



ECE410-Advanced Digital Logic Design

LAB 3: A Simple CPU Implemented with a Controller and Datapath

Fall 2023

DATES

Date	Time	Section	Demo Due Date	Report Due date
Session 1 06-Nov-2023 Session 2 20-Nov-2023 Session 3 04-Dec-2023	2:00 to 4:50 PM	D11	04-Dec-2023	08-Dec-2023
Session 1 08-Nov-2023 Session 2 22-Nov-2023 Session 3 06-Dec-2023	2:00 to 4:50 PM	D31	06-Dec-2023	08-Dec-2023
Session 1 10-Nov-2023 Session 2 24-Nov-2023 Session 3 08-Dec-2023	2:00 to 4:50 PM	D51	08-Dec-2023	08-Dec-2023

INTRODUCTION

In this lab you are required to modify and verify the functionality of a simple central processing unit (CPU) implemented using standard digital hardware components. The VHDL model for the simple CPU is provided as a starting point, and you are required to add new functionality to the existing design. In this lab you will gain insight into how a CPU that executes machine language instructions can be implemented using a controller and datapath architecture.


LEARNING OBJECTIVES

- To learn how to design and verify an instruction-handling datapath using conventional Register Transfer Level (RTL) hardware blocks for a simple CPU.
- To implement the Finite State Machine controller for a CPU that sequences the fetching, decoding, and execution of the binary instructions that are stored in the program memory.
- To integrate the controller and datapath to form a working CPU. Further, this design will be integrated with the seven-segment display of the FPGA to provide a rudimentary user interface.

PRE-LAB

Marks 5%

- Carefully read the lab manual before proceeding with the VHDL design.
- Read the description provided for the working CPU that is provided in this manual. Also, review the comments that are embedded in the VHDL code files.
- Draw a detailed RTL diagram of the datapath module that is being used in this lab. Be sure to include all the incoming control signals, the internal datapath and the outgoing status signals. Note that there is a

	<h1 style="text-align: center; color: blue;">ECE410-Advanced Digital Logic Design</h1>
<p>Fall 2023</p>	<h2 style="text-align: center; color: green;">LAB 3: A Simple CPU Implemented with a Controller and Datapath</h2>

template for the datapath given to you but that template is missing signals. Complete the entire datapath logic using this template.

- The simple CPU in this lab provides load, store, arithmetic, rotate and jump instructions. A partially verified instruction decoder state machine for some instructions will be given in the controller finite state machine (FSM) (controller.vhd). To implement the remaining instructions in this FSM, you must know the basic control flow for fetching, decoding and executing instructions.
- All of the pre-lab work will be checked by the LI/TA before the starting of the lab. In case of any problems or questions ask the LI/TA for clarification and help.
- Neglecting the pre-lab will very likely hinder you from completing the assigned tasks, so please make sure that you understand the logical operation of the controller and datapath, following their integration.
- You are strongly encouraged to follow the code template from the e-Class. However, you can choose to write your own code from scratch provided it satisfies all of the lab requirements.

NOTE: Pre-lab work will be conducted during the lab session under the guidance of the lab teaching staff. They will review your work, provide clarification on the assigned tasks as needed, and allocate marks for the completion and understanding of this work.

BACKGROUND

A digital computer is a digital system capable of executing data processing operations according to the instructions contained in a stored program. A program is a sequence of machine language instructions (encoded as binary vectors) that specify the operations, operands, and the default order in which the instructions are executed. Unlike other application-specific systems, a computer can perform different data processing tasks by simply modifying the stored program. The ability to execute arbitrary sequences of instructions from an instruction set is the most important property of the general-purpose, software-programmable computer.

Like other digital systems, the CPU of a computer consists of both sequential and combinational logic. As you will see in this lab, a very simple CPU can be constructed from counters, registers, multiplexers and an ALU which performs arithmetic and logical operations.

Once one has decided on the instruction set, one can proceed to design a datapath that will handle and execute all of the instructions in the set. Once a suitable datapath has been created to manipulate the stored instructions and data, the controller can be designed using an FSM that will assert the required control signals to the datapath. An example CPU architecture with controller and datapath is shown in Figure 1.



LAB 3: A Simple CPU Implemented with a Controller and Datapath

Fall 2023

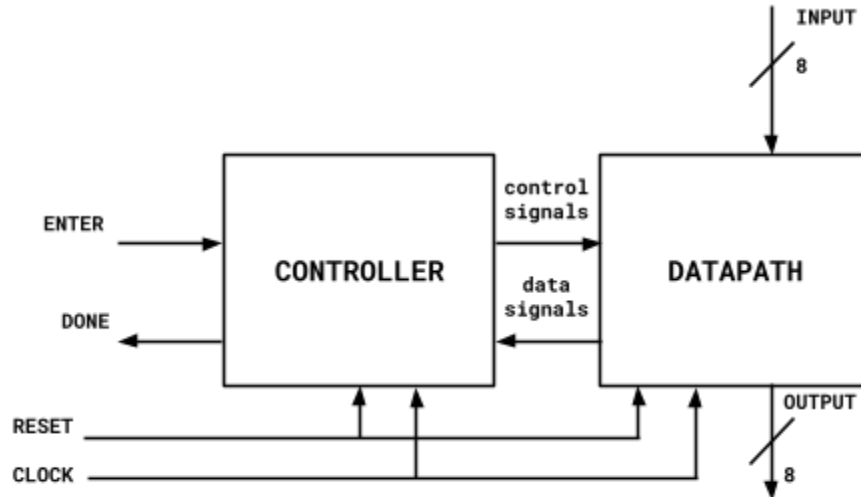


Figure 1: CPU architecture


The FSM in the control unit processes each instruction in three steps: **Fetch, Decode and Execute**.

