LAB 6: Elementary CPU Design: Fetching Instructions From a ROM

OBJECTIVE

The objective of this lab is to continue the design of an elementary central processing unit (CPU) that was started in Lab 4. In part 1 of this lab, a 2-bit instruction field will be used to control a simple state machine that in turn will be used to set the MUX lines in the ALU according to what type of instruction is designated for execution. In part 2, a 3-bit instruction field will be used. You will also add a program counter (PC) and a memory module to store the instructions of a "program" to be executed by the CPU.

MATERIALS

- Your entire 3701 kit including your PLD PCB Wires, Resistor packages, Switches (including your debounced switch circuit), LEDs, DAD/NAD
- VHDL ROM Alternative files:
  - Instructions: VHDL_ROM.pdf
  - Quartus archive file, Board_ROM.qar
  - Excel spreadsheet, ROM_contents.xlsx

INTRODUCTION - LAB 4 ALU MUX SIGNALs

The ALU designed in 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.

MSA1/ MSB1	MSA0/ MSB0	Bus Selected as Input to REGA/REGB
0	0	INPUT Bus
0	1	REGA Output Bus
1	0	REGB Output Bus
1	1	OUTPUT Bus

Table 1. MUX A and B settings.

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.

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

Table 2. MUX C settings.

Modify your lab 4 ALU design only if it did not work. We'll call this the **Lab 4* ALU**.

PART 1 INTRODUCTION: 1st ALU CONTROLLER

A state machine controller and Instruction Register (IR) are now added to the Lab 4* ALU 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. Note that the right shift is **NOT** sign-extended, i.e., a zero is shifted into the most significant bit of REGA with the SRA instruction. In this part of the lab, you will ultimately use Quartus to make the project **`LAB6_Part1`**.

IR1:0	Action	Instruction
00	Load INPUT bus => REGA	LDAA #In
01	Move REGA contents => REGB	TAB
10	Right Shift REGA 1 bit => REGA	SRA
11	Sum REGA & REGB => REGA	SUM_BA

Table 3. Part 1 instructions.

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 on the input of each flip-flops of the IR. When a 2-input MUX select line is low, select an IR output to pass through the MUX back into a D-FF input; when the select line is high, an INPUT bus signal should pass through a MUX and into a D-FF input.

PART 1 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* ALU.
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. (Submit the Quartus archive file **`LAB6_Part1.qar`**). Re-read, if necessary. Documents must be submitted through Canvas for every lab. All pre-lab material is to be submitted BEFORE the beginning of your lab.

LAB 6: Elementary CPU Design: Fetching Instructions From a ROM

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?

PART 1 IN-LAB REQUIREMENTS

None. You do **not** need to build the circuit of part 1.

HELPFUL HINTS

Debug as you design for a much better chance of success. When something goes wrong, i.e., 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.

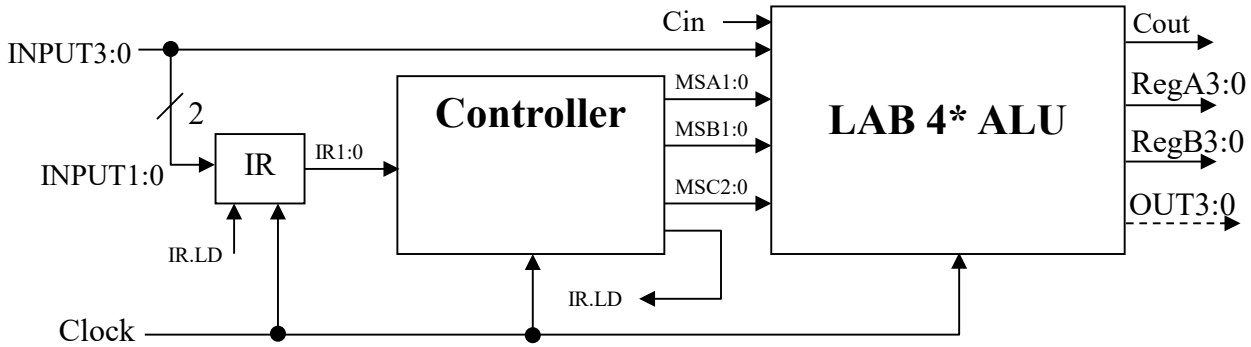

Figure 1. Block diagram of system components.

