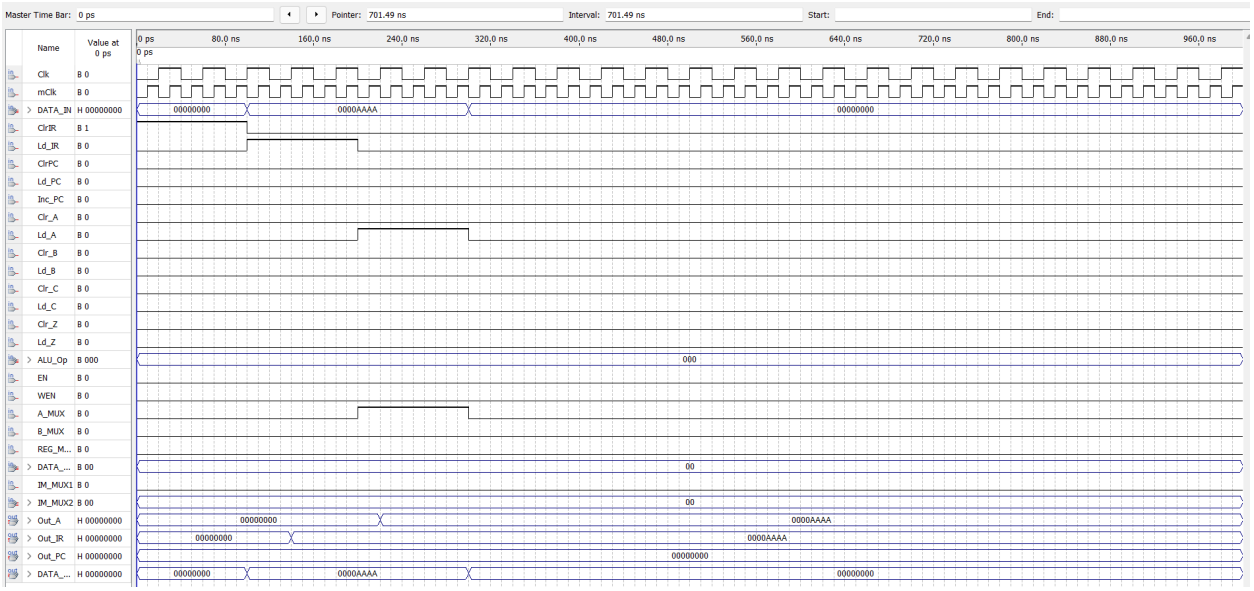

Figure 6: Waveform of the 32-bit CPU Data Path Design Component for operation: LDAI(load)

2)

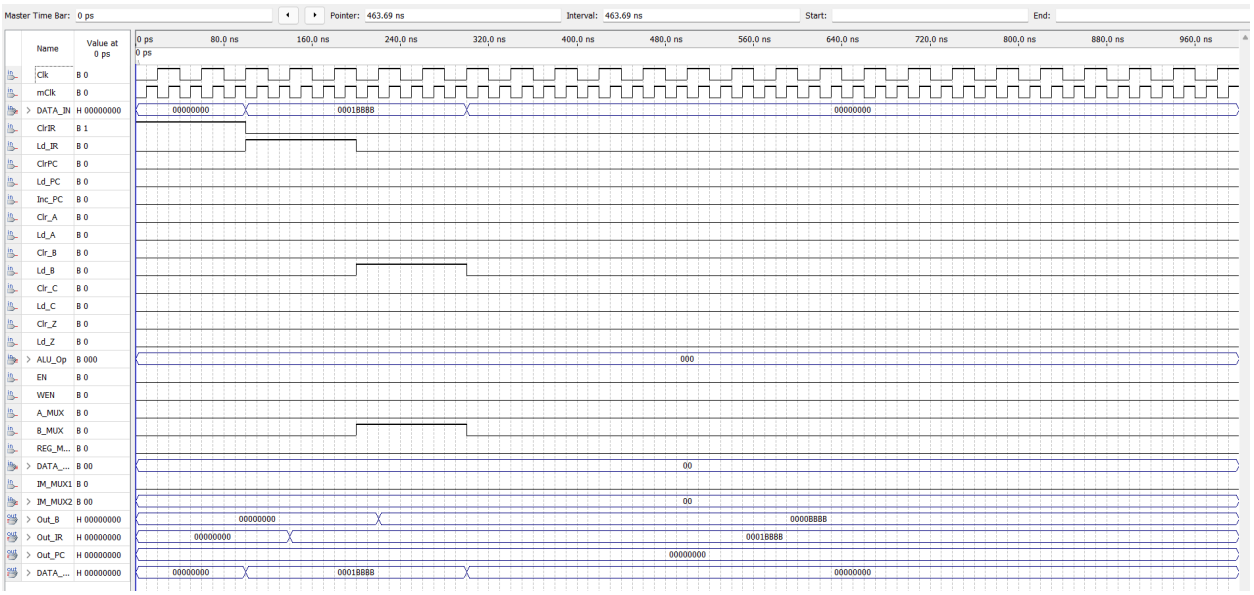

Figure 7: Waveform of the 32-bit CPU Data Path Design Component for operation: LDBI(load)

3)

