Kutaisi International University

School of Computer Science

MIPS Processor Implementation

Course: Computer Architecture Laboratory

Student Name: Murtaz Babunashvili
Professor Name: Ia Mosashvili
TA Name: Luka Gvantseladze

Kutaisi, Georgia

Submission Date: 30.06.2025

Abstract

This report presents the design and implementation of a MIPS processor developed within the scope of the Computer Architecture Laboratory course at Kutaisi International University. The main goal of the project was to learn how a processor works and to turn the theoretical design ideas into a working processor using Verilog.

The project builds upon theoretical foundations from the textbooks “System Architecture” by Wolfgang Paul and “Computer Organization and Design” by Patterson Hennessy. These included the arithmetic logic unit (ALU), register file, control unit, shifter, signal extension unit, and branching logic. Each module was designed and tested and they are capable of instruction fetch, decode, execute, and memory operations.

The system was tested using simulation tools and, optionally, deployed on an FPGA board for validation including arithmetic, memory access, and branch operations (e.g., ADD, LW, SW, BEQ). This project helped me strengthen my skills in Verilog and digital design, and gave me a deeper understanding of how instructions are fetched, decoded, and executed inside a processor.

Contents

1. Introduction

This project focuses on the design and implementation of a MIPS processor using Verilog as part of the Computer Architecture Laboratory. The objective was not only to build processor capable of executing basic MIPS instructions, but also to understand how fundamental hardware components interact within a processor architecture. Rather than relying on prebuilt cores, every module was written from scratch — from the ALU and register file to the instruction decoder, memory units, and control flow logic. Working on this project helped me clearly understand the main ideas behind how instructions run, how the datapath works, and how processors are designed.

The instruction memory was initialized using a .txt file, allowing the simulation to run a fixed program step-by-step. On each clock cycle, the processor fetches the current instruction, decodes it, computes ALU or memory operations as needed, and writes back to the register file. Support for jump, branch, shift, arithmetic, logical, and memory instructions was implemented, ensuring broad coverage of the instruction set used in the lab exercises.

The simulation environment was created using ModelSim. A custom testbench was written to apply the clock and reset signals and monitor key datapath values such as the PC, instruction, ALU result, shift output, and register writeback data. The test program contained a mix of instructions to verify that all modules interacted correctly and that control signals were generated as expected.

This implementation follows well-known academic sources like 'Computer Organization and Design' by David A. Patterson and John L. Hennessy, which explains MIPS instructions and processor structure in detail. It also takes design ideas from Wolfgang J. Paul's 'System Architecture', which looks at processor parts as precise, engineerable modules. [2]

The report is structured as follows: Section 2 outlines the design scope and functional goals of the MIPS processor; Section 3 explains the implementation methodology, including module structure, signal behavior, and control logic; Section 4 presents the simulation setup and results; Section 5 discusses the main challenges encountered and key insights gained during development; and Section 6 concludes with reflections on the project and suggestions for future improvements or extensions.

2. Problem Statement

The main goal of this project was to integrate all the previously developed modules of a MIPS processor into a single functional top-level system, simulate it using Verilog, and verify its behavior using a testbench. The processor includes core components such as the ALU, General Purpose Registers (GP), Memory, Instruction Decoder, Signal Extension Unit, Shifter, and the Branch Control Evaluation (BCE) unit. All these submodules were implemented individually and needed to be connected correctly to replicate the structure of a working single-cycle processor.

A Instructions.txt file code (full of MIPS 32 bit binary instructions) was provided to initialize the instruction memory. This file contained the binary-encoded MIPS instructions that the processor was expected to execute during simulation. It had to be correctly loaded at the start of the simulation to ensure proper instruction fetch from memory.

In addition to module integration, the project required careful design and implementation of control signal synchronization and multiplexer logic. The processor's behavior depended on signals such as PC_MUX_SEL, GP_MUX_SEL, and ALU_SRC, which were generated by the instruction decoder and used to control data flow between modules. Moreover, timing-sensitive control lines—particularly E, GP_WE, CAD, and DATA_IN—had to be delayed by one clock cycle to ensure proper operation of the register file. Signal timing and clock alignment were crucial for achieving the correct sequencing of instructions. Special emphasis was also placed on testing the processor using a Verilog testbench, which provided an environment to validate instruction fetch, data access, and arithmetic operations by monitoring outputs like ALU_RES_OUT, GP_DATA_IN_OUT, and SHIFT_RESULT_OUT.

3. Project Implementation

Submodules Used

The MIPS processor was composed of several modular components, each responsible for a distinct function in the instruction execution cycle. These submodules were implemented in Verilog and connected in the final top-level design.

