EE 2700 Project 2 – Microprocessor Design

This project may be done individually or in teams of two. You may consult others for general questions but not for specific issues. Cheating will not be tolerated and will result in a zero score for this part of the project.

In this project, you will design a simple microprocessor except for the control logic, which you will do in Project 3. The first step in any design is to get the requirements.

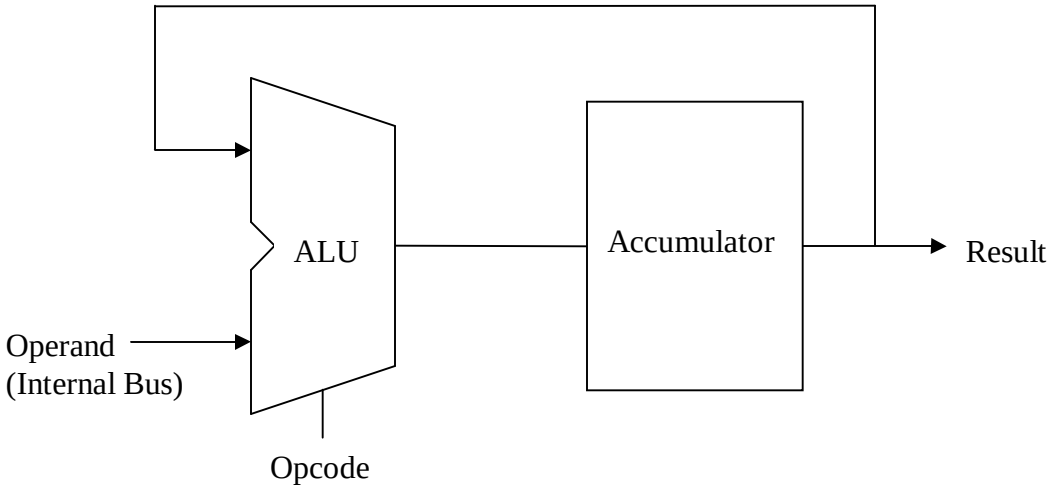
Requirements:

1. The microprocessor shall have synchronous memory interface consisting of:
   a) an 8-bit output bus for the memory address, asserted high.
   b) an 8-bit bi-directional bus for the memory data, also asserted high.
   c) a write output, asserted low. If write is asserted, the microprocessor shall drive the data bus with the data to be written to memory.
   d) a read output, asserted low. If read is asserted, the microprocessor shall sample the data bus. Read and write shall never be asserted at the same time.
2. The microprocessor shall use the memory interface to fetch instructions.
3. Instructions shall consist of a one-byte operation code (opcode) and, for some instructions, one byte of address or immediate data.
4. Instructions shall be fetched and executed in order, starting with address zero, until (a) an instruction is executed that changes the fetch address, or (b) an instruction is executed that causes the microprocessor to halt.
5. The microprocessor shall contain an 8-bit accumulator that receives the result of any arithmetic instruction.
6. The microprocessor shall contain a carry flag that is set or cleared based on whether or not an add (or add with carry) instruction results in a carry-out. All other instructions leave the carry flag unchanged.
7. The microprocessor shall support instructions to load the accumulator from memory and store the accumulator to memory.
8. The microprocessor shall support both immediate and direct addressing modes.
9. The accumulator shall be one operand for all arithmetic instructions. For operations with two operands, the second operand shall come from memory (either immediate or direct).
10. The microprocessor shall support the instructions shown in Table 1.
11. The microprocessor shall have an asynchronous reset (asserted low).
12. The microprocessor shall have a minimum clock speed of 20MHz.

Table 1.