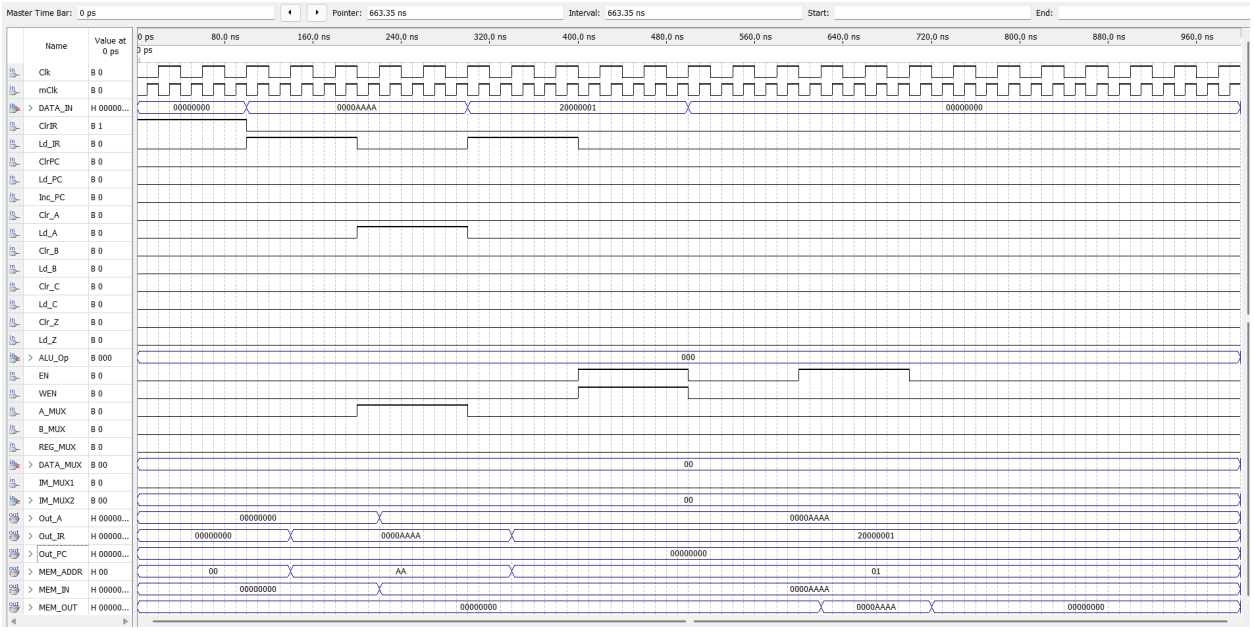

Figure 8: Waveform of the 32-bit CPU Data Path Design Component for operation: STA(storing)

4)

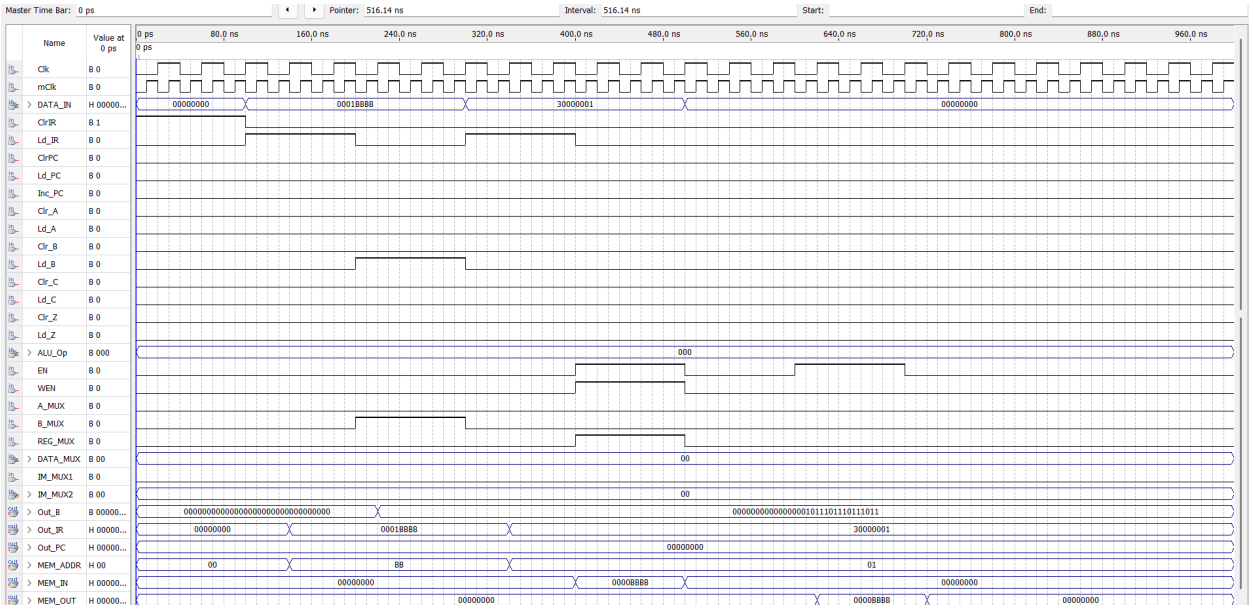

Figure 9: Waveform of the 32-bit CPU Data Path Design Component for operation: STB(storing)

5)

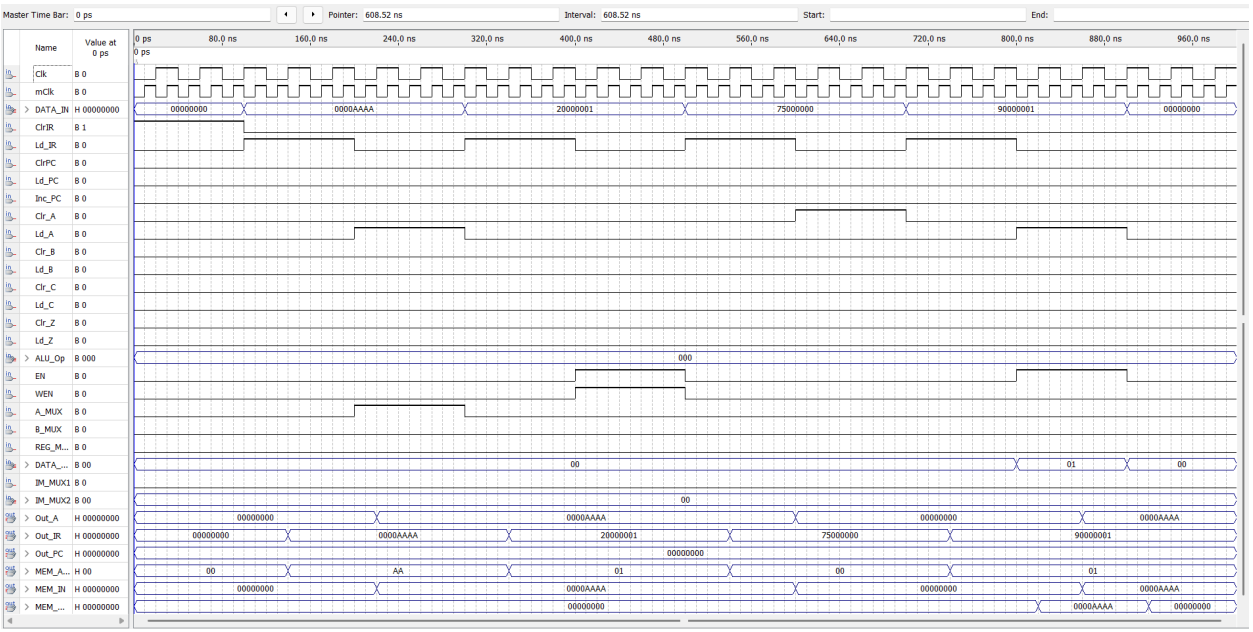

Figure 10: Waveform of the 32-bit CPU Data Path Design Component for operation: LDA(load)

