CSE 331 — Computer Organization

Project 3: Single-Cycle MIPS Processor (Verilog + Quartus)

Due Date: 24/12/2025

1) Overview

Design and implement a single-cycle MIPS32 processor supporting all integer instructions covered in class. The design will be structural/dataflow oriented, favoring continuous assignments (`assign`) and module composition. Behavioral modeling is required only where sequential state is essential (e.g., registers and memories). You will also write your own self-checking testbench.

Goal: A cleanly partitioned, synthesizable design with a minimal, readable datapath and control, verified by comprehensive tests.

---

2) Functional Requirements

- ISA scope: All integer instructions discussed in class, including:
  - Arithmetic: add, addu, sub, subu, addi, addiu
  - Logical: and, or, xor, nor, andi, ori, xori
  - Shifts: sll, srl, sra, sllv, srlv, srav
  - Compare: slt, sltu, slti, sltiu
  - Move/Upper: lui
  - Branch: beq, bne, bltz, bgez, blez, bgtz
  - Jump: j, jal, jr, jalr
  - Memory: lw, sw
- Word size: 32-bit
- Byte addressing with word-aligned instruction fetch
- Single-cycle CPU: each instruction completes in one clock cycle (no pipelining)
- Reset behavior: synchronous, active-high or active-low (document your choice)
- Endianness: [Specify; default to what was used in class]

Note: If your course's instruction list differs, use the set provided by your instructor/TAs. Include any clarifications in your README.

---

3) Design Constraints & Guidance

- Use `assign` for combinational logic and structural wiring wherever possible.
- Keep logic combinational unless state is required.
- Behavioral modeling is required for:
  - Program Counter (PC) register
  - Register File (2 read ports, 1 write port)
  - Instruction Memory and Data Memory
- Recommended:

    - Parameterize widths (default 32)
    - Separate Main Control from ALU Control
    - Keep instruction decode fields as wires, connected via `assign`
    - Use modules for adders, shifters, extenders to simplify wiring
- FPGA note: For synthesis on FPGA BRAMs, reads may be synchronous. For simulation, you may use behavioral arrays with `$readmemh`. Document any single-cycle compromises if using synchronous BRAM models.

---

4) High-Level Architecture

- Datapath
    - PC register and PC+4 adder
    - Instruction Memory (read-only during execution)
    - Register File (rs, rt reads; rd/rt write)
    - ALU (arithmetic, logical, set-less-than)
    - Shifter (logical/arithmetic, variable/immediate)
    - Sign/Zero Extender (immediates)
    - Branch target adder, branch decision logic
    - Jump/jal/jr/jalr target selection
    - Data Memory (load/store)
    - Write-back MUXing (ALU result vs. load data; jal link)
- Control
    - Main Control (op → control signals)
    - ALU Control (funct/op → ALU operation)
    - Misc: RegDst, ALUSrc, MemRead, MemWrite, MemToReg, RegWrite, Branch types, Jump types, Link bit

Keep the top-level minimal and wire-oriented, pushing behavior into memories and registers only.

---

5) Module Skeletons (ports and intent only)