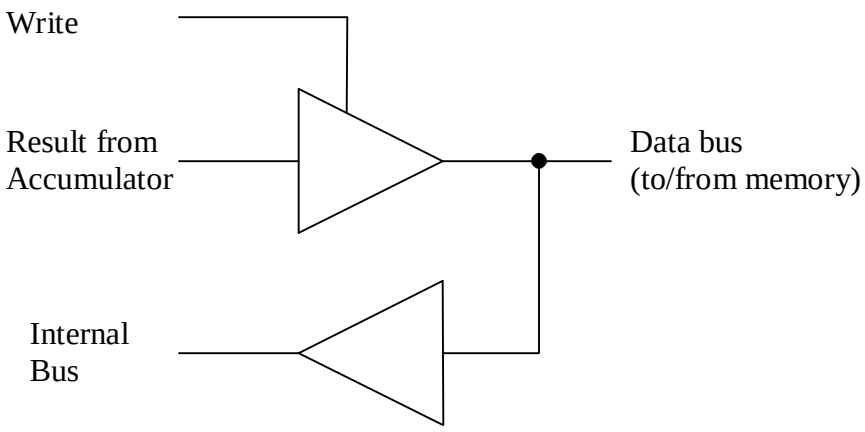
Opcode	Name	Description
LDI	Load Immediate	Load the next byte of the program into the accumulator
LDM	Load Memory	Treat the next byte in the program as an address, and load the contents of memory at that address into the accumulator.
ADDI	Add Immediate	Add the next byte of the program to the accumulator.
ADDM	Add Memory	Treat the next byte in the program as an address, and add the contents of memory at that address to the accumulator.
ADCI	Add Immediate with Carry	Add the next byte of the program plus the carry flag to the accumulator.
ADCM	Add Memory with Carry	Treat the next byte in the program as an address, and add the contents of memory at that address plus the carry flag to the accumulator.
XORI	Subtract Immediate	Compute the exclusive-or of the accumulator and the next byte in the program and put the result in the accumulator.
XORM	Subtract Memory	Treat the next byte in the program as an address, compute the exclusive-or of the accumulator and the memory at that address, and put the result in the accumulator.
STM	Store Memory	Treat the next byte in the program as an address, and store the contents of the accumulator at that address.
JMP	Jump	Load the next byte of the program into the program counter
JC	Jump if Carry	If the carry flag is set, load the next byte of the program into the program counter, otherwise discard that byte.
JNC	Jump if No Carry	If the carry flag is not set, load the next byte of the program into the program counter, otherwise discard that byte.
HALT	Halt	Stop running the program. This will be the last instruction executed in a program.

After you understand the requirements, the next step is to develop an algorithm or technique to meet the requirements. In this case, we will start with the accumulator and carry. Since the accumulator and carry flag are used the store the output of all arithmetic operations, it makes sense to make them out of D flip-flops and connect them directly to the ALU. Likewise, since they are also the source of at least one operand, they can be routed directly to one input of the ALU. (See figure on the next page.) It usually works best if the accumulator is actually 9-bits in size and includes the carry bit. If you wish, you can redesign your ALU in VHDL first.



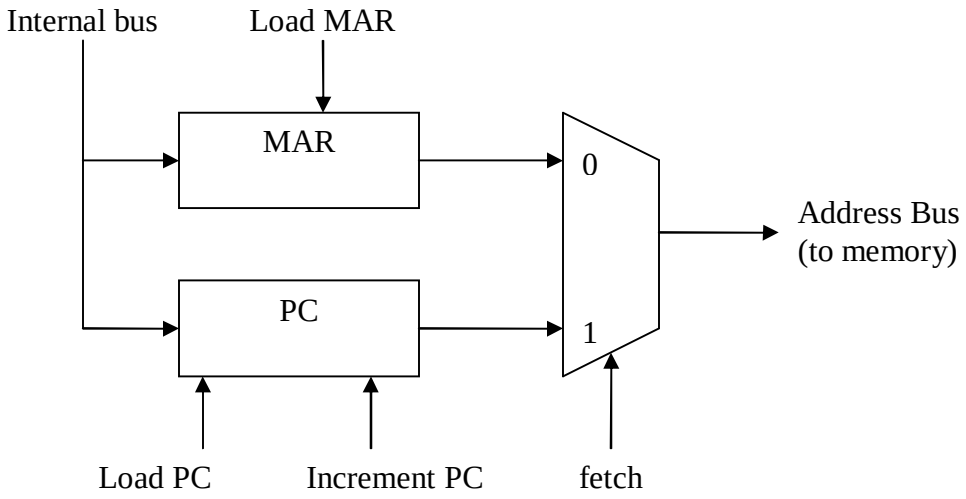
The accumulator will need a clock and possibly a clock enable input (because you may not want to update the accumulator on every clock cycle).

The second step is to create an interface for the memory. Data can either go out to the memory (write) or come in from the memory (read). This means you will need a bi-directional bus with line drivers. (See figure below.) Most of the time you will want to read the memory and put what you read onto an internal bus, but sometimes you will need to write the results from the accumulator to the memory. The memory interface uses the *write* (WR) signal (asserted low) to tell it when to write and when to read.



