Lane Department of Computer Science and Electrical Engineering

CpE 272 Digital Logic Laboratory

Design of a Simple CPU

Fall 2013

West Virginia University - College of Engineering and Mineral Recourses

Microprocessor Overview

A microprocessor can be thought of as a complex, programmable state machine that performs operations according to instructions that have been stored in memory. The processor retrieves an instruction from memory, decodes the instruction to determine what actions need to be performed, performs the necessary actions, and then begins retrieving the next instruction. On their own, individual instructions perform simple tasks, but when instructions are arranged in special sequences, extremely complex tasks can be performed. When instructions are arranged in this manner, they are collectively referred to as a *program*.

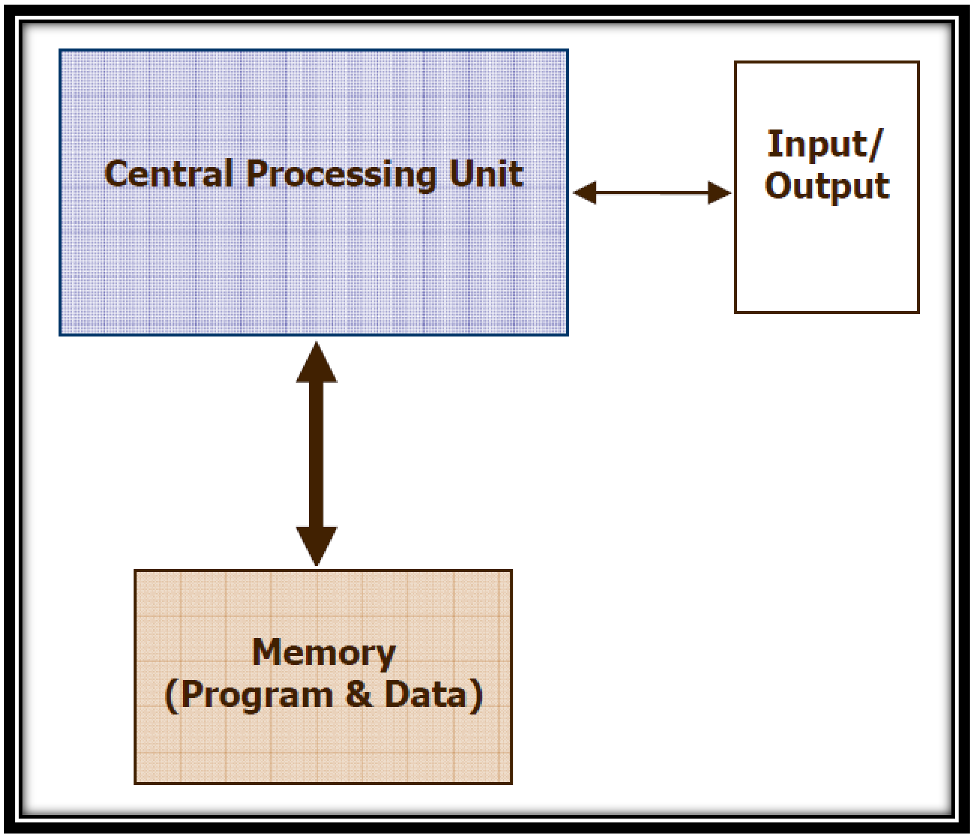
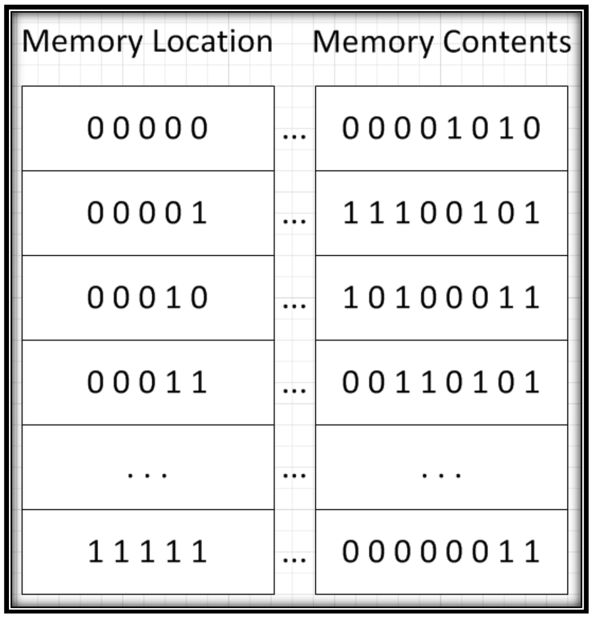


Figure 1: Microprocessor - High Level View

The architecture above follows the Von Nuemann model, in which program instructions and data are stored in the same memory space. This means that one must pay careful attention when creating program in order to avoid accidentally overwriting instructions or important data.

Memory Structure (RAM)

For this project, the memory module will be based on the memory component created in Lab #11. The memory address bus is 5-bits wide, allowing for a total of 32 memory locations. Each memory location contains an 8-bit value, which can be read from or written to, according to the "Write_Enable" bit.



Memory Location		Memory Contents
0 0 0 0 0	...	0 0 0 0 1 0 1 0
0 0 0 0 1	...	1 1 1 0 0 1 0 1
0 0 0 1 0	...	1 0 1 0 0 0 1 1
0 0 0 1 1	...	0 0 1 1 0 1 0 1
. . .	...	. . .
1 1 1 1 1	...	0 0 0 0 0 0 1 1

