Lab 6: Elementary CPU Design

OBJECTIVES

- Learn how to create a state machine to control a registered ALU (the RALU created in Lab 4), also called a datapath, and create a simple CPU.
- Understand how opcodes are translated into control words inside a CPU.
- Perform basic assembly programming and hand-assemble a program for a CPU.

INTRODUCTION

We have spent the semester so far learning about the building blocks used in all digital circuits. This has culminated in the creation of a RALU that can execute general math operations in Lab 4 and learning how to design state machines in Lab 5. By combining the concepts learned in Lab 4 and 5, we can create a simple central processing unit (CPU), the heart of any digital computer.

In this lab, you will create a controller for the RALU that you created at the end of Lab 4. During Lab 4, you controlled this datapath by providing the MUX select lines directly from your DAD. While this allowed fine grained control of exactly what your RALU was doing at any point in time, it is not an efficient way to encode what operations should be completed and it is error prone if the control word changes close to a rising clock edge. A much better solution is to create a state machine that can interpret short numbers representing each operation that the CPU can perform (called opcodes) and output the corresponding control word sequences to perform that operation. This abstracts away direct control of the RALU and allows you to think about your program as a series of operations that the controller can convert into the required control word values. You can also make a single opcode represent a series of operations to be done, such as an opcode that takes the 2's complement of the values in a register.

LAB STRUCTURE

In this lab, you will be creating and programming a simple CPU. In § 1, you will be creating a state machine that can control your Lab 4 ALU and convert four different opcodes to control word sequences that perform corresponding operations. You will then expand upon these concepts to create a CPU in § 2 that has eight possible opcodes and can read a program made from those opcodes from ROM.

REQUIRED MATERIALS

- Your entire lab kit (including your DAD)
- Note that in this lab, you can use the DAD to generate your inputs and to show your outputs (instead of switch and LED circuits); it is your choice!
- UF's DAD *Waveforms Tutorial*
- *DE10-Lite Pins*
- Document on website: *Explanation_of_Table4* from Lab 4
- *ROM Creation Tutorial*

SUPPLEMENTAL MATERIALS

- *DE10-Lite Manual*
- *DE10-Lite Schematic*
- *PLD on Breadboard Programming WARNING!*

PRE-LAB PROCEDURE

INTRODUCTION: LAB 4 REVIEW

The RALU designed in the second part of Lab 4 consisted of four 4-input MUXs on the inputs of REGA and four 4-input MUXs on the inputs of REGB. The select lines for these MUXs were designated MSA1:0 and MSB1:0, respectively. For a quick review, the MUXs selected a bus as shown in Table 1.

The outputs of REGA and REGB were then passed to a combinatorial logic block and the results of this were then passed to four 8-input MUXs. The select lines for these four MUXs were designated as MSC2:0. For review purposes, these (3) lines selected the functions shown in Table 2.

Table 1: Input source MUXs for Registers A and B.

MSA/ MSB1	MSA/ MSB0	Bus Selected as Input to REGA/REGB
0	0	INPUT Bus
0	1	REGA Bus
1	0	REGB Bus
1	1	OUTPUT Bus

Table 2: ALU function selection MUX (for MUX C).

MSC2:0	Action
000	REGA Bus to OUTPUT Bus
001	REGB Bus to OUTPUT Bus
010	complement of REGA Bus to OUTPUT Bus
011	bit wise AND REGA/REGB Bus to OUTPUT Bus
100	bit wise OR REGA/REGB Bus to OUTPUT Bus
101	sum of REGA Bus & REGB Bus to OUTPUT Bus
110	shift REGA Bus left one bit to OUTPUT Bus
111	shift REGA Bus right one bit to OUTPUT Bus **without** sign extension

There is no need to modify your lab 4 RALU design unless it does not work, although adding a reset input will be necessary in Part 2. We'll call this the **Lab 4* RALU**.

1. FIRST RALU CONTROLLER

A state machine controller and Instruction Register (IR) are now added to the Lab 4* RALU to facilitate the execution of simple instructions. See Figure 1 for the total system components of this section. The **IR** register contains **2 bits** that represent the four instructions shown in Table 3. In this part of the lab, you will ultimately use Quartus to make the project **LAB6_Part1**.

Table 3: Part 1 instructions.