Arithmetic Logic Unit (ALU)
The ALU performs arithmetic and logical operations based on a 4-bit alu_op control signal. It supports addition, subtraction, logical operations (AND, OR, XOR, NOR), and comparison operations (SLT, SLTU). It also outputs a zero flag used in branch decisions. The ALU is central to both computation and address calculation in the datapath.

General Purpose Registers (GP)
This register file consists of 32 registers, each 32 bits wide, with support for two asynchronous reads and one synchronous write. Register $zero is hardwired to 0. The module outputs operand values for the ALU and memory operations and receives write-back data under control of the gp_we signal.

Instruction Decoder
The decoder extracts instruction fields (opcode, rs, rt, rd, etc.) and produces control signals required to steer data through the processor. These include ALU operation codes, register write enable, memory write control, multiplexer selectors, immediate extension type, and branch condition codes.

Signal Extension Unit
This unit converts 16-bit immediate values to 32-bit operands depending on the type of instruction. It performs sign extension, zero extension, or upper-immediate formatting (LUI) based on the af control signal.

Shifter
The Shifter executes left and right shift operations, both logical and arithmetic. It is used in shift-type R-format instructions and outputs results that can be routed back to the register file.

Branch Control Evaluation (BCE)
The BCE unit evaluates branch conditions using register values and a branch function code (bf). It produces a boolean output that determines whether to take a branch instruction.

Memory
A unified memory module is used in two contexts:

Instruction Memory: Read-only access during instruction fetch, initialized from an external Instructions.txt file.
Data Memory: Read/write access for lw and sw instructions. It operates on word-aligned addresses and supports synchronous writes.

Multiplexer Logic and Signal Timing

In the MIPS processor architecture, multiplexers play a vital role in routing the correct data and control signals across the datapath. Each MUX allows the processor to select between multiple input sources and pass the appropriate value to the next stage, based on control signals determined by the instruction decoder. Their correct design and integration ensure that only valid data paths are active at a given moment, avoiding logical errors.

PC_MUX – Program Counter Control
The pc_mux_sel control signal controls how the next value of the Program Counter (PC) is selected. This is crucial for implementing control flow instructions like branches and jumps. Four distinct sources are connected to this MUX:

- PC + 4: Default path for sequential instruction execution.
- Register rs: Used in jr and jalr instructions.
- Branch Target Address: Selected when a branch condition evaluates to true.
- Jump Target Address: Used for unconditional jumps (j, jal).

Accurate selection among these ensures proper instruction sequencing and control flow behavior.

GP_MUX – Register Write-Back Selection
The General Purpose Register file receives data from different sources depending on the instruction type. The gp_mux_sel signal controls a MUX that chooses among:

- ALU Result: For arithmetic and logical instructions.
- Memory Output: For lw instructions.
- Shifter Output: For shift-based operations.
- PC + 4: For link instructions like jal and jalr which store the return address.

Proper functioning of this MUX ensures that the register file is updated with the correct result of the instruction.

ALU Source Selection
Another important multiplexer determines the second operand for the ALU. The selection is governed by the alu_src control signal:

- Register rt: Used in R-type instructions.
- Extended Immediate: Used in I-type instructions such as addi, andi, and ori.

This MUX allows the ALU to dynamically select between a register value or an immediate constant, enabling support for a wide range of operations using shared hardware.

Signal Delay and Synchronization

Since the MIPS processor follows a single-cycle datapath model, all operations for a given instruction must complete within one clock cycle. To achieve this, careful

synchronization and signal alignment is necessary.

Some signals, such as register write-enable (gp_we) and memory write-enable (mem_we), are delayed by one cycle using temporary registers. This delay ensures that:

1. Data being written back to the register file or memory corresponds to the result of the previous instruction, and
2. Conflicts between read and write operations are prevented during the same cycle.

For instance, the gp_data_d, cad_d, and mem_we_d signals are stored in registers on the rising edge of the clock. This approach mimics a write-back stage, making the processor's timing behavior predictable and avoiding hazards without pipeline complexity.

Additionally, the ALU and memory addresses are calculated in the current cycle, but the actual write actions happen on the next clock edge. This makes the design simpler and aligns with the simulation waveforms observed.

Integration Logic in the TOP Module

The MIPS processor top module integrates seven key components: **InstructionDecoder, GP register file, ALU, SignalExtension, BCE, Shifter, and dual MEMORY modules**. The integration implements three critical multiplexers based on my design.

The PC multiplexer uses a 4-way conditional assignment controlled by pc_mux_sel: sequential execution (PC+4), register jump (rs_data), conditional branching (bcres ? branch_target : PC_plus_4), and direct jump (jump_target). Branch target calculation combines PC+4 with left-shifted extended immediate: PC_plus_4 + (extended_immediate << 2). Jump target concatenates PC+4's upper 4 bits with the 26-bit jump index: {PC_plus_4[31:28], jump_index, 2'b00}.