Figure 2: Memory Layout

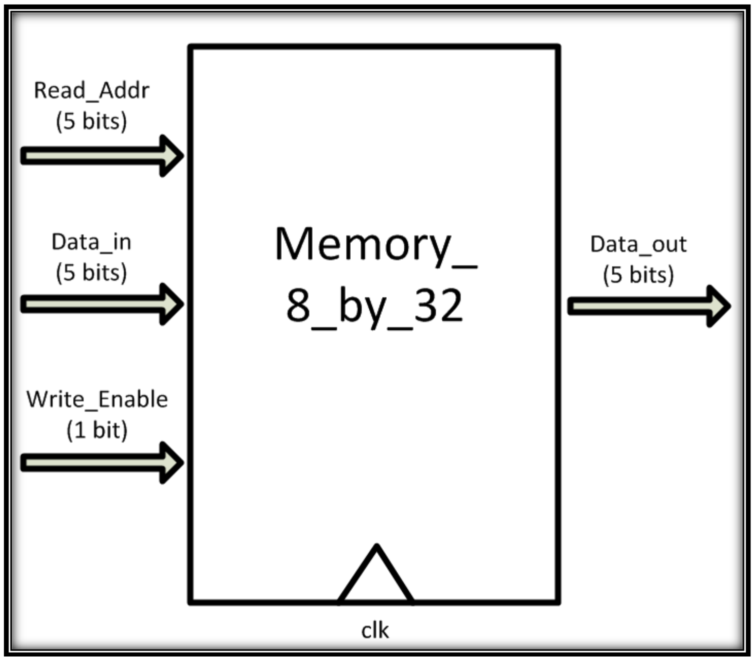
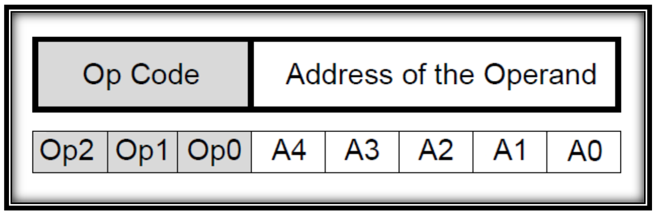


Figure 3: Memory Entity (VHDL)

Instruction Format

Each instruction is represented by an 8-bit value. The 3 most significant bits contain the "Op Code", which describes the operation that the instruction performs. The remaining 5 bits represent the memory address of the "Operand", which contains the data used by the instruction.

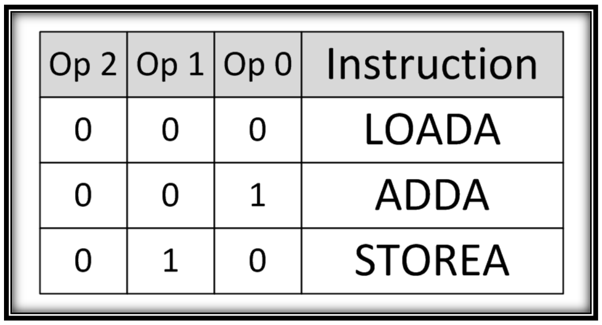


Op Code			Address of the Operand				
Op2	Op1	Op0	A4	A3	A2	A1	A0

Figure 4: Instruction Format

The CPU for this project will need to perform three distinct functions: LOADA, ADDA, and STOREA. Each instruction performs an operation involving an 8-bit register known as the "Accumulator". This register contains the output of the CPU's Arithmetic & Logic Unit (ALU), and is routed back to the ALU to serve as one of the two inputs. The Accumulator and ALU are described in further detail later in the handout.

The Op Codes for the three instructions are as follows:



Op 2	Op 1	Op 0	Instruction
0	0	0	LOADA
0	0	1	ADDA
0	1	0	STOREA

Figure 5: Instruction Op Codes

LOADA: Loads the data specified by the Operand into the Accumulator.
Example: LOADA 10h ; *Loads the Accumulator with the contents of memory location 10h*

ADDA: Adds the data specified by the Operand with the contents of the Accumulator.
Example: ADDA 10h ; *Adds the contents of memory location 10h to the value stored in the accumulator*

STOREA: Stores the contents of the Accumulator in the memory location specified by the Operand.
Example: STOREA 10h ; *Stores the contents of the Accumulator in memory location 10h*

CPU Architecture

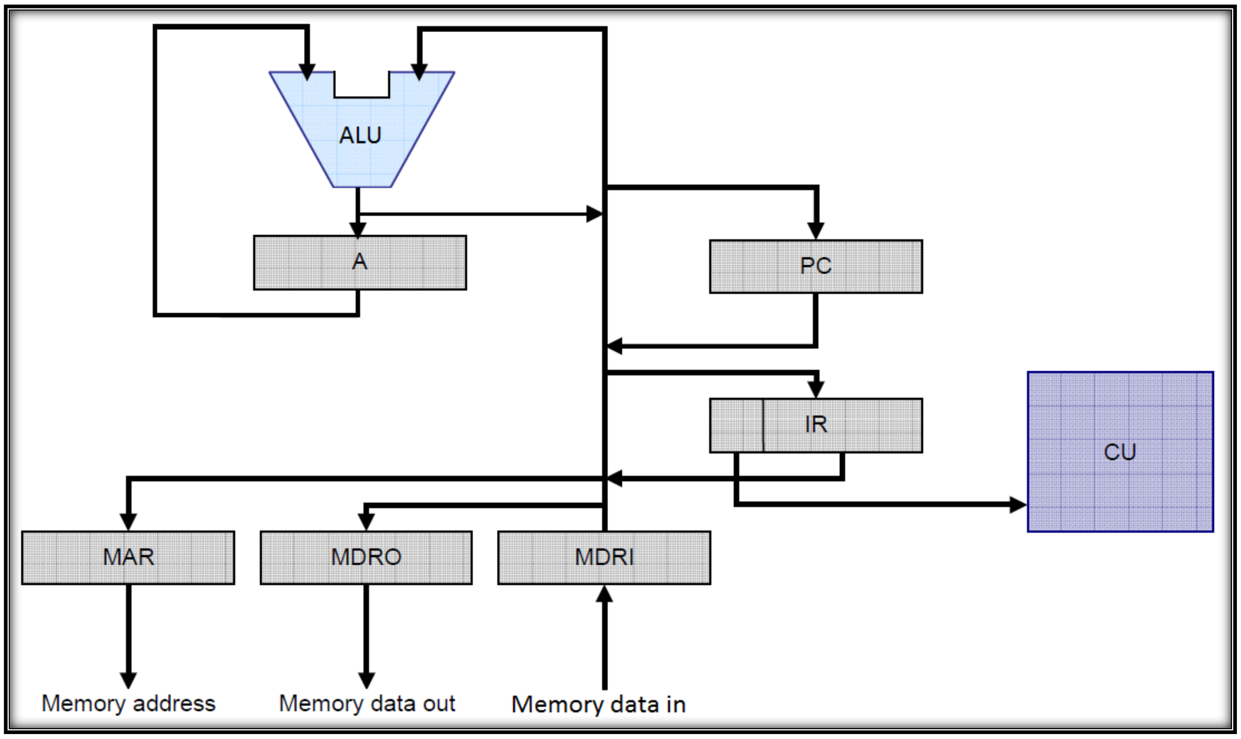


Figure 6: Simplified CPU Architecture

Figure 6 shows the simplified internal architecture of the CPU (memory module not shown). The CPU contains the following components:

A – Accumulator Register

IR – Instruction Register

CU – Control Unit

PC – Program Counter Register

ALU – Arithmetic & Logic Unit

MAR – Memory Address Register

MDRI – Memory Data Register (Input)

MDRO – Memory Data Register (Output)

Arithmetic & Logic Unit (ALU)

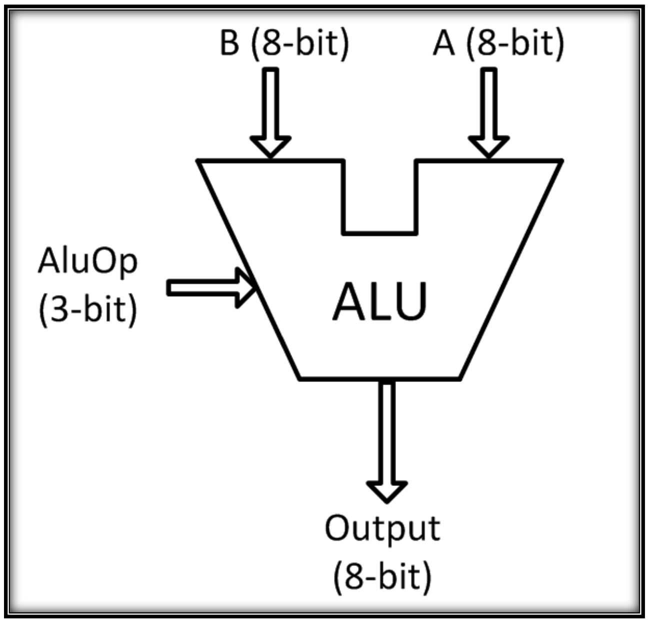
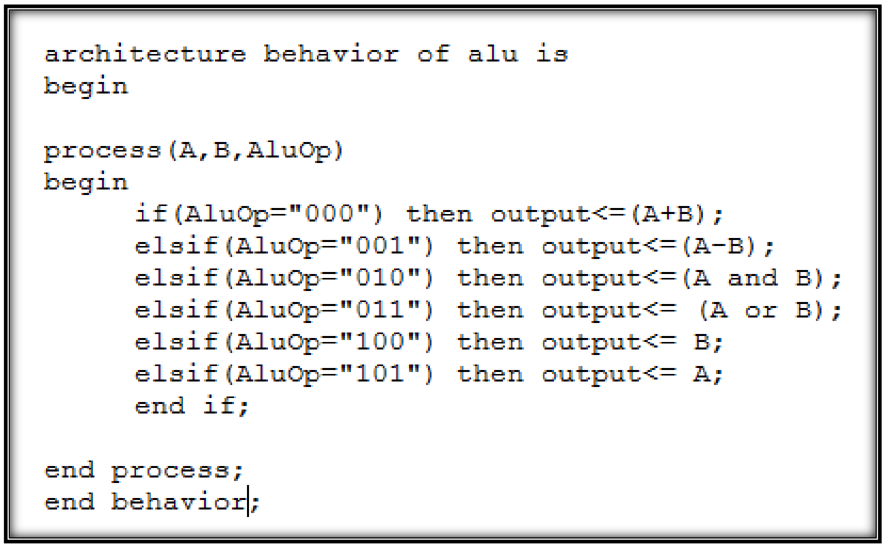


Figure 7: ALU Entity (VHDL)

The ALU is based on the ALU component created in Lab #11. The ALU performs operations on the 8-bit inputs, A and B, and outputs the 8-bit result. The possible operations include addition, subtraction, bitwise AND, bitwise OR, output A, and output B.

The ALU is created by invoking an instance of the component "alu" (port-mapping).



```
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 behavior;
```

Figure 8: ALU Behavioral Description

Program Counter (PC)

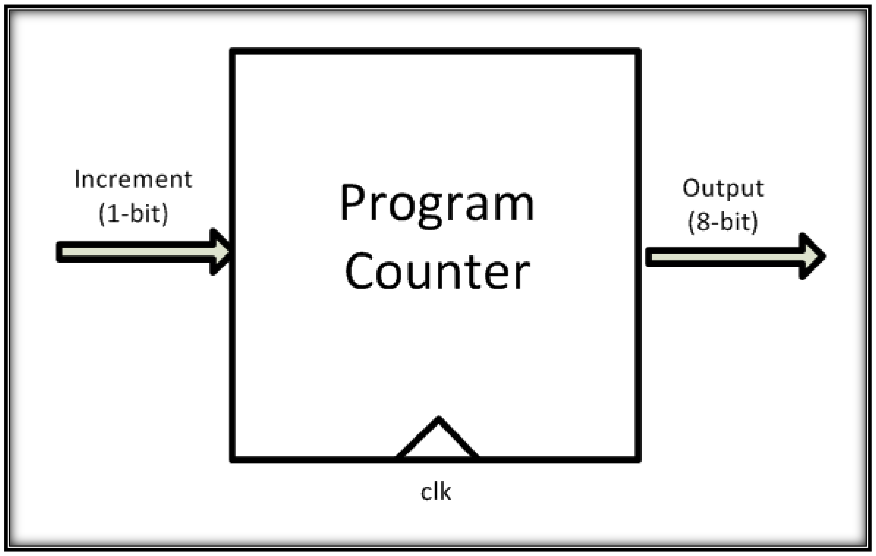


Figure 9: Program Counter Entity (VHDL)

The program counter (PC) is a special register that holds the 8-bit memory address of the next instruction to be executed. When a program begins, the memory address of the first instruction is stored in the PC. The CPU "fetches" the instruction from memory based on the contents of the PC. After the first instruction has been executed, the PC is incremented by one to point to the next instruction. To control this, the PC register has one input labeled "Increment". When Increment is set to one, the PC value is incremented by one. When Increment is set to zero, the PC value does not change.

