CSE 331 Computer Organization - Project 2

Single-Cycle MIPS Datapath Design

Due Date: December 03, 2025 Quiz

Project Overview

In this project, you will design and implement a simplified MIPS processor using a **single-cycle datapath architecture**. The processor will support R-type instructions and their I-type variants.

Key Difference: Unlike a traditional processor, your design will **NOT fetch instructions from memory**. Instead, instructions will be provided as **direct inputs** to your datapath. This simplifies the design and allows you to focus on the core datapath and control logic.

You will implement the entire design using **structural Verilog**, meaning you must instantiate modules and connect them hierarchically.

Important Constraints:

- ✅ Structural Verilog ONLY (module instantiation and wire connections)
- ❌ NO behavioral Verilog (`assign` statements are NOT allowed except for simple wire assignments)
- ❌ NO instruction memory or data memory
- ✅ Instructions come as external inputs
- ✅ ONLY memory element: Register File
- ✅ EXCEPTION: Register file MUST use `always@` blocks for sequential logic

---

Supported Instructions

Your processor must support the following instruction types:

R-Type Instructions

- `ADD` - Addition: `add $rd, $rs, $rt` → rd = rs + rt
- `SUB` - Subtraction: `sub $rd, $rs, $rt` → rd = rs - rt
- `AND` - Bitwise AND: `and $rd, $rs, $rt` → rd = rs & rt
- `OR` - Bitwise OR: `or $rd, $rs, $rt` → rd = rs | rt
- `SLT` - Set Less Than: `slt $rd, $rs, $rt` → rd = (rs < rt) ? 1 : 0
- `XOR` - Bitwise XOR: `xor $rd, $rs, $rt` → rd = rs ^ rt

I-Type Instructions

- `ADDI` - Add Immediate: `addi $rt, $rs, imm` → rt = rs + imm
- `ANDI` - AND Immediate: `andi $rt, $rs, imm` → rt = rs & imm
- `ORI` - OR Immediate: `ori $rt, $rs, imm` → rt = rs | imm
- `XORI` - XOR Immediate: `xori $rt, $rs, imm` → rt = rs ^ imm
- `SLTI` - Set Less Than Immediate: `slti $rt, $rs, imm` → rt = (rs < imm) ? 1 : 0

MIPS Instruction Format

R-Type Format (32 bits)

```
[31:26] [25:21] [20:16] [15:11] [10:6] [5:0]
opcode   rs      rt      rd     shamt  funct
  6bits  5bits   5bits   5bits  5bits  6bits
```

- **opcode**: Always `000000` for R-type
- **rs**: First source register
- **rt**: Second source register
- **rd**: Destination register
- **shamt**: Shift amount (not used in this project, set to 00000)
- **funct**: Function code (determines specific operation)

I-Type Format (32 bits)

```
[31:26] [25:21] [20:16] [15:0]
opcode   rs      rt     immediate
  6bits  5bits   5bits   16bits
```

- **opcode**: Determines I-type operation
- **rs**: Source register
- **rt**: Destination register (note: different from R-type!)
- **immediate**: 16-bit immediate value (sign-extended to 32 bits)

MIPS Instruction Encoding Reference

R-Type Instructions (opcode = 000000)

Instruction	Funct Code	Example
ADD	`100000` (0x20)	`add $1, $2, $3`
SUB	`100010` (0x22)	`sub $1, $2, $3`
AND	`100100` (0x24)	`and $1, $2, $3`
OR	`100101` (0x25)	`or $1, $2, $3`
XOR	`100110` (0x26)	`xor $1, $2, $3`
SLT	`101010` (0x2A)	`slt $1, $2, $3`

I-Type Instructions

Instruction	Opcode	Example
ADDI	`001000` (0x08)	`addi $1, $2, 100`
ANDI	`001100` (0x0C)	`andi $1, $2, 0xFF`
ORI	`001101` (0x0D)	`ori $1, $2, 0x10`
XORI	`001110` (0x0E)	`xori $1, $2, 0x0F`
SLTI	`001010` (0x0A)	`slti $1, $2, 50`

Top-Level Architecture

Your design must follow a **hierarchical modular structure**. The top-level module receives instructions as input rather than fetching them from memory.

```
Top Module (mips_single_cycle_datapath)
├── Register File (ONLY memory element)
├── Control Unit
├── ALU Control Unit
├── ALU (Arithmetic Logic Unit)
├── Sign Extender (for I-type immediates)
└── Multiplexers (for data path routing)
```

Note: No Program Counter (PC) or Instruction Memory needed since instructions are external inputs.