6)

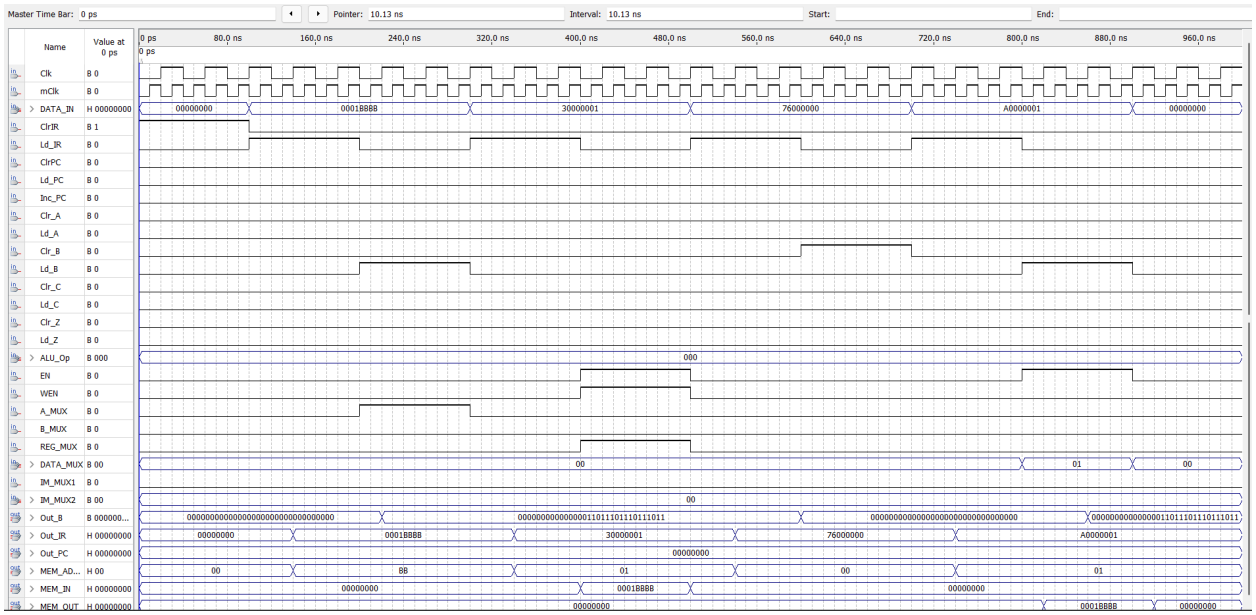

Figure 11: Waveform of the 32-bit CPU Data Path Design Component for operation: LDB(load)

7)

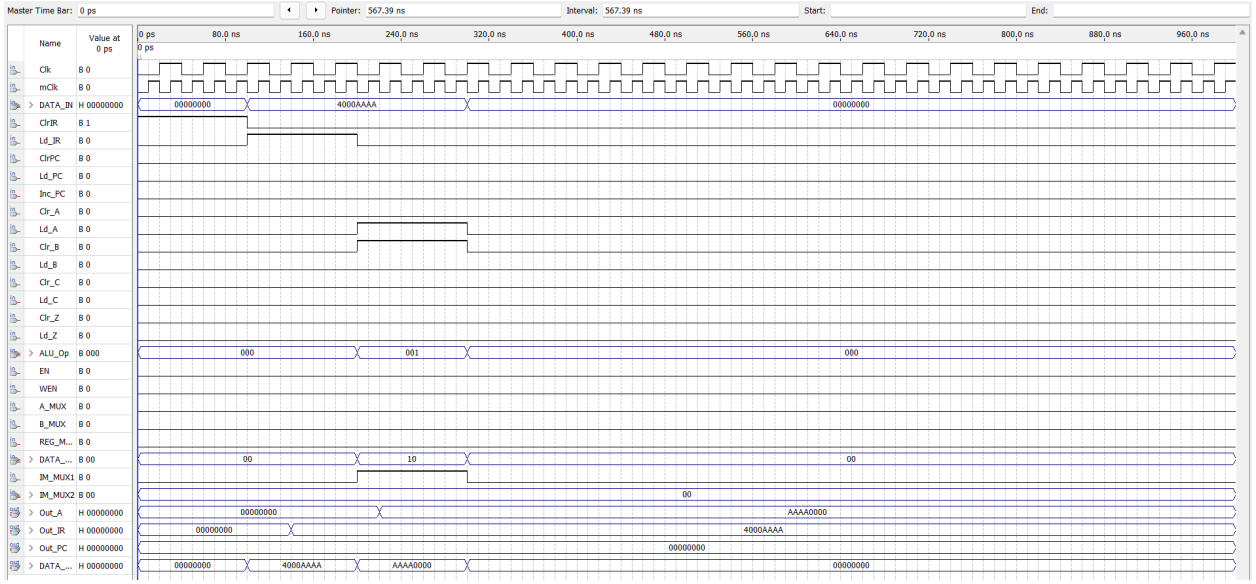

Figure 12 Waveform of the 32-bit CPU Data Path Design for operation: LUI(load-instr)

8)

