Lab 8 – Controlling the LC3 Datapath

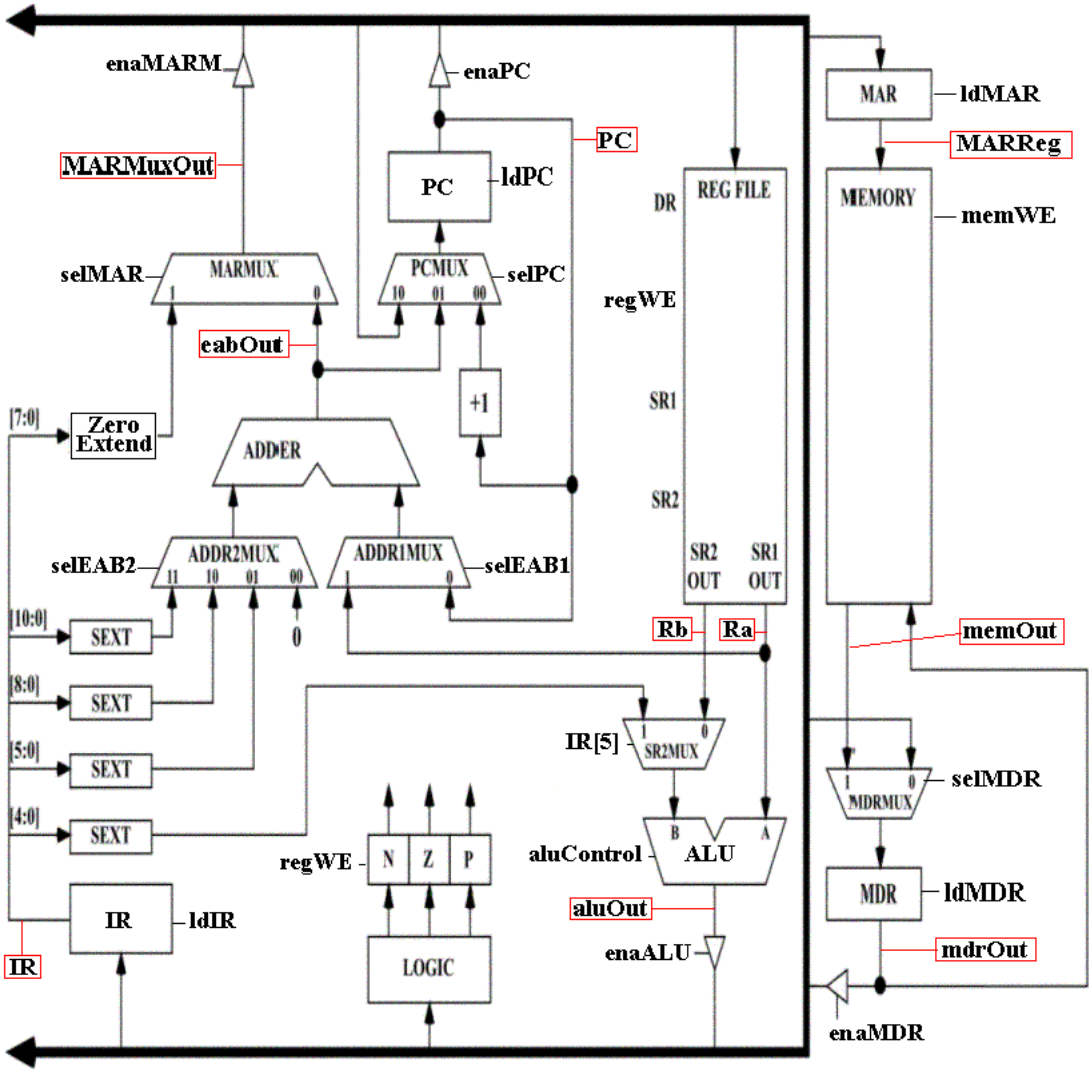
1 Objective

To understand and to exercise the Datapath design of the LC3.

