**Final Experiment- Design of a Simple CPU**

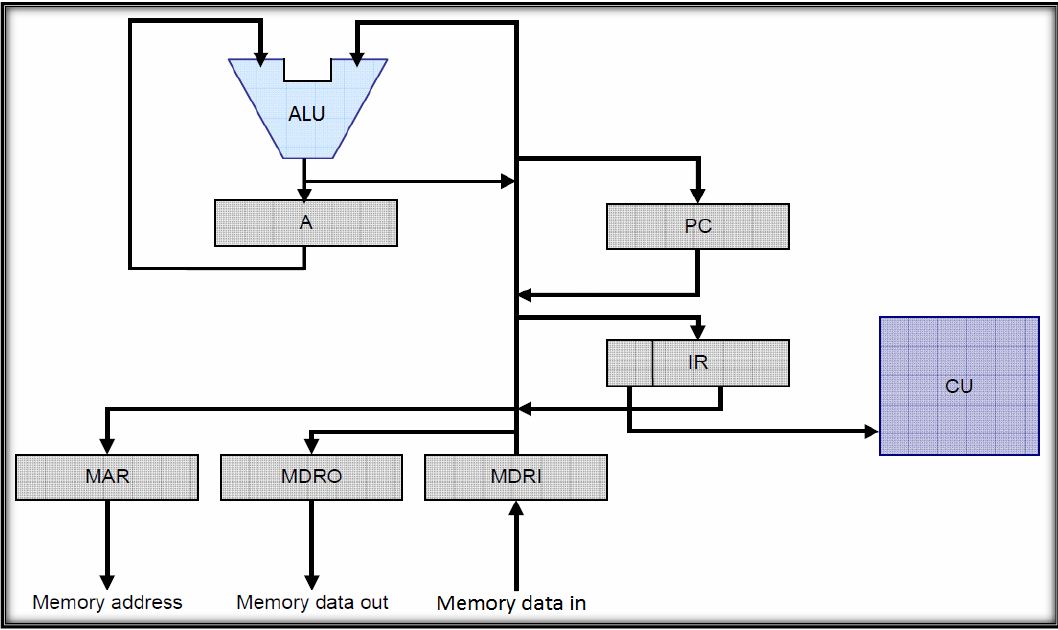
**By:** David Ghaphery and Olujide Jacobs

**Date Conducted:** November 28-December 9, 2013

**Introduction:**

In this experiment we were to create a simple Central Processing Unit, or CPU. A CPU is a very important part of a computer; it is where most every operation is carried out including arithmetic, logical, and input & output operations. Creating a CPU includes creating the various parts of a CPU including an arithmetic & logic unit (ALU), a control unit (CU), a program counter (PC), an instruction register (IR), a memory data in (MDRI), a memory data out (MDRO), a memory address register (MAR), and an accumulator (A). This experiment also required writing instructions fetch sequences for various tasks performed by the CPU, such as adding, storing, and loading data. Overall, the task of creating a CPU is somewhat complex, so a large portion of the VHDL code was provided to us.

**CPU Diagram:**

****

The above diagram is that of a CPU. The arrows shown describe the steps in which the CPU operates; these arrows are what our instruction fetch sequences were based on.

**Experiment:**

**Part 1-Simple CPU Template:**

The first of the two codes that we were required to display in the lab report was that of the Simple CPU Template. This code is shown below:

--Simple CPU template, This is the top level entity in your project

library ieee;

use ieee.std\_logic\_1164.all;

entity SimpleCPU\_Template is

--These are the Outputs that can be displayed on the FPGA, More port statements may be necessary,

--Depending on how you want to display each signal to the FPGA

port (

clk : in std\_logic;

pcOut : out std\_logic\_vector(0 to 7);

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;

--inialize 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 Pc

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(0 to 7)

);

end component;

--Inialize 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)

);

end component;

--The following signals will be used in your port map statements, don't use the port variables in your port maps

-- Connections : Need to be sorted

signal ramDataOutToMdri : std\_logic\_vector (7 downto 0);

-- MAR Multiplexer connections

signal pcToMarMux : std\_logic\_vector(7 downto 0);

signal muxToMar : std\_logic\_vector (7 downto 0);

-- RAM connections

signal marToRamReadAddr : std\_logic\_vector (4 downto 0);

signal mdroToRamDataIn : std\_logic\_vector (7 downto 0);