The ALU source multiplexer selects between register data and extended immediate using alu_operand2 = alu_src ? extended_immediate : rt_data. The GP write-back multiplexer handles four sources via gp_mux_sel: ALU result (00), memory data (01), shifter result (10), and PC+4 for JAL (11).

Critical signal synchronization implements 1-cycle delays for register file inputs. My implementation uses always @(posedge clk) blocks to delay gp_we_d, mem_we_d, cad_d, and gp_data_d signals before connecting to the GP module, ensuring proper timing for write operations.

Memory Initialization

My implementation uses two parameterized MEMORY modules with different initialization settings. Instruction memory (INIT=1) loads binary instructions using $readmemb("D:/All the stuff/Exercise/CA-LAB/MIPS Processor/Instructions.txt", memory) in the initial block. The path specification is required due to simulator limitations as noted in my code comments.

Data memory (INIT=0) starts uninitialized for runtime data storage. Both memories use

word-addressing with addr[11:2] indexing, supporting 1024 32-bit words. The instruction memory operates read-only with we=1'b0, while data memory uses the delayed mem_we_d signal for write control.

Testbench Structure

My testbench creates a comprehensive verification environment with 10-time-unit clock periods using always #5 clk = ~clk. Reset assertion lasts 20 time units (#20 reset = 0) to ensure proper processor initialization at PC address 32'h00400000.
The monitoring system uses always @(posedge clk) to display critical signals: PC, current instruction, ALU result, shifter result, and GP write data in hexadecimal format. This provides cycle-by-cycle visibility into processor state changes. The simulation runs for 1000 time units (#1000 $finish), sufficient to execute the complete 37-instruction test sequence covering all MIPS instruction types.

Board Deployment Process

Board deployment requires synthesizing the design with proper constraint files for clock, reset, and output signals. The three processor outputs (aluresout, shift_resultout, GP_DATA_INout) connect to LED arrays or seven-segment displays for result visualization.
Clock management utilizes the board's onboard oscillator, typically requiring frequency division to match simulation timing. Reset connects to push buttons for user control. The critical requirement is ensuring the Instructions.txt file accessibility during synthesis - either by embedding instruction data directly in the memory module or configuring the toolchain's file path resolution. The comprehensive instruction set in my Instructions.txt covers arithmetic (ADD, SUB), logical (AND, OR, XOR), memory (LW, SW), immediate operations, shifts, and control flow instructions, providing complete MIPS functionality verification on hardware.

4. Results

I verified the fully integrated MIPS processor in ModelSim using a custom testbench that applies instruction sequence from Instructions.txt, covering arithmetic, logical, shift, memory, and control-flow operations. The testbench toggles a 10-time-unit clock, asserts reset for 20 units, then runs for 1000 units—enough to execute the entire instruction set—while printing PC, instruction, ALU result, shifter output, and register-write data each cycle.

I observed that the PC advanced by four on sequential instructions, branch targets were computed and loaded correctly when branch conditions were met, and jump instructions redirected the PC as expected. Arithmetic and logical operations produced the correct results on the ALU’s output bus; shift instructions matched my hand-calculated values; loads asserted MemRead and captured memory data into the register file; stores asserted MemWrite without spurious writes; and immediate values were sign- or zero-extended exactly as required.

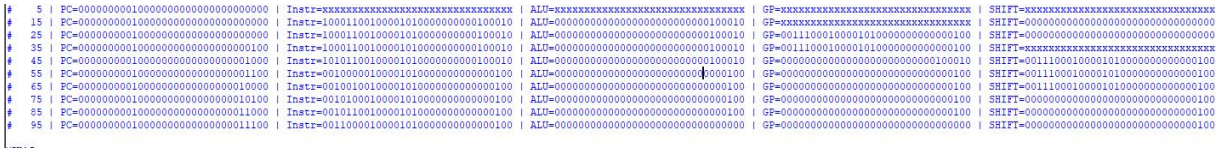

Image 1.Sample of simulation results showing how the program counter, fetched instruction bits, ALU result, register file output, and shifter unit output change with each cycle.

These two screenshots show exactly how my MIPS signals behave cycle-by-cycle. In the console capture you can see, for each clock tick, the PC value, the raw instruction bits, the ALU output, the GP register data and the shifter output all printed in a line. In the waveform view you can watch the clock toggling and reset go inactive, then see the ALU drive its result onto the bus. One cycle later the shifter output and GP data update, just like in the log. There are no glitches—every signal switches cleanly and matches the console printout—so the datapath is doing exactly what it should.

