Design of a Simple CPU

Author: Mustafa Al-Shebeeb

Co-Author: Jeffrey Thompson

Instructor: Michael Barchett

Introduction to Digital Logic and Design Laboratory

CPE 271L, Section 6

Date Preformed:

11/30/2023

Institution: West Virginia University - Statler College of Engineering and Mineral Resources

Lane Department of Computer Science and Electrical Engineering

Table of Contents

Introduction:

In this culminating project, students are tasked with applying their acquired knowledge in digital logic and design by constructing a simple Central Processing Unit (CPU). This project comprises nine essential components: Accumulator (A), Instruction Register (IR), Control Unit (CU), Program Counter (PC), Arithmetic Logic Unit (ALU), Memory Address Register (MAR), Memory Data Register Input (MDRI), Memory Data Register Output (MDRO), and Random Access Memory (RAM).

The functionality of the CPU is synchronized with a clock, which drives its operations. The CPU begins its execution by reading an 8-bit instruction from the first memory address. The first three bits of the instruction dictate the operation to be performed, while the subsequent five bits represent a memory location. The three designated operations are LOADA (000), ADDA (001), and STOREA (010).

The LOADA operation involves loading the value at the specified memory address into the accumulator. Similarly, the ADDA operation entails adding the memory value to the accumulator. Lastly, the STOREA operation involves storing the current value of the accumulator into the specified memory location. The RAM is preprogrammed with specific addresses intended to execute these three fundamental commands. Consequently, the CPU autonomously progresses through each memory address, interpreting and executing the corresponding instructions.

This project serves as a practical application of the principles learned throughout the semester, challenging students to synthesize their knowledge and implement a functional CPU. The subsequent sections of this report will delve into the design, implementation, and evaluation of the CPU, providing a comprehensive overview of the undertaken project.

Hardware Description:

The DE10-Lite Field Programmable Gate Array board, or FPGA board, was the main hardware used throughout the CPE 271 lab course. The boards are circuits that can be programmed to perform many simple tasks using input switches/buttons and output LED lights and a 7-segment display that can also be used to display the output.

The board has an internal clock that can be used when performing many tasks to slow the passing of an input to the circuit until a clock edge. They have been utilized in many of the labs to create adders, subtractors, and memory. They are very versatile, able to be programmed to do a wide array of tasks that are of lower complexity and then reprogrammed with another code just as easily.

The main cons of FPGA boards are their cost and they use a lot of power so they can be very costly in price and power, especially in high quantities. It is very interesting how huge the potential is in such a small board able to be reprogrammed infinitely to perform a seemingly infinite amount of tasks. A whole extra semester of working with the board would still not be enough time to explore all the possibilities there are when programming an FPGA board.

Software Description:

Quartus Prime was the main software used all semester, and where most of the final project was performed. Quartus Prime is a platform that is capable of designing, analyzing, and programming FPGA boards and other programmable devices. The software can be used to create logic circuits or create a VHDL file to write a code, both can then be programmed into the FPGA board.

Another feature of Quartus is that the FPGA board is not necessary because it has waveform simulations that can be used to test the circuit/code inside of Quartus itself. Behavioral modeling and port mapping are useful tools while coding in Quartus that help save time and make a more efficient code for a circuit. It has very few downfalls in the scope of this course except when working on a personal laptop the simulation could not be performed because of needed license purchases.

Quartus has been used in almost every lab for multiple tasks such as creating encoders, decoders, adders, and subtractors. The final lab was almost completely performed inside of Quartus besides the diagrams. It was used to add the needed code to the given VHDL files such as the counter variable to the program counter code and port mapping variables from the other component files to the highest order file, the CPU.

The code was compiled and tested inside of Quartus Prime as well using no FPGA to test it. Instead, the waveform simulation inside Quartus was used to view the outputs of all the variables along the clock cycle. This is why Quartus is such a valuable and versatile program as it can ask the programmer and the programmed device with no need of an actual physical device.

Description of Each VHDL File:

The project is essentially divided into various VHDL files, each containing the code for a distinct component of the Central Processing Unit(CPU). These components include the Control Unit, Program Counter, and the crucial Arithmetic Logic Unit (ALU), along with Memory files responsible for addressing and registering input and output. Port-mapping statements, specifically port-map statements, are incorporated into each component file, playing a pivotal role in generating overall results.

Starting with the Program Counter code, it manages an 8-bit memory address to execute instructions. Notably, the Memory component accommodates only a 5-bit address, restricting the instructions it can process. The code outputs when there's a positive edge and a clock event equals 1, causing the counter to increase by 1.

In the ALU code, the component operates on 8 inputs, producing a 9-bit output. Operations encompass addition, subtraction, bitwise AND, bitwise OR, Output A, and Output B. Input B is derived from the Accumulator, while input A is sourced from the MDRI. Sequential statements, particularly if statements, dominate the ALU code.

The Accumulator acts as a crucial element providing an 8-bit output from the ALU. It functions in a loop-like manner, with the ALU and accumulator alternating as input and output when clk and load equal 1, facilitating the transfer of provided input to its output.