-- MDRI connections

signal mdriOut : std\_logic\_vector (7 downto 0);

-- IR connection

signal irOut : std\_logic\_vector (7 downto 0);

-- ALU / Accumulator connections

signal aluOut: std\_logic\_vector (7 downto 0);

signal aToAluB : std\_logic\_vector (7 downto 0);

-- Control Unit connections

signal cuToALoad : std\_logic;

signal cuToMarLoad : std\_logic;

signal cuToIrLoad : std\_logic;

signal cuToMdriLoad : std\_logic;

signal cuToMdroLoad : std\_logic;

signal cuToPcIncrement : std\_logic;

signal cuToMarMux : std\_logic;

signal cuToRamWriteEnable : std\_logic;

signal cuToAluOp : std\_logic\_vector (2 downto 0);

begin

-- RAM

inst1: memory\_8\_by\_32 port map(clk => clk, Write\_Enable => cuToRamWriteEnable, Read\_Addr => marToRamReadAddr,

Data\_in => mdroToRamDataIn, Data\_out => ramDataOutToMdri);

-- Accumulator

inst2: reg port map(clk => clk, input => aluOut, load => cuToALoad, output => aToAluB);

-- ALU

inst3: alu port map(A => mdriOut, B => aToAluB, aluOp => cuToAluOp, output => aluOut);

-- Program Counter

inst4: pc port map(clk => clk, increment => cuToPcIncrement, output => pcToMarMux);

-- Instruction Register

inst5: reg port map(clk => clk, input => mdriOut, load => cuToirLoad, output => irOut);

-- MAR mux

inst6: twotoOneMux port map(A => pctoMarMux, B => "000"&irOut(4 downto 0) , address => cutomarmux, output => muxToMar);

-- Memory Access Register

inst7: reg port map(clk => clk, input => muxToMar, load => cuToMarLoad, output(4 downto 0) =>marToRamReadAddr );

-- Memory Data Register Input

inst8: reg port map(clk => clk, input => ramDataOutToMdri, load => cutoMdriLoad, output => mdriOut);

-- Memory Data Register Output

inst9: reg port map(clk => clk, input => aluOut, load => cutoMdroLoad, output => mdroToRamDataIn );

-- Control Unit

inst10: controlunit port map(clk => clk, opcode => irOut(7 downto 5), toaLoad => cutoALoad, tomarLoad => cutoMarLoad,

toirLoad => cutoIrLoad, tomdriLoad => cutoMdriLoad, tomdroLoad => cutoMdroLoad, topcIncrement => cutoPcIncrement,

tomarMux => cutoMarMux, toramWriteEnable => cutoRamWriteEnable, toaluOp => cutoAluOp);

--Here is where you connect the port statement to the matching signal to display it on the FPGA

--If you want to display the signal on LED's, just set it to the port statement port<=signal;

--If you want to send the signal to the seven segment display, initialize an instance of the sevenseg

--Then map i=>signal, o=>port , keep in mind i needs to be 4 bits and o 8 bits

--pcOut <= pcToMarMux;

mdriOutput <= mdriOut;

aout <= aToAluB;

inst11: sevenseg port map(i => pctoMarMux(3 downto 0), o => pcout);

end behavior;

This portion of the code is the main code that we used. It contained all the codes that we wrote including the codes for the CU, RAM, Seven Segment, ALU, Accumulator, PC, IR, MDRI, MDRO, and the MAR. This code creates the inputs and outputs for each component of the CPU, as shown in the CPU diagram. This portion of the code also contained instructions for mapping these different steps to the Altera DE2 board. In addition, it contained all of the signals that were used to connect the previous steps to the next steps of the CPU.

**Part 2- Control Unit Code:**

We were also required to provide the VHDL code for our control unit (CU); this code is given below:

-- 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

if (clk'event and clk = '1') then

case current\_state is

--Increment the pc and fetch the instruction, then load the IR with the fetched instruction

--Decode the instruction, use the diagram in the handout to determine the next states

when increment\_pc =>

current\_state <= load\_mar;

when load\_mar =>

current\_state <= read\_mem;

when read\_mem =>

current\_state <= load\_mdri;

when load\_mdri =>

current\_state <= load\_ir;

when load\_ir =>

current\_state <= decode;