IR1:0	Action	Instruction
00	Move REGA contents => REGB	TAB
01	Sum REGA & REGB => REGA	SUM_BA
10	Load INPUT bus => REGA	LDAA #In
11	Right Shift REGA 1 bit (**logical**, not arithmetic shift) => REGA	SRA

The flowchart (**NOT** an ASM) for the controller is shown in Figure 2. All instructions execute in one cycle (plus one cycle to load the IR register). I strongly encourage you to use VHDL for the combinatorial part of the controller.

Instruction Register Design

The IR is clocked like a typical bank of D flip-flops, however, it has a new feature; it can be loaded or not loaded depending on "IR.LD". When IR.LD is true, data is loaded into the register and when IR.LD is false, new data is not loaded into the register (hold condition). This register can be simply realized with a 2-input MUX (in Quartus, if you want a bdf component, try *21mux*) on the input of each flip-flop of the IR. When a 2-input MUX select line is false, select an IR output to pass through the MUX back into a D-FF input; when the select line is true, an INPUT bus signal should pass through a MUX and into a D-FF input.

Pre-Lab Requirements

1. Use the flowchart shown in Figure 2 to help you create an ASM chart. The ASM's outputs include the MUX select signals (instead of the description of actions).
2. Create a next state truth table. If you use the graphic design editor (block diagram/schematic file) for schematic entry in Quartus to create your controller, you **must** make K-maps and simplified logic equations for the controller. If you use

VHDL (and external flip-flops) to create the controller, you do **not** need to make K-maps or simplify the equations.

3. Using the block diagram/schematic editor in Quartus, add the IR and controller circuitry to your Lab 4* RALU.
4. Simulate and test all instructions created in the controller circuitry. As always, annotate your design simulation.
5. Turn in all the documents described above as stated in the *Lab Rules and Policies* document; re-read, if necessary. (Submit the Quartus archive file **LAB6_Part1.qar**). Documents must be submitted through Canvas for every lab. All pre-lab material is to be submitted as required (at least 15 minutes before the beginning of your lab).

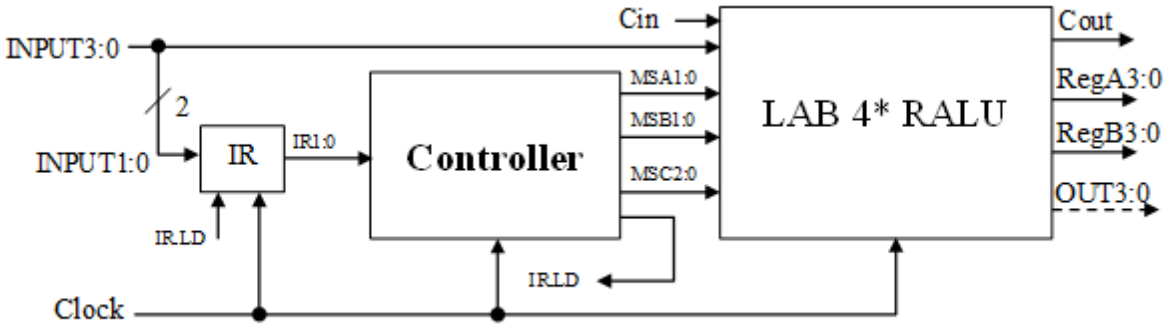

Figure 1. Block diagram of system components.