MDRI, an 8-bit register, retains a value read from memory and forwards it to the components for execution when needed.

The Control Unit, akin to the Program Counter, is a pivotal aspect of the project. It serves as a leader, orchestrating CPU component operations such as PC incrementation, memory/register access, and ALU operations. The Control Unit is modeled as a state machine with instructions representing different states, determining the active register and the subsequent state. Signals in the code ensure that all instructions are executed in the correct sequence.

The register file encompasses the code for various register components, including the accumulator, IR, MAR, MDRI, and MDRO. These components rely on the Register code for their port maps. The TwoToOneMux file precedes the MAR, selecting one out of two inputs based on Control Unit directives.

Problems Occurred and Solutions:

Most of the issues we faced while working on this project were in the Quartus Prime software. One issue was the 8-bit vectors (7 downto 0) would be initiated to a 5-bit(4 down to 0), We thought that we would need to individually connect each vector using 5 lines of code like this:

```
Marout(4)<=marToRamReadAddr(4);
Marout(3)<=marToRamReadAddr(3);
Marout(2)<=marToRamReadAddr(2);
Marout(1)<=marToRamReadAddr(1);
Marout(0)<=marToRamReadAddr(0);
```

After some troubleshooting it was found all that we had to do was to only use the first or last 5 bits of an 8 bit vector like this:

```
Marout (4 downto 0) <= marToRamReadAddr;
```

This cleared up our first issue for us but there was another main issue that we hit. As most of this project was performed in Quartus we met outside of lab times to do most of the work and it went smoothly up until the end. The waveform simulation would not run on our personal laptop without the proper licenses purchased. This was a major wall at the time as it stopped our progress in its tracks until the next time we could meet at lab time. We just had to look over and hope that we completed the code properly so it was able to be tested when we met in person. Luckily it was and the issue was resolved by simply logging onto Quartus on the lab computers to perform the waveform simulation. With the code working successfully the project was completed with only two major hiccups that were fixed through troubleshooting and patience.

Completed Code:

Added code is labeled with a Purple Square.

Program Counter:

```
--Program Counter Code
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
--Increments the program counter by 1 if there is a positive edge clock and increment =1
entity ProgramCounter is
port (
   output : out std_logic_vector(7 downto 0);
   clk : in std_logic;
   increment : in std_logic
);
end;

architecture behavior of ProgramCounter is
begin

process(clk,increment)
--Define a counter variable as an integer and initialize it to 0 (use variable counter: i
--INSERT CODE HERE

variable counter : integer := 0;

begin
   --Create an if statement to check for the condition of a positive edge clock and incre
   if (clk'event and clk = '1' and increment = '1') then
      --Increment counter variable by 1
      --INSERT CODE HERE
      counter := counter + 1;

      --Output the counter variable as a std logic vector of 8 bits
      --Use function conv_std_logic_vector(counter,8)
      --INSERT CODE HERE

      output <= conv_std_logic_vector(counter,8);

   end if;
end process;
end behavior;
```

Control Unit:

```
-- Control Unit Code
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity ControlUnit is

port (
    --Op code used for instructions (NOT the ALU Op)
    OpCode : in std_logic_vector(2 downto 0);
    --Clock Signal
    clk : in std_logic;
    --Load bits to basically turn components on and off at a given state
    ToALoad : out std_logic;
    ToMarLoad : out std_logic;
    ToIrLoad : out std_logic;
    ToMdriLoad : out std_logic;
    ToMdroLoad : out std_logic;
    ToPcIncrement : out std_logic := '0';
    ToMarMux : out std_logic;
    ToRamWriteEnable : out std_logic;
    --This is the ALU op code, look inside the ALU code to set this
    ToAluOp : out std_logic_vector (2 downto 0)
);

end;

architecture behavior of ControlUnit is
--Custom Data Type to Define Each State
type cu_state_type is (load_mar, read_mem, load_mdri, load_ir, decode,
                       ldaa_load_mar, ldaa_read_mem, ldaa_load_mdri, ldaa_load_a,
                       adaa_load_mar, adaa_read_mem, adaa_load_mdri, adaa_store_load_a,
                       staa_load_mdro, staa_write_mem,
                       increment_pc);

--Signal to hold current state
signal current_state : cu_state_type;

begin
--Defines the transitions in our state machine
process(clk)
begin