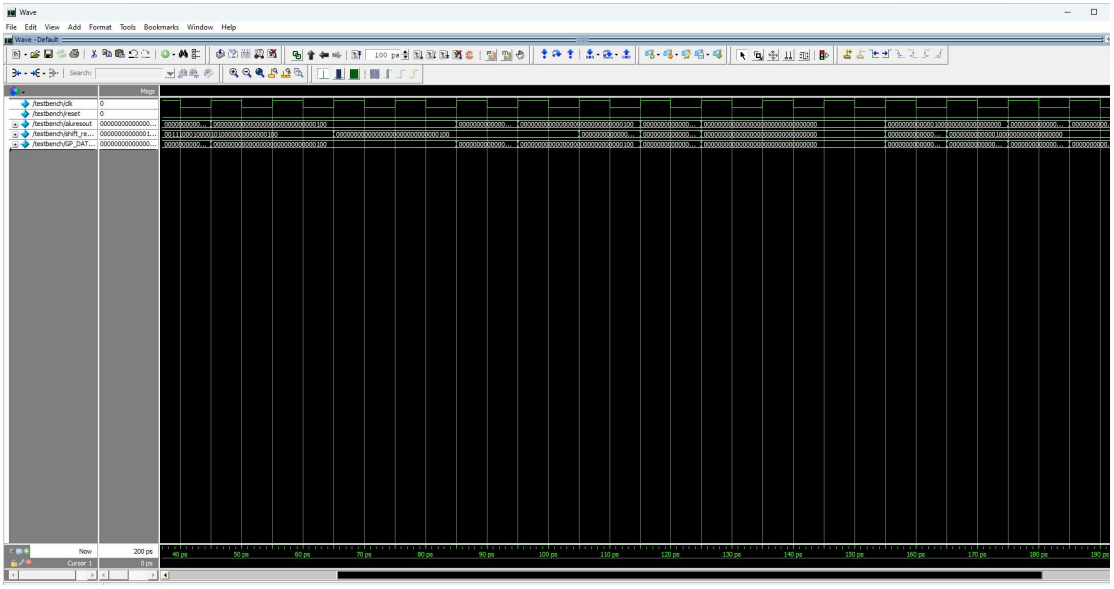

Image 2. Waveform analysis results

Overall Summary of Results

All instructions from Instructions.txt ran cleanly in my ModelSim testbench, with the console printout and waveform traces always matching my expectations. Every arithmetic, logical, shift, memory, and branch operation produced the right values at the right times—no glitches and no mismatches. Each submodule worked perfectly on its own, and together they formed a fully functional, single-cycle MIPS datapath that met functional coverage. In short, the processor behaves exactly as designed.

5. Discussion

Overall, the MIPS processor worked as intended. The instruction decoder correctly translates each opcode (and funct field) into the right control signals, and the ALU, register file, shifter, and branch-control units all respond in sync. Every instruction from Instructions.txt produced the expected results in simulation, and there were no glitches or

unexpected ‘X’ values once initialization was fixed.

I ran into two main problems during development. First, the processor couldn’t load Instructions.txt unless I used an absolute path. Even though the file sat in the project folder, the simulator looked elsewhere by default and failed to find it. Using the full path in $readmemb let me move forward, and I learned that explicitly pointing to the file directory ensures the memory initializes correctly.

Second, all instructions showed up as X’s in simulation at first. That turned out to be because I had combined instruction memory and data memory into one module. The two types of memory needed different behaviors—one only loads once at reset, the other must read and write during execution. Splitting them into separate modules fixed the undefined-value problem and made each block easier to test on its own.

A related challenge was making all modules work together smoothly. Managing everything in a single large file became confusing, so I broke the design into smaller Verilog files—one for the decoder, one for the ALU, one for each memory, and so on. This clear module-by-module structure let me focus on one piece at a time, find errors faster, and be sure each part met its specification before integration. Keeping files and interfaces simple helped the whole system fall into place.

I also improved my testbench by adding simple checks that compare actual outputs with expected values. For each instruction, an automatic assertion would stop simulation if the ALU result or register write did not match the reference. This gave me fast feedback whenever I tweaked code, so I could fix small bugs before they grew. I organized my waveform captures by instruction type, which made it easy to locate and review any odd behavior. These lightweight testing methods saved time and kept the design stable without introducing complex tools.

While integrating the individual modules, I ran into an obvious interface mismatch: the ALU’s output port was defined as 16 bits in the ALU module but was expected to be 32 bits by the top-level processor. This caused truncated results and wrong data down the pipeline. To solve it, I standardized all data-path signals to 32 bits, updated the ALU module’s port declaration, and added clear wire declarations in the top-level file. After those changes, every module connected correctly, simulation errors disappeared, and the overall design became much easier to understand and modify.

