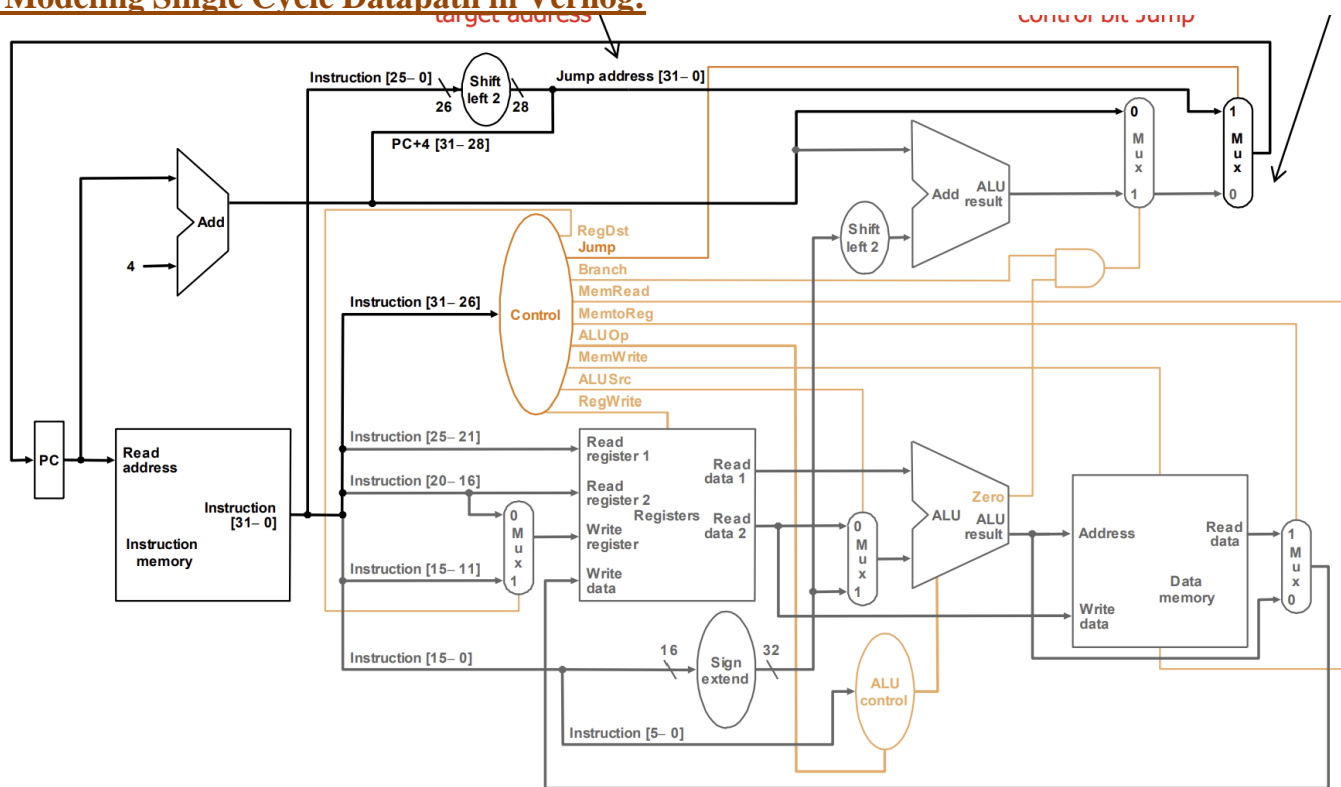


**Learning Objectives:**

- i) Modeling MIPS Single Cycle Datapath in Verilog
- ii) Designing modules for each hardware component and some support modules
- iii) Implementation of the modules in Verilog
- iv) Integrating these modules
- v) Writing test bench
- vi) Testing the circuit with R-format instruction.

Today we have to implement MIPS single cycle datapath design using Verilog. To accomplish this, first identify the hardware components from the datapath design and write separate modules for each component. Integrate these modules to realize the complete datapath design circuit. Test the correctness of the complete implementation by supplying MIPS r-format instruction as an input. Make use of modules such as ALU, Control Unit, Register File etc which you already made in previous labs.