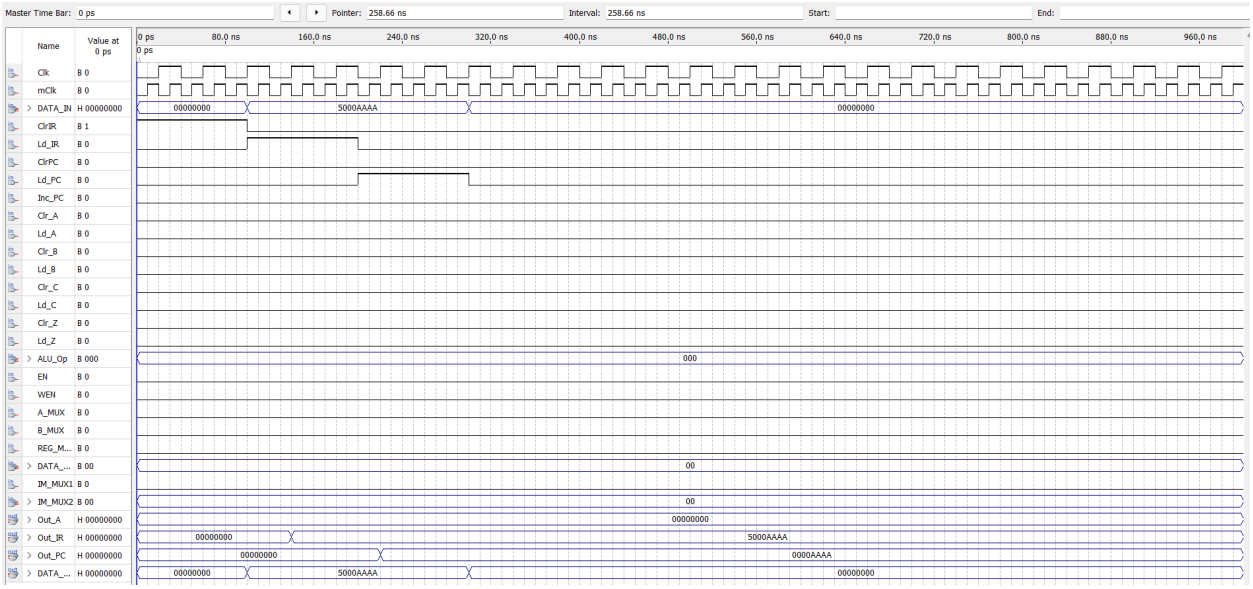

Figure 13: Waveform of the 32-bit CPU Data Path Design Component for operation: JMP

9)

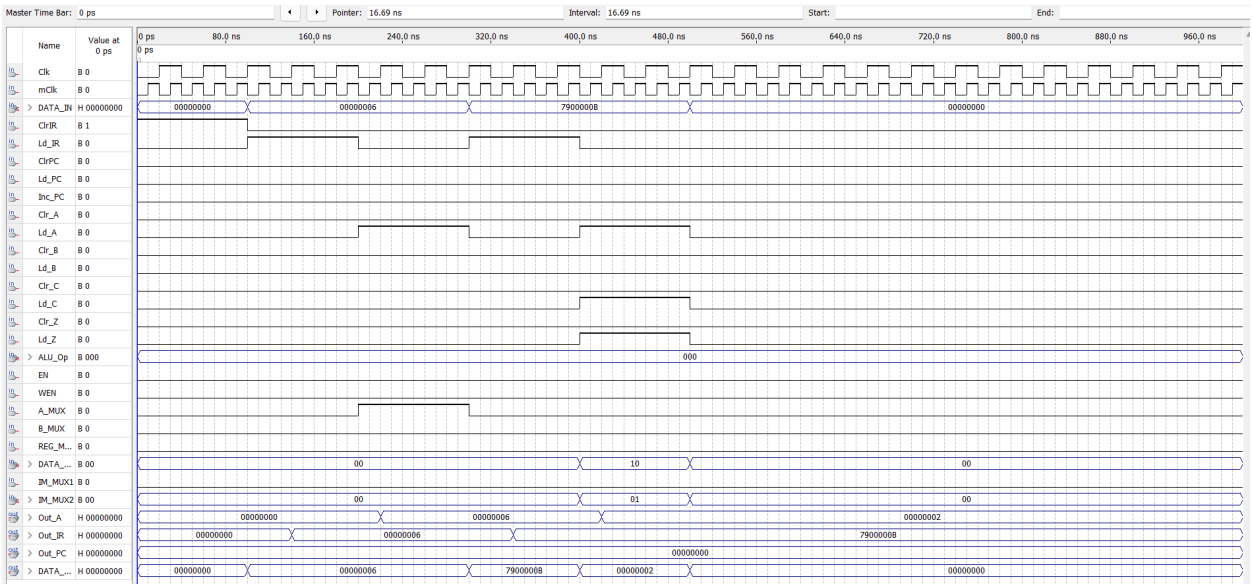

Figure 14: Waveform of the 32-bit CPU Data Path Design for operation:ANDI(and-intermediate)

10)

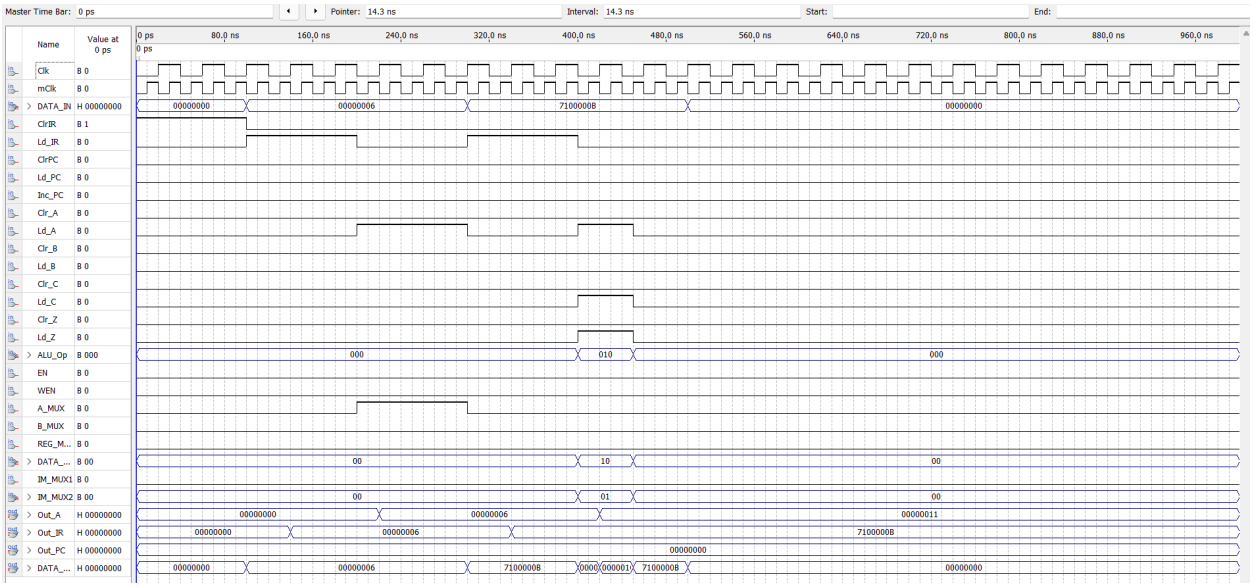

Figure 15: Waveform of the CPU Data Path Design for operation: ADDI(Add-intermediate)