The memory interface is not yet complete because we haven't considered what to put on the address bus. We are required to fetch instructions in sequential order, so we can generate the addresses for those instructions with a counter. This counter is historically called the "program counter" or PC. Since some instructions change the PC, the counter will need to be loadable.

Some of the required instructions transfer data to or from memory. This poses a double challenge. First, the PC should not increment on the cycle that data are transferred. This can be handled by a clock enable. Second, the memory address for the transfer needs to be supplied by the instruction. One way to handle this is to load the address from the instruction into a memory address register (MAR), then use a MUX to either select it or the PC. (See figure below.)



The last part of the microprocessor is the controller. This part interprets the instructions the microprocessor receives from memory and outputs all the required control signals. The details for this will have to wait until Project 3.

Now, consider how all these pieces play together. Suppose the PC points to an instruction that loads a byte into the accumulator from memory address 23. First, the PC is placed on the address and the read signal is asserted. The opcode of the instruction is input and given to the controller. The controller sees that the instruction is a load and that the memory address is in the second byte of the instruction. It increments the PC* and puts it on the address bus again, this time the results (in this case, 23) are placed in the MAR. On the next cycle, the value in the MAR is placed on the address bus, and the contents of memory are fed to the ALU, which has been told to pass its input directly to the accumulator. (*The PC is actually incremented at the end of the cycle in which it is used.)

At this point you have the requirements and should understand the concept and the algorithm. Your task is to complete the design except for the controller. First, make a functional partition. Next, compile a list of requirements for each module. (These lists should be fairly short except for the controller.) Third, repeat the design process for all the modules except for the controller. If a module is simple enough that it need not be partitioned, implement it in VHDL and create a schematic symbol for it in ISE. Remember that each module should be simulated separately before it is used.