```

```
45  if (clk'event and clk = '1') then
46      case current_state is
47          --Increment the pc and fetch the instruction,
48          --Decode the instruction, use the diagram in
49          when increment_pc =>
50              current_state <= load_mar;
51          when load_mar =>
52          --INSERT CODE HERE
53          current_state <= read_mem;
54
55          when read_mem =>
56          current_state <= load_mdri;
57
58          when load_mdri =>
59          current_state <= load_ir;
60
61          when load_ir =>
62          current_state <= decode;
63
64
65
66
67          --Decode Opcode to determine Instruction
68          --Assign current state based on the opCode
69          when decode =>
70          --INSERT CODE HERE
71
72          case opCode is
73
74          when "000" =>
75          current_state <= ldaa_load_mar;
76
77          when "001" =>
78          current_state <= adaa_load_mar;
79
80          when "010" =>
81          current_state <= staa_load_mdro;
82
83          -- when other =>
84          -- current_state <= (others => increment_pc)
85
86          when others =>
87          current_state <= increment_pc;
88
```

```
89  end case;
90
91          --Instructions, need to determine the ne
92          --Follow the path to perform each instru
93          --Where the state machine needs to go to
94          ---Load instruction
95          when ldaa_load_mar =>
96              current_state <= ldaa_read_mem;
97              --INSERT CODE HERE
98
99          when ldaa_read_mem =>
100             current_state <= ldaa_load_mdri;
101
102         when ldaa_load_mdri =>
103         current_state <= ldaa_load_a;
104
105         when ldaa_load_a =>
106         current_state <= increment_pc;
107
108
109         --Add Instruction
110
111         when adaa_load_mar =>
112             current_state <= adaa_read_mem;
113
114         when adaa_read_mem =>
115             current_state <= adaa_load_mdri;
116
117         when adaa_load_mdri =>
118             current_state <= adaa_store_load_a;
119
120         when adaa_store_load_a =>
121             current_state <= increment_pc;
122
123
124
125         --Store Instruction
126         when staa_load_mdro =>
127             current_state <= staa_write_mem;
128
129         when staa_write_mem =>
130             current_state <= increment_pc;
131
132
```

```
end case;
end if;
end process;

-- Defines what happens at each state, set to '1' if
-- Set Op Code accordingly based on ALU, different fr
-- Keep in mind when ToMarMux = 0 , MAR is loaded fro

process(current_state)
begin

   ToALoad <= '0';
   ToMdroLoad <= '0';
   ToAluOp <= "000";

   case current_state is
      --Turns on the increment pc bit
      when increment_pc =>
         ToALoad <= '0';
         ToPcIncrement <= '1';
         ToMarMux <= '0';
         ToMarLoad <= '0';
         ToRamWriteEnable <= '0';
         ToMdriLoad <= '0';
         ToIrLoad <= '0';
         ToMdroLoad <= '0';
         ToAluOp <= "000";
      --Loads MAR with address from program counter

      when load_mar =>
      --INSERT CODE HERE

         ToALoad <= '0';
         ToPcIncrement <= '0';
         ToMarMux <= '0';
         ToMarLoad <= '1';
         ToRamWriteEnable <= '0';
         ToMdriLoad <= '0';
         ToIrLoad <= '0';
         ToMdroLoad <= '0';
         ToAluOp <= "000";

      --Reads Address located in MAR
      when read_mem =>
      --INSERT CODE HERE

         ToALoad <= '0';
         ToPcIncrement <= '0';
         ToMarMux <= '0';
         ToMarLoad <= '0';
         ToRamWriteEnable <= '0';
         ToMdriLoad <= '0';
         ToIrLoad <= '0';
         ToMdroLoad <= '0';
         ToAluOp <= "000";

      --Load Memory Data Register Input
      when load_mdri =>
      --INSERT CODE HERE

         ToALoad <= '0';
         ToPcIncrement <= '0';
         ToMarMux <= '0';
         ToMarLoad <= '0';
         ToRamWriteEnable <= '0';
         ToMdriLoad <= '1';
         ToIrLoad <= '0';
         ToMdroLoad <= '0';
         ToAluOp <= "000";

      --Loads the Instruction Register with instru
      when load_ir =>
      --INSERT CODE HERE

         ToALoad <= '0';
         ToPcIncrement <= '0';
         ToMarMux <= '0';
         ToMarLoad <= '0';
         ToRamWriteEnable <= '0';
         ToMdriLoad <= '0';
         ToIrLoad <= '1';
         ToMdroLoad <= '0';
         ToAluOp <= "000";

      --Decodes The current instruction (everythin
      when decode =>
```

```
220         --Decodes The current instruction (everything sho
221         when decode =>
222         --INSERT CODE HERE
223
224            ToALoad <= '0';
225            ToPcIncrement <= '0';
226            ToMarMux <= '0';
227            ToMarLoad <= '0';
228            ToRamWriteEnable <= '0';
229            ToMdriLoad <= '0';
230            ToIrLoad <= '0';
231            ToMdroLoad <= '0';
232            ToAluOp <= "000";
233
234         --Loads the MAR with address stored in IR
235         when ldaa_load_mar =>
236         --INSERT CODE HERE
237
238            ToALoad <= '0';
239            ToPcIncrement <= '0';
240            ToMarMux <= '1';
241            ToMarLoad <= '1';
242            ToRamWriteEnable <= '0';
243            ToMdriLoad <= '0';
244            ToIrLoad <= '0';
245            ToMdroLoad <= '0';
246            ToAluOp <= "000";
247
248         --Reads Data in memory retrieved from Address in
249         when ldaa_read_mem =>
250         --INSERT CODE HERE
251
252            ToALoad <= '0';
253            ToPcIncrement <= '0';
254            ToMarMux <= '0';
255            ToMarLoad <= '1';
256            ToRamWriteEnable <= '0';
257            ToMdriLoad <= '0';
258            ToIrLoad <= '0';
259            ToMdroLoad <= '0';
260            ToAluOp <= "000";
261
```

```
262         --Loads the Memory data Register Input with dat
263         when ldaa_load_mdri =>
264         --INSERT CODE HERE
265
266            ToALoad <= '0';
267            ToPcIncrement <= '0';
268            ToMarMux <= '0';
269            ToMarLoad <= '0';
270            ToRamWriteEnable <= '0';
271            ToMdriLoad <= '1';
272            ToIrLoad <= '0';
273            ToMdroLoad <= '0';
274            ToAluOp <= "000";
275
276         --Loads the accumulator with data held in MDRI
277         when ldaa_load_a =>
278         --INSERT CODE HERE
279
280            ToALoad <= '1';
281            ToPcIncrement <= '0';
282            ToMarMux <= '0';
283            ToMarLoad <= '0';
284            ToRamWriteEnable <= '0';
285            ToMdriLoad <= '0';
286            ToIrLoad <= '0';
287            ToMdroLoad <= '0';
288            ToAluOp <= "000";
289
290         --Loads the MAR with address held in IR
291         when adaa_load_mar =>
292         --INSERT CODE HERE
293
294            ToALoad <= '0';
295            ToPcIncrement <= '0';
296            ToMarMux <= '1';
297            ToMarLoad <= '1';
298            ToRamWriteEnable <= '0';
299            ToMdriLoad <= '0';
300            ToIrLoad <= '0';
301            ToMdroLoad <= '0';
302            ToAluOp <= "001";
303
```

```
304         --Reads Memory based on address in MAR
305         when adaa_read_mem =>
306         --INSERT CODE HERE
307
308             ToALoad <= '0';
309             ToPcIncrement <= '0';
310             ToMarMux <= '0';
311             ToMarLoad <= '0';
312             ToRamWriteEnable <= '0';
313             ToMdriLoad <= '0';
314             ToIrLoad <= '0';
315             ToMdroLoad <= '0';
316             ToAluOp <= "001";
317
318         --Loads MDRI with data just read from memo
319         when adaa_load_mdri =>
320         --INSERT CODE HERE
321
322             ToALoad <= '0';
323             ToPcIncrement <= '0';
324             ToMarMux <= '0';
325             ToMarLoad <= '0';
326             ToRamWriteEnable <= '0';
327             ToMdriLoad <= '1';
328             ToIrLoad <= '0';
329             ToMdroLoad <= '0';
330             ToAluOp <= "001";
331
332         --Loads accumulator with data in MDRI
333         when adaa_store_load_a =>
334         --INSERT CODE HERE
335
336             ToALoad <= '1';
337             ToPcIncrement <= '0';
338             ToMarMux <= '0';
339             ToMarLoad <= '0';
340             ToRamWriteEnable <= '0';
341             ToMdriLoad <= '0';
342             ToIrLoad <= '0';
343             ToMdroLoad <= '0';
344             ToAluOp <= "001";
345
346         --Loads MDRO with data to be written to mem
347         when staa_load_mdro =>
348         --INSERT CODE HERE
349
350             ToALoad <= '0';
351             ToPcIncrement <= '0';
352             ToMarMux <= '1';
353             ToMarLoad <= '1';
354             ToRamWriteEnable <= '0';
355             ToMdriLoad <= '0';
356             ToIrLoad <= '0';
357             ToMdroLoad <= '1';
358             ToAluOp <= "010";
359
360         --Writes to memory the data stored in MDRO
361         when staa_write_mem =>
362         --INSERT CODE HERE
363
364             ToALoad <= '0';
365             ToPcIncrement <= '0';
366             ToMarMux <= '0';
367             ToMarLoad <= '0';
368             ToRamWriteEnable <= '1';
369             ToMdriLoad <= '0';
370             ToIrLoad <= '0';
371             ToMdroLoad <= '0';
372             ToAluOp <= "010";
373
374     end case;
375     end process;
376 end behavior;
377
```