When creating a program that will run on the CPU, it will be necessary to place the instructions in memory sequentially. If instruction #1 is in memory location 0, then instruction #2 will need to be in memory location 1, instruction #3 in memory location 2, and so on. Be cautious when loading and storing data to prevent instructions from being overwritten.

It is important to note that the PC contains an 8-bit memory address, but the memory component for this project is limited to 5-bit addresses. Keep this in mind when fetching the instruction from memory.

Accumulator (A)

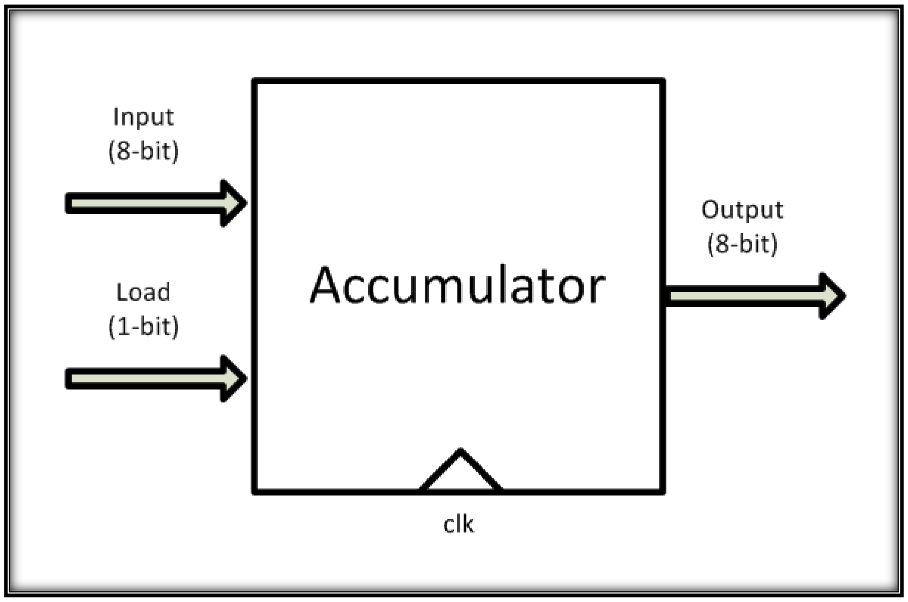


Figure 10: Accumulator Component (VHDL)

The Accumulator is a register that contains the 8-bit output of the ALU. Figure 6 shows that the output of the Accumulator serves as an input for the ALU, forming a feedback loop.

The "Load" input, in conjunction with the clock input, signifies when the Accumulator will latch its input to its output. "Load" performs the same role as the clock input of a D Flip-Flop, in that a high signal will set the output equal to the input, and a low signal will keep the output the same regardless of what happens at the input. The control unit controls the value of "Load".

The Accumulator is created by invoking an instance of the "reg" component (port-mapping).

```
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;
```

Figure 11: Register Behavioral Description

Memory Address Register (MAR)

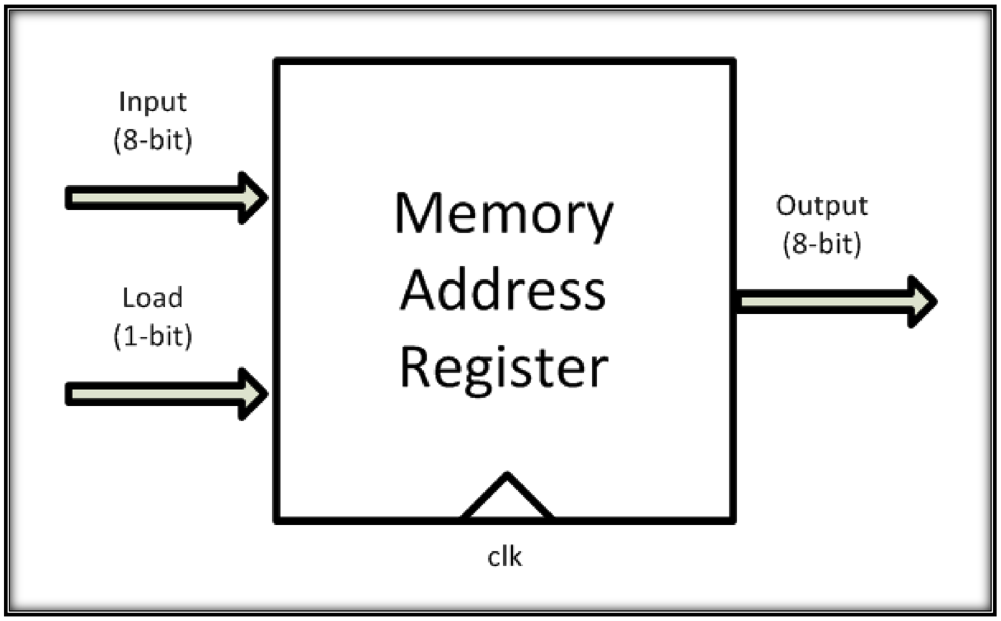
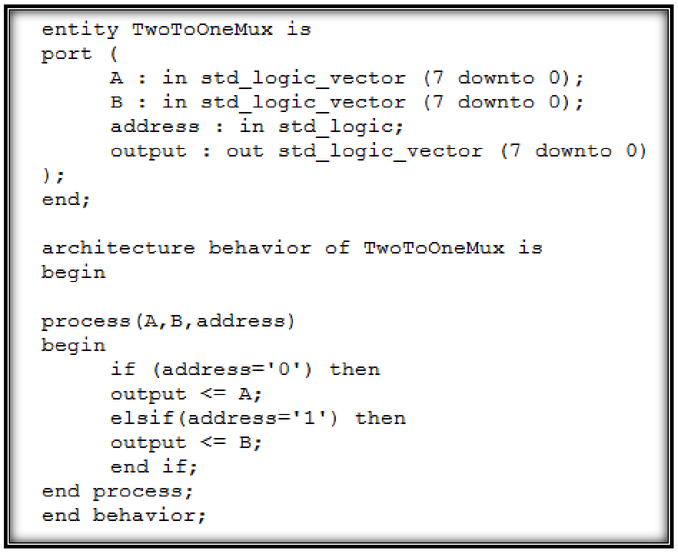


Figure 12: Memory Address Register Component (VHDL)

The Memory Address Register (MAR) is an 8-bit register that stores the value of a memory location. This memory location can be a location to retrieve data from, or a location to store data. Both the program counter and the instruction register need to access the MAR to read and write to memory. To handle both of these inputs, a multiplexer (TwoToOneMux) is placed in front of the MAR. The control unit determines which component has access to the MAR by controlling the “address” input.

