THE AMERICAN UNIVERSITY IN CAIRO

الجامعة الأمريكية بالقاهرة

School of Sciences and Engineering
Department of Computer Science and Engineering
Fall 2024

CSCE 3301: Computer Architecture

Project 01: Pipelined CPU

Dr. Cherif Salama

Submitted By:

Farida Bey - 900212071
Arwa Abdelkarim - 900221451

Date: Nov 13, 2024

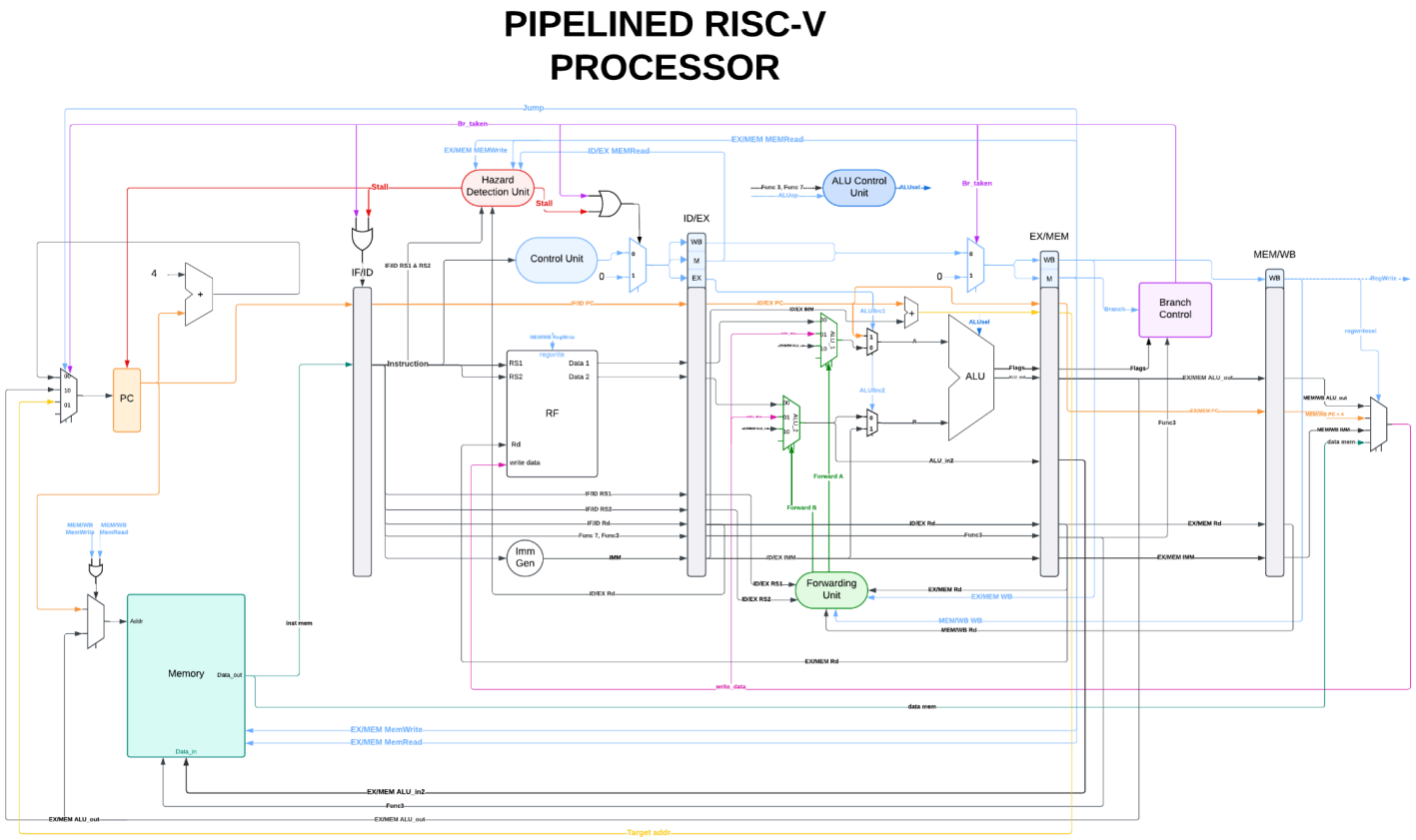
TABLE OF CONTENTS

GitHub repo link: https://github.com/arwaabdelkarim/RISC-V-Processor.git

Project Objectives:

The main aim of this milestone is to create a fully functional RV32I Pipelined processor with full hazard handling and a single memory which supports the 43 instructions including FENCE, ECALL and EBREAK where FENCE and EBREAK act as nops and ECALL acts as a halting instruction.