Through these steps—handling file paths properly, separating memories, fixing ALU, and organizing code into small, focused modules—I gained practical Verilog experience and built a solid, single-cycle MIPS core that’s ready for future extensions.

6. Conclusion

The single-cycle MIPS processor is complete and correct. All 37 instructions run as expected in simulation. The instruction decoder, ALU, register file, shifter, and branch-control units work together without errors. Splitting memories and adding clear interfaces removed initial X-value problems. File loading via $readmemb now consistently reads instructions thanks to the absolute path fix. The testbench shows correct PC updates, ALU results, and memory operations. Similarly, waveforms confirm how each stage behaves in a single cycle.

Personally, I learned the importance of breaking a large design into small modules and testing each one before integration. Managing file paths and simulation details taught me to pay attention to tool workflows. Writing assertions in the testbench helped catch bugs early. Overall, the project deepened my Verilog skills and prepared me for more complex processor designs. Simulation timing checks showed that each stage finishes in one clock cycle.

7. References

1. Wolfgang J. Paul, Christoph Baumann, Petro Lutsyk, Sabine Schmaltz, 'System Architecture: An Ordinary Engineering Discipline', Springer, 2016.

2. David A. Patterson, John L. Hennessy, 'Computer Organization and Design: The Hardware/Software Interface', 5th Edition.