Use these as starting points; do not over-specify internals here. Favor `assign` in combinational modules. Behavioral is required where noted.

```
// top-level CPU
module mips_cpu (
    input  wire        clk,
    input  wire        rst,          // active-[your choice]; document it
    // Optional debug visibility
    output wire [31:0] dbg_pc,
    output wire [31:0] dbg_instr
    // Add optional interfaces (I/O, halt) if needed
);
    // TODO: Structural wiring between submodules using assign and wires
endmodule
```

```
// main control unit: decode op (and possibly rt for special branches)
module control_unit (
    input  wire [5:0]  op,
    input  wire [5:0]  funct,
    input  wire [4:0]  rt,       // for bltz/bgez variants if needed
    output wire        reg_dst,
    output wire        alu_src,
    output wire        mem_to_reg,
    output wire        reg_write,
    output wire        mem_read,
    output wire        mem_write,
    output wire        branch,       // generic branch enable
    output wire [2:0]  branch_type,  // encodes beq, bne, bltz, bgez, blez, bgtz
    output wire        jump,
    output wire        link,         // jal/jalr
    output wire        jr,           // jr/jalr
    output wire [3:0]  alu_op        // to ALU control
);
    // TODO: Combinational decode using assign
endmodule
```

```
// ALU control: maps alu_op/funct to specific ALU operation encoding
module alu_control (
    input  wire [3:0]  alu_op,
    input  wire [5:0]  funct,
    output wire [4:0]  alu_sel   // select code for ALU
);
    // TODO: Combinational decode using assign
endmodule
```

```
// ALU: arithmetic/logical/compare/shift select (can split shifter out)
module alu (
    input  wire [31:0] a,
    input  wire [31:0] b,
    input  wire [4:0]  shamt,
    input  wire [4:0]  sel,
    output wire [31:0] y,
    output wire        zero,
    output wire        lt_signed,
    output wire        lt_unsigned
);
    // TODO: Combinational datapath using assign
endmodule
```

```
// Shifter (optional separate module)
module shifter (
```

```
    input  wire [31:0] a,
    input  wire [4:0]  shamt,
    input  wire [1:0]  mode, // 00 SLL, 01 SRL, 10 SRA
    output wire [31:0] y
);
    // TODO: Combinational using assign
endmodule
```

```
// Sign/zero extender
module imm_extender (
    input  wire [15:0] imm,
    input  wire        sign,     // 1=sign-extend, 0=zero-extend
    output wire [31:0] imm_ext
);
    // TODO: Combinational using assign
endmodule
```

```
// PC register (behavioral/sequential)
module pc_reg (
    input  wire        clk,
    input  wire        rst,
    input  wire [31:0] pc_next,
    output reg  [31:0] pc
);
    // TODO: Sequential always @(posedge clk)
endmodule
```

```
// Register file (behavioral): 2R/1W, $zero hard-wired to 0
module reg_file (
    input  wire        clk,
    input  wire        rst,
    input  wire        we,
    input  wire [4:0]  ra1,
    input  wire [4:0]  ra2,
    input  wire [4:0]  wa,
    input  wire [31:0] wd,
    output wire [31:0] rd1,
    output wire [31:0] rd2
);
    // TODO: behavioral array, async reads, sync write; force reg[0]=0
endmodule
```

```
// Instruction memory (behavioral for simulation; ROM-like)
module imem (
    input  wire [31:0] addr,      // word-aligned
```

```
    output wire [31:0] instr
);
    // TODO: behavioral read; $readmemh for initialization
endmodule
```

```
// Data memory (behavioral)
module dmem (
    input  wire        clk,
    input  wire        mem_read,
    input  wire        mem_write,
    input  wire [31:0] addr,      // word-aligned for lw/sw
    input  wire [31:0] wd,
    output wire [31:0] rd
);
    // TODO: behavioral array; define read/write timing model
endmodule
```

```
// Simple adder module (combinational)
module add32 (
    input  wire [31:0] a,
    input  wire [31:0] b,
    output wire [31:0] y
);
    // TODO: assign y = a + b;
endmodule
```

---

6) Integration Notes

- Instruction fetch: PC_next selection via mux chain (PC+4, branch target if taken, jump target, jr/jalr)
- Branch decision: use ALU flags (zero, lt_signed, lt_unsigned) and branch_type
- Writeback: select among ALU result, memory data, link address (PC+4) for jal/jalr
- Destination register: rd vs. rt, or $31 for link
- Immediate handling: choose sign/zero extension per instruction
- Shifts: immediate shamt vs. variable shifter using rs

---

7) Testbench Requirements (you write the TB)

- Self-checking: no manual waveform grading required for correctness
- Provide:
  - Clock and reset generation
  - Program load into IMEM via `$readmemh` (or parameterized init file)
  - Optional Data Memory pre-load
  - Assertions or checks on:
    - Architectural state (selected registers, memory locations)

        - Control flow (branch/jump correctness)
        - Edge cases (overflows if applicable to your chosen semantics, slt/sltu, shifts)
    - End-of-test with PASS/FAIL summary

Suggested TB skeleton:

```
`timescale 1ns/1ps
module mips_cpu_tb;
    reg  clk = 0;
    reg  rst = 1;
    wire [31:0] dbg_pc;
    wire [31:0] dbg_instr;

    // Clock gen
    always #5 clk = ~clk;

    // DUT
    mips_cpu dut (
        .clk(clk),
        .rst(rst),
        .dbg_pc(dbg_pc),
        .dbg_instr(dbg_instr)
    );

    // Reset and run
    initial begin
        rst = 1;
        repeat (2) @(posedge clk);
        rst = 0;

        // Run for fixed cycles or until a sentinel condition
        repeat (2000) @(posedge clk);
        $display("TIMEOUT: Test did not complete");
        $finish;
    end

    // TODO: Load program, monitor state, self-check via $fatal/$error/$display
endmodule
```

Provide at least:

- A directed test program covering each instruction class
- A mixed program verifying realistic sequences
- A README describing how to run tests and expected outcomes

---

8) Build & Run (Quartus/Simulation)

- Simulation
    - Compile all Verilog files
    - Use `$readmemh("prog.hex", ...)` for IMEM init

    - Run testbench; ensure PASS/FAIL prints
- Synthesis
    - Create Quartus project
    - Set top module to `mips_cpu`
    - Add pin constraints only if you include I/O
    - Note: If using FPGA BRAMs, document any changes to memory timing

---

9) Checklist

- ☐ All integer instructions implemented and verified
- ☐ Behavioral only where state is required (PC, RegFile, Memories)
- ☐ Combinational modules use `assign`
- ☐ Self-checking testbench passes
- ☐ README updated