The MAR is created in the same manner as the Accumulator by invoking an instance of the “reg” component. The multiplexor is based on the code below:



```
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;
```

Figure 13: MAR Multiplexor Entity and Behavioral Description (VHDL)

Memory Data Register: Input & Output (MDRI & MDRO)

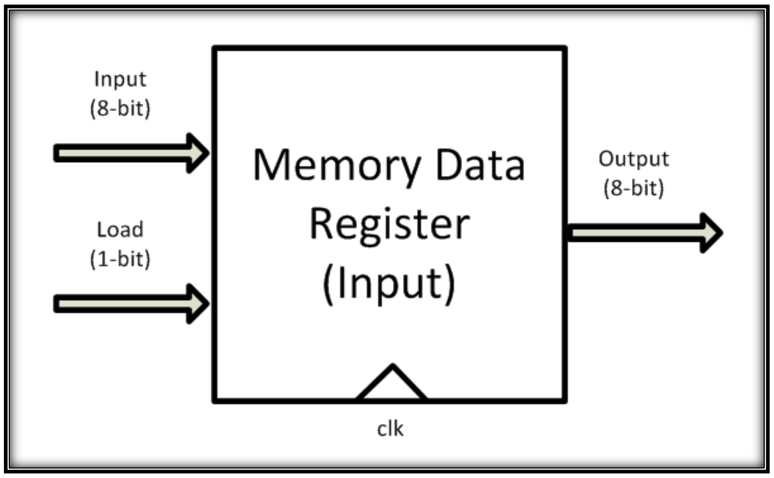


Figure 14: Memory Data Register Input Component (VHDL)

The Memory Data Register Input (MDRI) is an 8-bit register that holds a value read from memory. When an instruction reads from memory, this is where the result will temporarily be stored before being passed along to component that will use it.

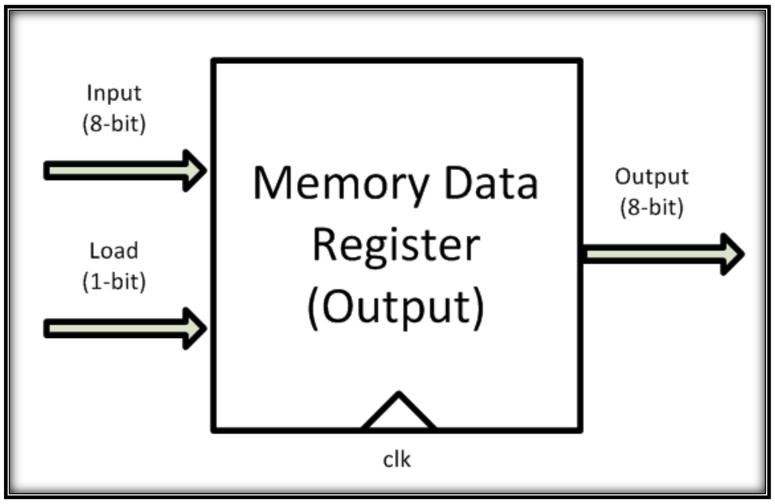


Figure 15: Memory Data Register Output Component (VHDL)

The Memory Data Register Output (MDRO) is an 8-bit register that holds a value that is to be written to memory. When an instruction writes to memory, this is where the value will be stored before being stored in memory.

Both the MDRI and the MDRO are created by invoking an instance of the “reg” component (port-mapping).

Instruction Register (IR)

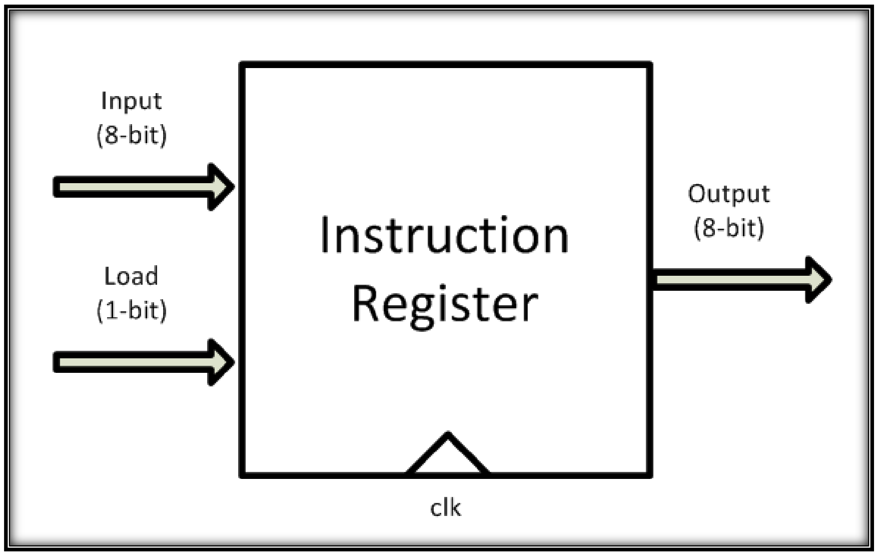


Figure 16: Instruction Register Component (VHDL)

The Instruction Register (IR) is an 8-bit register that contains the instruction that is currently being executed. The format of the instructions is described in Figure 4. The IR is created by invoking an instance of the "reg" component (port-mapping).