Fetch cycle: Read the next instruction from the memory into the Instruction Register (IR) in the datapath.

Decode cycle: Extract the first op-code bits from the new instruction and then determine what the current instruction is. Then jump to the first state in the controller state sequence that has been assigned for executing that instruction.

Execute cycle: Starting in that particular state, advance from state to state asserting the appropriate control signals that sequence the datapath to execute that instruction.

Instructions will be stored in the program memory (not shown above) and that is external to the CPU. In this lab, the program memory will actually be located inside the FPGA and implemented as a 32-word memory, where each word contains 8 bits. The instructions used in the provided CPU are shown in Table 1 with their respective opcodes and description. The number of implemented instructions is determined by the number of bit patterns used to encode all of the instructions. In the simple CPU all instructions are encoded using 1 byte except for the instructions that have a memory address or immediate data as the second operand. A one-byte instruction will be encoded as: xxxx 0010, where xxxx is a 4-bit operation code (opcode) and the last three bits indicate the CPU register number. For all the instructions the accumulator will be the first operand by default. The accumulator is a register that is used in many simple CPU designs to receive the result of many of the operations, like arithmetic operations. If the second operand is the register (R[0]: R[7], in the simple CPU) then the last three bits in the encoded instruction specify the register number. For example: [STA R\[2\]](#), A is a one-byte store instruction which stores the contents of the accumulator into the register R[2].

	<h1>ECE410-Advanced Digital Logic Design</h1>
<p>Fall 2023</p>	<h2>LAB 3: A Simple CPU Implemented with a Controller and Datapath</h2>

The results for all the arithmetic operations in the simple CPU will be stored in the accumulator. Consider the two-byte LDI Instruction. The second byte is an immediate unsigned binary value that is to be loaded into the accumulator.

Instructions, like Increment, Decrement and AND, will perform the required operation on the accumulator and will store the result back into the accumulator overwriting its existing contents. For the Jump Instruction used in this lab, the first four bits will specify the opcode and the next four bits specify either absolute or relative Jump. In our case, the last four bits as “0000” specifies absolute jump. In case of relative jump, there will be a sign and magnitude parameter that indicates the displacement from the current instruction location as specified in the Program Counter (PC). In our case, the JMPZ instruction is an absolute Jump to the provided address in program memory and this will be encoded as a two-byte instruction. Encoding of the first byte “1101 0000” specifies 1101 as the opcode for JMPZ and 0000 as absolute Jump. The second byte encoded for this instruction specifies the memory location to which the PC jumps if the JMPZ condition is true. Two conditional flags are used in this design, *zero* and *positive* flag. These flags are set or reset depending on the value of the accumulator when the accumulator is written. The JMPZ instruction a.k.a Jump if Zero, reads the value of the zero flag and decides whether to jump or not.

For the rotate right ROTR the first 4-bits specify the opcode, and the last 4 bits indicate the number of bits to shift right.



ECE410-Advanced Digital Logic Design

LAB 3: A Simple CPU Implemented with a Controller and Datapath

Fall 2023

#	Instruction	Opcode	Byte encoding	Operation (with explanation)
1	INA	0001	0001 0000	Read the 8-bit user input into the accumulator
2	LDI A, imm	0010	0010 0000 xxxx xxxx	Load the immediate value xxxx xxxx into the accumulator
3	LDA A, R[rrr]	0011	0011 0rrr	Load the value from register R[rrr] into the accumulator
4	STA R[rrr], A	0100	0100 0rrr	Store the value from the accumulator into register R[rrr]
5	ADD A, R[rrr]	0101	0101 0rrr	Add the contents of the accumulator and the register R[rrr]. Store the result into the accumulator
6	SUB A, R[rrr]	0110	0110 0rrr	Subtract the contents in register R[rrr] from the accumulator. Store the result back into the accumulator.
7		0111		left for implementation
8	ROTR A	1000	1000 00xx	Rotate the accumulator contents left by xx-bits
9	INC A	1001	1001 0000	Increment the value stored in the accumulator
10	DEC A	1010	1010 0000	Decrement the value stored in the accumulator
11	AND A, R[rrr]	1011	1011 0rrr	Bitwise AND operation of the accumulator and register R[rrr]
12		1100		left for implementation
13	JMPZ	1101	1101 0000 000a aaaa	This instruction checks the zero flag. If zero_flag=1 i.e.(A=0) then absolute jump to the memory address 000a aaaa.
14	OUTA	1110	1110 0000	Display the 8-bit result from the accumulator
15	HALT	1111	1111 0000	Stop/Halt execution

Table 1 Lab 3 CPU Instructions



LAB 3: A Simple CPU Implemented with a Controller and Datapath

Fall 2023

Note:

- **A**: accumulator
- **R[x]**: any of R[0]: R[7] with rrr=000 : 111
- **aaaaa**: specifies a memory address in the program memory
- **xxxx xxxx**: specifies the immediate data operand.

