CO502 - Part 3 (Integration)

Group 1

Table of Contents

Overview

This report presents the full integration of a 5-stage pipelined RISC-V processor. The final design consolidates the previously developed components—datapath, control unit, instruction/data memory with basic caching, and an assembler—into a cohesive, functional CPU. The design adheres to the canonical 5-stage RISC-V pipeline:

- **IF** – Instruction Fetch
- **ID** – Instruction Decode
- **EX** – Execute
- **MEM** – Memory Access
- **WB** – Write Back

Each pipeline stage is functionally separated by pipeline registers, enabling concurrent instruction execution and improved throughput.

Instruction and Data Flow Summary

The processor starts by fetching instructions from memory using the Program Counter (PC). The flow proceeds as follows:

- **IF Stage**: The PC fetches an instruction from instruction memory and increments to the next sequential address.
- **ID Stage**: The instruction is decoded; control signals are generated, and the register file is accessed. Immediate values are extended as needed.
- **EX Stage**: The ALU executes arithmetic or logical operations. Branch decisions are evaluated.
- **MEM Stage**: Load/store instructions interact with data memory. Memory addresses are computed in EX and used here.
- **WB Stage**: Results from ALU or memory are written back into the register file, completing the instruction.

The pipeline is designed for sequential execution of simple RISC-V instructions, with support for arithmetic, logic, load/store, and control flow.

Integrated Datapath

The integrated datapath implements a classic 5-stage pipelined RISC-V processor architecture, comprising Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB) stages. Each stage is modularized and connected using pipeline registers, enabling synchronized data flow between stages and allowing concurrent execution of multiple instructions. The following components and modules are instantiated and interconnected to form the complete datapath:

1. Instruction Fetch (IF)

This stage is responsible for generating the Program Counter (PC) and fetching the next instruction from instruction memory.

- Program Counter (pc.v): Holds the current instruction address. The PC is updated every clock cycle unless stalled by a BUSYWAIT signal.
- PC Adder (adder_32b_4.v): Increments the current PC by 4 to point to the next sequential instruction.
- PC Multiplexer (mux_32b_2to1.v): Selects the next PC value based on normal execution (PC + 4) or branch/jump targets computed in EX.
- IF/ID Pipeline Register: Holds the current PC and instruction to be passed to the decode stage, with stall logic on BUSYWAIT or branch flush conditions.

2. Instruction Decode (ID)

In this stage, the instruction is decoded, register values are read, and control signals are generated.

- Register File (reg_files.v): Provides two read ports for source operands (rs1, rs2) and a write port for write-back results. Operates synchronously with the clock.
- Immediate Generator (sign_extender.v): Extracts and sign-extends immediate values from the instruction based on its type (I-type, S-type, B-type, etc.), using the control signal IMM_SEL_ID.
- Control Unit (control_unit.v): Decodes opcode, funct3, and funct7 to generate control signals such as ALU operation (ALU_OP_ID), memory access signals, branch conditions, and multiplexing selectors for ALU operands and write-back sources.
- ID/EX Pipeline Register: Captures all necessary signals and data, including operand values, immediate, destination register address, and control signals, and forwards them to the execute stage.

3. Execute (EX)

This stage performs the actual computation using the ALU and resolves branch decisions.

- ALU Operand Multiplexers (mux_32b_2to1.v): Select between register value or PC for operand 1, and between register value or immediate for operand 2, based on control signals (DATA1_ALU_SEL_EX and DATA2_ALU_SEL_EX).
- Arithmetic Logic Unit (alu.v): Executes the specified operation using ALU_OP_EX, producing the result to be used in memory operations or write-back.
- Branch Unit (branch_logic.v): Evaluates branch or jump conditions using source operands and branch control signals (BRANCH_JUMP_EX), and asserts PC_MUX_SEL_EX if branching is required.
- EX/MEM Pipeline Register: Forwards the ALU result, control signals, and operands needed for the memory access stage.

4. Memory Access (MEM)

This stage interacts with data memory based on the instruction type.

- Memory Address (DMEM_ADDR_MA): Generated from the ALU output of the EX stage.
- Memory Write Data (DMEM_DATA_WRITE_MA): Carries the second register operand to be written into memory for store instructions.
- Control Signals (DMEM_READ_MA, DMEM_WRITE_MA): Drive memory read or write operations, depending on the instruction.
- PC + 4 Generator (adder_32b_4.v): Recomputed in this stage to be available for instructions that require it in write-back (e.g., JAL).
- MEM/WB Pipeline Register: Captures ALU output, memory read data, and control signals to be used in the final stage.