Control Unit (CU)

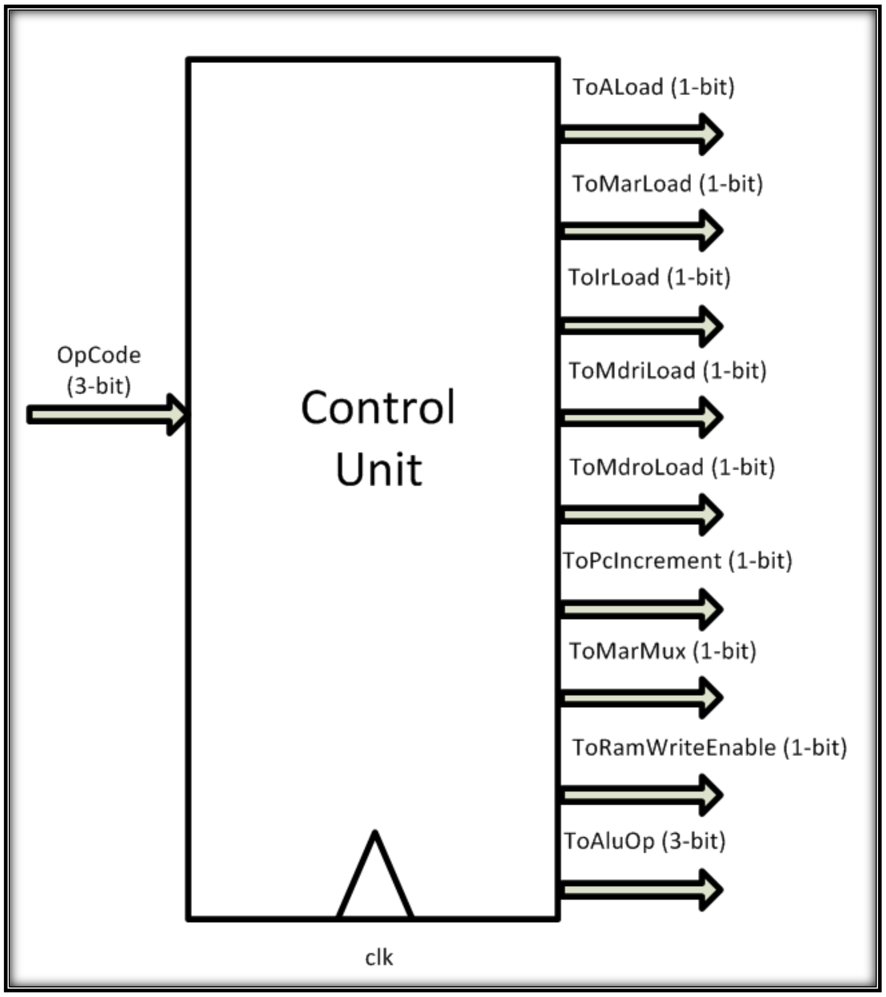


Figure 17: Control Unit Entity (VHDL)

The Control Unit (CU) coordinates all of the operations of the CPU, including incrementing the PC, accessing memory/registers, and ALU operations. The CU can be modeled as a state machine in which each step of an instruction represents a state. For example, fetching an instruction from memory takes five steps (clock cycles), so there will be five state transitions involved in the fetch process. At each state, the control unit must define which registers are active and what the next state will be.

Instruction Fetch

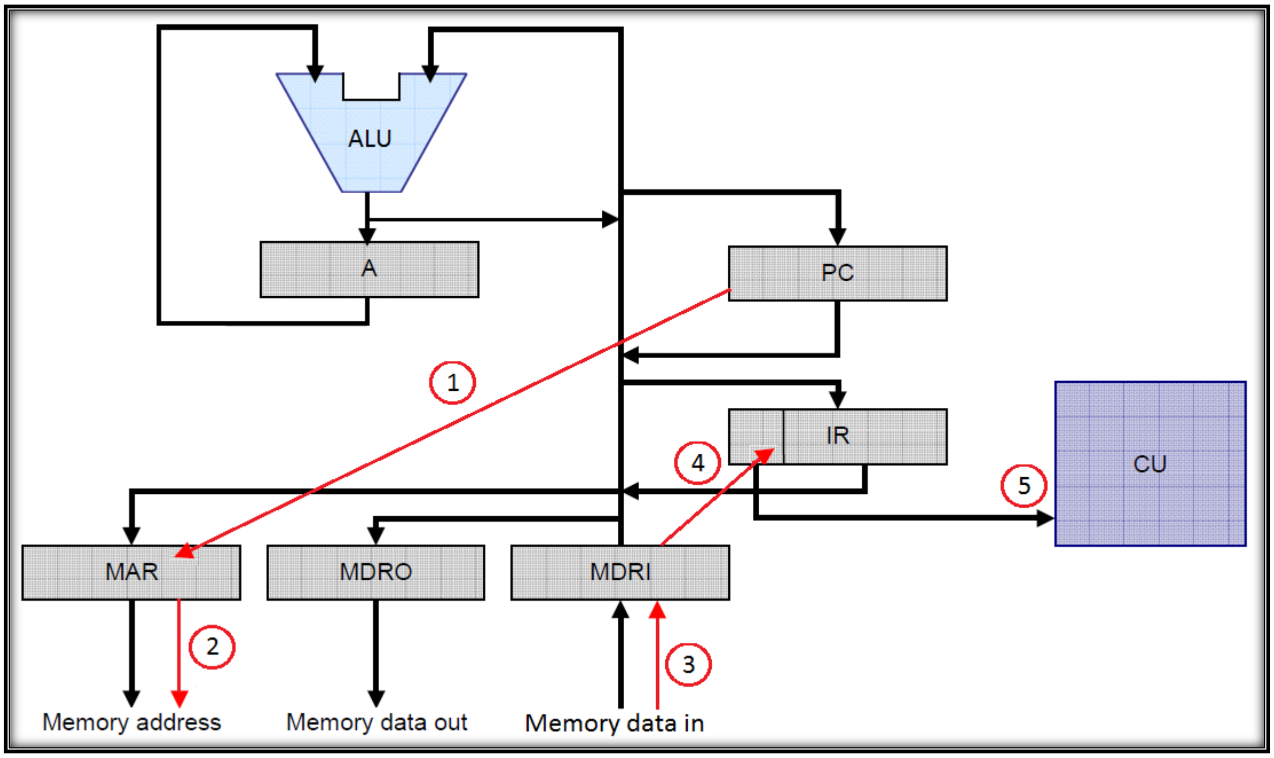


Figure 18: Instruction Fetch Sequence

In order to execute a program instruction, the CPU must first fetch the instruction from memory. The following steps are taken to fetch an instruction:

1) The program counter contains the address of the next instruction, load that address into MAR.

