LAB # 5
Designing Microarchitecture-I

Objective:

- To design the following modules in Verilog for RISC-V microarchitecture:
    - ✔ Control Unit
    - ✔ Register File
    - ✔ Data Memory
    - ✔ Instruction Memory
    - ✔ Program Counter

System Module (Hardware/Software)

- Visual Studio Code

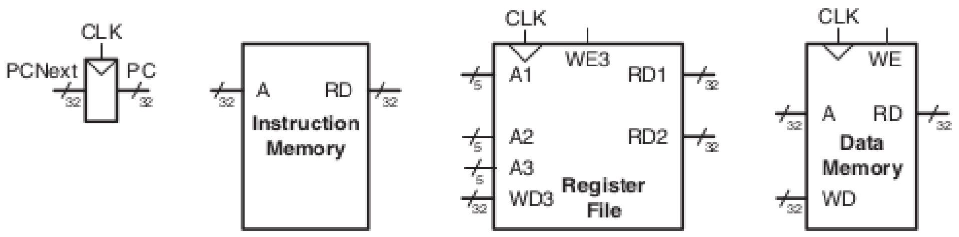
Theoretical Background:

Introduction

In this lab we will make Verilog modules for RISC-V four state elements that are data memory, instruction memory, program counter and register file.

State Elements

A good way to design a complex system is to start with hardware containing the state elements. These elements include the memories and the architectural state (the program counter and registers). Then, add blocks of combinational logic between the state elements to compute the new state based on the current state. The instruction is read from part of memory; load and store instructions then read or write data from another part of memory. Hence, it is often convenient to partition the overall memory into two smaller memories, one containing instructions and the other containing data. Figure below shows a block diagram with the four state elements: the program counter, register file, and instruction and data memories.



In above Figure heavy lines are used to indicate 32-bit data busses. Medium lines are used to indicate narrower busses, such as the 5-bit address busses on the register file. Narrow blue lines are used to indicate control signals, such as the register file write enable. We will use this convention throughout the chapter to avoid cluttering diagrams with bus widths. Also, state elements usually have a reset input to put them into a known state at start-up. Again, to save clutter, this reset is not shown.

The **program counter** is an ordinary 32-bit register. Its output, PC, points to the current instruction. Its input, PCNext, indicates the address of the next instruction.

The **instruction memory** has a single read port. It takes a 32-bit instruction address input, A, and reads the 32-bit data (i.e., instruction) from that address onto the read data output, RD.

The 32-element × 32-bit **register file** has two read ports and one write port. The read ports take 5-bit address inputs, A1 and A2, each specifying one of $2^5 = 32$ registers as source operands. They read the 32-bit register values onto read data outputs RD1 and RD2, respectively. The write port takes a 5-bit address input, A3; a 32-bit write data input, WD; a write enable input, WE3; and a clock. If the write enable is 1, the register file writes the data into the specified register on the rising edge of the clock.

The figure below shows the 32 registers available in RISC-V Microarchitecture.

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0–2	x5–7	Temporary registers
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0–1	x10–11	Function arguments / Return values
a2–7	x12–17	Function arguments
s2–11	x18–27	Saved registers
t3-6	x28–31	Temporary registers

The **data memory** has a single read/write port. If the write enable, WE, is 1, it writes data WD into address A on the rising edge of the clock. If the write enable is 0, it reads address A onto RD.

The instruction memory, register file, and data memory are all read combinationally. In other words, if the address changes, the new data appears at RD after some propagation delay; no clock is involved. They are written only on the rising edge of the clock. In this fashion, the state of the system is changed only at the clock edge. The address, data, and write enable must be set up sometime before the clock edge and must remain stable until a hold time after the clock edge. Because the state elements change their state only on the rising edge of the clock, they are synchronous sequential circuits.

The microprocessor is built of clocked state elements and combinational logic, so it too is a synchronous sequential circuit. Indeed, the processor can be viewed as a giant finite state machine, or as a collection of simpler interacting state machines.

Memory declaration in Verilog
Example:
reg [15:0] mem1 [511:0];

We declared an array called mem1, it has 512 locations. Each location is 16 bits wide.

Now consider
B = mem1 [3];

Here the data stored at mem [3] is assigned to B.

$readmemh(“memfile.hex”,mem1);
Reads a .hex file into array mem1.

Procedure:

Open Visual Studio Code and perform the procedure mentioned in lab #01 in order to make a Verilog module and test bench.

Lab Tasks:

1. **Write Verilog code for each of the following state elements. Attach the code.**
    - **Register File**
    - **Data Memory**
    - **Instruction Memory**
    - **Program Counter**

Conclusion:

What have you learnt from this lab?

____
____
____
____

Learning Outcomes:

Upon successful completion of the lab, students will be able to:

LO1: Declare a memory in Verilog.
LO2: Design data and instruction memories, register file and program counter for RISC-V.