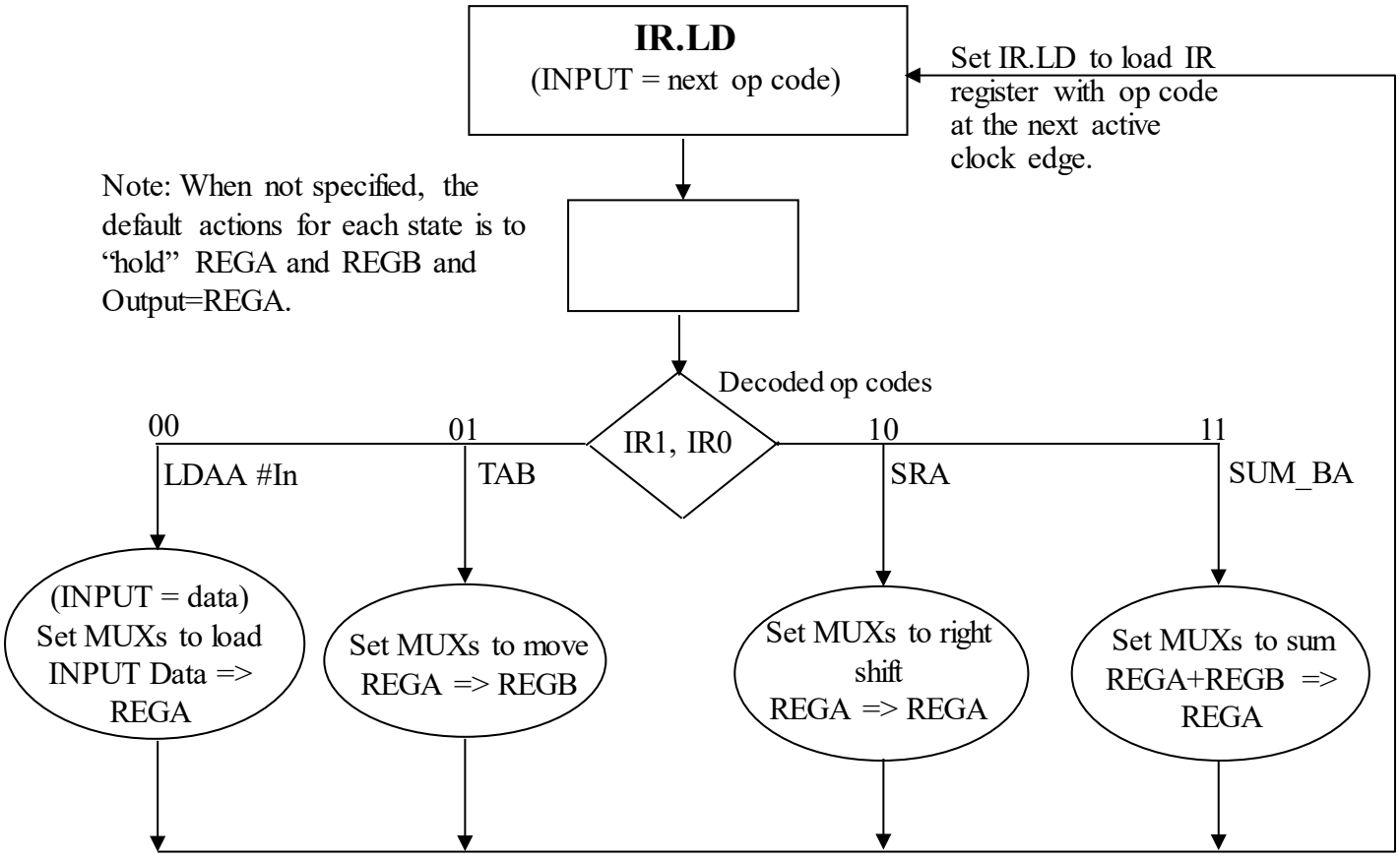

Figure 2. Controller flowchart (**not** an ASM)

LAB 6: Elementary CPU Design: Fetching Instructions From a ROM

PART 2 INTRODUCTION: 2nd ALU CONTROLLER

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 ***op codes*** (i.e., 00, 01, 10, or 11) and ***data manually***. The op codes and inputs were entered between every active clock transition with the switches at INPUT3:0. In this section (part 2), the op code 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 instruction 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`** and then **`LAB6_Part2_VHDL_ROM`**).

SPECIFICATIONS

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

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 $3E70.** (See Pre-Lab question 1.)
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 is 0000. 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.
3. The instruction register is increased to 3 bits (IR2:0), (from part 1 of this lab) with the following definition:

IR2:0	Instruction	Function
000	TAB	Copy A to B (transfer A to B)
001	LDAA #data	Load A with input data
010	SAR	Shift A right 1 bit , store in A (**no** sign extension)
011	SAL	Shift A left 1 bit, store in A
100	ABA	Load A with A plus B plus Cin; update Cout
101	JMP Addr	Load PC with input address
110	Future use	
111	Future use	

Table 4. Part 2 instructions.

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. This is accomplished by incrementing the PC register or "Inc PC," as shown in the Figure 4 flowchart.
2. Notice that an additional state is necessary for the LDAA instruction in order to read the memory a second time to obtain the data to placed 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). Shift register A right (SRA) from part 1 is also spelled differently here, now as SAR. In both parts, sign extension is **not** used for this shift.
4. 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).
5. The other new instruction is a shift register A left, spelled 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_32k.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 aide; i.e., put in 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.