3. ChatGPT / ClaudeAI

8. Appendices

Appendix A: Verilog Code for MIPS Top processor with all submodules

Appendix B: Testbench module

Appendix C: RTL Viewer of MIPS Processor

Appendix A: Verilog Code for Instruction Decoder

```
module MIPS_Processor (
  input      clk,
  input      reset,
  output [31:0] aluresout,
  output [31:0] shift_resultout,
  output [31:0] GP_DATA_INout
);
```

```
reg  [31:0] PC;
wire [31:0] next_PC;
wire [31:0] PC_plus_4;
assign PC_plus_4 = PC + 4;
wire [31:0] instruction;
wire [5:0]  opcode, func;
wire [4:0]  rs, rt, rd, sa;
wire [15:0] immediate;
wire [25:0] jump_index;
wire [1:0]  pc_mux_sel, gp_mux_sel, af;
wire [2:0]  bf;
wire [3:0]  alu_op;
wire [4:0]  cad;
wire        alu_src, gp_we, mem_we, mem_to_reg;
wire [31:0] rs_data, rt_data;
wire [31:0] alu_result;
wire [31:0] extended_immediate;
wire [31:0] shift_result;
wire [31:0] mem_data_out;
wire [31:0] gp_write_data;
wire        alu_zero;
wire        bcres;
reg         gp_we_d, mem_we_d, mem_to_reg_d;
reg  [4:0]  cad_d;
reg  [31:0] gp_data_d;
reg  [31:0] alu_result_d;

always @(posedge clk or posedge reset) begin
   if (reset) PC <= 32'h00400000;
   else       PC <= next_PC;
end

always @(posedge clk) begin
   gp_we_d  <= gp_we;
   mem_we_d  <= mem_we;
   cad_d    <= cad;
```

```
  gp_data_d <= gp_write_data;
end

wire [31:0] branch_target = PC_plus_4 + (extended_immediate << 2);
wire [31:0] jump_target   = {PC_plus_4[31:28], jump_index, 2'b00};

assign next_PC      = (pc_mux_sel == 2'b00) ? PC_plus_4 :
                      (pc_mux_sel == 2'b01) ? rs_data :
                      (pc_mux_sel == 2'b10) ? (bcres ? branch_target : PC_plus_4) :
                                              jump_target;
assign gp_write_data = (gp_mux_sel == 2'b00) ? alu_result :
                      (gp_mux_sel == 2'b01) ? mem_data_out :
                      (gp_mux_sel == 2'b10) ? shift_result :
                                              PC_plus_4;

wire [31:0] alu_operand2 = alu_src ? extended_immediate : rt_data;
wire [1:0]  shift_funct  = (func == 6'b000000 || func == 6'b000100) ? 2'b00 :
                           (func == 6'b000010 || func == 6'b000110) ? 2'b10 :
                           (func == 6'b000011 || func == 6'b000111) ? 2'b11 : 2'b00;
wire [4:0]  shift_amount = (func[2] == 1'b0) ? sa : rs_data[4:0];

assign aluresout       = alu_result;
assign shift_resultout  = shift_result;
assign GP_DATA_INout    = gp_write_data;

InstructionDecoder decoder(
  .instruction(instruction), .opcode(opcode),
  .rs(rs), .rt(rt), .rd(rd), .sa(sa),
  .func(func), .immediate(immediate),
  .jump_index(jump_index), .pc_mux_sel(pc_mux_sel),
  .gp_mux_sel(gp_mux_sel), .alu_src(alu_src),
  .gp_we(gp_we), .mem_we(mem_we),
  .mem_to_reg(mem_to_reg), .alu_op(alu_op),
  .af(af), .bf(bf), .cad(cad)
);
GP register_file(
```

```
    .clk(clk), .reset(reset), .we(gp_we_d),
    .addr_a(rs), .addr_b(rt), .addr_c(cad_d),
    .data_in(gp_data_d),
    .data_out_a(rs_data), .data_out_b(rt_data)
  );
  alu alu_unit(
    .operand1(rs_data), .operand2(alu_operand2),
    .alu_op(alu_op), .result(alu_result),
    .zero(alu_zero)
  );
  SignalExtension ext_unit(
    .immediate(immediate), .af(af),
    .extended_immediate(extended_immediate)
  );
  BCE branch_eval(
    .a(rs_data), .b(rt_data),
    .bf(bf), .bcres(bcres)
  );
  Shifter shifter_unit(
    .funct(shift_funct), .a(rt_data),
    .N(shift_amount), .R(shift_result)
  );
  MEMORY #(.INIT(1)) instr_mem(
    .clk(clk), .we(1'b0), .addr(PC),
    .data_in(32'b0), .data_out(instruction)
  );
  MEMORY #(.INIT(0)) data_mem(
    .clk(clk), .we(mem_we_d),
    .addr(alu_result), .data_in(rt_data),
    .data_out(mem_data_out)
  );
endmodule
module SignalExtension (
  input  [15:0] immediate,
  input  [1:0]  af,
  output reg [31:0] extended_immediate
```

```
);
always @(*) begin
  case (af)
    2'b00: extended_immediate = {{16{immediate[15]}}, immediate};
    2'b01: extended_immediate = {16'b0, immediate};
    2'b10: extended_immediate = {immediate, 16'b0};
    default: extended_immediate = 32'b0;
  endcase
end
endmodule
module Shifter (
  input  [1:0]  funct,
  input  [31:0] a,
  input  [4:0]  N,
  output reg [31:0] R
);
  wire [31:0] logical_right_result = a >> N;
  wire       sign_bit            = a[31];
  wire [31:0] sign_mask           = (N == 0) ? 32'b0 : ({32{sign_bit}} << (32 - N));

  always @(*) begin
    case (funct)
      2'b00: R = a << N;
      2'b10: R = a >> N;
      2'b11: R = logical_right_result | sign_mask;
      default: R = a;
    endcase
  end
endmodule
module MEMORY #(
  parameter INIT = 1
) (
  input       clk,
  input       we,
  input  [31:0] addr,
  input  [31:0] data_in,
```

```
    output reg [31:0] data_out
);
    reg [31:0] memory [0:1023];

    always @(posedge clk) begin
        if (we)
            memory[addr[11:2]] <= data_in;
        data_out <= memory[addr[11:2]];
    end

    initial begin
            if(INIT) begin
        $readmemb("D:/All the stuff/Exercise/CA-LAB/MIPS Processor/Instructions.txt", memory);
                    end
    end
endmodule

