Midterm Project Report

Design and Implementation of a Sequential Processor Using Verilog

Laboratory: Computer Organization and Architecture
2024–2025

Instructor: Luka Karoniani

Submission Date: *April 3, 2025*

Submitted by: Luka Khachapuridze

Introduction

This project focuses on the design and implementation of a sequential MIPS processor based on the MIPS Instruction Set Architecture (ISA), using Verilog HDL in Quartus Prime 20.1. The processor is synthesized and tested on the Cyclone V GTX Starter Kit from Altera (Intel), demonstrating its functionality through instruction execution and register-level operations.

Sequential digital circuits play a crucial role in system architecture, as they provide the ability to store and process data over time. Unlike combinational circuits, which produce outputs solely based on current inputs, sequential circuits incorporate memory elements that retain state information. This characteristic is essential for building components such as registers, program counters, and control units — all of which are fundamental to the operation of a processor. In modern digital systems, sequential logic enables the implementation of instruction cycles, timing control, and complex decision-making based on past events.

Two **addiu** instructions are being tested on board - **addiu $rs $rt imm**

Key Components:

1. *Flip-Flops* $\rightarrow$ *chainging the execute bit* $E \in B \rightarrow E = \neg E$ *every clock cycle*
2. *Decoder* $\rightarrow$ *Instruction decoder* : *Decoding the* 32*bit instruction* : $instruction[31:0] \in B^{32}$
3. *Counter* $\rightarrow pc \in B^8$: *program counter*(**should have been 32, but due to the lack of space)**
4. *Incrementor* $\rightarrow$ *Incrementing* $pc \in B^8$ *every clock cycle* : $fetching$ *the next instruction* $from$ *memory*
5. *Shift Register* $\rightarrow$ *Register* for *shifting operations.*

The implementation is based on a 5-stage MIPS-ISA architecture.

System Architecture & Design Approach

Structural design was used to define the top-level organization of the processor by instantiating and connecting various functional components. The `baseline` module serves as the main structural unit, where the `cpu_logic` module is instantiated. This top-level module(cpu_logic) handles the orchestration of the instruction fetch, decode, and execute stages.

```
wire [15:0] short_display;
cpu_logic cpu
(
    .clock(CLOCK_50_B5B),
    .clockEna(SW[9]),
    .reset(SW[8]),
    .short_display(short_display)
    //.SUCCESSFULL(LEDG[5])
);
```

Listing 1: cpu_logic instantiation in baseline

Below is an example of structural instantiation of the instruction decoder module within `cpu_logic`:

```
instruction_decod decoder (
    .instruction(instruction),
    .opc(opc),
    .rs(rs),
    .rt(rt),
    .rd(rd),
    .shamt(shamt),
    .funct(funct),
    .imm(imm),
    .iindex(iindex),
    .finalType(type)
);
```

Listing 2: Structural instantiation of the decoder module

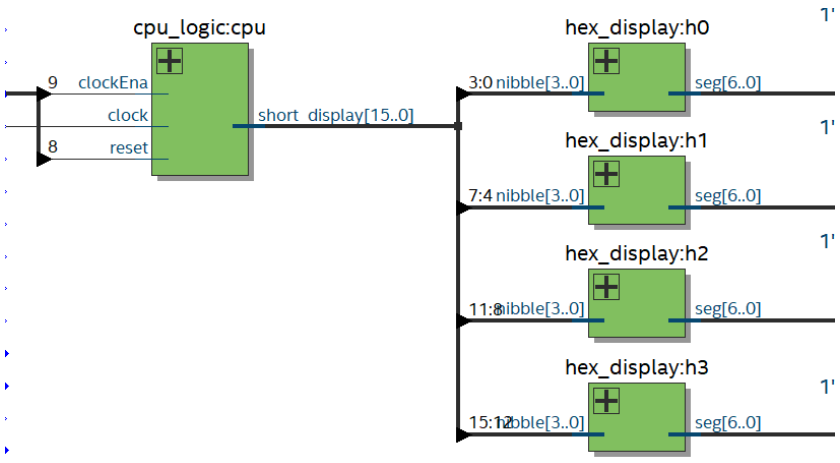

Figure 1: Baseline structure instantiating `cpu_logic` and `hex_display` modules to observe outputs.

Behavioral design is used in modules where the internal operation is described using high-level control structures such as if, case, and always blocks. Instead of focusing on how components are connected, behavioral design describes what the system should do over time. In this project, behavioral design is applied in modules that manage control flow, sequential logic, and register updates.

A typical example is the reset button that changes the value of each register to 0. (pc updates in cpu_logic):

```
always @(posedge clk or posedge rst) begin
if( rst ) begin
    for( i = 1; i <32; i=i+1)
        registers[i] <= 32'b0;
end
```

Listing 3: Changing the register value to 0 after reset

Component Design and Implementation

As we have already mentioned, the project involves several main components: Flip-Flops, Decoder, Incrementor, Counter, Shift Register.