Figure 2 shows the state transition decoding sequence that is used for three of the implemented instructions.

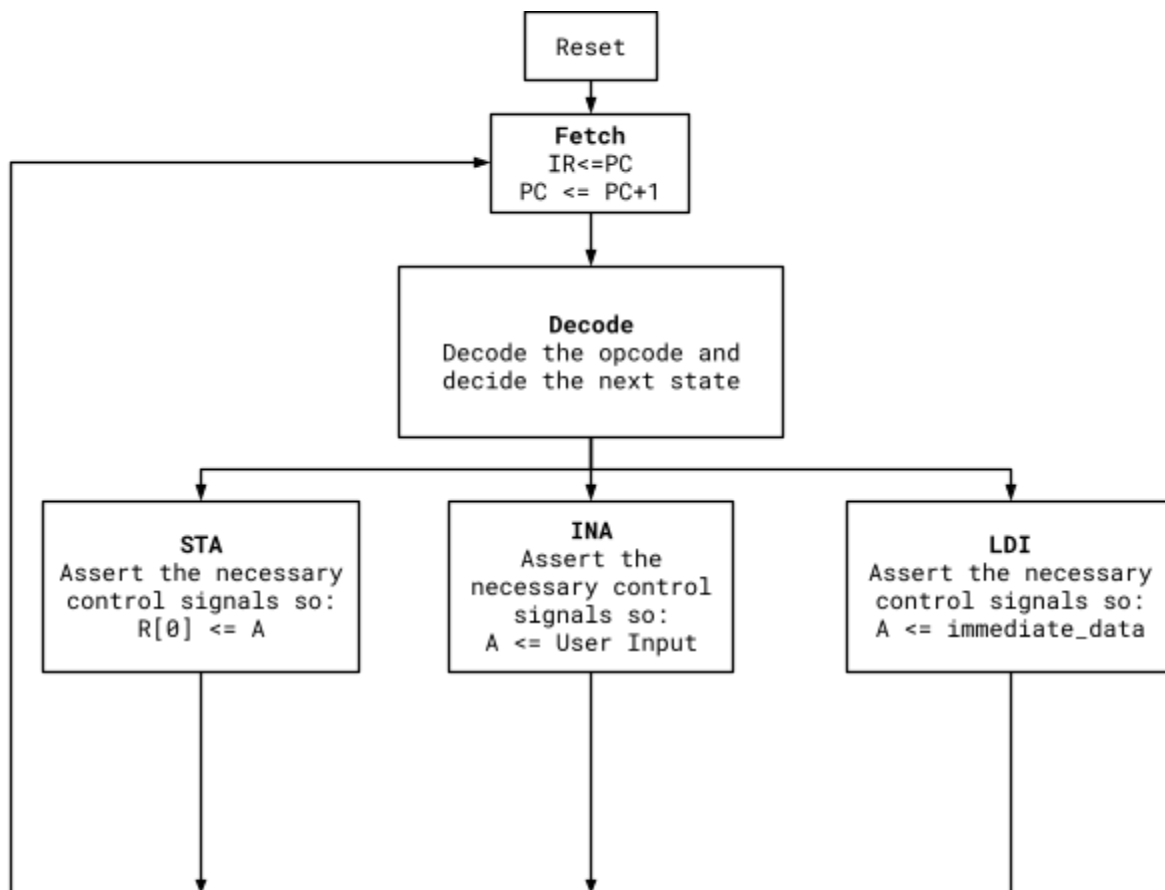


Figure 2: Instruction decoding state sequence for the initial instructions: STA, INA, HALT



ECE410-Advanced Digital Logic Design

LAB 3: A Simple CPU Implemented with a Controller and Datapath

Fall 2023

The simple CPU is designed using the standard controller-datapath architecture. The associated divide-and-conquer strategy allows you to simplify the problem by isolating the FSM that implements the data processing algorithm (the controller) from the datapath hardware that directly handles the program instructions including the embedded immediate data values. The datapath in the provided VHDL code is described structurally, while the control unit is described behaviorally. You will also be required to modify the datapath so that it can execute new instructions.

Figure 3 below shows the datapath template that you will use to identify the missing signals. The explanation for all the RTL components used in this datapath is given below. Take, as a starting point reference, the datapath component VHDL files to complete this datapath logic, and refer to the detailed explanations of each component on the following page for additional guidance.