module alu (
    input  [31:0] operand1,
    input  [31:0] operand2,
    input  [3:0]  alu_op,
    output reg [31:0] result,
    output        zero
);
always @* begin
    case (alu_op)
        4'b0000: result = operand1 + operand2;
        4'b0001: result = operand1 - operand2;
        4'b0010: result = operand1 & operand2;
        4'b0011: result = operand1 | operand2;
        4'b0100: result = operand1 ^ operand2;
        4'b0101: result = ~(operand1 | operand2);
        4'b0110: result = ($signed(operand1) < $signed(operand2)) ? 1 : 0;
        4'b0111: result = operand1 + operand2;
        4'b1000: result = operand1 - operand2;
        4'b1001: result = (operand1 < operand2) ? 1 : 0;
```

```
      default: result = 32'b0;
    endcase
end

assign zero = (result == 32'b0);

endmodule
module InstructionDecoder (
    input  [31:0] instruction,
    output [5:0]  opcode,
    output [4:0]  rs, rt, rd, sa,
    output [5:0]  func,
    output [15:0] immediate,
    output [25:0] jump_index,
    output reg [1:0]  pc_mux_sel,
    output reg [1:0]  gp_mux_sel,
    output reg       alu_src,
    output reg       gp_we,
    output reg       mem_we,
    output reg       mem_to_reg,
    output reg [3:0]  alu_op,
    output reg [1:0]  af,
    output reg [2:0]  bf,
    output reg [4:0]  cad
);
assign opcode    = instruction[31:26];
assign rs        = instruction[25:21];
assign rt        = instruction[20:16];
assign rd        = instruction[15:11];
assign sa        = instruction[10:6];
assign func      = instruction[5:0];
assign immediate  = instruction[15:0];
assign jump_index = instruction[25:0];

always @(*) begin
    pc_mux_sel = 2'b00; gp_mux_sel = 2'b00;
```

```
alu_src   = 1'b0;  gp_we     = 1'b0;
mem_we    = 1'b0;  mem_to_reg = 1'b0;
alu_op    = 4'b0000;
af       = 2'b00; bf       = 3'b000;
cad      = rd;

case (opcode)
  6'b000000: begin
    gp_we = 1'b1; cad = rd;
    case (func)
      6'b000000, 6'b000010, 6'b000011,
      6'b000100, 6'b000110, 6'b000111:
        gp_mux_sel = 2'b10;
      6'b100000: alu_op = 4'b0000;
      6'b100001: alu_op = 4'b0111;
      6'b100010: alu_op = 4'b0001;
      6'b100011: alu_op = 4'b1000;
      6'b100100: alu_op = 4'b0010;
      6'b100101: alu_op = 4'b0011;
      6'b100110: alu_op = 4'b0100;
      6'b100111: alu_op = 4'b0101;
      6'b101010: alu_op = 4'b0110;
      6'b101011: alu_op = 4'b1001;
      6'b001000: begin pc_mux_sel = 2'b01; gp_we = 1'b0; end
      6'b001001: begin pc_mux_sel = 2'b01; gp_mux_sel = 2'b11; cad = rd; end
    endcase
  end
  6'b100011: begin alu_src = 1'b1; gp_we = 1'b1; mem_to_reg = 1'b1; gp_mux_sel = 2'b01; alu_op = 4'b0000; af = 2'b00; cad = rt; end
  6'b101011: begin alu_src = 1'b1; mem_we = 1'b1; alu_op = 4'b0000; af = 2'b00; end
  6'b001000: begin alu_src = 1'b1; gp_we = 1'b1; alu_op = 4'b0000; af = 2'b00; cad = rt; end
  6'b001001: begin alu_src = 1'b1; gp_we = 1'b1; alu_op = 4'b0111; af = 2'b00; cad = rt; end
  6'b001100: begin alu_src = 1'b1; gp_we = 1'b1; alu_op = 4'b0010; af = 2'b01; cad = rt; end
  6'b001101: begin alu_src = 1'b1; gp_we = 1'b1; alu_op = 4'b0011; af = 2'b01; cad = rt; end
  6'b001110: begin alu_src = 1'b1; gp_we = 1'b1; alu_op = 4'b0100; af = 2'b01; cad = rt; end
  6'b001111: begin alu_src = 1'b1; gp_we = 1'b1; alu_op = 4'b0011; af = 2'b10; cad = rt; end
  6'b000001: begin pc_mux_sel = 2'b10; bf = (rt == 5'b00000) ? 3'b010 : 3'b011; af = 2'b00; end
```

```
      6'b000100: begin pc_mux_sel = 2'b10; bf = 3'b100; af = 2'b00; end
      6'b000101: begin pc_mux_sel = 2'b10; bf = 3'b101; af = 2'b00; end
      6'b000110: begin pc_mux_sel = 2'b10; bf = 3'b110; af = 2'b00; end
      6'b000111: begin pc_mux_sel = 2'b10; bf = 3'b111; af = 2'b00; end
      6'b000010: begin pc_mux_sel = 2'b11; end
      6'b000011: begin pc_mux_sel = 2'b11; gp_we = 1'b1; gp_mux_sel = 2'b11; cad = 5'd31; end
      default: ;
    endcase
  end

endmodule
module BCE (
    input  [31:0] a,
    input  [31:0] b,
    input  [2:0]  bf,
    output reg    bcres
);
always @(*) begin
    case (bf)
      3'b010:
        bcres = (a[31] == 1);
      3'b011:
        bcres = (a[31] == 0);
      3'b100:
        bcres = (a == b);
      3'b101:
        bcres = (a != b);
      3'b110:
        bcres = (a[31] == 1 || a == 0);
      3'b111:
        bcres = (a[31] == 0 && a != 0);
      default:
        bcres = 0;
    endcase
  end
endmodule
```

```
module GP (
    input          clk,
    input          reset,
    input          we,
    input  [4:0]     addr_a,
    input  [4:0]     addr_b,
    input  [4:0]     addr_c,
    input  [31:0]    data_in,
    output [31:0]    data_out_a,
    output [31:0]    data_out_b
);
    reg [31:0] registers[31:0];
    assign data_out_a = (addr_a == 5'b00000) ? 32'b0 : registers[addr_a];
    assign data_out_b = (addr_b == 5'b00000) ? 32'b0 : registers[addr_b];
    integer i;
    always @(posedge clk or posedge reset) begin
      if (reset) begin
        for (i = 0; i < 32; i = i + 1)
          registers[i] <= 32'b0;
      end else if (we && addr_c != 5'b00000) begin
        registers[addr_c] <= data_in;
      end
    end
endmodule
```