Module Specifications

You need to design the following modules. Each module description provides the **interface** and **functionality** but NOT the implementation—you must design the internal logic yourself.

---

1. Register File

File: `register_file.v`

Functionality:

- 32 general-purpose registers ($0-$31)
- $0 is hardwired to zero
- Two asynchronous read ports
- One synchronous write port
- Write occurs on rising clock edge when RegWrite is high
- **This is the ONLY memory element in your design**

Interface:

```
module register_file(
    input wire clk,
    input wire reset,
    input wire reg_write,           // Write enable
    input wire [4:0] read_reg1,     // rs
    input wire [4:0] read_reg2,     // rt
    input wire [4:0] write_reg,     // rd (R-type) or rt (I-type)
    input wire [31:0] write_data,
    output wire [31:0] read_data1,
    output wire [31:0] read_data2
);
```

Requirements:

- Register storage must use `always@(posedge clk)`
- Register $0 must ALWAYS output zero (hardwired)
- Reads are combinational (no clock dependency)
- Reset should clear all registers (except $0 which is always zero)
- Think about how to implement read logic structurally

---

2. Sign Extender

File: `sign_extender.v`

Functionality:

- Extends 16-bit immediate value to 32 bits
- Performs sign-extension (MSB is replicated)
- Used for I-type instructions

Interface:

```
module sign_extender(
    input wire [15:0] immediate_in,   // 16-bit immediate from instruction
    output wire [31:0] immediate_out  // 32-bit sign-extended value
);
```

Requirements:

- Must replicate bit [15] to fill bits [31:16]
- Bits [15:0] remain unchanged
- Implement using structural components (muxes, bit replication modules)
- NO `assign immediate_out = {{16{immediate_in[15]}}, immediate_in};`

---

3. Control Unit

File: `control_unit.v`

Functionality:

- Decodes the opcode field (bits [31:26])
- Generates control signals for datapath components
- Determines operation type (R-type vs I-type)

Interface:

```
module control_unit(
    input wire [5:0] opcode,
    output wire reg_dst,      // 0: rt (I-type), 1: rd (R-type)
    output wire alu_src,      // 0: register, 1: immediate
    output wire reg_write,    // Enable register file write
    output wire [1:0] alu_op  // ALU operation type indicator
);
```

Control Signal Truth Table:

Opcode	Instruction Type	RegDst	ALUSrc	RegWrite	ALUOp
000000	R-type	1	0	1	10
001000	ADDI	0	1	1	00
001100	ANDI	0	1	1	11
001101	ORI	0	1	1	11
001110	XORI	0	1	1	11
001010	SLTI	0	1	1	01

Requirements:

- Decode based on 6-bit opcode
- Must be implemented structurally (build decoder from gates)
- Think about how to create a decoder circuit

---

4. ALU Control Unit

File: `alu_control.v`

Functionality:

- Generates specific ALU control signals
- Uses funct field (bits [5:0]) and ALUOp from main control
- Determines exact operation (add, sub, and, or, xor, slt)

Interface:

```
module alu_control(
    input wire [1:0] alu_op,        // From Control Unit
    input wire [5:0] funct,         // From instruction [5:0]
    output wire [3:0] alu_control   // To ALU
);
```

ALU Control Logic:

ALUOp	Funct	Instruction	ALU Control	ALU Action
00	xxxxxx	ADDI	0010	ADD
01	xxxxxx	SLTI	0111	SLT
10	100000	ADD	0010	ADD
10	100010	SUB	0110	SUB
10	100100	AND	0000	AND
10	100101	OR	0001	OR
10	100110	XOR	0011	XOR
10	101010	SLT	0111	SLT
11	xxxxxx	ANDI/ORI/XORI	See funct	Bitwise ops

Requirements:

- Must differentiate between similar operations (e.g., ADD vs SUB)
- Implement as combinational logic using gates and decoders
- No `assign` statements for logic expressions

---

5. Arithmetic Logic Unit (ALU)

File: `alu.v`

Functionality:

- Performs arithmetic and logical operations
- Supports: ADD, SUB, AND, OR, XOR, SLT
- Generates zero flag

Interface:

```
module alu(
    input wire [31:0] a,            // First operand (from rs)
    input wire [31:0] b,            // Second operand (from rt or immediate)
    input wire [3:0] alu_control,   // Operation select
    output wire [31:0] result,      // ALU result
```

```
    output wire zero                // Zero flag (result == 0)
);
```

ALU Control Codes:

ALU Control	Operation
0000	AND
0001	OR
0010	ADD
0011	XOR
0110	SUB
0111	SLT (set less than)

Requirements:

- Implement each operation as a separate module (adder, subtractor, AND gate array, etc.)
- Use a multiplexer to select the correct result based on alu_control
- Build adder/subtractor structurally (consider ripple-carry or other designs)
- NO behavioral operators like `+`, `-`, `&`, etc.—build them from gates
- Zero flag = 1 if result is all zeros

---

6. Multiplexer (2-to-1, 5-bit)

File: `mux2to1_5bit.v`

Functionality:

- Selects between rt and rd for register write address
- Used for RegDst control

Interface:

```
module mux2to1_5bit(
    input wire [4:0] input0,    // rt (I-type)
    input wire [4:0] input1,    // rd (R-type)
    input wire select,          // RegDst
    output wire [4:0] out
);
```

Requirements:

- Must be built from basic gates or bit-level multiplexers
- Create 1-bit mux first, then instantiate 5 times

---

7. Multiplexer (2-to-1, 32-bit)

File: `mux2to1_32bit.v`

Functionality:

- Selects between two 32-bit inputs based on select signal
- Used for ALUSrc control (register vs immediate)

Interface:

```
module mux2to1_32bit(
    input wire [31:0] input0,   // Register data
    input wire [31:0] input1,   // Immediate data
    input wire select,          // ALUSrc
    output wire [31:0] out
);
```

Requirements:

- Must be built from basic gates or bit-level multiplexers
- Create 1-bit mux first, then instantiate 32 times
- NO `assign out = select ? input1 : input0;`

---

8. Adder (32-bit)

File: `adder_32bit.v`

Functionality:

- Adds two 32-bit numbers
- Used for ALU addition operation

Interface:

```
module adder_32bit(
    input wire [31:0] a,
    input wire [31:0] b,
    output wire [31:0] sum,
    output wire carry_out
);
```

Requirements:

- Design structurally (full adders chained together)
- Consider ripple-carry architecture
- Build full-adder from basic gates first

---

9. Subtractor (32-bit)

File: `subtractor_32bit.v`

Functionality:

- Subtracts two 32-bit numbers (a - b)
- Can be built using adder with two's complement

Interface:

```
module subtractor_32bit(
    input wire [31:0] a,
    input wire [31:0] b,
    output wire [31:0] difference,
    output wire borrow_out
);
```

Requirements:

- Implement using two's complement method
- Use your adder module
- Build structurally

---

10. Top-Level Module

File: `mips_single_cycle_datapath.v`

Functionality:

- Instantiates all components
- Connects all modules with wires
- Implements the complete MIPS datapath
- **Receives instruction as external input** (no instruction memory)

Interface:

```
module mips_single_cycle_datapath(
    input wire clk,
    input wire reset,
    input wire [31:0] instruction,  // Instruction provided as input
    output wire [31:0] alu_result,  // ALU output
    output wire [31:0] write_data   // Data written to register
);
```

Internal Signals to Extract from Instruction:

```
wire [5:0] opcode;      // instruction[31:26]
wire [4:0] rs;          // instruction[25:21]
wire [4:0] rt;          // instruction[20:16]
wire [4:0] rd;          // instruction[15:11]
wire [4:0] shamt;       // instruction[10:6] (not used)
wire [5:0] funct;       // instruction[5:0]
wire [15:0] immediate;  // instruction[15:0]
```

Requirements:

- All connections must be explicit wire declarations
- Extract instruction fields properly
- Connect control signals correctly
- Ensure proper signal routing between components
- Output the ALU result and register write data for verification

---

Design Hierarchy Guidelines

Low-Level Components (Build First)

1. Basic gates (AND, OR, NOT, XOR) - if not using Verilog primitives
2. 1-bit multiplexer
3. 1-bit full adder
4. Comparator circuits (for SLT)

Mid-Level Components

1. 5-bit multiplexer (for RegDst)
2. 32-bit multiplexer (for ALUSrc)
3. 32-bit adder (using 1-bit full adders)
4. 32-bit subtractor
5. 32-bit bitwise operations (AND, OR, XOR arrays)
6. 32-bit comparator (for SLT)
7. Sign extender

High-Level Components

1. ALU (integrating arithmetic and logic modules)
2. ALU Control Unit
3. Main Control Unit
4. Register file

Top-Level Integration

1. MIPS single-cycle datapath (connecting everything)

Testing Requirements

Create a testbench (`testbench.v`) that:

1. Instantiates your MIPS single-cycle datapath
2. Generates clock signal
3. Applies reset
4. **Provides instructions as inputs** (one per clock cycle)
5. Monitors outputs (instruction, ALU result, register write data)
6. Verifies correct execution of at least:
    - 3 R-type instructions
    - 3 I-type instructions

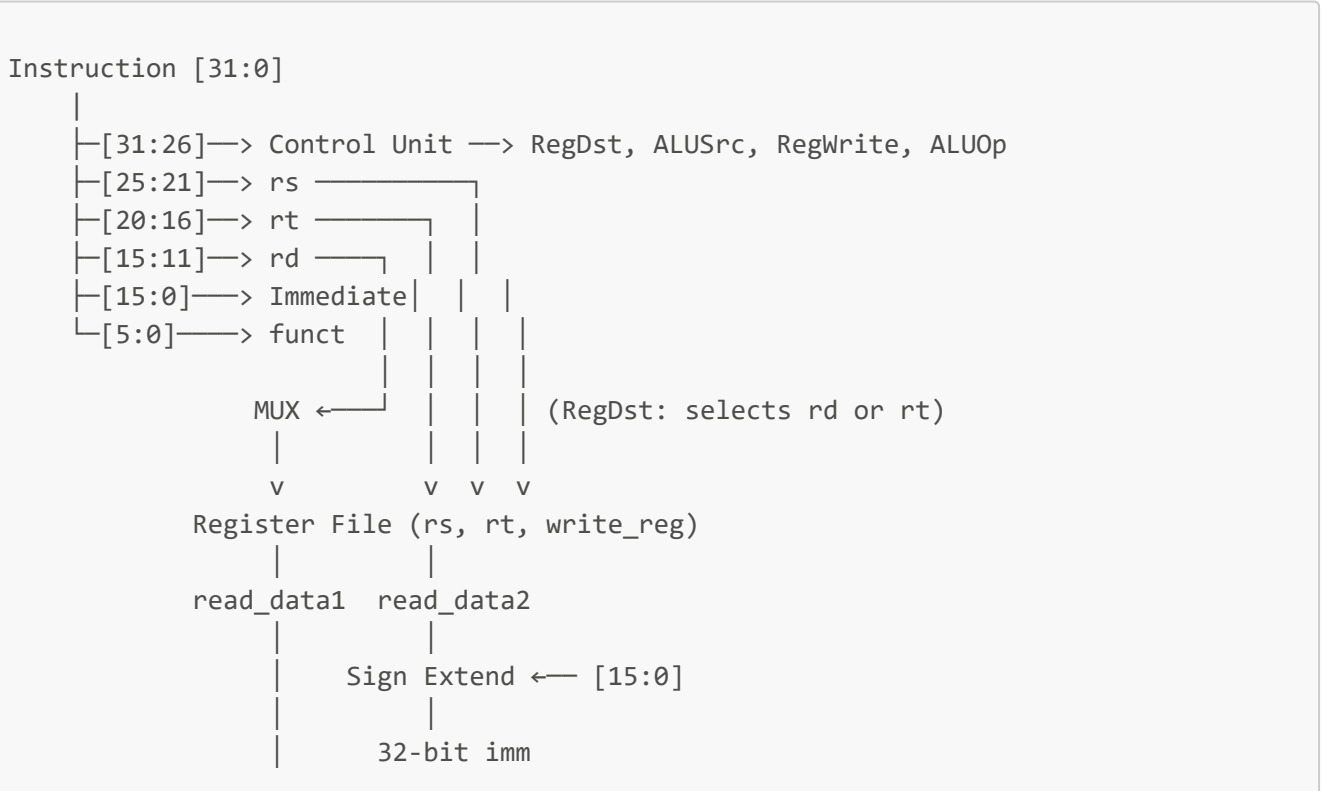
Example Test Sequence:

```
// Example testbench structure
module testbench;
    reg clk;
    reg reset;
    reg [31:0] instruction;
    wire [31:0] alu_result;
    wire [31:0] write_data;

    // Instantiate datapath
    mips_single_cycle_datapath uut(
        .clk(clk),
        .reset(reset),
        .instruction(instruction),
        .alu_result(alu_result),
        .write_data(write_data)
    );

    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    // Test sequence
    initial begin
        // Reset
        reset = 1;
        instruction = 32'h00000000;
        #10 reset = 0;

        // Test 1: addi $1, $0, 10 (R1 = 10)
        // opcode=001000, rs=00000, rt=00001, imm=0000000000001010
        instruction = 32'b001000_00000_00001_0000000000001010;
        #10;
        $display("ADDI: R1 should be 10, Result = %d", write_data);

        // Test 2: addi $2, $0, 20 (R2 = 20)
        instruction = 32'b001000_00000_00010_0000000000010100;
        #10;
```

```
        $display("ADDI: R2 should be 20, Result = %d", write_data);

        // Test 3: add $3, $1, $2 (R3 = R1 + R2 = 30)
        // opcode=000000, rs=00001, rt=00010, rd=00011, shamt=00000, funct=100000
        instruction = 32'b000000_00001_00010_00011_00000_100000;
        #10;
        $display("ADD: R3 should be 30, Result = %d", write_data);

        // Test 4: sub $4, $2, $1 (R4 = R2 - R1 = 10)
        instruction = 32'b000000_00010_00001_00100_00000_100010;
        #10;
        $display("SUB: R4 should be 10, Result = %d", write_data);

        // Test 5: and $5, $1, $2 (R5 = R1 & R2)
        instruction = 32'b000000_00001_00010_00101_00000_100100;
        #10;
        $display("AND: R5 = %d", write_data);

        // Test 6: ori $6, $5, 5 (R6 = R5 | 5)
        instruction = 32'b001101_00101_00110_0000000000000101;
        #10;
        $display("ORI: R6 = %d", write_data);

        $finish;
    end
endmodule
```

