Computer Architecture Lab 08

Memory System and Address Decoding – Instruction/Data Access

Introduction

This lab is intended as an exercise in integrating memory systems (instruction and data memory) into your processor, and implementing a system of address decoding to access data and I/O memory outside the processor.

Objectives

* To understand and implement memory systems in a processor design.
* To integrate instruction memory (ROM) for storing and fetching RISC-V instructions.
* To implement data memory (RAM) for performing load and store operations.
* To design and implement address decoding logic that differentiates between data memory, and I/O regions (such as LEDs and buttons).
* To verify correct memory access by reading and writing data, and displaying the results using LEDs.

Sections

|  |  |
| --- | --- |
| Section |  |
| a) Task 1 | 40 |
| You will learn how to take button input to change the state of internal variables. |
|  |
| c) Task 2: | 40 |
| In this section, you will develop modules to write outputs to output peripherals such as Seven Segment LED on the FPGA. |
|  |
| d) Task 3: | 100 |
| In this section, you will develop a counter with user-configurable delay values. |
|  |

Task 1: Data Memory

|  |
| --- |
| Data memory is the off-chip memory from which data is read and written through control signals. Although not a part of the functioning units of a processor, it is an important component that must communicate with the processor for storage of important information.  It is controlled by a combinational read signal for reading data and a sequential write signal for writing data to a specific memory location, defined by the address bus signal.    The size of the address signal depends on the size of the memory. |

Your task is to define the data memory unit with the control signals given, along with a 64-bit address signal. You must be able to combinationally (no dependence on clock) read data memory into a 32-bit readData upon a memRead signal, and sequentially (governed by clock) write data from the 32-bit writeData upon a memWrite signal.

Provide your module here:

|  |
| --- |
|  |

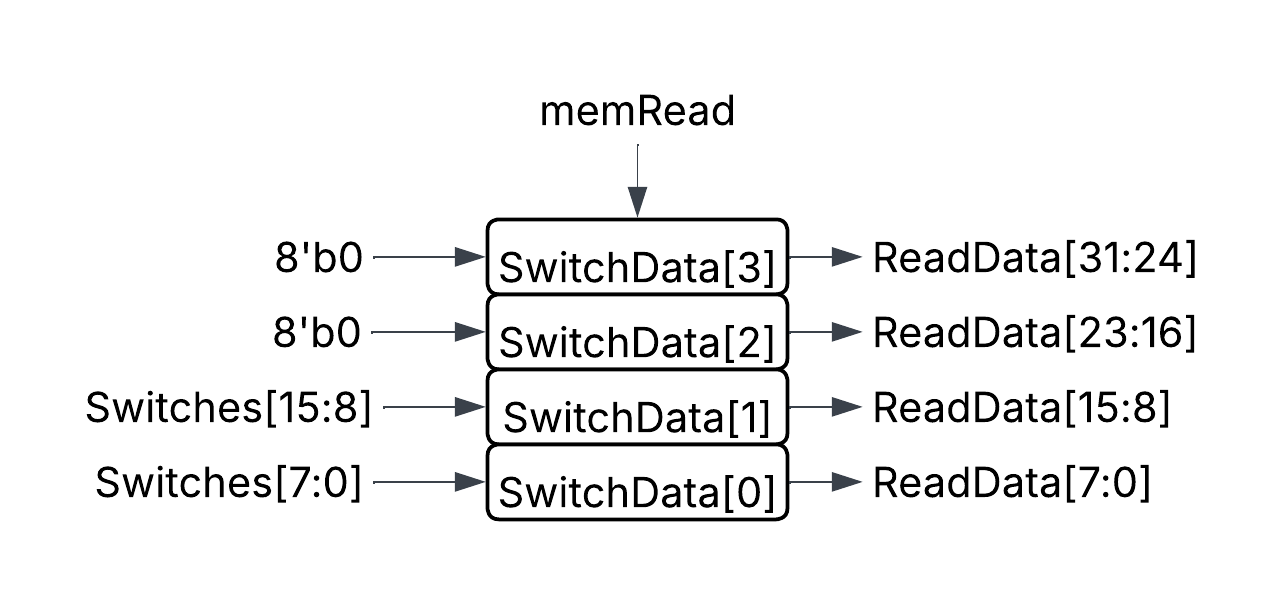
Task 2: Memory Mapped I/O

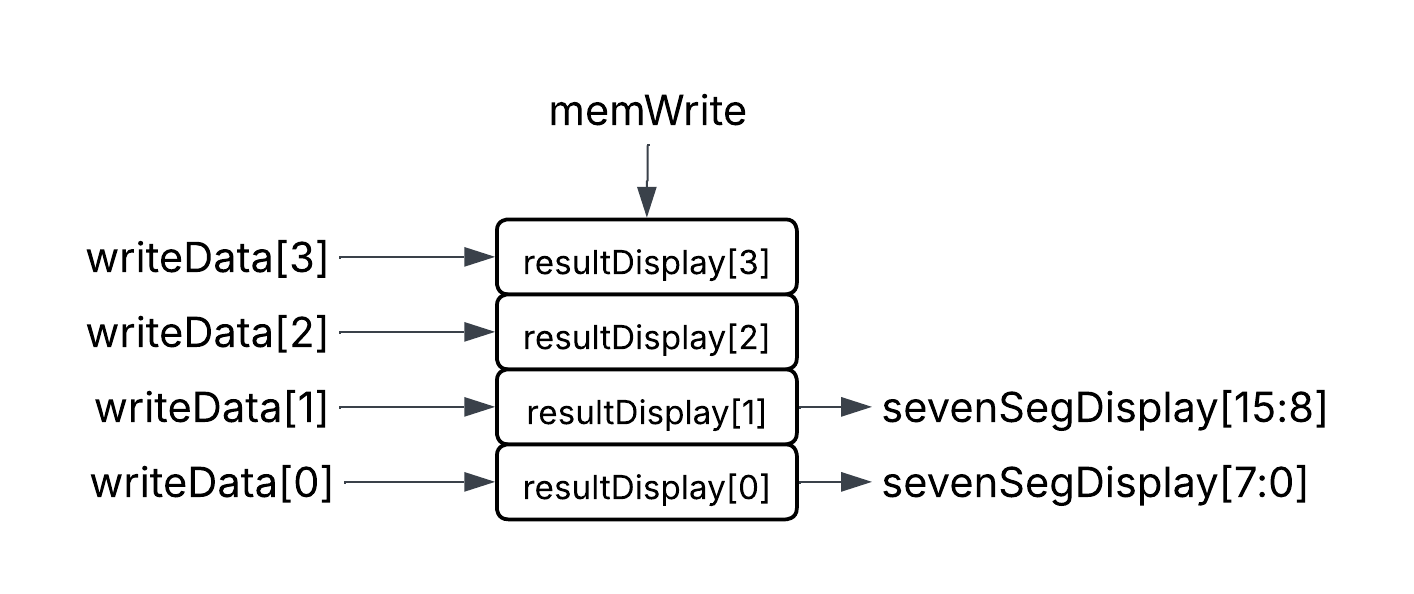
|  |
| --- |
| In many processors (including RISC-V), I/O devices are accessed through specific address ranges instead of special instructions.  This means that some memory addresses don’t belong to RAM. Instead, when those addresses are accessed, they correspond to hardware devices like LEDs, buttons, or seven-segment displays.  For example:  0x0000\_0000 – 0x0000\_FFFF → Data Memory (RAM)  0x1000\_0000 – 0x1000\_0FFF → Switches (Input)  0x1000\_1000 – 0x1000\_1FFF → Seven Segment Display (Output)  When your processor executes a load instruction from 0x1000\_0000, it should fetch the switch input. When it executes a store instruction to 0x1000\_1000, it should update the seven-segment display. The same memRead, memWrite, and memAddress lines that go to memory are also shared with these I/O modules. It’s the job of address decoding to decide which module actually responds. This concept is discussed further in the next task. |

Your task now is to create a switch module, and adapt your seven segment module to have them behave as memory devices and respond to read and write requests.

* Switches:  
  Should behave like a *read-only memory location*.  
  When the processor performs a read at the “switch address,” the module should return the current state of the switches.
* Seven Segment Display:  
  Should behave like a *write-only memory location*.  
  When the processor writes to its address, it should display the corresponding value on the seven-segment display.

Here’s how this may work:





Here’s what you are to do:

Modify your modules to respond only when:

* readEnable (or memRead) is asserted (for switches).
* writeEnable (or memWrite) is asserted (for seven-segment display).

Make reads combinational for input devices (like switches), and writes sequential (clock-dependent) for output devices (like seven segments). This mimics the behaviour of memory: reads happen instantly, but writes are synchronized.

(Hint: Look at your data memory module from Task 1: you already have memRead, memWrite, writeData, and readData signals. How can switches or seven segments fit into that same pattern? In both cases, think about how you can represent 32-bit reads and writes even if your hardware device is only 8 or 16 bits wide)

Provide both your modules here:

|  |
| --- |
| module switches(  input clk, rst,  input [15:0] btns,  input [31:0] writeData, // not to be written  input writeEnable, // not to be used  input readEnable,  input [63:0] memAddress,  input [15:0] switches,    output reg [31:0] readData  ); |
| module segTop(  input clk,  input rst,  input [31:0] writeData,  input writeEnable,  input readEnable, // not to be used  input [63:0] memAddress,    output reg [31:0] readData = 0, // not to be read  output [6:0] seg, // 7-segment segments (a-g)  output [3:0] an  ); |

Task 3: Address Decoding

|  |
| --- |
| What Is Address Decoding?  A processor sends every memory or I/O operation (read or write) with:   * An address (which location it’s trying to access) * A data value (to be written or read) * Control signals (memRead, memWrite)   But not every address belongs to the same device. When the CPU issues a request, address decoding ensures that only one device responds, depending on the address range. This is done by checking certain bits of the address (often the high-order bits) to *select* which device should be active.  How it works:   1. The Address Bus carries the address from the CPU to all connected memory and I/O devices. 2. Each device is given a unique address range. 3. Decoding Logic (combinational) examines the address and produces *select signals*:    * selDataMem → activates when the address falls inside the RAM range    * selSwitch → activates when inside the switch range    * selLed or selSeg → activates when inside the display range 4. These select signals then control:    * Which device receives the writeEnable    * Which device’s readData is forwarded back to the CPU   These selection signals then become control signals to multiplexers and demultiplexers to determine which device must be written to or read at a given time. |