2) A memory read is performed at the address loaded in MAR.

3) The result of the memory read is loaded into MDRI.

4) The value held in MDRI is loaded into IR.

5) The instruction held in IR is decoded.

After the instruction has been decoded, the CPU can then begin executing the instruction. Remember, for this project there are only three possible instructions that can be performed: LOADA, ADDA, STOREA. The Control Unit's next state after decoding will depend on which of these instructions is to be performed.

LOADA Instruction

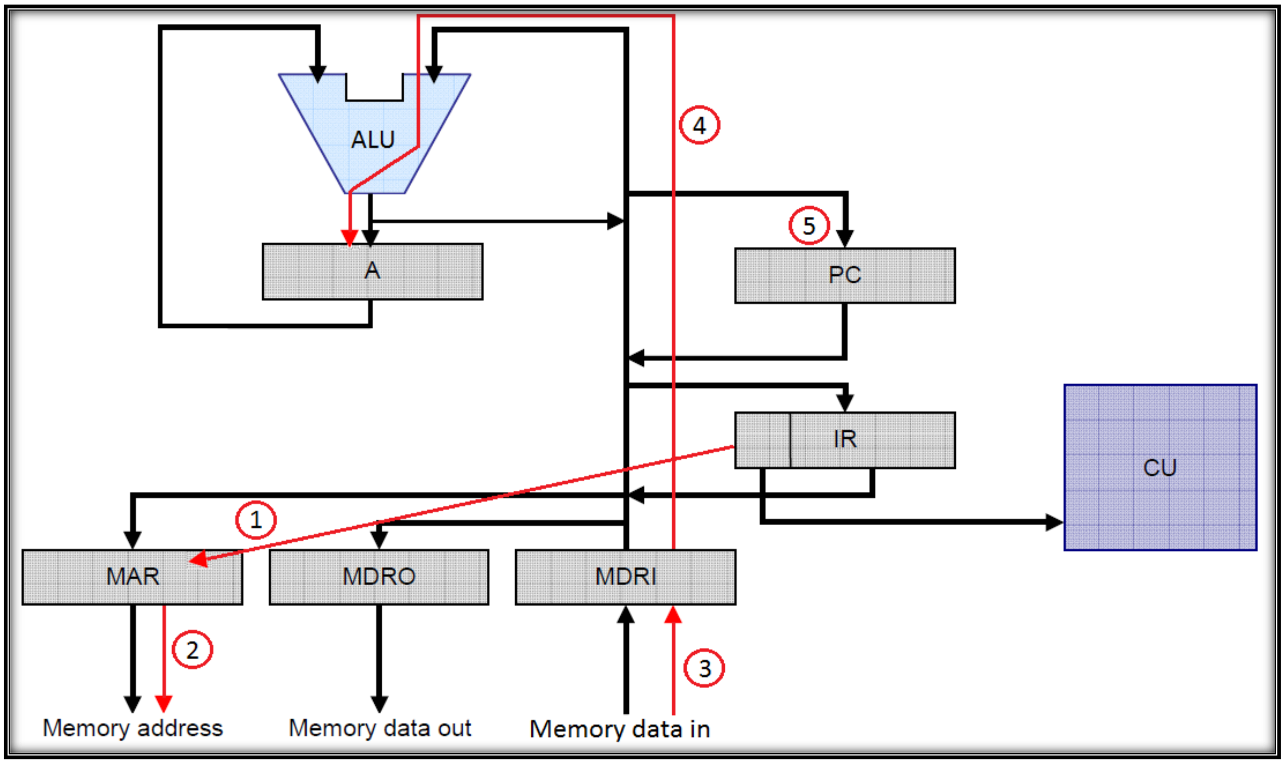


Figure 19: LOADA Instruction Sequence

The LOADA instruction loads the value from the specified memory location into the Accumulator (A). The following steps are performed to complete this instruction:

1) Load MAR with the lower 5-bits of IR
2) A memory read is performed at the address loaded in MAR.
3) The result of the memory read is loaded into MDRI.
4) Load the accumulator with the contents of MDRI.
5) Increment PC

The correct ALU op code will need to be set by the Control Unit in order to load the appropriate ALU input into the accumulator.