11)

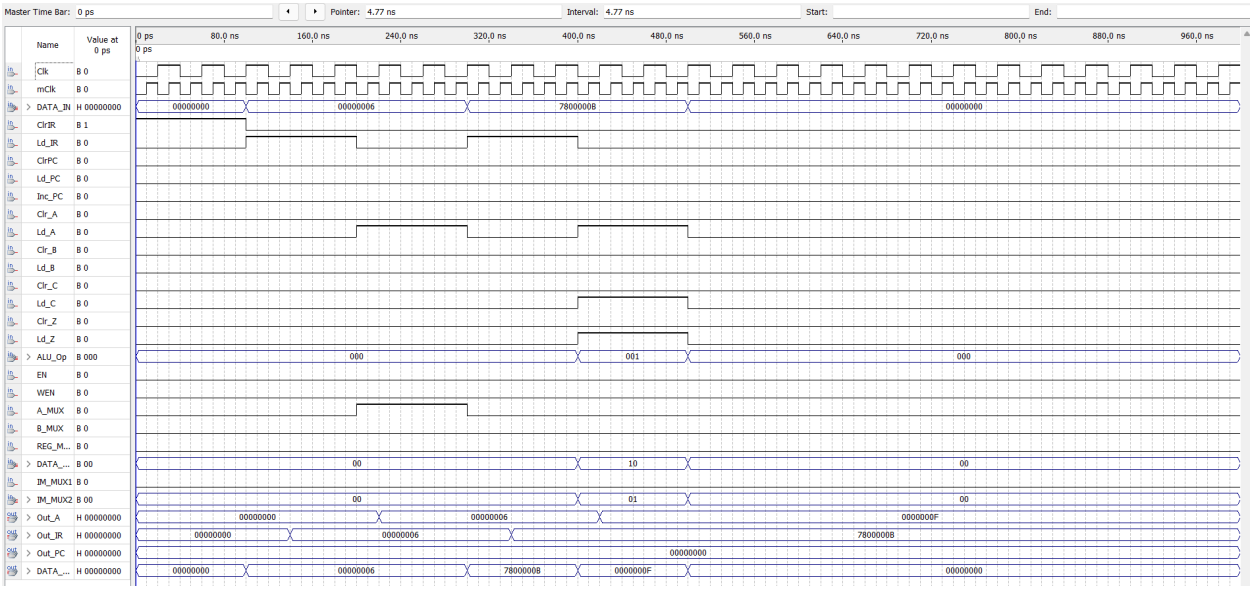

Figure 16: Waveform of the 32-bit CPU Data Path Design for operation: ORI(Or-intermediate)

12)

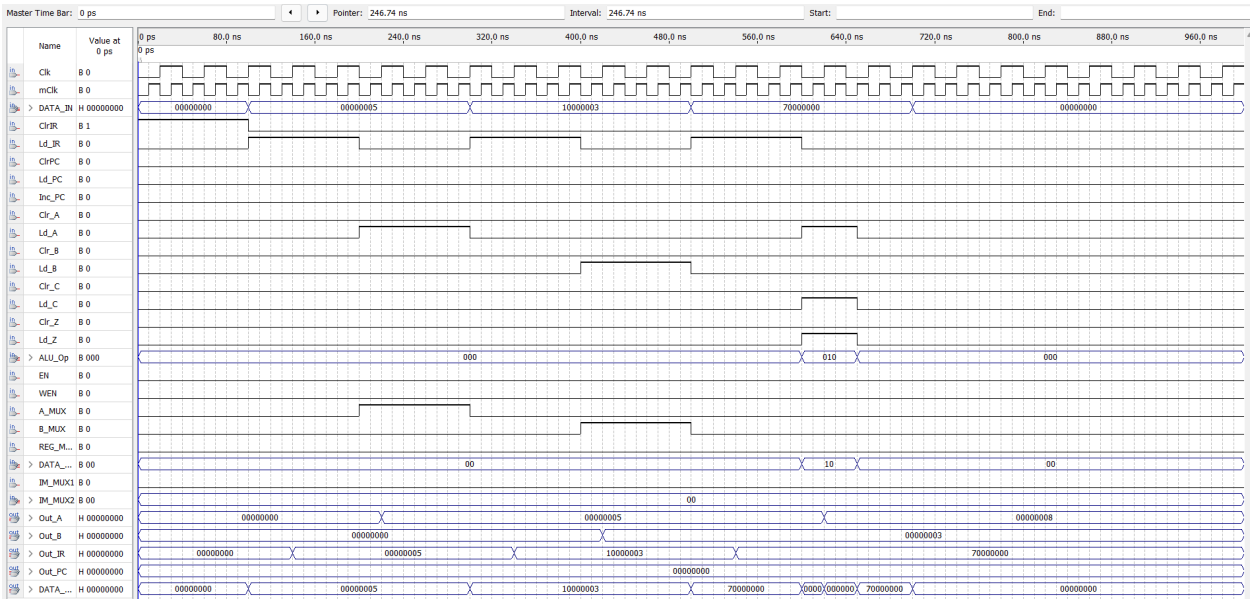

Figure 17: Waveform of the 32-bit CPU Data Path Design for operation: ADD

13)