5. Write Back (WB)

The final stage determines what data is written back to the register file.

- Write-Back Multiplexer (mux_32b_3to1.v): Selects between ALU result, memory read data, or PC+4 (for link instructions) based on WB_SEL_WB.
- Register File Write: The selected value is written to the destination register if WRITE_EN_WB is asserted.

Pipeline Register Implementation

To maintain proper instruction flow and data consistency across the pipelined processor, each stage is decoupled using dedicated pipeline registers. These registers ensure that all necessary data and control signals are propagated forward at every clock cycle, while supporting stalling and flushing when necessary. Each pipeline register module is designed to be sensitive to the clk, rst, and busywait signals, providing robust synchronization and control.

1. IF/ID Pipeline Register (if_id_pipeline_reg.v)

This register separates the Instruction Fetch (IF) and Instruction Decode (ID) stages. It holds the Program Counter (PC) and fetched instruction (instr_in) until the decode stage is ready.

- On reset (rst), all outputs are cleared.
- If the busywait signal is high (e.g., due to memory stalls), the register holds its current state to prevent incorrect advancement of pipeline data.

2. ID/EX Pipeline Register (id_ex_pipeline_reg.v)

This module transfers decoded control signals, source operand values, the sign-extended immediate, and the destination register address to the Execute (EX) stage.

- Supports stalling using the busywait signal.
- Holds ALU operand selectors (data1_alu_sel, data2_alu_sel), ALU control (aluop), and write-back configuration.
- Includes branching control and memory read/write signals, ensuring EX stage has complete context.

3. EX/MEM Pipeline Register (ex_mem_pipeline_reg.v)

This pipeline register bridges the EX and MEM stages by storing the ALU result, operand for memory write, control signals, and destination register index.

- On rst, all stored values are cleared.
- Includes memory control (mem_read, mem_write) and write-back selectors.
- Ensures that results of computation and necessary metadata are correctly forwarded to the Memory Access stage.

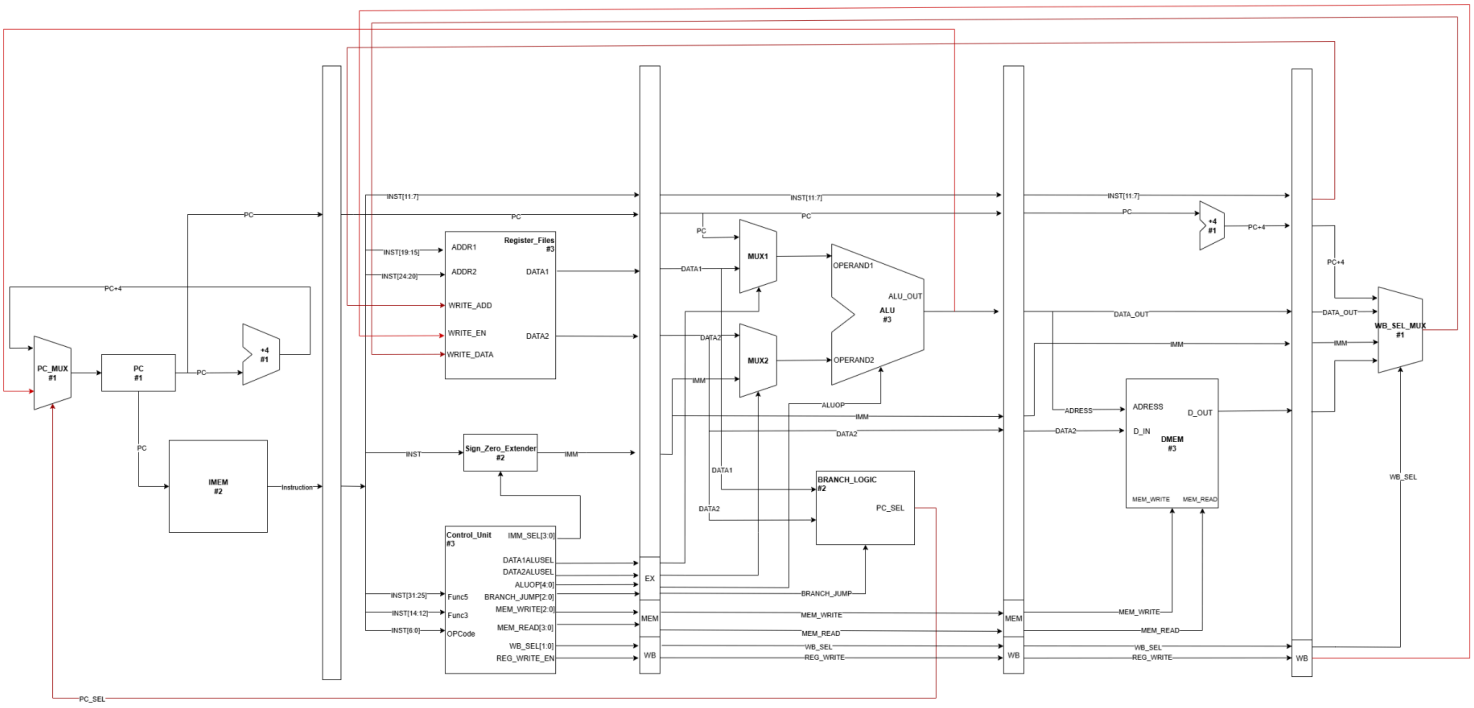
4. MEM/WB Pipeline Register (mem_wb_pipeline_reg.v)