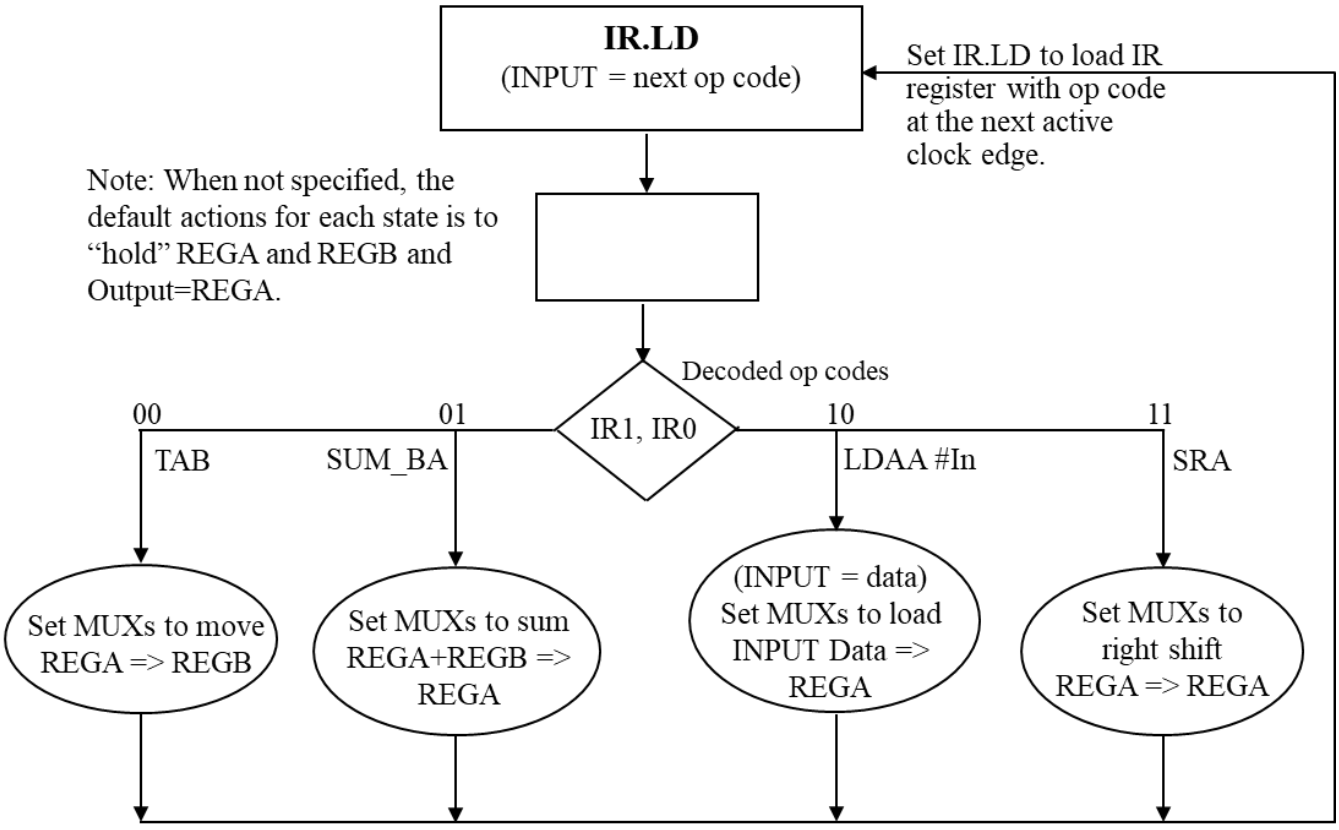

Figure 2. Controller flowchart (**not** an ASM)

PART 1 PRE-LAB QUESTIONS

1. Why did we require the new instruction register in this design?
2. In this section of the lab, you are setting the INPUT bus by hand. If you wanted to read or fetch this value from memory, what could you add to do this automatically for you every CLK cycle?
3. How would you add more instructions (i.e., 8 instead of 4) to the controller?

HELPFUL HINTS

Debug as you design for a much better chance of success. When a design does not work as expected, don't panic! Think of some experiments that you can do to break the problem down into pieces in order to isolate the error. A useful tool for debugging a design is to add outputs for some of the internal signals, i.e., signals that are neither outputs nor inputs of your design. This will allow you to "peer inside" a design both in simulation and with the actual hardware.

2. SECOND RALU CONTROLLER WITH ROM

The main difference between parts 1 and 2 of the lab is in the way the inputs are generated. In part 1, you input the opcodes (i.e., 00, 01, 10, or 11) and data manually. The opcodes and inputs were entered between every active clock transition with the switches at INPUT3:0. In this section (part 2), the opcode and data will be stored in memory. Your controller will control the signals in such a manner that the op code and data are automatically fetched from memory; the outputs of this memory are inputs INPUT3:0. A program counter (PC) will coordinate the sequencing of the instructions by stepping through the addresses in an appropriate manner. In this part of the lab, you will ultimately use Quartus to make the project LAB6_Part2.

SPECIFICATIONS

No changes will be made to MUXA, MUXB, MUXC, REGA or REGB from the Lab 4* RALU.