**I. Modeling Single Cycle Datapath in Verilog:**



**Fig.1 Single Cycle Datapath Design Circuit**

In order to model datapath in verilog we have to first identify the hardware components in the datapath and implement them in separate verilog module. Following are the hardware components for which you have to implement verilog modules:

1. Instruction Memory
2. Register File
3. ALU
4. Control Unit and ALU Control
5. Data Memory
6. Sign Extender
7. Shifter
8. Adders (required for computing next PC)
9. Multiplexers

In addition to these modules write few modules to integrate all these hardware components. For example, modules for reading from instruction memory, reading from and writing to register file or data memory etc. Also write testbench module to test the complete implementation for r-format instruction.

## **II. How to integrate sub-modules for running r-format instruction?**

Write a module SCDataPath() to integrate sub-modules to realize the complete single cycle datapath circuit for running r-format instruction. To accomplish this, follow the steps below:

1. Model Program Counter (PC) by instantiating module for 5-bit register. Populate PC with some address. Let us restrict size of instruction memory to 32 locations each of width 32-bit.
2. Write a module to initialize Instruction Memory with some r-format instructions and instantiate it in this module. Also write module to initialize data memory to zeros which is required for lw/sw instructions.
3. Instantiate a module for reading instruction word from instruction memory by passing PC as an argument.
4. Instantiate module for Control Unit by passing the instruction word obtained in above step. This will generate required control signals.
5. Instantiate register file module by passing required input arguments which will give two 32-bit values.
6. Then instantiate module for ALU Control, ALU etc appropriately.
7. In r-format instruction, data memory is not required. But it is required in lw/sw instructions.
8. For storing back the output of ALU to register in register file, instantiate module for writing result to register file.
9. You will also have to instantiate all other modules like Sign Extender, Shifter, module for computation of PC and multiplexers. Depending upon the instruction format, multiplexers along with appropriate control signals should choose correct values. For example, ALU's second source operand has two different values which are given to (2X1) mux. Control signal ALUSrc will decide whether ReadData2 is the second source operand of ALU or Sign extended offset depending upon type of instruction.

**Note:** For running other instruction formats like lw/sw, beq etc., sequence of instantiation of sub-modules will change. You can easily figure out the sequence of instantiation for

other instructions if you successfully do atleast one type of instruction. **End of Note**

### III. TestBench module:

Use the following testbench to verify the processor working. Remember that the program from the instruction memory will be summing up the first 10 numbers and storing them in \$t0.

```
module tb_SCDatapath;
    reg clk, rst;

    // Instantiate the datapath
    SCDatapath processor(clk, rst);

    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // 10 time units per clock cycle
    end

    // Reset sequence
    initial begin
        rst = 0;
        #12 rst = 1; // Release reset slightly after first clock edge
    end

    initial begin
        $display("Time
| PC      Instr      | $t0      $t1      $t2      $t3      $t4");
        $display("-----");
        $display("-----");
    end

    // Monitor key signals every cycle
    always @(posedge clk) begin
        $display("t=%0t | PC=0x%08h instr=0x%08h | t0=%h t1=%h t2=%h t3=%h
t4=%h",
            $time,
            processor.addr,
            processor.instr,
            processor.RegFile.regvalues[8],
            processor.RegFile.regvalues[9],
            processor.RegFile.regvalues[10],
            processor.RegFile.regvalues[11],
            processor.RegFile.regvalues[12]);

        // === Uncomment these for deeper debug ===
        // $display("  ALU: ALUres=%h Read1=%h Read2=%h Zero=%b",
        //     processor.ALU_result, processor.ReadData1,
        //     processor.ReadData2, processor.zero);
        // $display("  WB : WriteReg=%0d WriteData=%h", processor.WriteReg,
        //     processor.WriteData);
        // $display("  Control: RegDst=%b ALUSrc=%b MemtoReg=%b RegWrite=%b
        //     MemRead=%b MemWrite=%b Branch=%b ALUOp0=%b ALUOp1=%b Jump=%b",
        //     processor.RegDst, processor.ALUSrc, processor.MemtoReg,
        //     processor.RegWrite, processor.MemRead, processor.MemWrite,
```

```
        //      processor.Branch, processor.ALUOp[0], processor.ALUOp[1],  
processor.Jump);
```

```
        $display("-----  
-----");
```

```
    end
```

```
    // Run just enough cycles to complete the program
```

```
    initial begin
```

```
        #800 $finish;
```

```
    end
```

```
endmodule
```

---