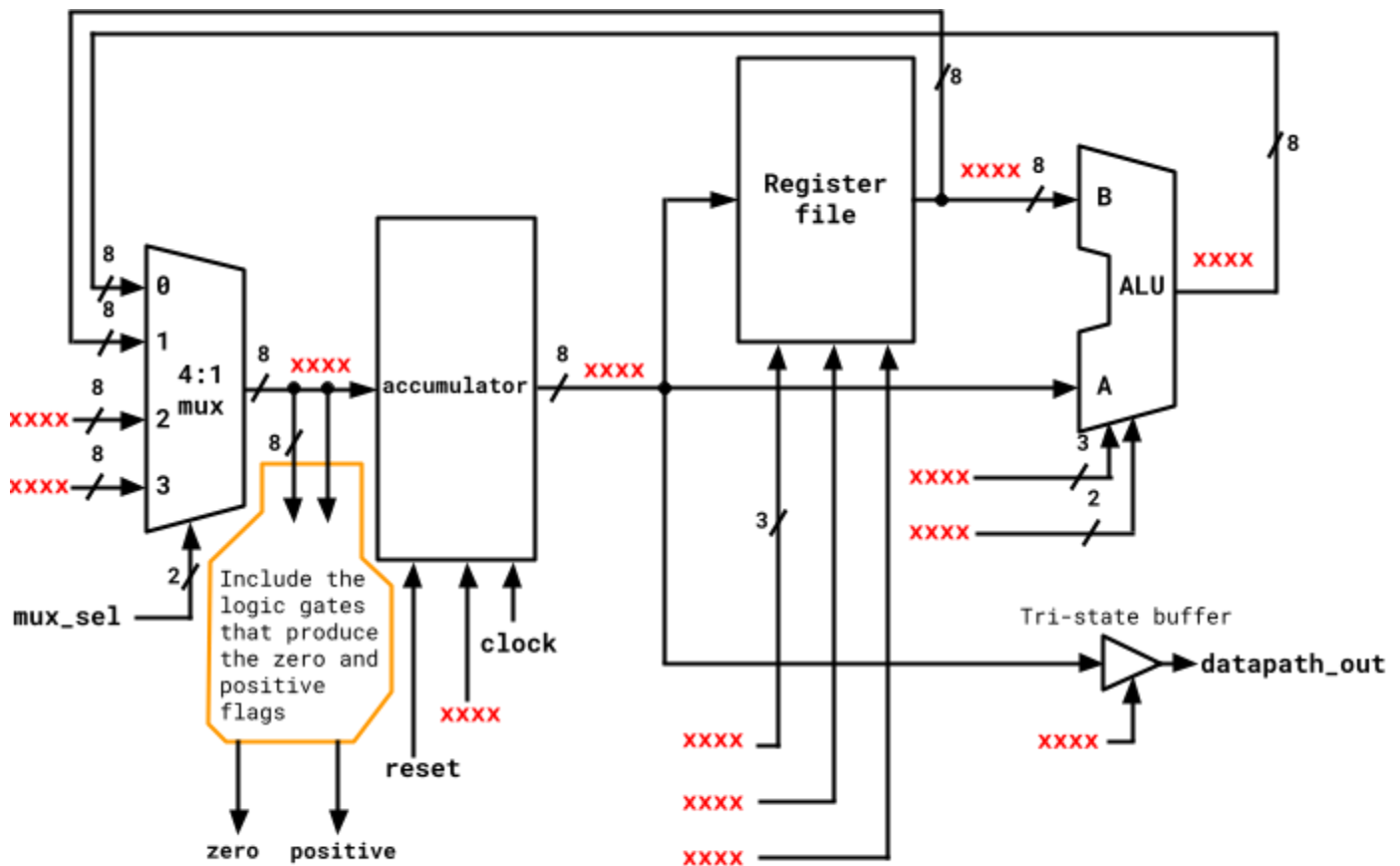


Figure 3: Datapath template



ECE410-Advanced Digital Logic Design

LAB 3: A Simple CPU Implemented with a Controller and Datapath

Fall 2023

Datapath components:

There is a separate VHDL file for each of the components (Multiplexer, Accumulator, Register file, ALU and tri-state buffer) in the data path file.

4:1 Multiplexer: This multiplexer selects between four 8-bit inputs to be written to the accumulator, according to Table 2.

Multiplexer inputs	
#	name
0	ALU output
1	Register file output
2	Immediate data
3	User input

Table 2: multiplexer inputs

Z and P flags: Use the most significant bit in the accumulator to check for the positive number flag at the output of the 4:1 mux. To check for the zero flag, you will need to provide a logic in the datapath that asserts the zero flag to '1' when the new number in the accumulator is 0.

Accumulator: The accumulator is an 8-bit register with enable and clear (reset) control input signals. The accumulator output goes to three different places:

- 1) the register file.
- 2) the A input to the ALU.
- 3) output buffer for display to the user..

Register File: This component contains 8 registers that are each 8-bits wide. The registers can be written by selecting the one register using the three address bits and one write enable line.