1. As shown in Figure 3, a 32k × 8 EEPROM (or Flash) is added. The instructions and data will be stored in this EEPROM **starting in location $2B70.**
2. A program counter (PC) is added. PC is a 4-bit up-counter with a synchronous count enable signal (PC_INC). If PC_INC is TRUE, the counter will increment by 1 at the next active clock transition. If PC_INC is FALSE, the counter holds its current value. The count after 1111_2 is 0000_2. Another synchronous signal, PC_LD, is used to load the counter from the INPUT bus. The 74'161 (with **asynchronous** clear) is ideally suited to function as a 4-bit PC. (The 74'163 is identical except that it has a **synchronous** clear.)
3. The instruction register (IR1:0) from part 1 of this lab is increased to 3 bits (IR2:0), as shown in Table 4.

Table 4. Part 2 instructions.

IR2:0	Instruction	Function
000	TAB	Copy A to B (transfer A to B)
001	ABA	Load A with A plus B plus Cin; update Cout
010	LDAA #data	Load A with input data
011	SAR	Shift A right 1 bit, store in A (**logical**, not arithmetic shift)
100	SAL	Shift A left 1 bit, store in A
101	JMP Addr	Load PC with input address
110	Future use	
111	Future use	

Changes to the ASM chart:

1. All the manual switching that you would need to do if part 1 of this lab was built (e.g., setting the INPUT = next op code or data) can be better accomplished by incrementing the address on the ROM. The ROM will have the information that (in part 1 of this lab) would have come from switches (or your DAD). This is accomplished by incrementing the PC register or "Inc PC," as shown in the Figure 4 flowchart.
2. An additional state is necessary for the LDAA instruction in order to read the memory a second time to obtain the data to place in register A.
3. The instruction SUM_BA from part 1 is now spelled ABA (which stands for add A to B and put the result into A).

4. Note that the right shift is a **logical** shift, **NOT** an arithmetic shift, i.e., a zero is shifted into the most significant bit of REGA with the SAR instruction. Shift register A right (SRA) from part 1 is spelled differently here, now as SAR.
5. The first new instruction is called "JMP Addr". JMP Addr consists of two nibbles where the first is the opcode and the second is an address. This instruction forces the PC to load a 4-bit address read from memory (2nd nibble).
6. The other new instruction is shift register A left (SAL). A zero should be shifted into the right most bit with this shift.

MIF FILE CREATION INFORMATION

When you write code in pre-lab part 4, below, you will hand assemble your code and put it into the "rom_**32**k.mif" file. A sample mif file (rom_8k.mif) can be found on the website and another mif file, rom_1k.mif, in the tutorial. These files can be used as templates to generate your own file. Key points related to these files are:

1. The comments are surrounded by "%"symbols. The left most number represents address space followed by the hex value to the right. For example:

```
"memory address" :"memory value"     %comment%
```

2. The last line of code in the "rom_32k.mif," file (after your program) should zero out the remaining data in ROM. In the rom_8k.mif file, the end of memory was filled with $FF using the line below:

```
[8A..1FFF]  :   00;
```

Your last line will be

```
[XX..7FFF]  :  0;
```

In the above, **XX** represents the next address after the last address of your code. This will initialize all your remaining unused memory to a known value of zero.

PRE-LAB REQUIREMENTS

1. Create an ASM chart using the Figure 4 flowchart as an aid, i.e., utilize the actual signals to control the PC, IR, and the MUXs. Complete the ASM diagram, by also including the required elements for the SAR and SAL instructions.
2. Create a next state truth table. If you use the block diagram/schematic editor in Quartus to create your controller, you **must** make K-maps and use simplified logic equations for the controller. If you use VHDL (and external flip-flops) to create the controller, you do **not** need to make K-maps or simplify the equations.
3. Add an **active-low asynchronous RESET** signal to all registers and the counter. You should use this to initialize all flip-flops (and the 4-bit counter) to zero before beginning your testing. (You should design every state machine so that you can start it in a known state. For this lab, the known state has state bits of all zeros.)
4. Hand assemble the program in Table 5 and complete the table, adding the address on the left and the machine codes (the values that will be stored in the ROM). The successive columns of A and B should contain the changing values of RegA and RegB as the program is executed and the loop (created by the JMP instruction) causes the code to repeat.

Table 5: Program to assemble.