Design of the Datapath:



Implementation Overview:

- ALU_Control_Unit.v: Control unit for the ALU. This module uses ALUOp, func3, and parts of the instruction to determine the operation the ALU should execute.
- Branch_Control_Unit.v: Control unit for branching. This module evaluates the func3 field and ALU flags to determine whether a branch should be taken, based on the provided Branch_signal.
- Control_Unit.v: Control module that decodes the opcode to generate control signals for the processor. These signals (Branch, Jump, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc_1, ALUSrc_2, RegWrite, RegWriteSel) control the data flow and operations based on the instruction type, directing ALU operations, memory access, register writes, and program flow.
- The ForwardUnit handles data hazards in pipelined processors by forwarding data when needed. It compares source registers (ID_EX_Rs1, ID_EX_Rs2) with destination registers (EX_MEM_Rd, MEM_WB_Rd) in later stages. Based on matches and write-enable signals, it sets forwardA and forwardB to direct data from the correct pipeline stage.
- The hazard_detection module controls stalls to manage structural hazards caused by a single memory design. When EX_MEM_MemRead or EX_MEM_MemWrite is active, it sets stall to 1, pausing execution to prevent conflicts. This prevents data and memory access conflicts within the pipeline.
- The Imm_Gen module extracts and sign-extends the immediate value from a 32-bit instruction based on its opcode. It supports different instruction types like arithmetic, load/store, and branch by generating the appropriate 32-bit immediate. The module uses a case statement to handle various opcodes, such as `OPCODE_Arith_I`, `OPCODE_Store`, `OPCODE_LUI`, `OPCODE_AUIPC`, `OPCODE_JAL`, `OPCODE_JALR`, and `OPCODE_Branch`, to compute the immediate value for each.
- The Memory module handles memory read and write operations. It reads from memory based on MemRead and func3, supporting byte, half-word, and word access. For writes, it uses MemWrite and func3 to store data in memory. It manages a single memory unit to prevent structural hazards.
- The Reg_File module represents a 32-register file that supports read and write operations. It reads data from two registers and writes data to a destination register if it is enabled. On reset, all registers are set to zero.
- The ALU_32 module performs 32-bit arithmetic and logic operations (addition, subtraction, AND, OR, XOR, shifts, SLT, SLTU). It generates flags for carry, zero, sign, and overflow. The ALUsel input selects the operation, and ALU_A and ALU_` are the operand inputs. Shift operations use the `shamt` input, with the result output on ALU_out.

- DFF.v: A D flip-flop module with a clock (clk), reset (rst), and data input (D). It outputs a value (Q), which is updated on the rising edge of the clock, and reset to 0 when rst is high.

- Full_Adder.v: A full adder module that adds two 1-bit inputs (A, B) and a carry input (Cin). It outputs a sum and a carry-out (Cout).
- N_Bit_RCA.v: A ripple carry adder (RCA) that performs the addition of two N-bit numbers (A, B). It uses Full_Adder instances to compute the sum, carry-out, and overflow signals for the entire width.
- N_Bit_Register.v: A parameterized N-bit register that can store a value, triggered by a clock signal (clk). It supports a load signal (load) and resets the value using the reset (rst) signal. The module uses a 2-to-1 multiplexer and D flip-flops to store and update the register value.
- In addition to the described modules, we modified the N_Bit_Register and DFF modules to N_Bit_Reg_IF_ID and DFF_IF_ID, respectively. This modification was necessary to properly handle the reset behavior of the IF/ID pipeline register. Instead of simply setting the register values to zero, which was mistakenly interpreted as a valid instruction, we introduced the concept of a "NOP" (No Operation) instruction. By passing a NOP instruction during reset, we ensure that the reset state is appropriately handled, and the IF/ID pipeline register correctly reflects a valid idle state without being misread as an actual instruction.

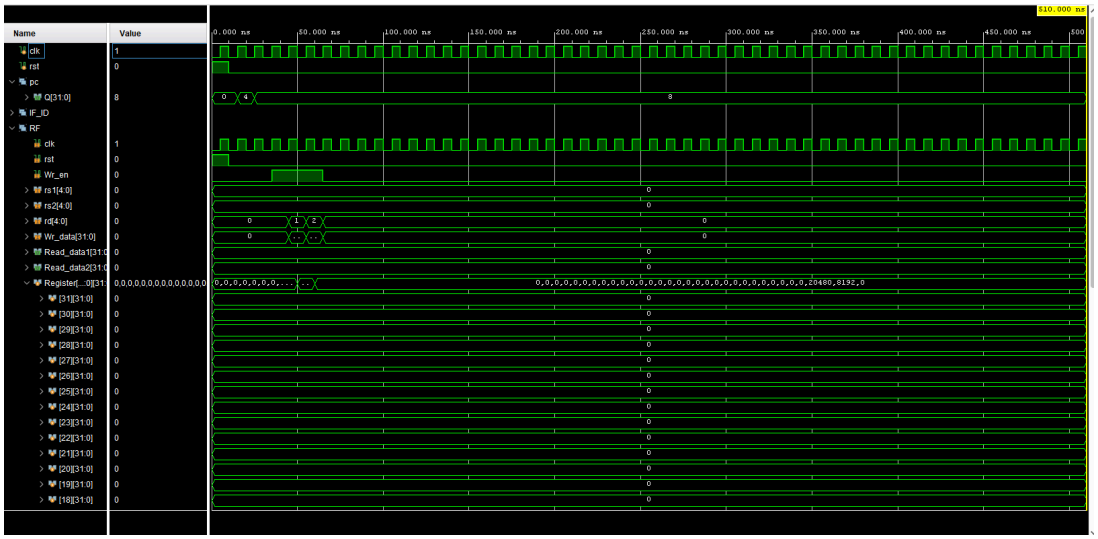
Bonus:

- We implemented a different approach for the single memory to handle the structural hazards. We stalled whenever we accessed the memory instead of changing the clock.
- We implemented the random test generator which writes the final test in a .hex file to then be called in our memory.