This register transfers final results to be written back to the register file in the Write-Back stage.

- Stores the memory read data (mem_data_in), ALU result (alu_result_in), and PC (used in JAL/JALR-like instructions).
- Also carries the reg_write signal and wb_sel control to determine which result is written back.
- On reset or stall (busywait), it retains or clears its outputs as needed.

Each pipeline register is carefully synchronized and encapsulated to reflect a clean separation of pipeline stages, enabling high-performance instruction throughput and modular control. The

inclusion of busywait support in all registers allows the design to gracefully handle stalls, particularly during memory access latency or instruction hazards.



Updated Data Path

Control Logic

The control unit interprets instruction opcodes and generates control signals for all modules:

- **Main Control Unit**: Decodes opcodes to drive register write, ALU operation, memory access, etc.
- **ALU Control**: Uses funct3 and funct7 fields for detailed ALU selection.
- **Pipeline Signal Propagation**: Control signals are passed forward using pipeline registers alongside data.

Signals generated include:

- RegWrite, ALUSrc, MemWrite, MemRead, MemToReg, Branch, Jump

The separation of control logic per stage enhances modularity and debugging.

Instruction & Data Memory

The processor design incorporates two distinct memory components: Instruction Memory and Data Memory, each serving a specific role in the execution pipeline.

- Instruction Memory is a read-only module that holds the program's machine code. It is accessed during the Instruction Fetch (IF) stage using the current value of the Program Counter (PC). For every valid PC address, the instruction memory outputs a corresponding 32-bit instruction. As a ROM, it does not support dynamic modification during execution, ensuring the integrity of the loaded program throughout the simulation.
- Data Memory, in contrast, supports both read and write operations and is used during the Memory Access (MEM) stage. The memory address is derived from the ALU result generated in the Execute stage, representing either the load or store target. The actual memory operation—read or write—is determined by control signals MemRead and MemWrite, which are decoded from the instruction during the decode stage. For simulation purposes, the data memory is modeled behaviorally, providing controlled and observable memory interactions necessary for verifying the processor's correctness under test programs.

Assembler

A custom assembler was implemented to translate human-readable RISC-V instructions into binary format for memory initialization.

- Written in Python.
- Supports all major instruction types:
    - **R-type**: add, sub, and, or, sll
    - **I-type**: addi, lw, jalr
    - **S-type**: sw
    - **B-type**: beq, bne, blt, bge
    - **U-type**: lui, auipc
    - **J-type**: jal

Binary output is written to memory initialisation files for simulation. Label resolution, immediate parsing, and error checks are supported.

The assembler is capable of handling labels, immediate values, and register aliases (e.g., a0, x10, ra, etc.), making it easier for developers to write readable and structured code. It uses a two-pass approach:

1. First pass: Scans the input .s assembly file to collect label positions and count instructions.
2. Second pass: Converts each instruction into a 32-bit binary representation based on opcode, funct3/funct7, and operand parsing.

An instruction set definition is loaded from an external CSV file (RV32IM.csv), allowing the assembler to be easily extended or modified. Instruction types and encodings are extracted and used dynamically, making the assembler modular and scalable.