Addr		Mach Codes	A	B	A	B	A	B	A	B
$2B70	LDAA #7									
	TAB									
	LDAA #3									
	ABA									
	SAR									
	ABA									
	ABA									
	JMP 5									
	ABA									
	TAB									

You can separately simulate the flash memory portion of your design in Quartus, just as you can separately test the controller, PC, and you already tested the Lab4 RALU design. Read through the documentation in the ***ROM Creation Tutorial***, available on our website. Also see the below section on MIF file creation.

Implement the design in Quartus (**LAB6_Part2**) and simulate the execution of this program. Use the ***ROM Creation Tutorial*** to create the ROM: 1-PORT. You should start your simulation with a reset.

You must make a MIF file for this program. If you did not simulate a ROM, you would need to input the op codes and data in the *.vwf simulation waveform file (just as you did in Lab 4), which is much harder than making the ROM and MIF file. Use a **functional compilation and simulation**. As always, annotate your design simulation. Outputs should include the state bits, the registers, INPUT3:0, OUT3:0, IR.LD, IR, PC, PC_LD, and PC_INC. During debugging, you should also add the MUX select lines.

Note that the memory clock should be **at least twice as fast as the state machine clock** to assure that the ROM data is available at the proper time. (See the ***ROM Creation Tutorial*** for more information.)

The simulation technique used above will be used again in Lab 7.

Pre-lab Simulation and Programming Summary:

- Create a **32k** × 8 ROM with the program's machine codes.
  - Create a MIF file with these machine codes.
  - Simulate this design (**LAB6_Part2**).
  - Program this design to your DE10-Lite

PRE-LAB QUESTIONS

1. Why do we need the extra states in the LDAA and JMP instruction paths?
2. What do you need to do to the address lines to get your program to start at address $37D0 (instead of $2B70)?

PRE-LAB PROCEDURE SUMMARY

1. Design a RALU controller that can interpret 4 opcodes in § 1.
2. Design a more advanced RALU controller that can interpret 8 opcodes and read instructions from ROM in § 2.
3. Program your Part 2 CPU with a sample program at the end of § 2.

IN-LAB PROCEDURE

You will demonstrate your Table 5 program executing on your Part 2 RALU controller on your DE-10 Lite board. Use a **debounced clock** for your design. To facilitate this demo, please display REGA and REGB on two seven segment displays on your DE10-Lite.