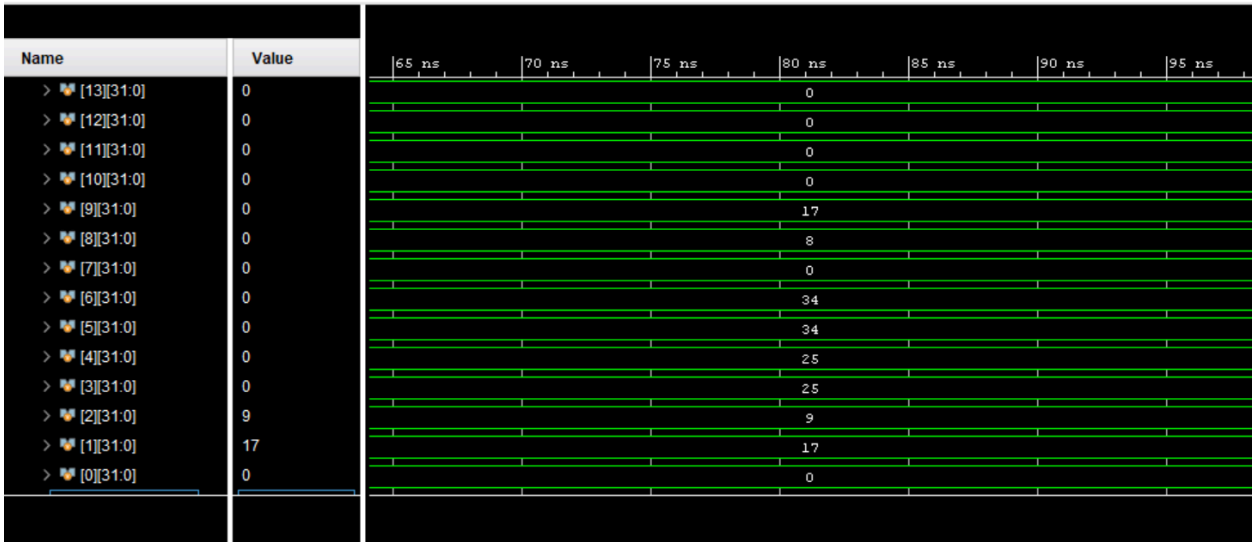
Test:

- Test 1:
  lui x1, 2 #x1 = 8192
  auipc x2, 5 #x2 = 20484
  jal x3, 8 # x3 = 12 ; PC => 16
  ecall #Halt PC

  {Mem[3], Mem[2], Mem[1], Mem[0]} = 32'b00000000000000000010000010110111;
  {Mem[7], Mem[6], Mem[5], Mem[4]} = 32'b00000000000000000101000100010111;
  {Mem[11], Mem[10], Mem[9], Mem[8]} = 32'b00000000000000000000000001110011;



- Test 2:
  (INST)
  000000000000_00000_010_00001_0000011 ; //lw x1, 0(x0)
  000000000100_00000_010_00010_0000011 ; //lw x2, 4(x0)
  000000001000_00000_010_00011_0000011 ; //lw x3, 8(x0)
  0000000_00010_00001_110_00100_0110011 ; //or x4, x1, x2
  0_000000_00011_00100_000_0100_0_1100011; //beq x4, x3, 4
  0000000_00010_00001_000_00011_0110011 ; //add x3, x1, x2
  0000000_00010_00011_000_00101_0110011 ; //add x5, x3, x2
  0000000_00101_00000_010_01100_0100011; //sw x5, 12(x0)
  000000001100_00000_010_00110_0000011 ; //lw x6, 12(x0)
  0000000_00001_00110_111_00111_0110011 ; //and x7, x6, x1
  0100000_00010_00001_000_01000_0110011 ; //sub x8, x1, x2
  0000000_00010_00001_000_00000_0110011 ; //add x0, x1, x2
  0000000_00001_00000_000_01001_0110011 ; //add x9, x0, x1
  (DATA)
  00000000000000000000000000010001
  00000000000000000000000000001001
  00000000000000000000000000011001



- Test 3:
  addi x1,x0,-32
  addi x2,x0,44
  addi x31,x0,2
  addi x25,x0,2
  add x2,x2,x1
  sub x2,x1,x2
  xor x3,x2,x1
  or x4,x3,x1
  and x5,x4,x1
  addi x10,x9,5
  xori x11, x10,5
  ori x12,x11,2
  andi x13,x12,2