2 Introduction

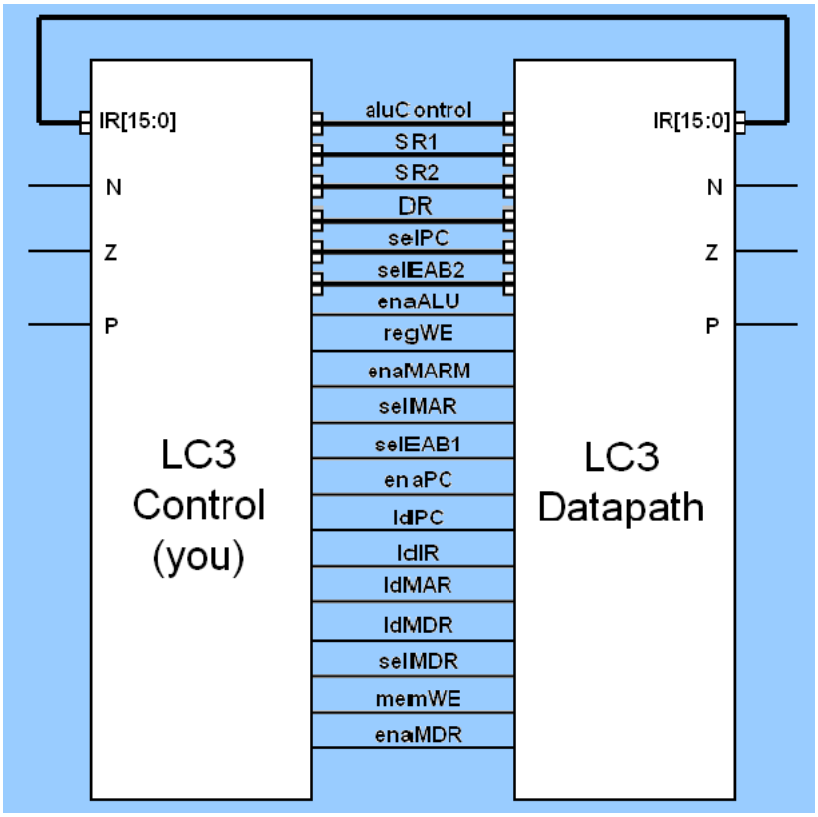
In this assignment you will exercise the Datpath for the LC3 with complete instructions. You will assemble the full Datapath using the Functional Blocks you created during the last lab and the provided memory module (Memory). This memory module already includes the MAR and MDR registers.

The LC-3 Datapath is as follows:



You are to use the modules you designed and tested in the last lab to build the Datapath as shown above. The Datapath module is to be written in VHDL code.

For this lab, there is **NO** control logic. You will act as the control logic, i. e.you will read the instruction which has been loaded into the IR, determine what instruction it is, and then set the control signals appropriately to make the Datapath execute the instruction.



The LC3 control (you) has 4 inputs:

Instruction Register	IR[15:0]
Status flags	N, Z and P

The LC3 control also has 21 control signal outputs which are inputs to the LC3 Datapath:

system clock	clk
reset signal	reset
ALU control	aluControl (2 bits)
Enable ALU onto Bus	enaALU
Source 1 Register address	SR1 (3 bits)
Source 2 Register address	SR2 (3 bits)
Destination Register address	DR (3 bits)
Register write enable	regWE
PC select control	selPC (2 bits)
Enable Memory address Mux onto Bus	enaMARM
Memory address select control	selMAR

Effective Address Block Select 1	selEAB1
Effective Address Block Select 2	selEAB2 (2 bits)
Enable PC onto Bus	enaPC
Load Program Counter	ldPC
Load Instruction Register	ldIR
Load Memory Address Register	ldMAR
Load Memory Data Register	ldMDR
Memory data select control	selMDR
Memory write enable	memWE
Enable Memory Data Register onto Bus	enaMDR

3 Preparation

Do the following:

1. Using the modules you built in the last lab, construct a Datapath module.
2. Add the supplied Memory module to your Datapath.
3. Add the supplied Memory Contents module to your Datapath.
4. Don't forget to add a Tri-state Bus Driver between the output of the memory module and the Bus.