Key features include:

- Robust register mapping, supporting both x0-x31 and ABI names like t0, s1, a5, etc.
- Support for pseudoinstructions such as nop, ecall, and ebreak.
- Flexible immediate parsing, handling decimal, hexadecimal (0x), and binary (0b) formats.
- Accurate PC-relative address resolution for branch and jump instructions (beq, jal, etc.).
-
- Hexadecimal output generation written line-by-line into memory initialisation files for simulation

Padding support to ensure fixed-length memory files, e.g., 1024 lines by default.

Error checking is integrated at multiple stages—operand count verification, label resolution, register format validation, and immediate value checks—ensuring reliable and consistent output. The assembler outputs console logs mapping each instruction to its corresponding machine code, which is useful for debugging and educational insights.

Test Programs

The processor was validated using a range of test programs:

- **Arithmetic Tests**: Basic add, sub, and, or between registers.
- **Memory Tests**: Load (lw) and store (sw) with various offsets.
- **Branch Tests**: Conditional branches with known outcomes.
- **Jump Tests**: jal, jalr functionality and return address storage.
- **Combined Program**: A complete routine including computation, loops, and memory interaction.

Manual NOP instructions were inserted to resolve data hazards.

Timing Considerations

Each pipeline stage is assumed to complete in **1 clock cycle**, resulting in a **5-cycle latency per instruction** (excluding stalls).

- **Memory Access Delay**: Modelled as 1 cycle.
- **ALU Execution**: 1 cycle per operation.
- **Clock Frequency**: Relative simulation clock used; no real-time MHz target defined.
- **Pipeline Start-up**: Initial latency before steady throughput due to fill cycles.

No dynamic hazard resolution or cycle-level cache simulation was used.

Design Choices and Limitations

Design Decisions

- **Fully Modular Pipeline**: Clear separation between pipeline stages using dedicated registers.
- **Simplified Control Logic**: Opcode decoding drives ALU, memory, and register controls.
- **Basic Caching**: Modeled for instruction/data memory to simulate realistic behavior.
- **Custom Assembler**: Supports full instruction set used for testing.

Limitations

- **No Hazard Detection**: Manual NOPs inserted to prevent data and control hazards.
- **No Forwarding/Bypassing Logic**: True pipelining disabled without software support.
- **No Exceptions or Interrupts**: Not part of the implemented control logic.
- **No Multiply/Divide/Floating Point**: ALU limited to basic operations only.

Appendix: Key Code Snippets

CPU Testbench (Verilog)

```
module cpu_tb();

    // Inputs
    reg CLK;
    reg RST;

    // Outputs
    wire [31:0] PC, INST, DMEM_DATA_READ, DMEM_DATA_WRITE, DMEM_ADDR;
    wire [3:0] DMEM_READ;
    wire [2:0] DMEM_WRITE;
    wire BUSYWAIT;

    // Instantiate the CPU module
    cpu cpu_inst(
        .INST_IF(INST),
        .CLK(CLK),
        .RST(RST),
        .DMEM_DATA_READ_MA(DMEM_DATA_READ),
        .PC_IF(PC),
        .DMEM_ADDR_MA(DMEM_ADDR),
        .DMEM_DATA_WRITE_MA(DMEM_DATA_WRITE),
        .DMEM_READ_MA(DMEM_READ),
        .DMEM_WRITE_MA(DMEM_WRITE),
        .BUSYWAIT_IN(BUSYWAIT),
        .BUSYWAIT_OUT()
    );

    // Instantiate the instruction memory module
    imem imem_inst(
        .clk(CLK),
        .rst(RST),
        .pc(PC),
        .instr(INST)
    );

    // Instantiate the data memory module
    dmem dmem_inst(
        .clock(CLK),
        .reset(RST),
        .read(DMEM_READ),
        .write(DMEM_WRITE),
```

```
        .address(DMEM_ADDR),
        .writedata(DMEM_DATA_WRITE),
        .readdata(DMEM_DATA_READ),
        .busywait(BUSYWAIT)
    );

    // Clock generation
    always #5 CLK = ~CLK;

    // Testbench logic
    initial begin
        // Open a file for logging
        $dumpfile("cpu_tb.vcd");
        $dumpvars(0, cpu_tb);

        // Initialize inputs
        CLK = 0;
        RST = 1; // Assert reset

        // Wait for global reset
        #15;
        RST = 0; // Deassert reset

        // Run the simulation for a few clock cycles
        #2000;

        // End simulation
        $finish;
    end

endmodule
```