LAB 6: Elementary CPU Design: Fetching Instructions From a ROM

3. Add an **active-low asynchronous RESET** signal to all registers and counters. You should use this to initialize all flip-flops 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 below program and complete the table. The successive columns of A and B are for the changing values of RegA and RegB as the program is executed and the loop causes the code to repeat.

Addr		Mach Codes	A	B	A	B	A	B	A	B
$3E70	LDAA #7									
	TAB									
	LDAA #3									
	SAL									
	ABA									
	SAR									
	TAB									
	JMP 5									
	LDAA $F									

Implement the design in Quartus (**LAB6_Part2**) and simulate the execution of this program. You should start your simulation with a reset.

You can simulate the flash memory portion of your design in Quartus. Read through the documentation on ***ROM Creation Tutorial*** on our website. Also see the below section on MIF file creation.

You must make a MIF file for this program. If you did not simulate a ROM (or flash memory), you would need to input the op code 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. You can**not** compile an entire 32k × 8 EEPROM with any devices in MAX X family, but you can compile and simulate a 32k x 8 EEPROM if you select the Cyclone V family and the device at the bottom of the list in Quartus in the *Assignments* | *Device* menu. Use a **functional compilations 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 might also need to add the MUX select lines.

Note that the memory clock should be **at least twice as fast as the state machine clock** in order 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.

Although there is a flash ROM on your OOTB PLD PCB, we will instead use the ***VHDL ROM alternative*** procedure described on the website and put the entire design in Quartus (**LAB6_Part2_VHDL_ROM**). In addition to the one done in part 4, above, you must also do a second simulation with the ***VHDL ROM alternative***.

5. Download your design to the PLD board. **Warning**: Tri-state all unused PLD pins, as described in the Quartus Tutorial.
6. Appropriately connect the flash, PLD PCB, LEDs, and switches on your bread board; verify that it functions as specified in the Pre-Lab Requirements. You will need a debounced switch for the CLK input.
7. Run the program.

Pre-lab Simulation and Programming Summary:

- Create a 32k ×8 ROM with the program's machine codes, using a Cyclone V device.
  - Create a MIF file with these machine codes.
  - Simulate this design.
- Use the VHDL ROM alternative for the ROM with your MAX 10. Simulate this design.
  - Program this design to your MAX 10 PLD

PART 2 PRE-LAB QUESTION

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 $4370 (instead of $3E70)?

PART 2 IN-LAB REQUIREMENTS

1. Demo the program (step 12 in the prelab).
2. Write a new program (given to you by your Alpha); handle assemble the program; program your flash; run the program.