ECE410-Advanced Digital Logic Design

LAB 3: A Simple CPU Implemented with a Controller and Datapath

Fall 2023

ALU- This ALU implements the 8 operations shown in table 3:

ALU address bits (2:0)	Operation
000	Pass A to output
001	Bitwise AND
010	Rotate Left
011	Rotate Right
100	ADD
101	SUB
110	INC
111	DEC

Table 3: ALU operations

Output Buffer: This output buffer has an enable signal which, when asserted, causes the output of the accumulator to be stored into the buffer. The buffer will be displayed to the user on the two-digit seven segment LED display as the CPU output.

Program Memory:

Instructions to be executed are typically stored sequentially in an external memory such as ROM. However, to keep our design simple, instead of having an external memory chip, we implement the memory along with the CPU inside the FPGA; specifically, the program memory is implemented as a 32-word array with each array word being 8 bits wide.

The project component hierarchy is shown in Figure 4 on the next page.

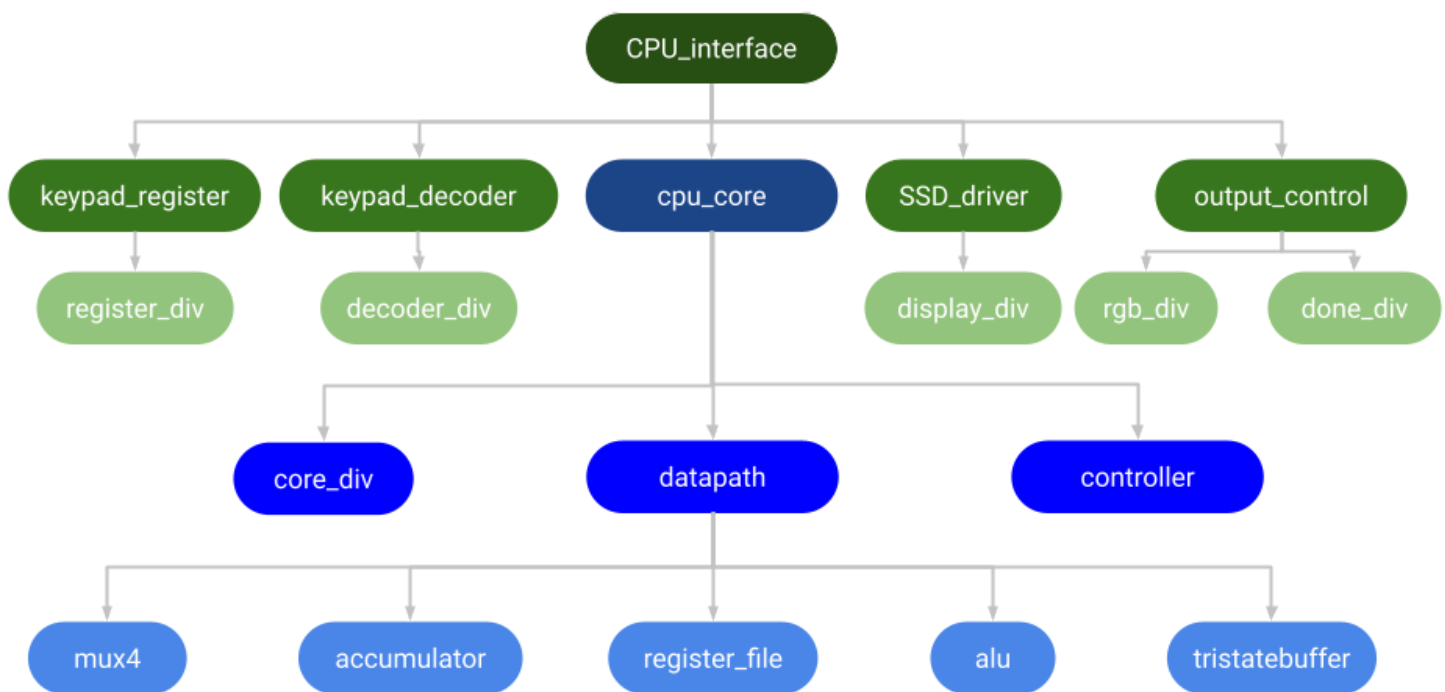



Figure 4: CPU component hierarchy tree

Note: in Figure 4, the components are color-coded to show related components and to delineate their objectives.

- **Green Blocks** are associated with the CPU top-level interface. Do not modify or alter anything related to the components with green blocks. They are foundational and essential to the project's top-level structure. unintended consequences or complications in your project may arise from modifying these blocks.
- **Blue Blocks:** These represent the components associated with the CPU core (controller and datapath). Focus your work and modifications on these files only.

 Fall 2023	ECE410-Advanced Digital Logic Design
	LAB 3: A Simple CPU Implemented with a Controller and Datapath

PART 1 DATAPATH

Marks 25%

The datapath component files from the resources are incomplete. Similarly, not all of the components have testbenches. Your first task is to complete and test each component individually to make sure that the lowest level entities behave correctly. Then you are to finish the datapath design by putting all the components together in a structural fashion in the datapath.vhd file following the names used in the prelab work.