MIPS Datapath Diagram

Your datapath should follow this general structure:



```
                    |          |
                    |    MUX ←┘ (ALUSrc: selects rt or immediate)
                    |     |
                    v     v
                     ALU
                      |
                      v
                alu_result ——> write_data (back to Register File)
```

Simplified Architecture Notes

Since you are **NOT using instruction memory**, your design is simpler:

✅ **What you HAVE:**

- Register file (only memory element - 32 registers)
- Instruction as input (32-bit)
- Control logic (Main Control + ALU Control)
- ALU and datapath
- Sign extender for immediates

❌ **What you DON'T HAVE:**

- Program Counter (PC)
- Instruction Memory
- Data Memory
- PC increment logic
- Branch/Jump logic

This allows you to focus on:

- Proper MIPS instruction decoding
- Register file operation ($0 hardwired to zero)
- ALU functionality
- Control signal generation
- Immediate sign-extension
- Datapath connectivity

Hints and Tips

1. **Start Small:** Build and test 1-bit components before scaling to 32-bit
2. **Draw First:** Sketch your datapath diagram before coding
3. **Test Incrementally:** Verify each module individually before integration
4. **Use Parameters:** Define bit widths as parameters for flexibility
5. **Consistent Naming:** Use MIPS naming conventions (rs, rt, rd, funct, opcode)
6. **Verify Register $0:** Make absolutely sure $0 always outputs zero
7. **Sign Extension:** Test with both positive and negative immediates
8. **Instruction Encoding:** Create a reference table - encode manually first

9. **Check Connectivity:** Use waveform viewer to trace signals
10. **RegDst Signal:** Remember R-type writes to rd, I-type writes to rt

---

Common Pitfalls to Avoid

❌ Confusing rs, rt, rd positions between R-type and I-type
❌ Forgetting to sign-extend the 16-bit immediate
❌ Not hardwiring register $0 to zero
❌ Wrong ALU control codes
❌ Mixing up RegDst (should select rd for R-type, rt for I-type)
❌ Using behavioral Verilog operators instead of structural design

---

Prohibited Techniques

❌ `assign result = a + b;` (behavioral arithmetic)
❌ `assign out = (select) ? a : b;` (conditional operator)
❌ `assign y = &x;` (reduction operators)
❌ `assign extended = {{16{imm[15]}}, imm};` (concatenation for sign-extend)
❌ Any behavioral ALU operations
❌ Using memory arrays except in register file

✅ **Allowed:** Module instantiation, wire declarations, `always@` for register file only

---

Recommended Development Order

1. **Part 1 (Nov 18-22):**
    - Design basic 1-bit components (gates, mux, full adder)
    - Build 32-bit adder and subtractor
    - Test arithmetic modules
2. **Part 2 (Nov 23-27):**
    - Implement register file with `always@` blocks - **TEST THOROUGHLY**
    - Create sign extender structurally
    - Build control units (use truth tables)
    - Design and test ALU
3. **Part 3 (Nov 28-Dec 02):**
    - Integrate everything in top module
    - Create comprehensive testbench
    - Debug and verify with waveforms
    - Document your design