--Decode Opcode to determine Instruction

--Assign current state based on the opCode

when decode =>

if OpCode = "000" then

current\_state <= ldaa\_load\_mar;

elsif Opcode = "001" then

current\_state <= adaa\_load\_mar;

elsif Opcode = "010" then

current\_state <= staa\_load\_mdro;

end if;

--Instructions, need to determine the next state to implement each instruction

--Follow the path to perform each instruction as described in the handout, and determine

--Where the state machine needs to go to implement the instruction

---Load instruction

when ldaa\_load\_mar =>

current\_state <= ldaa\_read\_mem;

when ldaa\_read\_mem =>

current\_state <= ldaa\_load\_mdri;

when ldaa\_load\_mdri =>

current\_state <= ldaa\_load\_a;

when ldaa\_load\_a =>

current\_state <= increment\_pc;

--Add Instruction

when adaa\_load\_mar =>

current\_state <= adaa\_read\_mem;

when adaa\_read\_mem =>

current\_state <= adaa\_load\_mdri;

when adaa\_load\_mdri =>

current\_state <= adaa\_store\_load\_a;

when adaa\_store\_load\_a =>

current\_state <= increment\_pc;

--Store Instruction

when staa\_load\_mdro =>

current\_state <= staa\_write\_mem;

when staa\_write\_mem =>

current\_state <= increment\_pc;

end case;

end if;

end process;

-- Defines what happens at each state, set to '1' if we want that component to be on

-- Set Op Code accordingly based on ALU, different from the instruction op code, look at the actual ALU code

-- Keep in mind when ToMarMux = 0 , MAR is loaded from PC address, when ToMarMux = 1, MAR is loaded with IR address

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 =>

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 =>

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 =>

ToALoad <= '0';

ToPcIncrement <= '0';

ToMarMux <= '0';

ToMarLoad <= '0';

ToRamWriteEnable <= '0';

ToMdriLoad <= '1';

ToIrLoad <= '0';

ToMdroLoad <= '0';

ToAluOp <= "000";

--Loads the Instruction Register with instruction fetched from Memory

when load\_ir =>

ToALoad <= '0';

ToPcIncrement <= '0';

ToMarMux <= '0';

ToMarLoad <= '0';

ToRamWriteEnable <= '0';

ToMdriLoad <= '0';

ToIrLoad <= '1';

ToMdroLoad <= '0';

ToAluOp <= "000";

--Decodes The current instruction (everything should be off for this)

when decode =>

ToALoad <= '0';

ToPcIncrement <= '0';

ToMarMux <= '0';

ToMarLoad <= '0';

ToRamWriteEnable <= '0';

ToMdriLoad <= '0';

ToIrLoad <= '0';

ToMdroLoad <= '0';

ToAluOp <= "000";

--Loads the MAR with address stored in IR

when ldaa\_load\_mar =>

ToALoad <= '0';

ToPcIncrement <= '0';

ToMarMux <= '1';

ToMarLoad <= '1';

ToRamWriteEnable <= '0';

ToMdriLoad <= '0';

ToIrLoad <= '0';

ToMdroLoad <= '0';

ToAluOp <= "000";

--Reads Data in memory retrieved from Address in MAR

when ldaa\_read\_mem =>

ToALoad <= '0';

ToPcIncrement <= '0';

ToMarMux <= '0';

ToMarLoad <= '0';

ToRamWriteEnable <= '0';

ToMdriLoad <= '0';

ToIrLoad <= '0';

ToMdroLoad <= '0';

ToAluOp <= "000";

--Loads the Memory data Register Input with data read from memory

when ldaa\_load\_mdri =>

ToALoad <= '0';

ToPcIncrement <= '0';

ToMarMux <= '0';

ToMarLoad <= '0';

ToRamWriteEnable <= '0';

ToMdriLoad <= '1';

ToIrLoad <= '0';

ToMdroLoad <= '0';

ToAluOp <= "000";

--Loads the accumulator with data held in MDRI

when ldaa\_load\_a =>

ToALoad <= '1';

ToPcIncrement <= '0';

ToMarMux <= '0';

ToMarLoad <= '0';

ToRamWriteEnable <= '0';

ToMdriLoad <= '0';

ToIrLoad <= '0';

ToMdroLoad <= '0';

ToAluOp <= "101";

--Loads the MAR with address held in IR

when adaa\_load\_mar =>

ToALoad <= '0';

ToPcIncrement <= '0';