Simple CPU Code:

```
--Simple CPU template, This is the top level entity in
library ieee;
use ieee.std_logic_1164.all;

entity SimpleCPU_Template is
--These are the Outputs that can be displayed on the FP
--Depending on how you want to display each signal to t
port (
    clk : in std_logic;
    pcOut : out std_logic_vector(7 downto 0);
    marOut : out std_logic_vector (7 downto 0);
    irOutput : out std_logic_vector (7 downto 0);
    mdriOutput : out std_logic_vector (7 downto 0);
    mdroOutput : out std_logic_vector (7 downto 0);
    aOut : out std_logic_vector (7 downto 0);
    incrementOut : out std_logic
);

end;

architecture behavior of SimpleCPU_Template is
--Initialize our memory component
component memory_8_by_32
port(
    clk: in std_logic;
    Write_Enable: in std_logic;
    Read_Addr:  in std_logic_vector  (4 downto 0);
    Data_in:    in std_logic_vector  (7 downto 0);
    Data_out:   out std_logic_vector(7 downto 0)
);
end component;
--initialize the alu
component alu port (

    A : in std_logic_vector       (7 downto 0);
    B : in std_logic_vector       (7 downto 0);
    AluOp : in std_logic_vector      (2 downto 0);
    output : out std_logic_vector (7 downto 0)
);
end component;
--initialize the registers
component reg
port (
    input : in std_logic_vector      (7 downto 0);
    output : out std_logic_vector (7 downto 0);
    clk : in std_logic;
    load : in std_logic
);
end component;
--initialize the program counter
component ProgramCounter
port (
    increment : in std_logic;
    clk : in std_logic;
    output : out std_logic_vector (7 downto 0)
);
end component;
--initialize the mux
component TwoToOneMux
port (
    A : in std_logic_vector       (7 downto 0);
    B : in std_logic_vector       (7 downto 0);
    address : in std_logic;
    output : out std_logic_vector (7 downto 0)
);
end component;
--initialize the seven segment decoder
component sevenseg
port(
    i : in std_logic_vector(3 downto 0);
    o : out std_logic_vector(7 downto 0)
);
end component;

-- initialize control unit
component ControlUnit
port (
    OpCode : in std_logic_vector(2 downto 0);
    clk : in std_logic;
    ToALoad : out std_logic;
    ToMarLoad : out std_logic;
    ToIrLoad : out std_logic;
    ToMdriLoad : out std_logic;
    ToMdroLoad : out std_logic;
    ToPcIncrement : out std_logic;
    ToMarMux : out std_logic;
    ToRamWriteEnable : out std_logic;
    ToAluOp : out std_logic_vector (2 downto 0)
```

```
87      ToRamWriteEnable : out std_logic;
88      ToAluOp : out std_logic_vector (2 downto 0)
89  );
90  end component;
91
92  --The following signals will be used in your port map state
93
94  -- Connections : Need to be sorted
95  signal ramDataOutToMdri : std_logic_vector (7 downto 0);
96
97  -- MAR Multiplexer connections
98  signal pcToMarMux : std_logic_vector(7 downto 0);
99  signal muxToMar : std_logic_vector  (7 downto 0);
100
101 -- RAM connections
102 signal marToRamReadAddr : std_logic_vector   (4 downto 0);
103 signal mdroToRamDataIn : std_logic_vector (7 downto 0);
104
105 -- MDRI connections
106 signal mdriOut : std_logic_vector   (7 downto 0);
107
108 -- IR connection
109 signal irOut : std_logic_vector     (7 downto 0);
110
111 -- ALU / Accumulator connections
112 signal aluOut: std_logic_vector  (7 downto 0);
113 signal aToAluB : std_logic_vector   (7 downto 0);
114
115 -- Control Unit connections
116 signal cuToALoad : std_logic;
117 signal cuToMarLoad : std_logic;
118 signal cuToIrLoad : std_logic;
119 signal cuToMdriLoad : std_logic;
120 signal cuToMdroLoad : std_logic;
121 signal cuToPcIncrement : std_logic;
122 signal cuToMarMux : std_logic;
123 signal cuToRamWriteEnable : std_logic;
124 signal cuToAluOp : std_logic_vector (2 downto 0);
125 begin
126
127
128
129
130
```

```
132 --PORT MAP STATEMENTS GO HERE
133 -- Create port map statements for each co
134 -- RAM
135 --INSERT CODE HERE
136
137 ram: memory_8_by_32 port map (
138
139 clk => clk,
140 Write_Enable => cuToRamWriteEnable,
141 Read_Addr => marToRamReadAddr,
142 Data_in => mdroToRamDataIn,
143 Data_out => ramDataOutToMdri
144
145 );
146
147 -- Accumulator
148 --INSERT CODE HERE
149
150 --acc: alu port map (
151 --
152 --A => mdriOut,
153 --B => aToAluB,
154 --aluOp => cuToAluOp,
155 --output => aluOut
156 --
157 --);
158
159 acc: reg port map (
160
161 clk => clk,
162 load => cuToALoad,
163 input => aluOut,
164 output => aToAluB
165
166 );
167
168
169
170 -- ALU
171 --INSERT CODE HERE
172
173 --input b is connected to the out aToAlu
174
175 aluop: alu port map (
```