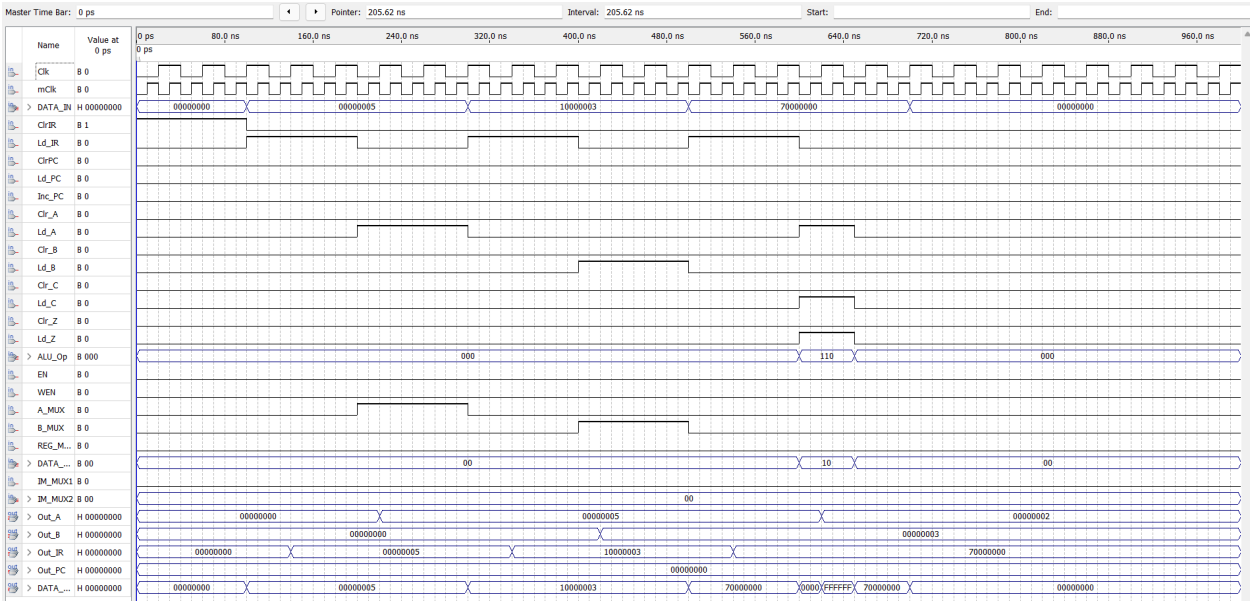

Figure 18: Waveform of the 32-bit CPU Data Path Design for operation: SUB

14)

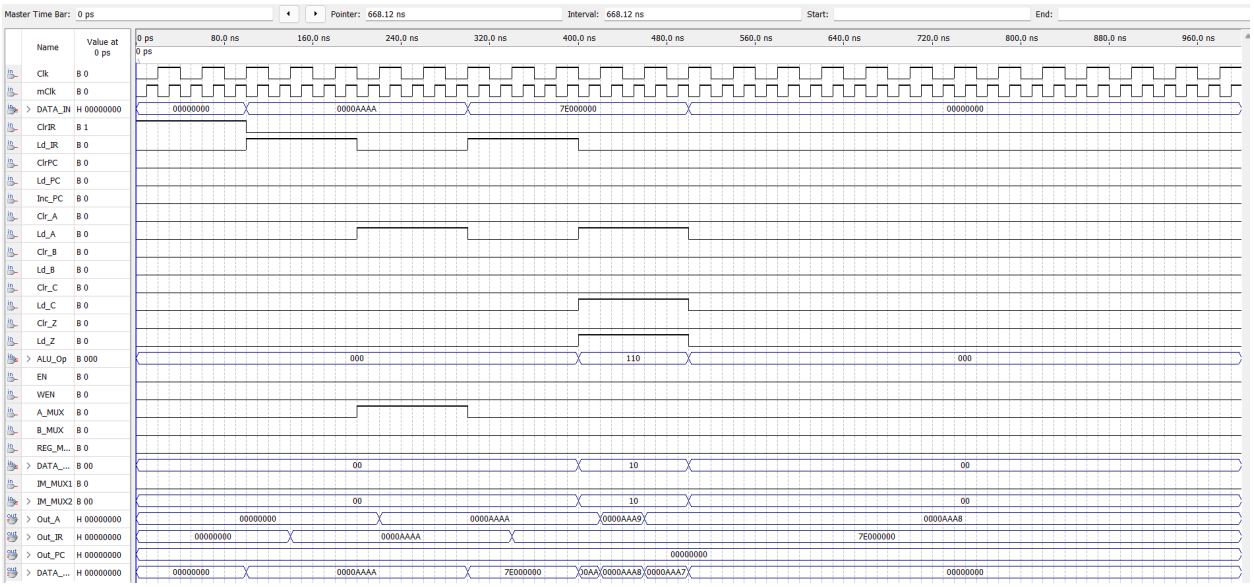

Figure 19: Waveform of the 32-bit CPU Data Path Design for operation: DECA(Decrement)

15)

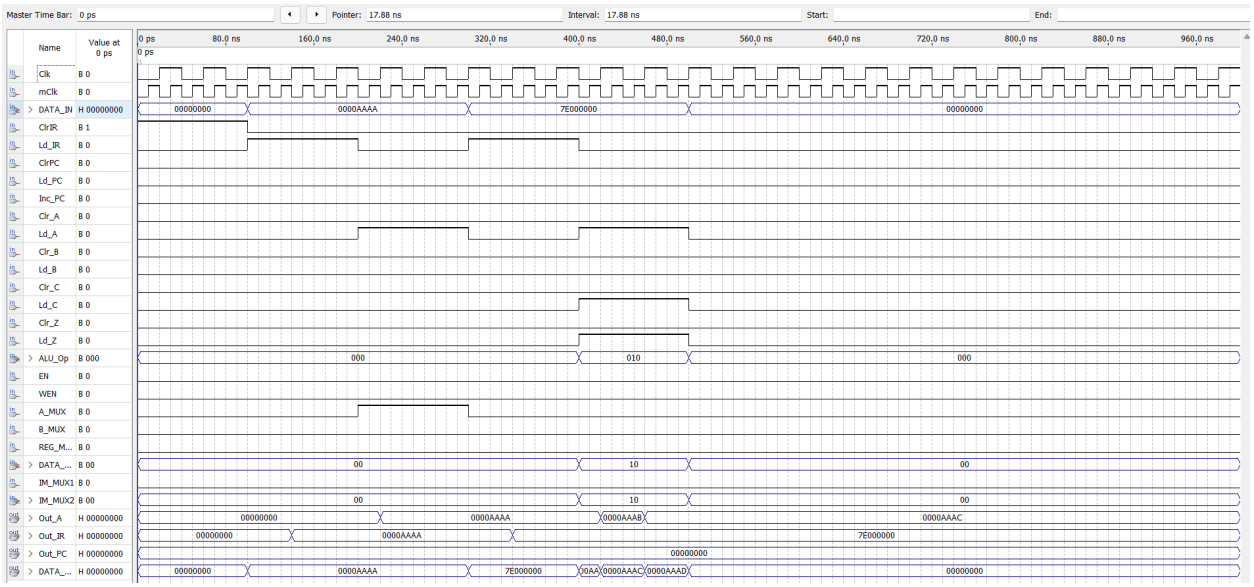

Figure 20: Waveform of the 32-bit CPU Data Path Design for operation: INCA(Increment)

16)