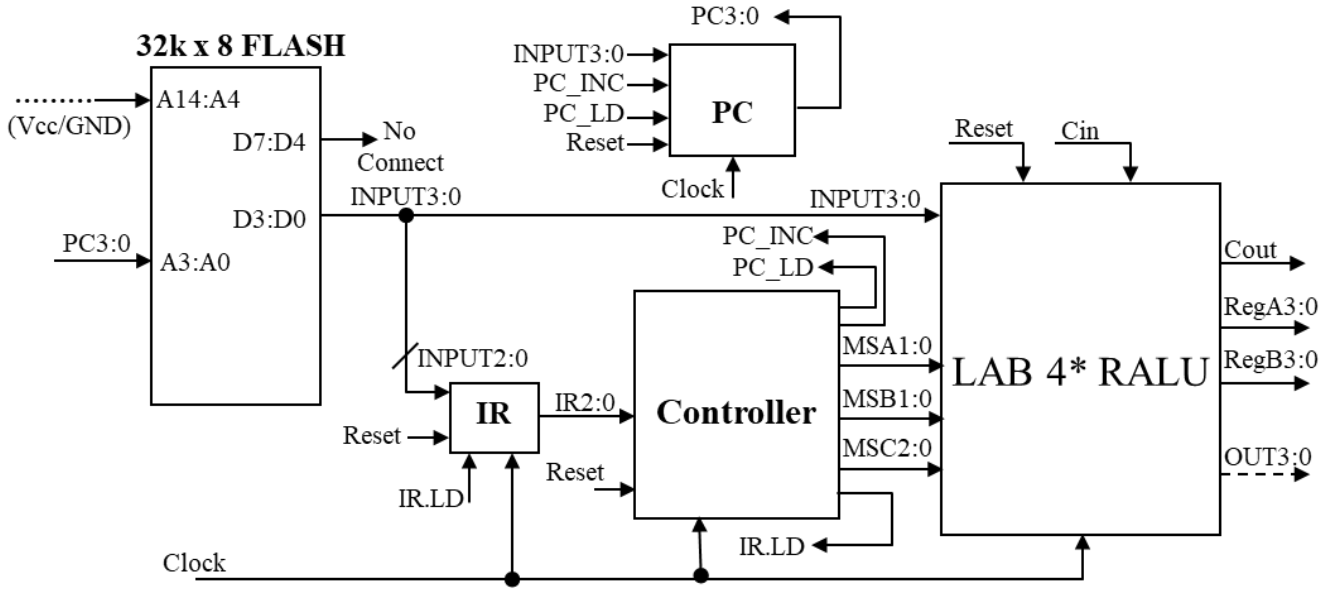

Figure 3. Block diagram of system hardware.

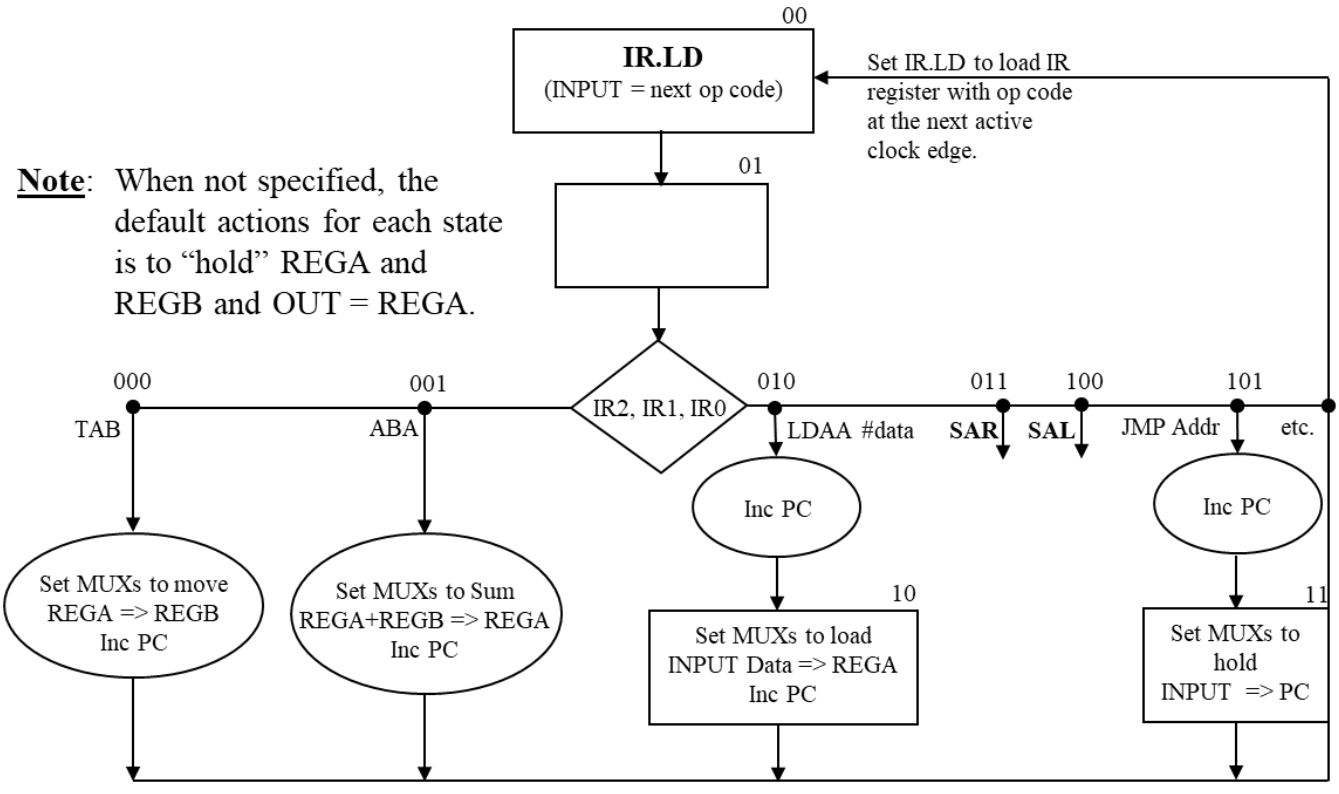

Figure 4. Controller flowchart (**not** an ASM).