LAB 6: Elementary CPU Design: Fetching Instructions From a ROM

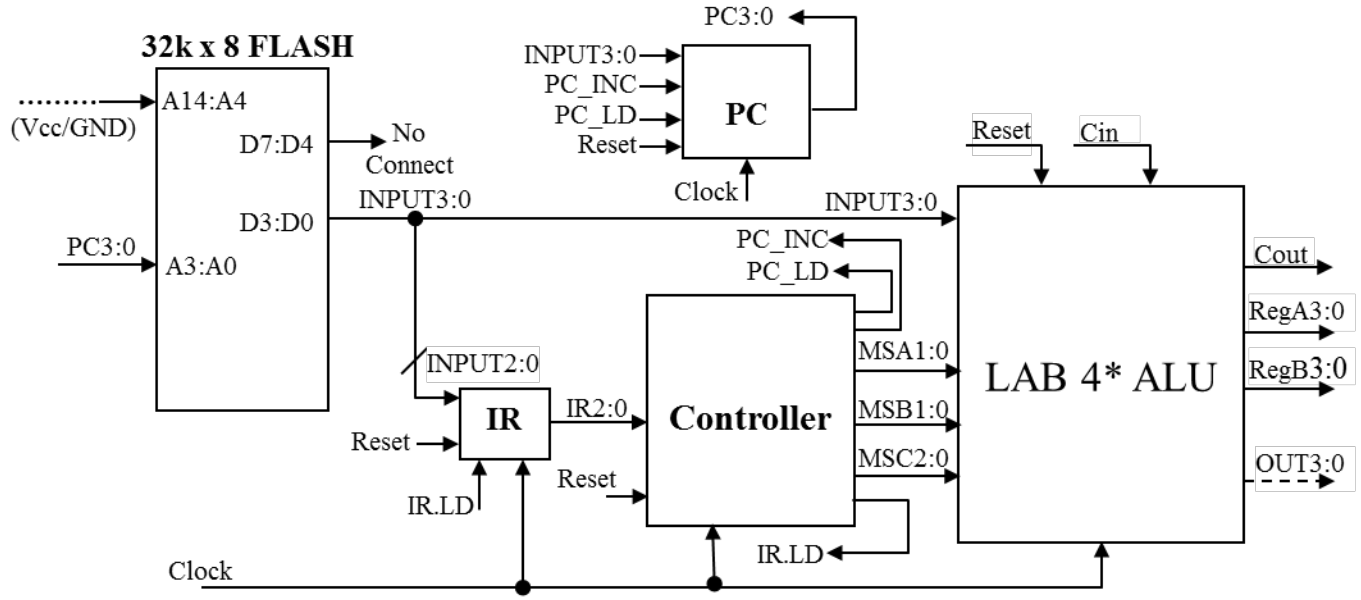

Figure 3. Block diagram of system hardware.

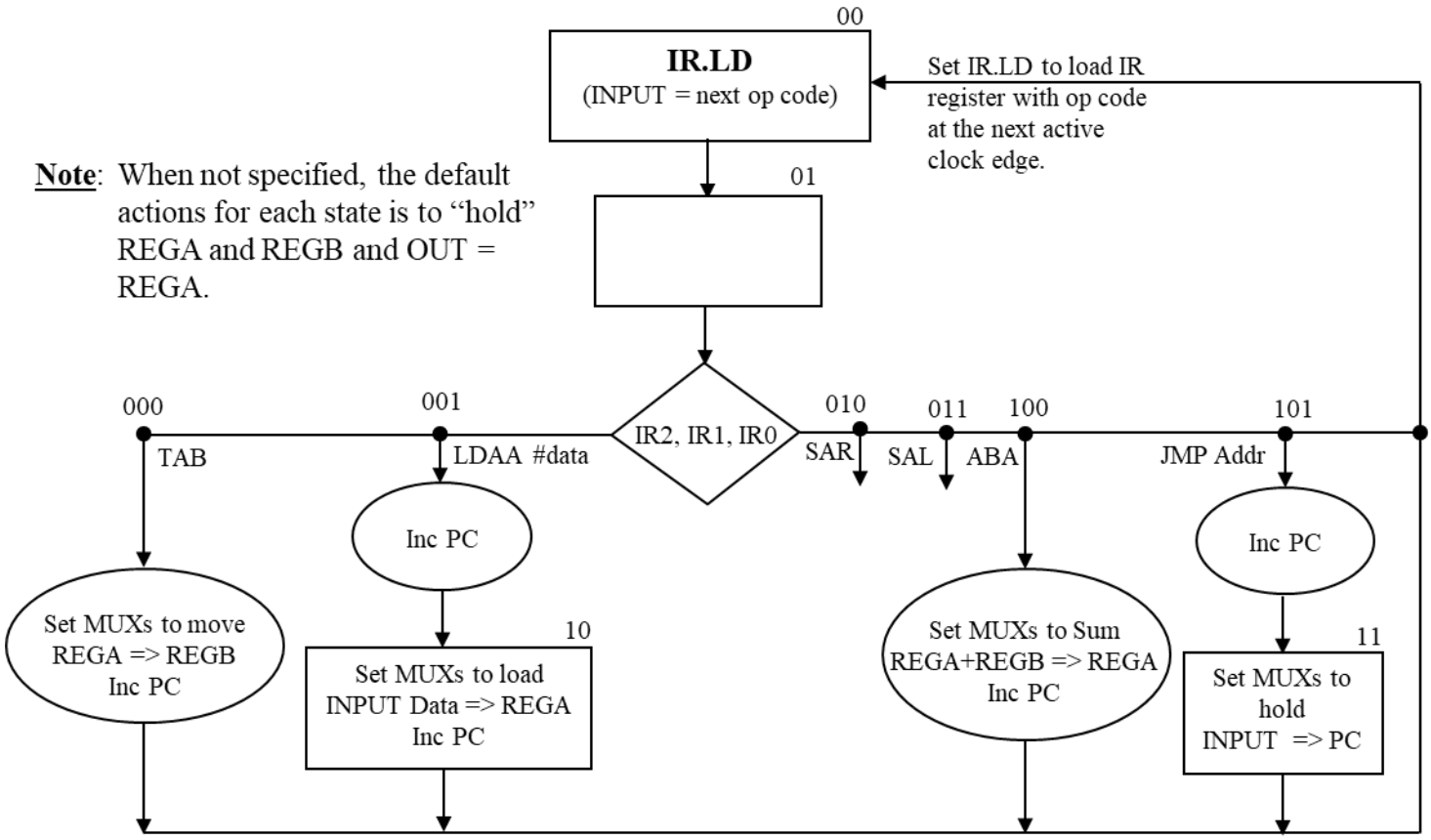

Figure 4. Controller flowchart (**not** an ASM).