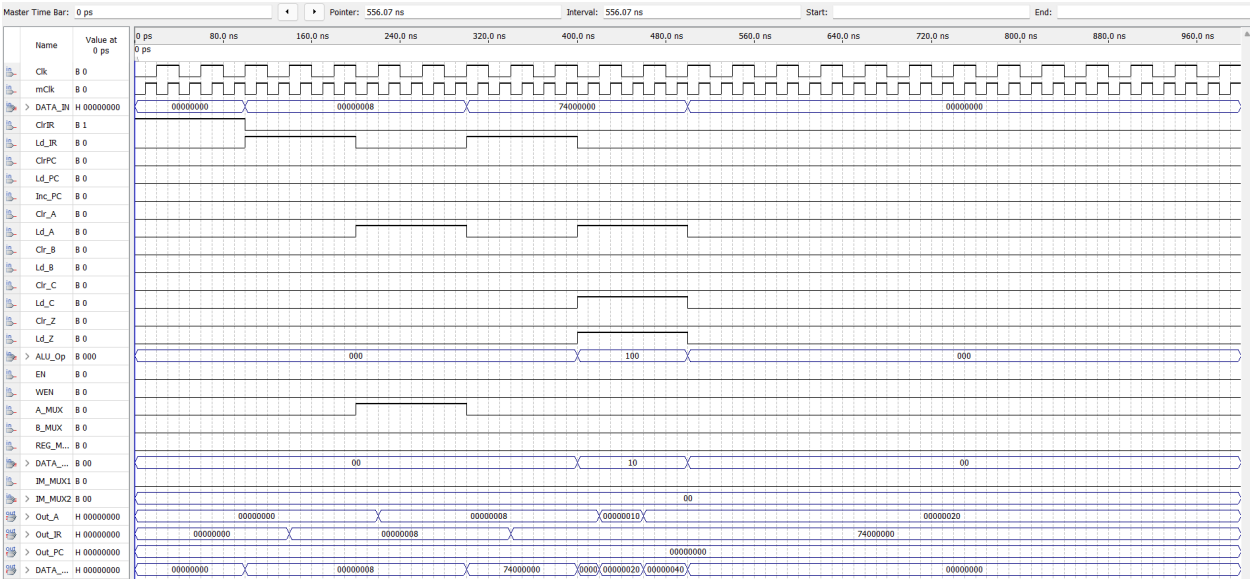

Figure 21: Waveform of the 32-bit CPU Data Path Design for operation: ROL(Rotate Left)

17)

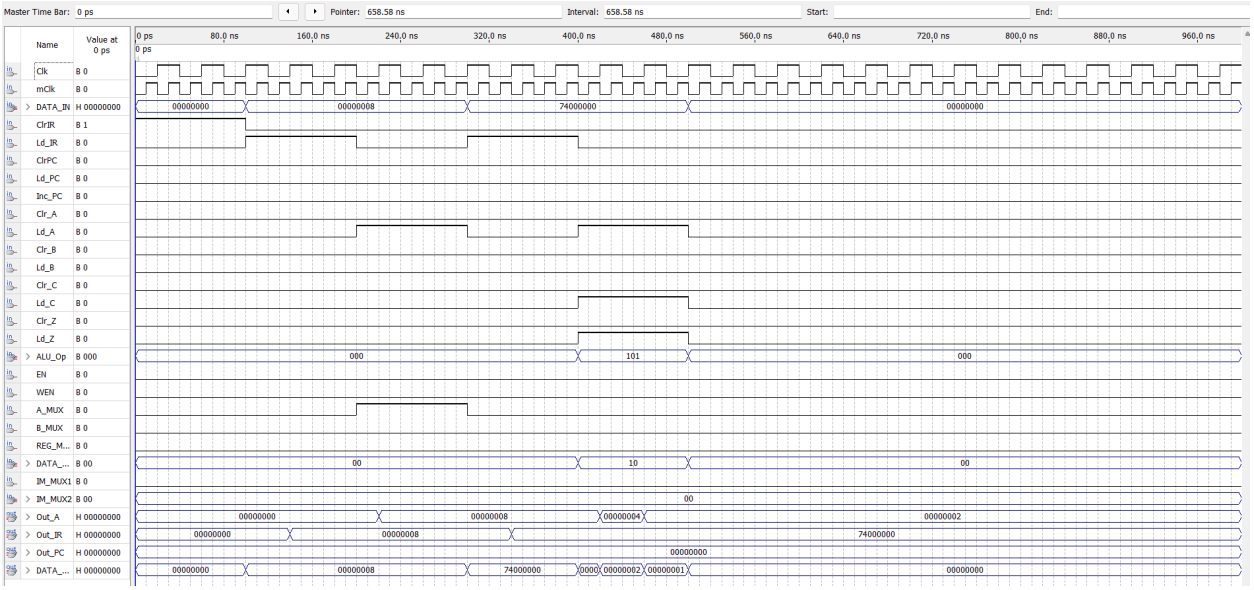

Figure 22: Waveform of the 32-bit CPU Data Path Design for operation: ROR(Rotate Right)

18)