Instruction Encoding Function for R-Type Instructions (Python)

```
    def handle_r_type(self, opcode: str, operands: List[str]) -> str:
        """
        Encode R-type instructions (e.g., ADD, SUB).
        Format: funct7 (7 bits) | rs2 (5 bits) | rs1 (5 bits) | funct3 (3 bits) | rd (5 bits) | opcode (7
    bits)
        """
        if len(operands) != 3:
            raise ValueError(f"R-type instruction {opcode} requires exactly 3 operands")
```

```
    rd = self.parse_register(operands[0])
    rs1 = self.parse_register(operands[1])
    rs2 = self.parse_register(operands[2])
    inst_info = self.inst_data[opcode]

    # Construct binary string for the instruction
    instruction = (
        inst_info['funct7'] +          # funct7 (7 bits)
        self.to_binary(5, rs2) +       # rs2 (5 bits)
        self.to_binary(5, rs1) +       # rs1 (5 bits)
        inst_info['funct3'] +          # funct3 (3 bits)
        self.to_binary(5, rd) +        # rd (5 bits)
        inst_info['opcode']            # opcode (7 bits)
    )
    return instruction
```

Explanation:

- Converts human-readable assembly operands into bit fields.
- Uses instruction format specifications from a CSV file for opcode, funct3, and funct7 bits.
- Returns a 32-bit binary string representation of the instruction.

Immediate Parsing and Two's Complement Handling

```
def to_binary(self, num_digits: int, num: int) -> str:
    """
    Convert an integer (signed) into a fixed-width binary string.
    Uses two's complement representation for negative values.
    """
    if num < 0:
        num = (1 << num_digits) + num  # Two's complement conversion

    mask = (1 << num_digits) - 1
    return format(num & mask, f'0{num_digits}b')
```

Explanation:

- Ensures that signed immediate values fit properly into their fixed field sizes (e.g., 12 bits for I-type).

- Properly converts negative values using two's complement representation.
- Masks higher bits to avoid overflow.

Parsing Register Strings

```
def parse_register(self, reg_str: str) -> int:
    """
    Convert register name (e.g., 'a0', 'x10', 't2') to register number (0-31).
    Raises ValueError if invalid register.
    """
    reg_str = reg_str.strip().lower()

    # Handle xN format
    if reg_str.startswith('x') and reg_str[1:].isdigit():
        reg_num = int(reg_str[1:])
        if 0 <= reg_num <= 31:
            return reg_num

    # Handle ABI names
    if reg_str in self.registers:
        return self.registers[reg_str]

    raise ValueError(f"Invalid register: {reg_str}")
```

Explanation:

- Supports both numeric (`x0` to `x31`) and ABI-standard register names (`a0`, `t1`, `s0`, etc.).
- Enables flexible input in assembly source files.

Example of Assembler Label Resolution and PC-Relative Addressing

```
def handle_b_type(self, opcode: str, operands: List[str], pc: int) -> str:
    """
    Encode B-type branch instructions (e.g., BEQ, BNE) with PC-relative offset.
    operands: rs1, rs2, label
    """
    rs1 = self.parse_register(operands[0])
    rs2 = self.parse_register(operands[1])

    label = operands[2]
    if label in self.label_positions:
```

```
            target_pc = self.label_positions[label]
            imm = (target_pc - pc - 1) * 4  # PC-relative byte offset, instruction aligned
        else:
            imm = self.parse_immediate(label)

        inst_info = self.inst_data[opcode]
        imm_bin = self.to_binary(13, imm)  # 13-bit immediate including sign

        # Immediate bits mapped to instruction fields as per RISC-V spec
        instruction = (
            imm_bin[0] +              # imm[12]
            imm_bin[2:8] +             # imm[10:5]
            self.to_binary(5, rs2) +   # rs2
            self.to_binary(5, rs1) +   # rs1
            inst_info['funct3'] +      # funct3
            imm_bin[8:12] +            # imm[4:1]
            imm_bin[1] +               # imm[11]
            inst_info['opcode']        # opcode
        )
        return instruction
```

Explanation:

- Computes relative branch offsets based on label positions and current PC.
- Encodes immediate fields correctly according to RISC-V bit layout for B-type instructions.

Sample Assembly Code and Corresponding Hex Output

Assembly Instruction	Hexadecimal Machine Code
addi x1, x0, 1	00100093
addi x1, x0, 1	00100093
addi x2, x0, 1	00000113
addi x5, x0, 6	00600393
add x3, x1, x2	002081b3
lw x5, 0(x1)	0032a023

beq x7, x7, label	00038463
jal x0, label	fe5ff06f

Note: The above codes correspond to test program instructions verifying various instruction types including arithmetic, load, branch, and jump.