Feel free to read through the VHDL code in the Memory module to familiarize yourself with how it is built. This module defines a block of memory which has been preloaded with a short program. The program begins executing code at location 0. When the program terminates it branches unconditionally back to location 0 and starts over.

When you simulate the Datapath, you will likely want to view many of the signals which are internal to the modules but which do not come out to the I/O ports of the modules. To view these signals, you can either drill down to the level of heirarchy in the structure browsers (a heirarchical view) or you can add them to your waves view will the full path name. For example, if you have designed an ALU module which has an internal signal named aluIn2, you can add a wave to the DO file to display this signal. If you had instantiated the ALU module with the name ALU0, the pathname to the signal would be ALU0/aluIn2. The declaration of the wave in the DO file would be as follows:

```
add wave /ALU0/aluIn2
```

3.1 Simulation

For this lab you will be creating a large testbench that will execute all of the instructions contained in the memory. You will first reset the machine and then start fetching instructions from memory address 0. You will not be creating any control circuitry for this, you will act as the control circuitry yourself and execute the instructions.

4 Procedure and Functionality

Writing very long test benches will require discipline and careful consideration of what is going on. You will have to cut and paste the same sets of tests to execute the instructions. The following tips may help you in this process.

1. Write simulation cycle that will be your inactive state. This should set all the enable and load signals to their "inactive" state. Call this before and after every fetch to make sure that all control signals get turned off. For example:

```
-- Set all of the signals that drive onto the bus or
-- load registers to their inactive state
enaALU <= '0';
enaMARM <= '0';
enaPC <= '0';
enaMDR <= '0';
ldIR <= '0';
ldPC <= '0';
ldMAR <= '0';
ldMDR <= '0';
regWE <= '0';
memWE <= '0';
wait for PERIOD;
```

2. Write the reuired simulation cycles for an instruction fetch. The first cycle will be to make all of the signals inactive using the lines above. Then add the assignments to manipulate the signals required to fetch the instruction into IR, then add the inactive cycle again. As a part of fetching an instruction, your fetch maninpulations should increment the PC (it fits into the 2nd or 3rd cycle of the fetch just fine).

   This will need to be repeated for every instruction simulation.

3. The first part of your testbench should include signal assignments which will clear the machine (PC ¡= 0) and test that it works. (Don't forget to stimulate the clk and reset signals).

4. Next fetch an instruction (adding the sequence determined above).

5. Now, look at the binary pattern which was loaded into the IR, write it down, and decide what instruction it is. In order to follow your program, put in a comment after each instruction is fetched which indicates what the instruction is.

6. Next, modify the test bench to execute the fetched instruction. Note: the testbench is executing the instruction - not any circuitry you have designed. The point of fetching it into the IR in this assignment is so you can look at it and figure out what it instruction it is.

Now, repeat the three previous steps until you have a testbench that will fetch and execute all the instructions preloaded into the memory. You will know you are done when the PC goes back to location 0.

An additional suggestion – don't have any of your signals changing near clock edges. At the top of your testbench, run for a partial clock cycle so the simulation doesn't stop on a clock transition edge. From this point on, you can run the simulation for full clock cycles. The point is to NOT have any input signals change coincident with the clock edges. This will clear up any question about which signals really change first.

When you are finished, print out your simulation results and testbench files and then, using colored pencils, clearly mark the execution of each instruction. Clearly show what the disassembled instructions are and how your simulation shows that they work.

Document your VHDL code, testbench files and output waveforms for each functional block. Don't forget your TOC, assumptions and anomalies page.

To pass of this lab, show a TA the waveforms generated by simulating the testbench files you created.