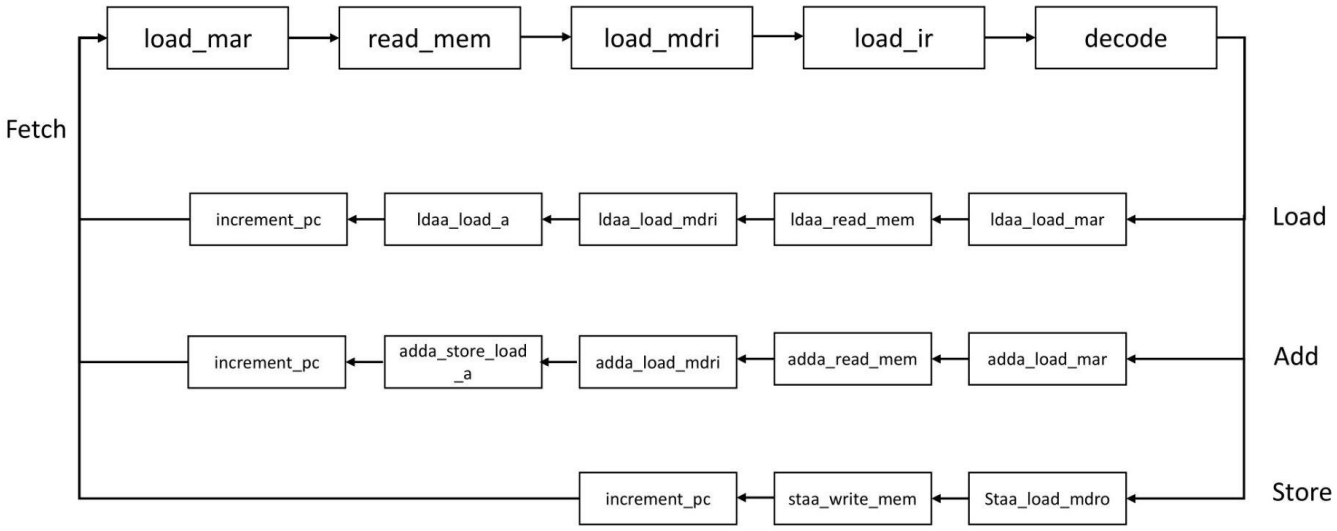
```
175 aluop: alu port map (
176
177 A => mdriOut,
178 B => aToAluB,
179 aluOp => cuToAluOp,
180 output => aluOut --this output of the
181
182 );
183
184 -- Program Counter
185 --INSERT CODE HERE
186
187 pc: ProgramCounter port map (
188
189 increment => cuToPcIncrement,
190 clk => clk,
191 output => pcToMarMux
192
193 );
194
195 -- Instruction Register
196 --INSERT CODE HERE
197
198 ir: reg port map (
199
200 input => mdriOut,
201 output => irOut,
202 clk => clk,
203 load => cuToIrLoad
204
205 );
206
207 -- MAR mux
208 --INSERT CODE HERE
209
210 mux: TwoToOneMux port map (
211
212 A => pcToMarMux,
213 B => irOut,
214 address => cuToMarMux,
215 output => muxToMar
216
217 );
218
```

```
220 -- Memory Access Register
221 --INSERT CODE HERE
222
223 mar: reg port map(
224
225 input => muxToMar,
226 output (4 downto 0) => marToRamReadAddr
227 clk => clk,
228 load => cuToMarLoad
229
230 );
231
232
233
234 -- Memory Data Register Input
235 mdri: reg port map(
236
237 input => ramDataOutToMdri,
238 output => mdriOut,
239 clk => clk,
240 load => cuToMdriLoad
241
242 );
243
244 -- Memory Data Register Output
245 --INSERT CODE HERE
246
247 mdro: reg port map(
248
249 input => aluOut,
250 output => mdroToRamDataIn,
251 clk => clk,
252 load => cuToMdroLoad
253
254 );
```

```
256 -- Control Unit
257 --INSERT CODE HERE
258
259 cu: ControlUnit port map(
260
261    OpCode => irOut(7 downto 5),
262    clk => clk,
263    ToALoad => cuToALoad,
264    ToMarLoad => cuToMarLoad,
265    ToIrLoad => cuToIrLoad,
266    ToMdriLoad => cuToMdriLoad,
267    ToMdroLoad => cuToMdroLoad,
268    ToPcIncrement => cuToPcIncrement,
269    ToMarMux => cuToMarMux,
270    ToRamWriteEnable => cuToRamWriteEnable,
271    ToAluOp => cuToAluOp
272
273 );
274
275 --REMAINING CODE GOES HERE
276 --Here is where you connect the port statement to the
277 --If you want to display the signal on LED's, just set
278 --If you want to send the signal to the seven segment
279 --Then map i=>signal, o=>port , keep in mind i needs t
280
```