ADDA Instruction

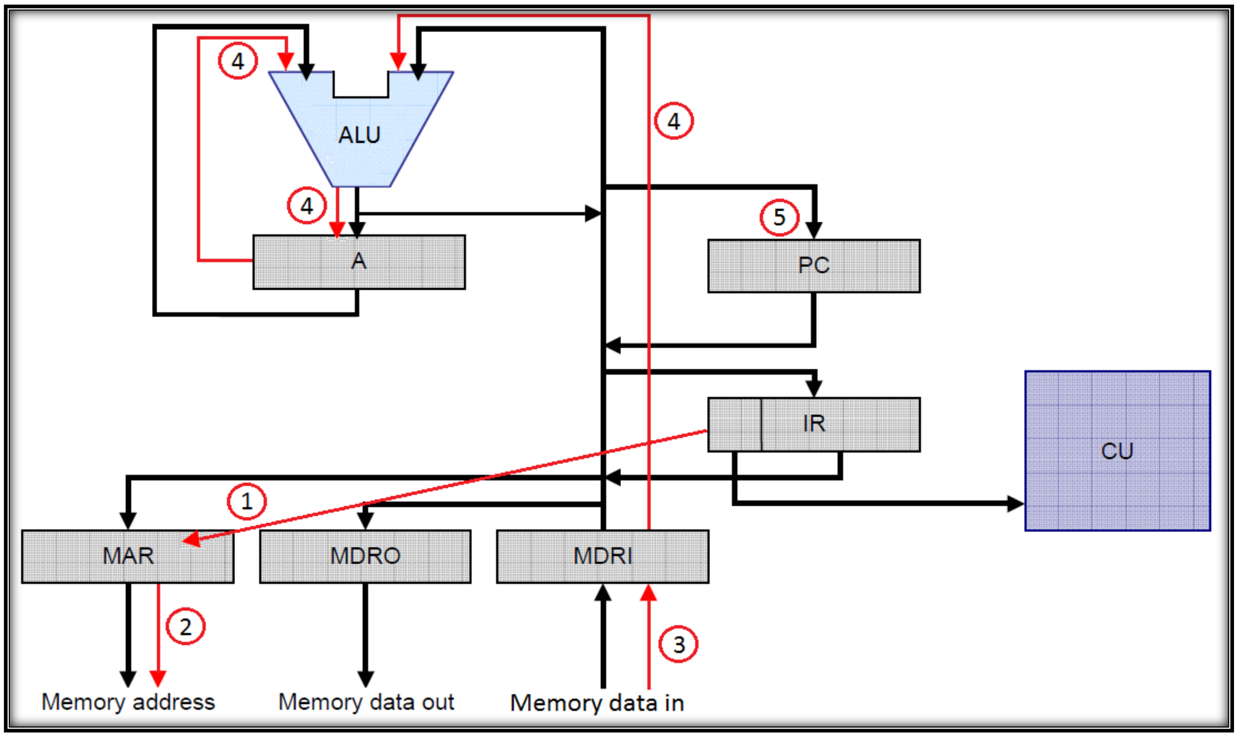


Figure 20: ADDA Instruction Sequence

The ADDA instruction adds the value from the specified memory location to the value in the Accumulator. The result is stored in the Accumulator. The following steps are performed to complete this instruction:

1) Load MAR with the lower 5-bits of IR
2) A memory read is performed at the address loaded in MAR.
3) The result of the memory read is loaded into MDRI.
4) Perform addition of MDRI contents and accumulator contents, and store the result in the accumulator.
5) Increment PC

STOREA Instruction

The STOREA Instruction stores the contents of the accumulator in the specified memory location. As part of the final project, students will need to determine the steps required to complete this instruction.

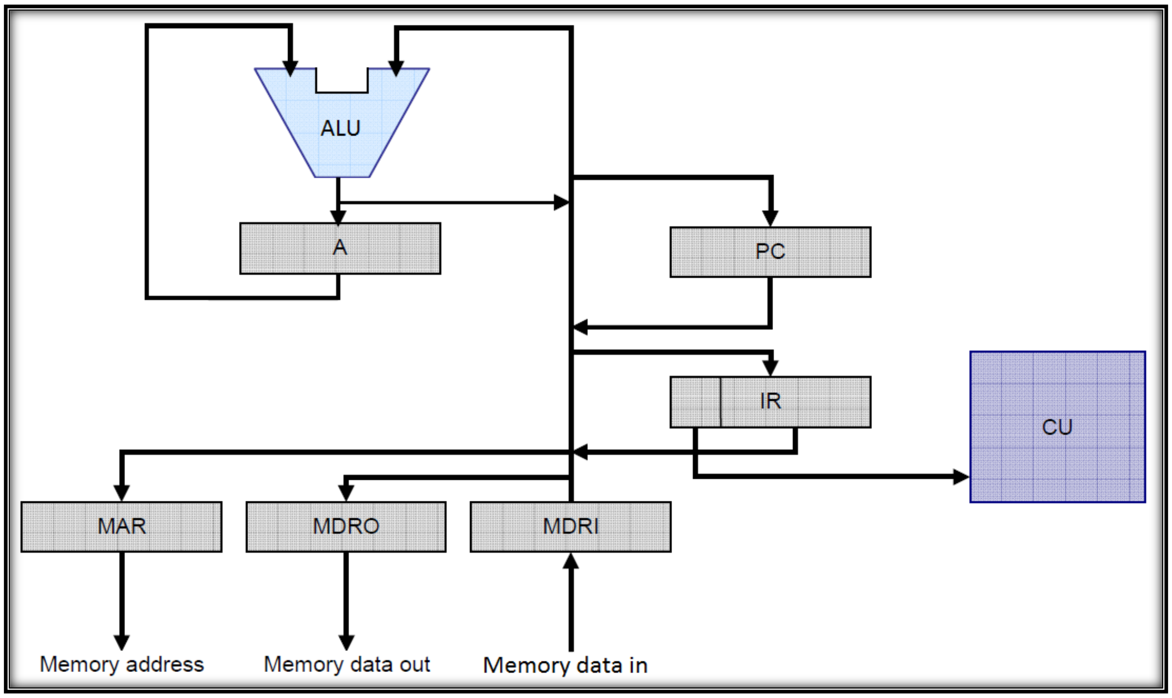


Figure 21: STOREA Instruction Sequence (FILL IN!)

Steps (FILL IN!):

1)

2)

3)

Tasks

Due to the complexity of this project, a large portion of the code has been provided. Take your time reading through the handout and studying the provided code. Your instructor will clarify any portions of the provided code that are unclear.

Critical areas of the provided code are incomplete, and it will be your task to fill in these areas.

Tasks to complete:

1) Complete the behavioral description of the Program Counter.

2) Create a complete diagram of the CPU, including all components and connections **as they are labeled in the code**.

3) Define the Control Unit state transitions as well as the corresponding output for each state (Have to define sequence for STOREA first).

4) Instantiate all of the CPU components through port-mapping, and create the appropriate connections. Drawing the CPU diagram beforehand will make this significantly easier!

5) Demonstrate a functioning CPU by displaying all of the SimpleCPU_Template outputs on the Altera Board while running a program. (Your instructor will give you the memory contents for the program).

Bonus points will be given for the following items:

1) Implementing additional instructions (Subtract, Shift, Rotate, Decrement, etc.).

2) Utilizing all of the seven segment displays to display the various program outputs.

3) Using your functioning CPU to execute interesting/complex programs in spite of the limited amount of RAM available.