Flip-Flops:

Flip-flops play a crucial role in the current design, particularly in controlling the execution flow of the processor. In this implementation, the signal E is toggled ($E = \neg E$) on every rising edge of the system clock (CLOCK_50_B5B), functioning as an internal execution phase flag. When E = 0, the processor is in the fetch phase, and when E = 1, it enters the execute phase. This flip-flop-based control mechanism helps synchronize the processor's operation across clock cycles without using a complex finite state machine.

```
always @(posedge clock or posedge reset) begin
    if(reset) begin
        E <=0;
        pc <= 8'b0;
    end
    else begin
        E <= ~E;
        if(E) begin
            pc <= nextpc;
        end
    end
end
```

Listing 4: Changing the register value to 0 after reset

Instruction Decoder:

The instruction decoder is responsible for interpreting the 32-bit binary instruction fetched from memory. Outputting the crucial information like: opc, instruction type and etc.

$instruction[31:0] \rightarrow output\ [5:0]\ opc;\ output\ [4:0]\ rs;\ output\ [4:0]\ rt;\ output\ [4:0]\ rd;\ output\ [4:0]\ shamt;\ output\ [5:0]\ funct;\ output\ [15:0]\ imm;\ output\ [25:0]\ iindex;\ output\ [1:0]\ finalType;$

```
function [1:0] Type (input [5:0] opcode);
    casez (opcode)
        6'b000000: Type = 2'b01; // R TYPE INSTRUCTION:
           OPC IS ALWAYS 0;
        6'b000010, 6'b000011: Type = 2'b10;  // J TYPE
           INSTRUCTION 2 OPCS
        default: Type = 2'b00;  // all the other valid
           opcs are I type
    endcase
endfunction
wire [1:0] type;
// getting the opcode
assign opc = instruction[31:26];
// getting the type ( I,R,J)
assign type = Type(opc);
// assigning the register values (rs rt rd)
assign rs = (type == 2'b10 ? 5'b00000 : instruction
   [25:21]);
assign rt = (type == 2'b10 ? 5'b00000 : instruction
   [20:16]);
assign rd = (type == 2'b01 ? instruction[15:11] : 5'
   b00000);
    ...
```

Listing 5: Changing the register value to 0 after reset

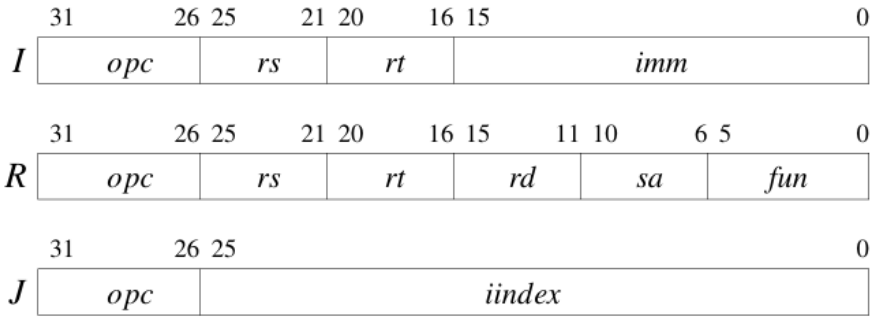

Figure 2: Instructions from 'System Architecture An Ordinary Engineering Discipline'

+0	+1	+2
00100000001000100000000001111111	00100000010000100000000001111111	00100000001000100000000001111111
00100000001000100000000001111111	00100000001000100000000001111111	00100000001000100000000001111111

Figure 3: program.mif 32 bit instructions

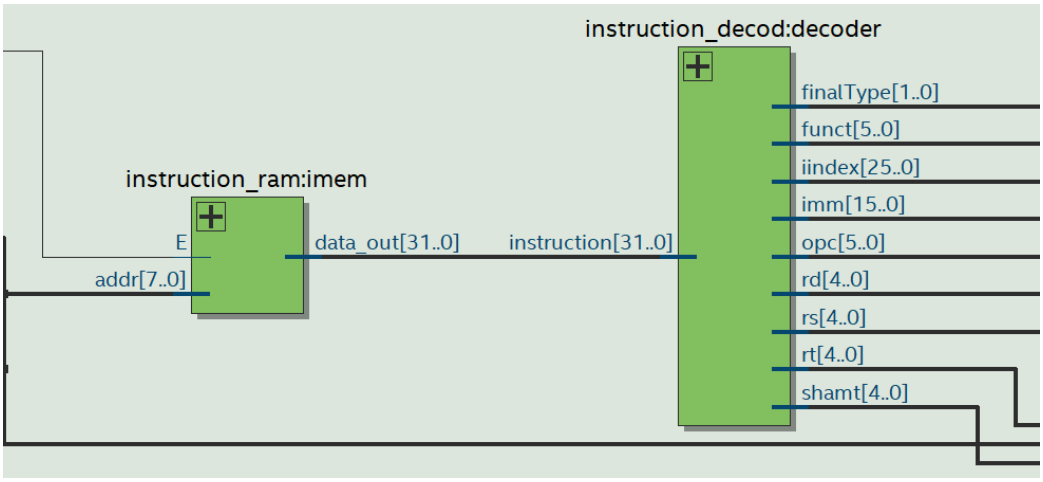

Figure 4: input - instruction[31:0] with the following outputs

Incrementor:

At each cycle $E = \neg E \quad \rightarrow pc = next\_pc$:

```
if(E) begin
    pc <= nextpc;
end
```

Listing 6: assigning the nexp_pc at each cycle

And the *next_pc* is being calculated in the execute phase:

```
if( type == 2'b00) begin
casez(opc)
    6'b001000:
        begin
            registers[rt] <= registers[rs] + imm;
            regvalue <= registers[rt];
            nextpc <= pc + 1;
        end
    ...
```

Listing 7: assigning the nexp_pc at each cycle

Counter:

Program counter plays a crucial role in the implementation. Pc is an input for instruction_ram that outputs the $instruction[31:0] \in B^{32}$:

```
    (* ram_init_file = "program.mif" *)
reg [31:0] mem [0:255];

always @(negedge E) begin
    data_out <= mem[addr];
 end

endmodule
```

Listing 8: assigning the nexp_pc at each cycle

Shift Register:

We need shift register for shifting operations:
$srl\ rd\ rs\ sa \longrightarrow rd = srl(rt, sa)$

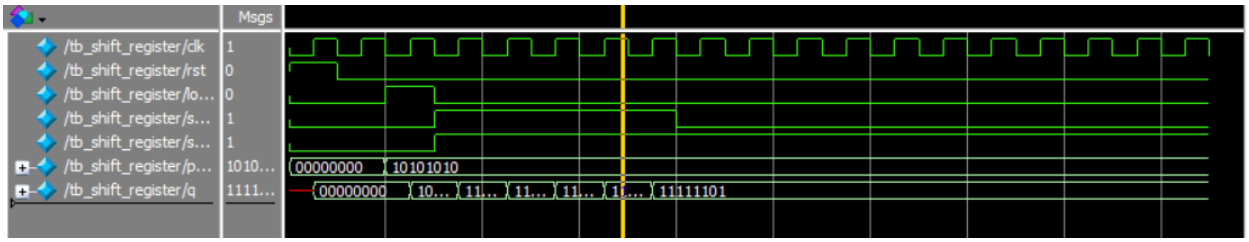

Figure 5: Rtl Simulatiron Waves

Debugging:

One of the primary debugging methods in digital system design is waveform analysis, which allows designers to visually inspect signal behavior over time. By simulating the design and viewing signal transitions (such as clock, control lines, register values, etc.) in tools like ModelSim, developers can verify correct operation, detect timing issues, and trace logical errors. Additional methods include using testbenches to apply controlled input scenarios and assertions to automatically detect violations of expected behavior during simulation.

Challenges Faced:

1. Adding Ram for instructions

2. Difficulty simulating Verilog code in Quartus due to missing ModelSim path

3. Verifying Correctness of the instruction execution

Solutions:

1. I used Google to learn how to create a 256-word program.mif file, which was then written manually to initialize the instruction memory in Quartus. This file stores the machine code for testing the processor's functionality.

2. Configured the correct ModelSim path in Quartus under EDA Tool Options

3. Added hex displays: h0,h1,h2,h3 To see the output of the instructions

Conclusion:

The project significantly strengthened my skills in hardware description languages, particularly Verilog, and deepened my understanding of digital system design. I gained practical experience in implementing a sequential MIPS processor based on the MIPS instruction set architecture, reinforcing key concepts such as instruction decoding, register manipulation, and control signal management. Additionally, I learned how to work with the Cyclone V GTX Starter Kit by Altera, including setting up the development environment, writing memory initialization files, and observing hardware behavior through waveform analysis and output displays. This hands-on experience was instrumental in bridging theoretical knowledge with real-world hardware implementation. See the full overview on Figure 6:

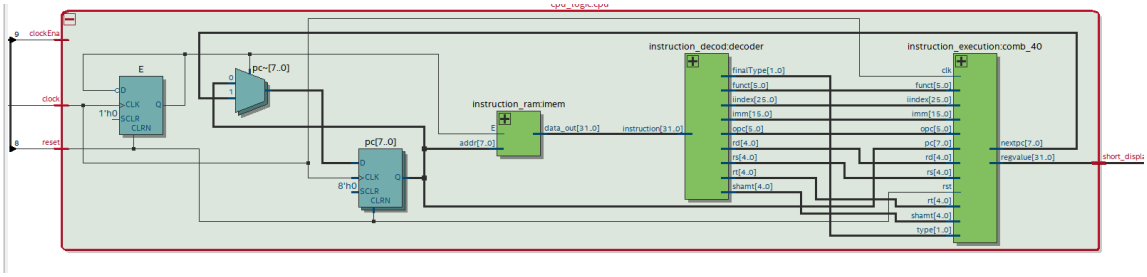

Figure 6: The full Overview of the project

References:

1. **Verilog_Manual**: Digital VLSI System Desgin ⟶ provided by Professor Ia Mosasvhili

2. **System Architecture An Ordinary Engineering Discipline** ⟶ provided by Professor Wolfgang Paul

3. **Verilog Tutorial**: https://www.chipverify.com/tutorials/verilog

4. **Google**
5. **ChatGpt**