Once all the modules except the controller have been tested and have schematic symbols, connect them together in an ISE schematic. Without a controller, the control lines will all have to come from the test fixture, so make them inputs for now. Make use of the test fixture fragment below to test your circuit.

```
-- clock process
process
begin
	clk <= '0';
	wait for 10ns;
	for i in 1 to 80 loop
		clk <= not clk;
		wait for 5ns;
	end loop;
	wait;
end process;

-- Stimulus process
process
begin
	rst <= '1';

	-- PC = 00, LDI  9E
	rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_hold;
	data <= "XXXXXXXX"; wait for 5ns; rst <= '0'; wait for 6ns;
	rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_load;
	data <= "10011110"; wait for 10ns;

	-- PC = 02, ADDI AA (A = 48, C=1)
	rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_hold;
	data <= "XXXXXXXX"; wait for 10ns;
	rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_add;
	data <= "10101010"; wait for 10ns;

	-- PC = 04, STO  3F
	rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_hold;
	data <= "XXXXXXXX"; wait for 10ns;
	rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '1'; op <= my_hold;
	data <= "00111111"; wait for 10ns;
	-- data should be "01001000"
	rd <= '0'; wr <= '1'; fetch <= '0'; inc_pc <= '0'; ld_pc <= '0'; ld_mar <= '0'; op <= my_hold;
	data <= "ZZZZZZZZ"; wait for 5ns;
	assert data = "01001000" report "incorrect data value"; wait for 5ns;

	-- PC = 06, ADC  3F (A = 91, C = 0)
	rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_hold;
	data <= "XXXXXXXX"; wait for 10ns;
	rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '1'; op <= my_hold;
	data <= "00111111"; wait for 10ns;
```

```
rd <= '1'; wr <= '0'; fetch <= '0'; inc_pc <= '0'; ld_pc <= '0'; ld_mar <= '0'; op <= my_addc;
data <= "01001000"; wait for 10ns;
-- PC = 08, STO  3F
rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_hold;
data <= "XXXXXXXX"; wait for 10ns;
rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '1'; op <= my_hold;
data <= "00111111"; wait for 10ns;
-- data should be "10010001"
rd <= '0'; wr <= '1'; fetch <= '0'; inc_pc <= '0'; ld_pc <= '0'; ld_mar <= '0'; op <= my_hold;
data <= "ZZZZZZZZ"; wait for 5ns;
assert data = "10010001" report "incorrect data value"; wait for 5ns;

-- PC = 0A ADCI 7B (A = 0C, C = 1)
rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_hold;
data <= "XXXXXXXX"; wait for 10ns;
rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_addc;
data <= "01111011"; wait for 10ns;

-- PC = 0C, XOR  3F (A = 9D, C = 1)
rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_hold;
data <= "XXXXXXXX"; wait for 10ns;
rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '1'; op <= my_hold;
data <= "00111111"; wait for 10ns;
rd <= '1'; wr <= '0'; fetch <= '0'; inc_pc <= '0'; ld_pc <= '0'; ld_mar <= '0'; op <= my_xor;
data <= "10010001"; wait for 10ns;

-- PC = 0E, ADCI 4A (A = E8, C = 0)
rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_hold;
data <= "XXXXXXXX"; wait for 10ns;
rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_addc;
data <= "01001010"; wait for 10ns;

-- PC = 10, STO  3E
rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_hold;
data <= "XXXXXXXX"; wait for 10ns;
rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '1'; op <= my_hold;
data <= "00111110"; wait for 10ns;
-- data should be "11101000"
rd <= '0'; wr <= '1'; fetch <= '0'; inc_pc <= '0'; ld_pc <= '0'; ld_mar <= '0'; op <= my_hold;
data <= "ZZZZZZZZ"; wait for 5ns;
assert data = "11101000" report "incorrect data value"; wait for 5ns;

-- PC = 12, LD   3F
rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_hold;
data <= "XXXXXXXX"; wait for 10ns;
rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '1'; op <= my_hold;
data <= "00111111"; wait for 10ns;
rd <= '1'; wr <= '0'; fetch <= '0'; inc_pc <= '0'; ld_pc <= '0'; ld_mar <= '0'; op <= my_load;
data <= "10010001"; wait for 10ns;

-- PC = 14, XORI FF
```

```
        rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_hold;
        data <= "XXXXXXXX"; wait for 10ns;
        rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_xor;
        data <= "11111111"; wait for 10ns;

        -- PC = 16, ADD  3E (C =1)
        rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_hold;
        data <= "XXXXXXXX"; wait for 10ns;
        rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '1'; op <= my_hold;
        data <= "00111110"; wait for 10ns;
        rd <= '1'; wr <= '0'; fetch <= '0'; inc_pc <= '0'; ld_pc <= '0'; ld_mar <= '0'; op <= my_add;
        data <= "11101000"; wait for 10ns;

        -- PC = 18, JC   28
        rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_hold;
        data <= "XXXXXXXX"; wait for 10ns;
        rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '0'; ld_pc <= '1'; ld_mar <= '0'; op <= my_hold;
        data <= "00101000"; wait for 5ns;
        assert address = x"19" report "incorrect address value"; wait for 5ns;

        -- PC = 28, ADDI 01
        rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_hold;
        data <= "XXXXXXXX"; wait for 10ns;
        rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_add;
        data <= "00000001"; wait for 5ns;
        assert address = x"29" report "incorrect address value"; wait for 5ns;

        -- PC = 2A, STO 3E
        rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_hold;
        data <= "XXXXXXXX"; wait for 10ns;
        rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '1'; op <= my_hold;
        data <= "00111110"; wait for 10ns;
        -- data should be 57
        rd <= '0'; wr <= '1'; fetch <= '0'; inc_pc <= '0'; ld_pc <= '0'; ld_mar <= '0'; op <= my_hold;
        data <= "ZZZZZZZZ"; wait for 5ns;
        assert data = "01010111" report "incorrect data value"; wait for 5ns;

        -- PC = 2C, JMP 12
        rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_hold;
        data <= "XXXXXXXX"; wait for 10ns;
        rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '1'; ld_mar <= '0'; op <= my_hold;
        data <= "00010010"; wait for 10ns;

        -- PC = 12
        rd <= '1'; wr <= '0'; fetch <= '1'; inc_pc <= '1'; ld_pc <= '0'; ld_mar <= '0'; op <= my_hold;
        data <= "XXXXXXXX"; wait for 5ns;
        assert address = x"12" report "incorrect address value"; wait for 5ns;
        wait;
    end process;
```

Verify that your circuitry works on the test fixture given without reporting errors.

Turn in the following on or before the due date given on the course website.

1. The top level schematic showing the functional partition.
2. Intermediate level schematics if any exist.
3. VHDL code for each VHDL module.
4. Test fixture code for each module.
5. Simulation results for each module
6. Simulation results for the entire circuit.

> *Note: You do not need to turn in schematics, test fixtures or simulation results for the ALU.*

This week during your lab period, your lab instructor will be in lab to assist you with your design. *Make sure you have written all your VHDL code and captured your schematic(s) before coming to lab*.