PART 2 CONTROLLER

Marks 25%

The instructions INA, LDI and STA have already been implemented in the controller.vhd file for you. Design and integrate the remaining instructions for the FSM into the controller (you will need to get approval from a TA or LI before implementing them). Additionally, you are required to design and implement two instructions of your own choice. Should you need inspiration or guidance in conceptualizing useful instructions, please refer to the suggestions listed in the Appendix of this manual. Read the comments in controller.vhd carefully. There is an explanation of how the example instructions proceed through the fetch, decode, and execute cycle. Ensure that both the remaining predefined instructions and your custom instruction follow the same three cycles

PART 3 CPU CORE

Marks 10%

With the datapath and controller finished separately, bring them together in the cpu_core.vhd file and write a testbench in terms of the clock signal to simulate (you may use an 8-ns period as given in the template of the testbench) to simulate the working CPU. Feel free to use any other clock timing.

- Simulation waveforms must clearly show the state transitions for every instruction implemented and the corresponding output signals.
- You must write the entire constraint file by yourself. Get the template for the .xdc file from the lab resources and map the entity (cpu_core) input and output ports from top_level_interface.vhd.


PART 4 TEST PROGRAMS

Marks 15%

In the appendix section of this manual, you'll find a sample test program intended to validate the functionality of your CPU design. Begin by implementing and thoroughly understanding this program. Following that, you are tasked with creating a custom program. This program must:

1. Your program must include at the minimum the use of the following instructions: JMPZ, INA, STA, and at least one arithmetic instruction (ADD or SUB).
2. Incorporate the instructions that you've added to the CPU.
3. Consist of at least 20 instructions.
4. Bonus marks will be awarded for the creativity level and complexity of the program.

Your aim is to demonstrate the practical utility of your custom instruction within the context of a larger program.

	<h1>ECE410-Advanced Digital Logic Design</h1>
<p>Fall 2023</p>	<h2>LAB 3: A Simple CPU Implemented with a Controller and Datapath</h2>

LAB REPORT

Marks 20%

A formal lab report will be required as the previous labs, but with the following additions:

- A state transition diagram of the CPU after all the instructions have been implemented.
- The final datapath of this 8-bit CPU that shows all the control and status signals shown.
- Simulation results of all the cpu_core components.
- Explanation of the test programs used to validate the CPU.
- Explanation of your designed instruction with examples and code.
- Answer the questions on the worksheet.
- Results discussion and conclusions.

REFERENCES

- [Zybo Z7 Documentation, Tutorials and Example Projects](#),
- [Zybo Z7 Reference Manual](#)
- [ECE410 VHDL Reference from the Lectures](#)
- [Pmod SSD: Seven-segment Display Reference Manual](#)
- [Pmod KYPD Reference Manual](#)
- [VHDL essentials Github repository](#)
- [VHDL Essentials Youtube Playlist](#)



ECE410-Advanced Digital Logic Design


LAB 3: A Simple CPU Implemented with a Controller and Datapath

Fall 2023

APPENDIX

CPU test program:

```
0:  IN A
1:  STA R[0], A -- you may use any of the R[0]: R[7]
2:  LDA A, R[0] -- will be the same used in PC(1)
3:  DEC A
4:  STA R[0], A -- will be the same used in PC(1)
5:  OUT A
6:  JMPZ x0C
7:  x0C
8:  LDI A, x00
8:  x00
10: JMPZ x02
11: x02
12: LDI A, x0F -- you may use any 8-bit value
13: x0F
14: STA R[0], A -- will be the same used in PC(1)
15: LDI A, xAA -- you may use any 8-bit value
16: xAA
17: AND A, R[0] -- will be the same used in PC(1)
18: OUT A
19: INC A
20: STA R[0], A -- will be the same used in PC(1)
21: LDI A, x0F
22: x0F
23: ADD A, R[0] --R[x] will be the same used in PC(1)
24: OUT A
25: HALT
```

	<h1 style="text-align: center; color: blue;">ECE410-Advanced Digital Logic Design</h1>
<p>Fall 2023</p>	<h2 style="text-align: center; color: green;">LAB 3: A Simple CPU Implemented with a Controller and Datapath</h2>

Suggested instructions to implement on the simple CPU:

- **Bitwise (NOT):** Perform a bitwise logical NOT operation on the contents of the accumulator.
- **Relative jump (JMPR):** Allows the program counter (PC) to jump forward or backward from the current location by a specified offset rather than to a fixed address.
- **Test and set (TAS):** If the value in register R[rrr] is 0, set the accumulator to 1 and also set R[rrr] to 1. Otherwise, set the accumulator to 0.
- **Conditional skip (CSKIP):** Skip the next instruction if the value in the accumulator is equal to the value in register R[rrr].
- **Exchange (XCHG):** Swap the values between the accumulator and register R[rrr].
- **Shift left (SHL):** Shift the bits of the accumulator to the left.