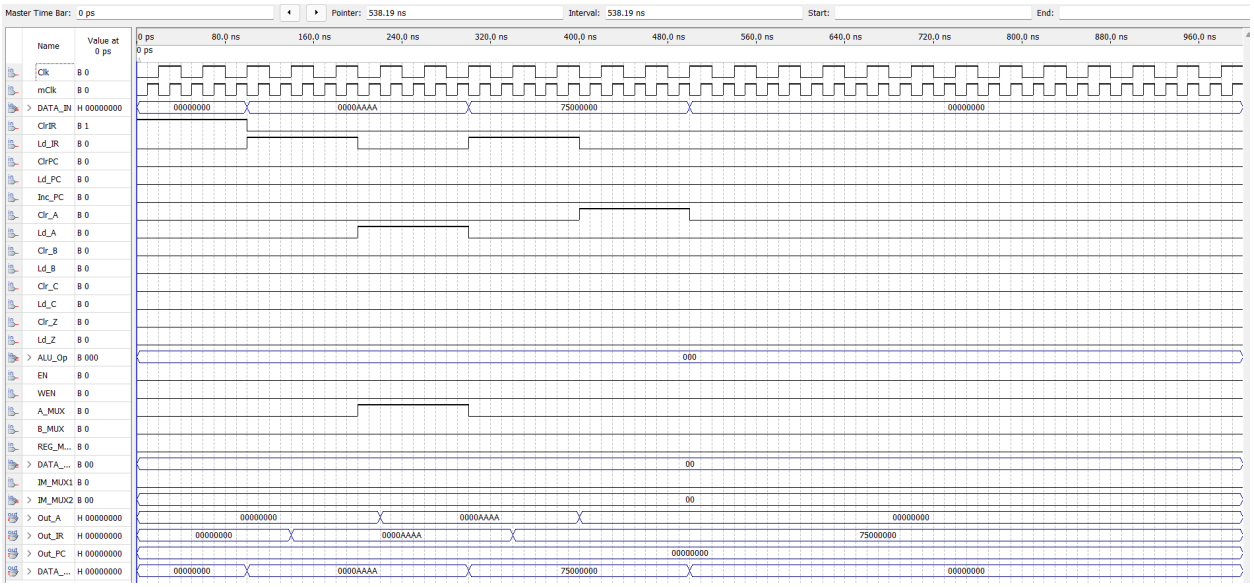

Figure 23: Waveform of the 32-bit CPU Data Path Design for operation: CLRA (Clear A reg)

19)

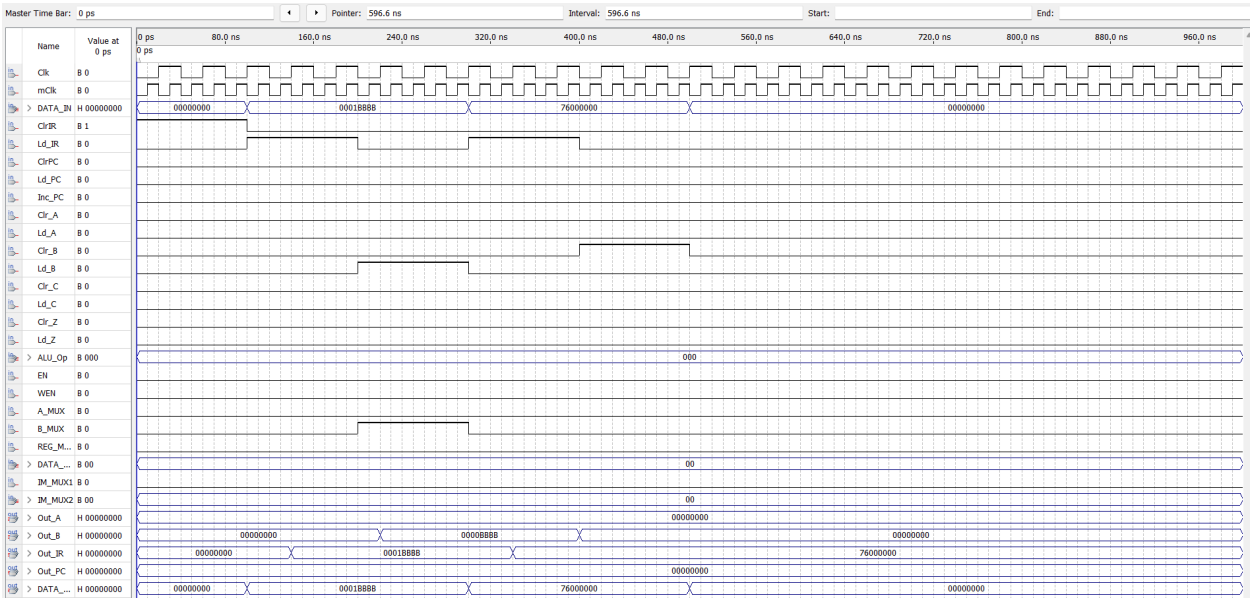

Figure 24: Waveform of the 32-bit CPU Data Path Design for operation: CLRB(Clear B reg)

Final END Questions:

```
--END Questions:
--1) a)INCA: The Increment Function is implemented by placing Reg A as the first ALU input, by loading and opening the MUX(IM_MUX1) And the other input for the ALU
--           is a single value '1'(by setting IM_MUX2 to 2). Reg A and '1' are added in the ALU(OpCode = 010)
--   b)ADDI: The Adding with intermediate function is implemented by enabling the mux and loading reg A as one of the input. The other input would be the DATA_IN
--           that is directly loaded to the IR reg, from there it makes it's way to the second input(where IM_MUX2 is set as 1). Reg A and the DATA_IN value in
--           Reg IR are then added (OpCode = 010)
--   c)LDBI: The Loading B with intermediate function is implemented by first loading IR reg with the DATA_IN value. Then the B_mux and load for Reg B is enabled
--           allowing the DATA_IN from IR Reg to make its way there, and thus the DATA_IN value is loaded into reg B .
--   d)LDA: The Load memory with A function takes the first DATA_IN input and enables the A_MUX and loades reg A with it. Now this function uses the Data Memory
--          unit; the reg A is used as the data_in for memory, and the next DATA_IN from the cpu is the specified address to where the contents of reg A will be
--          stored. The address(last 2 of the 8 hex) comes form the IR reg and the data memory unit is in writing mode(WEN=1 and EN=1), where the contents of reg A
--          will be written in the address. Later reg A is cleared and data_in is no longer in use. Now the same address is accessed by loading it in the IR reg
--          again, but this time it is in read mode(just EN=1), and the memory unit generates output(content of regA that was written previously).

 --2) The Maximum Reliable clock speed is determined by the delays that indicate the time required to execute the process. Most specifically the longest
    -- combinational delay from the time it takes to progress through the combinational logic gate circuit, that is the "Critcial Path Delay".
    -- The clock period, T_clk must be atleast as long as T_criticalpathdelay: Tclk >= T_criticalpathdelay
    -- The Reliable clock speed also includes the setup time, since data signals must be settled before the clock edge, and data must remain stable after the clock
    -- cycle ends(T_setup):
    -- Thus, the acceptable Maximum Reliable clock speed must include two factors, the critical path delay and setup time:
    -- T_clk = T_criticalpathdelay + T_setup

--3) The relaible limit for the data-path clock is(in frequency):
--   f_reliable_clk = 1 / (T_criticalpathdelay + T_setup)
--   f_reliable_clk = 1 / (20 ns + 1 ns) [The criticla path delay is assumed to be the current clock cycle that is successful(20ns); the setup time is assumed to
--                                        be 1ns as per usual cirumstances]
--   f_reliable_clk = 1 / (20 ns + 1 ns) = 1 / (21 ns) = 47.6 MHz
```

Figure 5 – 2: END Questions for Data Path Design Component