ToMarMux <= '1';

ToMarLoad <= '1';

ToRamWriteEnable <= '0';

ToMdriLoad <= '0';

ToIrLoad <= '0';

ToMdroLoad <= '0';

ToAluOp <= "000";

--Reads Memory based on address in MAR

when adaa\_read\_mem =>

ToALoad <= '0';

ToPcIncrement <= '0';

ToMarMux <= '0';

ToMarLoad <= '0';

ToRamWriteEnable <= '0';

ToMdriLoad <= '0';

ToIrLoad <= '0';

ToMdroLoad <= '0';

ToAluOp <= "000";

--Loads MDRI with data just read from memory

when adaa\_load\_mdri =>

ToALoad <= '0';

ToPcIncrement <= '0';

ToMarMux <= '0';

ToMarLoad <= '0';

ToRamWriteEnable <= '0';

ToMdriLoad <= '1';

ToIrLoad <= '0';

ToMdroLoad <= '0';

ToAluOp <= "000";

--Loads accumulator with data in MDRI

when adaa\_store\_load\_a =>

ToALoad <= '1';

ToPcIncrement <= '0';

ToMarMux <= '0';

ToMarLoad <= '0';

ToRamWriteEnable <= '0';

ToMdriLoad <= '0';

ToIrLoad <= '0';

ToMdroLoad <= '0';

ToAluOp <= "000";

--Loads MDRO with data to be written to memory (this data comes from the accumulator)

when staa\_load\_mdro =>

ToALoad <= '0';

ToPcIncrement <= '0';

ToMarMux <= '1';

ToMarLoad <= '1';

ToRamWriteEnable <= '0';

ToMdriLoad <= '0';

ToIrLoad <= '0';

ToMdroLoad <= '1';

ToAluOp <= "100";

--Writes to memory the data stored in MDRO

when staa\_write\_mem =>

ToALoad <= '0';

ToPcIncrement <= '0';

ToMarMux <= '0';

ToMarLoad <= '0';

ToRamWriteEnable <= '1';

ToMdriLoad <= '0';

ToIrLoad <= '0';

ToMdroLoad <= '0';

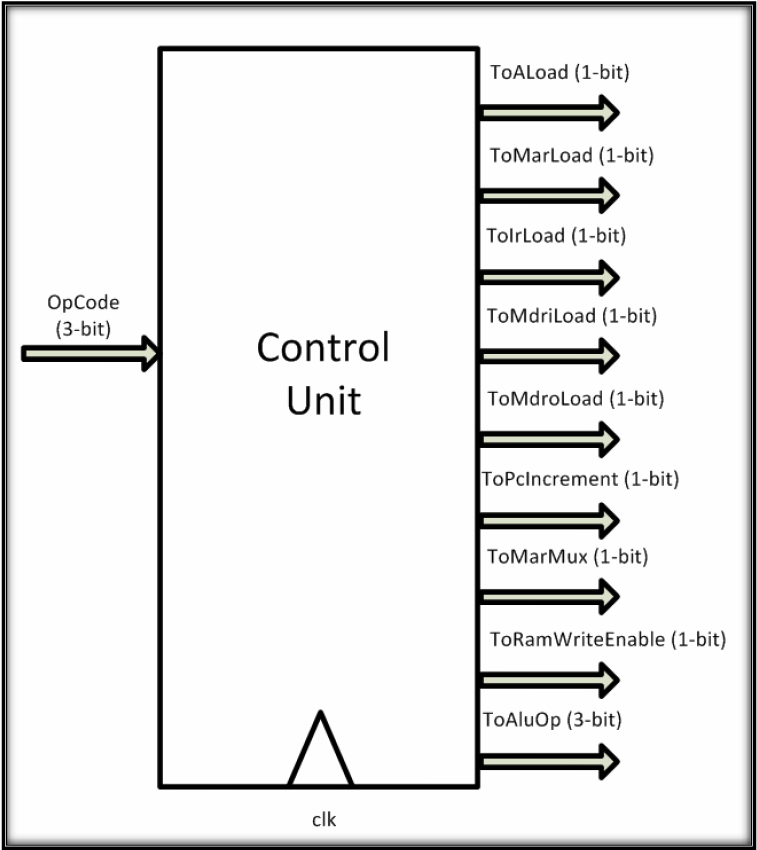
ToAluOp <= "100";

end case;