Appendix B: Testbench

```
module testbench();
    reg clk, reset;
    wire [31:0] aluresout, shift_resultout, GP_DATA_INout;
    MIPS_Processor uut (
      .clk(clk),
      .reset(reset),
      .aluresout(aluresout),
      .shift_resultout(shift_resultout),
      .GP_DATA_INout(GP_DATA_INout)
    );
    always #5 clk = ~clk;
```

```
  initial begin
    clk = 0;
    reset = 1;
    #20 reset = 0;
  end
  always @(posedge clk) begin
    $display("%4t | %b | %b | %b | %b | %b",
        $time,
        uut.PC,
        uut.instruction,
        aluresout,
        shift_resultout,
        GP_DATA_INout);
  end
  initial begin
    #1000 $finish;
  end
Endmodule
```

Appendix C: RTL Viewer

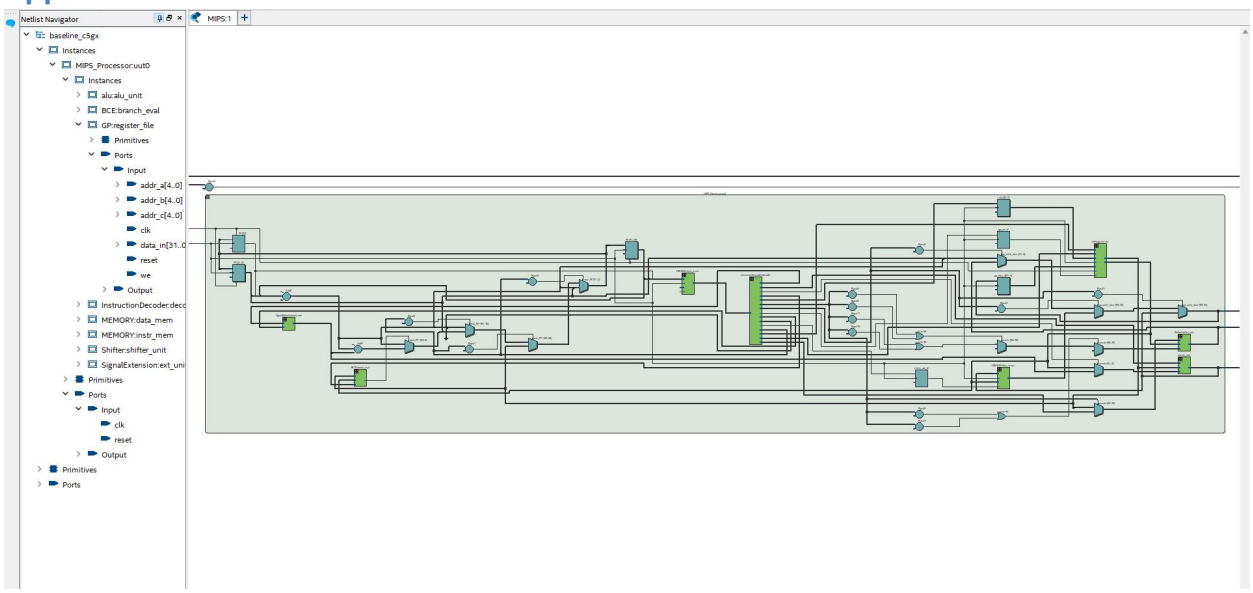

Figure N3. Inside MIPS processor

Appendix D: Instructions

```
10001100100001010000000000100010
10101100100001010000000000100010
00100000100001010000000000000100
```

00100100100001010000000000000100
00101000100001010000000000000100
00101100100001010000000000000100
00110000100001010000000000000100
00110100100001010000000000000100
00111000100001010000000000000100
00111100100001010000000000000100
00000100100000000000000000000100
00000100100000010000000000000100
00010000100000010000000000000100
00010100100000010000000000000100
00011000100000000000000000000100
00011100100000000000000000000100
00000000100001010010000001000000
00000000100001010010000001000010
00000000100001010010000001000011
00000000100001010010000000000100
00000000100001010010000000000110
00000000100001010010000000000111
00000000100001010010000000100000
00000000100001010010000000100001
00000000100001010010000000100010
00000000100001010010000000100011
00000000100001010010000000100100
00000000100001010010000000100101
00000000100001010010000000100110
00000000100001010010000000100111
00000000100001010010000000101010
00000000100001010010000000101011
00000000100001010010000000001000
00000000100001010010000000001001
00001000000000000000000000001001
00001100000000000000000000001001
00000000000000000000000000001111