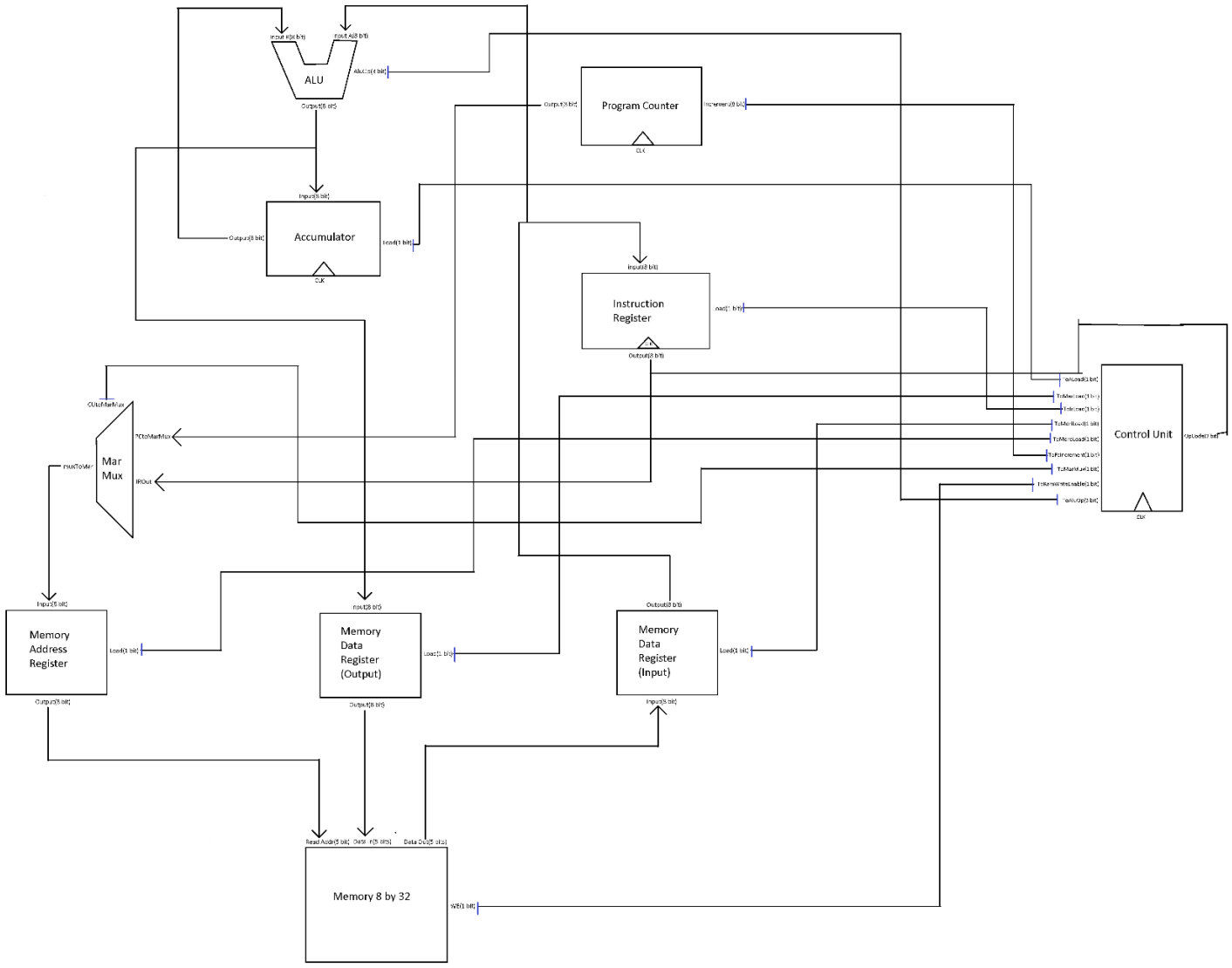
```
290
291 pcOut <= pcToMarMux;
292
293 marout (4 downto 0) <= marToRamReadAddr;
294
295 irOutput <= irOut;
296
297 mdriOutput <= mdriOut;
298
299 mdroOutput <= mdroToRamDataIn;
300
301 aOut <= aToAluB;
302
303 incrementOut <= cuToPcIncrement;
304
305
306
307 end behavior;
308
```