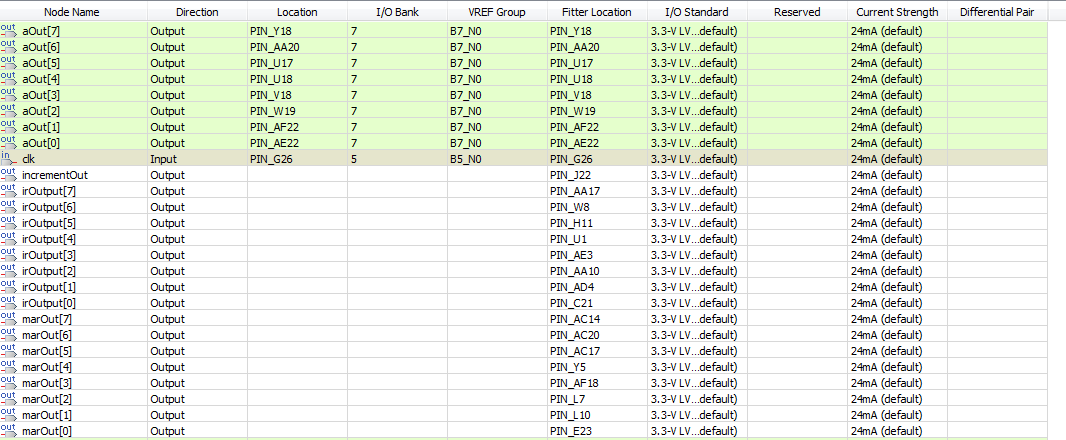
end process;

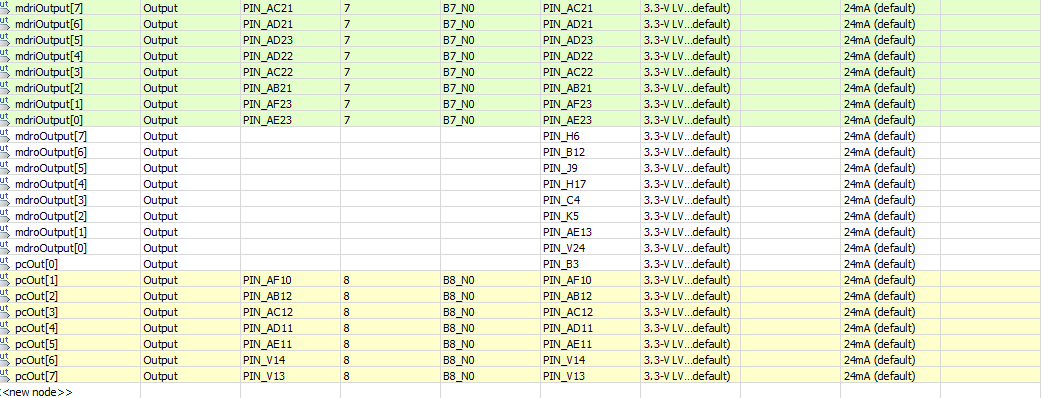
end behavior;

The control unit is responsible for coordinating every process within a CPU. We used the OpCode shown at the beginning of this code for every step of the CU. Within the current state code above, a ‘1’ indicated that that particular instruction was being implemented. When the “ToAluOp” shows a “000” the CU is loading data, when is shows a “001” it is adding data, when it shows a “010” it is storing data, when it shows “100” it is writing data stored in MDRO to memory, and when it shows a “101” it is loading data into the accumulator. The diagram on the following page shows the layout of a CU.



**Pin Assignments for VHDL Project:**

****

****

As you can see from the pictures on the previous page, we assigned the accumulator outputs to green LEDs and the memory data in outputs to red LEDs. We mapped the clock to a push button and the program counter outputs to the seven segment display.

**Results:**

When we pressed the push button (which was connected to the clock of the CPU) the seven segment displayed a “0.” When we continued to press the push button, the seven segment display started to count up to nine. The red LEDs held the value read from memory. After the button was pushed continuously, the values displayed on the red LEDs were then stored and displayed on the green LEDs. Then, the red LEDs would display a different binomial number after more pushes of the button. The new binomial number displayed by the red LEDs was then added to the previous number displayed on the red LEDs, which was stored and displayed by the green LEDs; this new number was once again displayed by the green LEDs. This process continued as we continued to push the push button.