CPU_core block diagram:

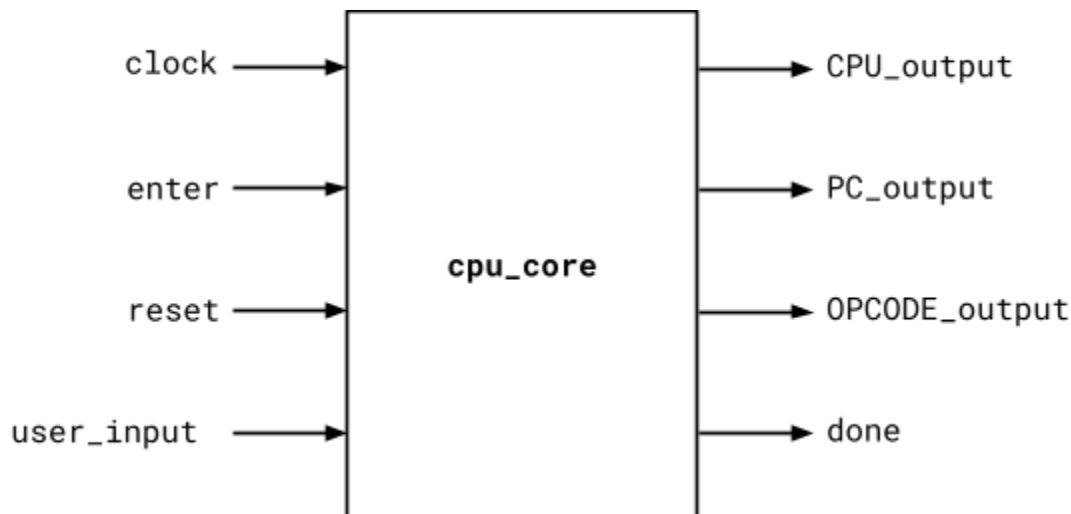


Figure 5: Block Diagram of CPU core



CPU_interface block diagram:

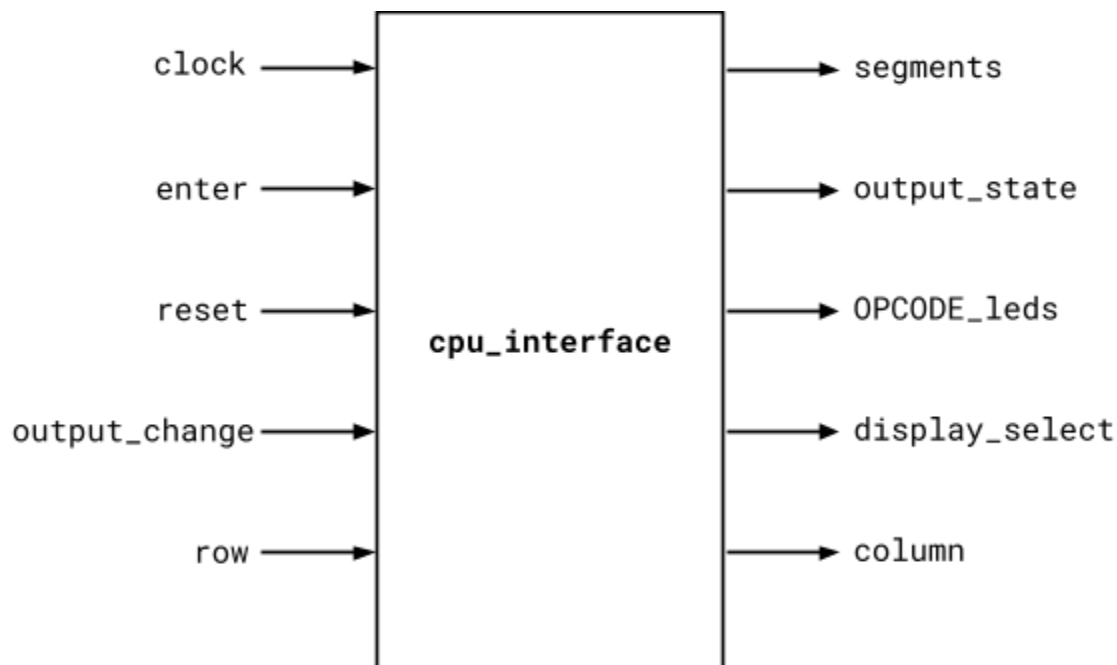


Figure 6: Block Diagram of top level entity



LAB 3: A Simple CPU Implemented with a Controller and Datapath

Fall 2023

CPU interface detailed diagram:

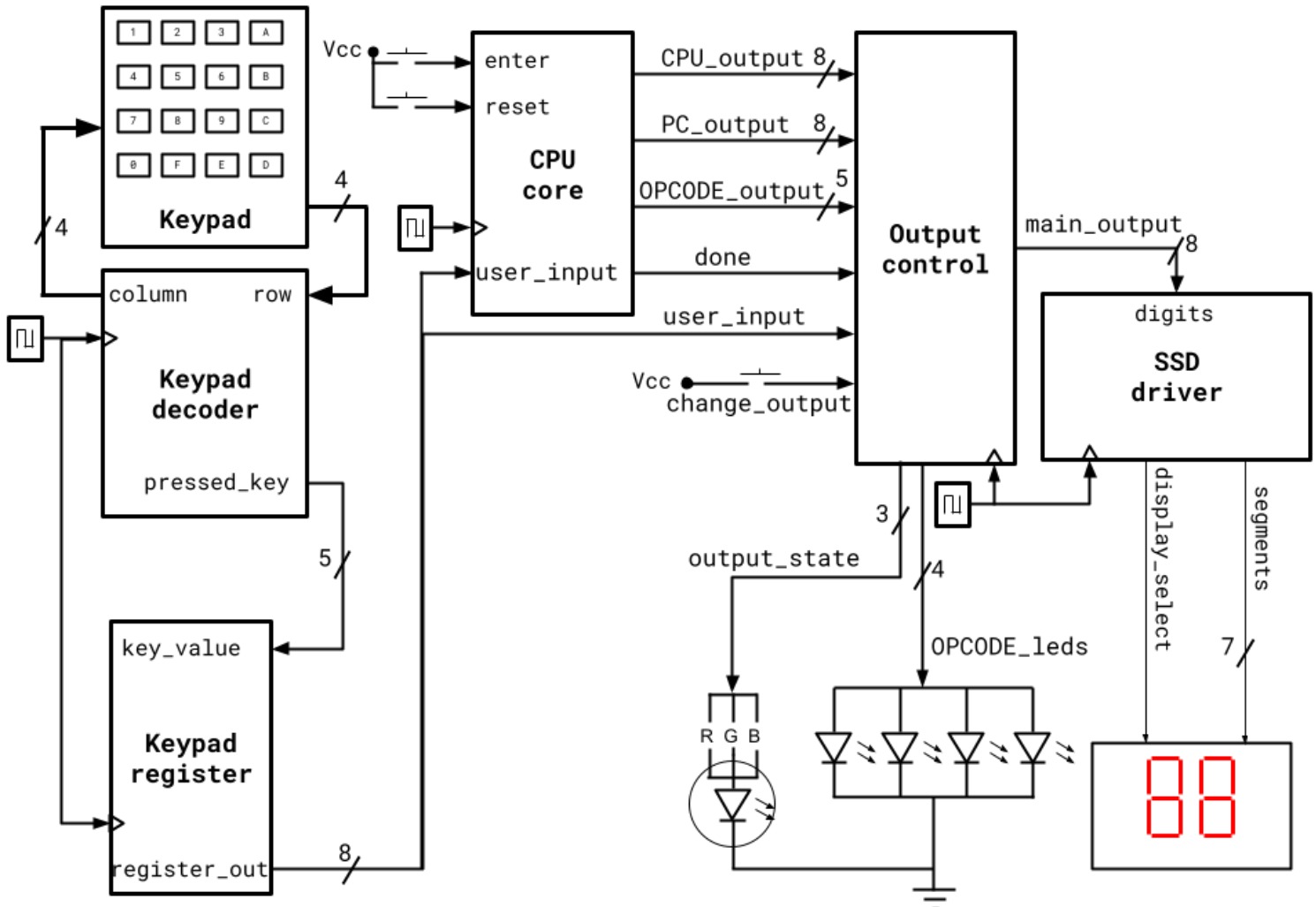



Figure 7: CPU interface schematic diagram

 Fall 2023	<h1 style="text-align: center; color: blue;">ECE410-Advanced Digital Logic Design</h1>
	<h2 style="text-align: center; color: green;">LAB 3: A Simple CPU Implemented with a Controller and Datapath</h2>

CPU Interface Operation:

As illustrated in Figure 7, the top-level entity encompasses several peripheral components that facilitate user interaction with the simple CPU.

Input Components:

- **Keypad:** Connected to a dedicated decoder and shift register (referred to as the "keypad decoder" and "keypad register", respectively). These components retain the information of the last two keys pressed. The combined output from the keypad register results in an 8-bit signal directed straight to the CPU's user input.
- **Control Buttons:** Two distinct buttons – 'enter' and 'reset' – are integrated with the CPU. The 'enter' button allows users to manually advance through instructions, while the 'reset' button returns the CPU to its initial state.

Output Components:

- **Output Control & SSD Driver:** These blocks manage the CPU's output. The 'output control' determines which data to display on the seven-segment display (SSD), based on the 'output_state' signal. The 'SSD driver' then drives this data to the SSD.
- **RGB LED Indication:** The RGB LED indicates the nature of data being displayed on the SSD:
 - **Red:** Displays user input coming in from the keypad register.
 - **Green:** Showcases the CPU output.
 - **Blue:** Reveals the program counter value.
- **OPCODE Indication:** The green LEDs consistently represent the OPCODE of the instruction currently in execution. When the CPU reaches a halt state, these LEDs will flash, notifying the user that the program has concluded its operation.