Finite State Machine Diagram:



Block Diagram with Components and Connections:

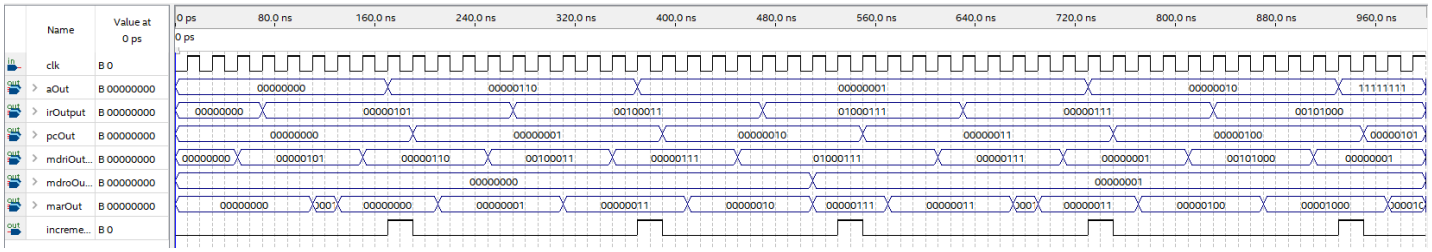
Base CPU Circuit



Results:

Simple CPU's Table

Memory Data					
RAM	**Binary**	**OpCode**	**Address**	**Value**	**Accumulator**
0	00000101	000 – loadA	00101 – 5	6	6
1	00100011	001 – addA	00011 – 3	7	13
2	01000111	010 – storeA	00111 – 7	13	13
3	00000111	000 – loadA	00111 – 7	13	13
4	00101000	001 – addA	01000 – 8	1	14
5	00000110	000-loadA	00110 - 6	20	20
6	00010100	000-loadA	10100 - 20	9	9
7	00001101	000-loadA	01101 - 13	0	0
8	00000001	000-loadA	00001 - 1	3	3



Waveform Simulation

Conclusion:

This project effectively incorporates fundamental concepts acquired throughout the semester, such as ALU, memory, and finite state machines, to construct a "simple" CPU. Despite successfully completing the project, the intricacy and complexity of the CPU remain quite challenging, especially for an introductory digital logic design course. Enhancing the CPU could involve expanding its operational capabilities or incorporating a user interface for executing desired tasks.

Appendix:

ALU:

```
--Arithmetic Logic Unit Code
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
-- 8 bit operands and output
ENTITY alu IS
PORT(
A : in std_logic_vector       (7 downto 0);
B : in std_logic_vector       (7 downto 0);
AluOp : in std_logic_vector      (2 downto 0);
output : out std_logic_vector (7 downto 0)
);

end alu;


-- decode op code, perform operation,
architecture behavior of alu is
begin
process(A,B,AluOp)
begin
if(AluOp="000") then output<=(A+B);
elsif(AluOp="001") then output<=(A-B);
elsif(AluOp="010") then output<=(A and B);
elsif(AluOp="011") then output<= (A or B);
elsif(AluOp="100") then output<= B;
elsif(AluOp="101") then output<= A;

end if;
end process;
end;

```

Register:

```
--Register component for CPU
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity reg is
port (
    input : in std_logic_vector      (7 downto 0);
    output : out std_logic_vector (7 downto 0);
    clk : in std_logic;
    load : in std_logic
);
end;

architecture behavior of reg is
begin

process(clk,load)
begin
   if (clk'event and clk = '1' and load = '1') then
      output <= input;
   end if;
end process;
end behavior;

```

Memory:

```
-- 8 By 32 Memory Array
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity memory_8_by_32 is

Port(
    clk:     in std_logic;
    Write_Enable: in std_logic;
    Read_Addr:  in std_logic_vector  (4 downto 0);
    Data_in:    in std_logic_vector  (7 downto 0);
    Data_out:   out std_logic_vector(7 downto 0));
    end memory_8_by_32;

    architecture behavior of memory_8_by_32 is
    type ram_type is array(0 to 31) of std_logic_vector(7 downto 0);
    --instructions / data go into memory here
    signal Z: ram_type:=("00000101","00100011","01000111","00000111",
    "00101000","00000110","00010100","00001101","00000001","10110100",
    "10001010","10101010","10101001","00000000","10100101","01010101",
    "10101110","10110100","10001010","10101010","10101001","00000000",
    "10100101","01010101","10101110","10110100","10001010","10101010",
    "10101001","00000000","10100101","01010101");
    Begin
    Process(clk,Read_Addr, Data_in, Write_Enable)
    Begin
    --Read from memory
    if(clk'event and clk='1' and Write_Enable='0') then
    Data_out<=Z(conv_integer(Read_Addr));
    --Write to Memory
    elsif(clk'event and clk='1' and Write_Enable='1') then
    Z(conv_integer(Read_Addr))<=Data_in;
    end if;
    end process;
    end;

```

Two-To-One Mux:

```
--Mux used to create a shared connection between PC an
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity TwoToOneMux is
port (
    A : in std_logic_vector       (7 downto 0);
    B : in std_logic_vector       (7 downto 0);
    address : in std_logic;
    output : out std_logic_vector (7 downto 0)
);
end;

architecture behavior of TwoToOneMux is
begin

process(A,B,address)
begin
    if (address='0') then
    output <= A;
    elsif(address='1') then
    output <= B;
    end if;
end process;
end behavior;

```

Seven Segment Display:

```
1  --Seven Segment Display, keep in mind the output he
2  --The CPU component outputs, when connecting the pi
3  library ieee;
4  use ieee.std_logic_1164.all;
5
6  entity sevenseg is
7  port(
8      i : in std_logic_vector(3 downto 0);
9      o : out std_logic_vector(7 downto 0)
10 );
11 end sevenseg;
12
13 architecture logic of sevenseg is
14 begin
15 o <= "00000001" when i="0000" else
16     "01001111" when i="0001" else
17     "00010010" when i="0010" else
18     "00000110" when i="0011" else
19     "01001100" when i="0100" else
20     "00100100" when i="0101" else
21     "00100000" when i="0110" else
22     "00001111" when i="0111" else
23     "00000000" when i="1000" else
24     "00000100" when i="1001" else
25     "00001000" when i="1010" else
26     "01100000" when i="1011" else
27     "00110001" when i="1100" else
28     "01000010" when i="1101" else
29     "00110000" when i="1110" else
30     "00111000" when i="1111";
31 end;
32
```