8-Bit 5-Stage Pipelined RISC-Style Processor with Hazard Handling

The processor features a 5-stage pipeline with a custom RISC style 14-bit instruction set architecture (ISA) with 16 instructions, and comprehensive hazard management including flushing for control hazards and forwarding/stalling for data hazards. This processor was designed using Verilog and implemented on a Basys 3 FPGA board.

Features:

- 5-stage pipelining (Fetch, Decode, Execute, Memory, Writeback).
- 8-bit data path.
- 14-bit custom RISC style ISA (16 instructions).
- 4 General Purpose Registers(GPRs).
- Control hazard handling (flushes, jumps and branches).
- Data hazard handling (forwarding, stall, custom path).
- Instruction cycle counter on 7-seg display

Block Diagram

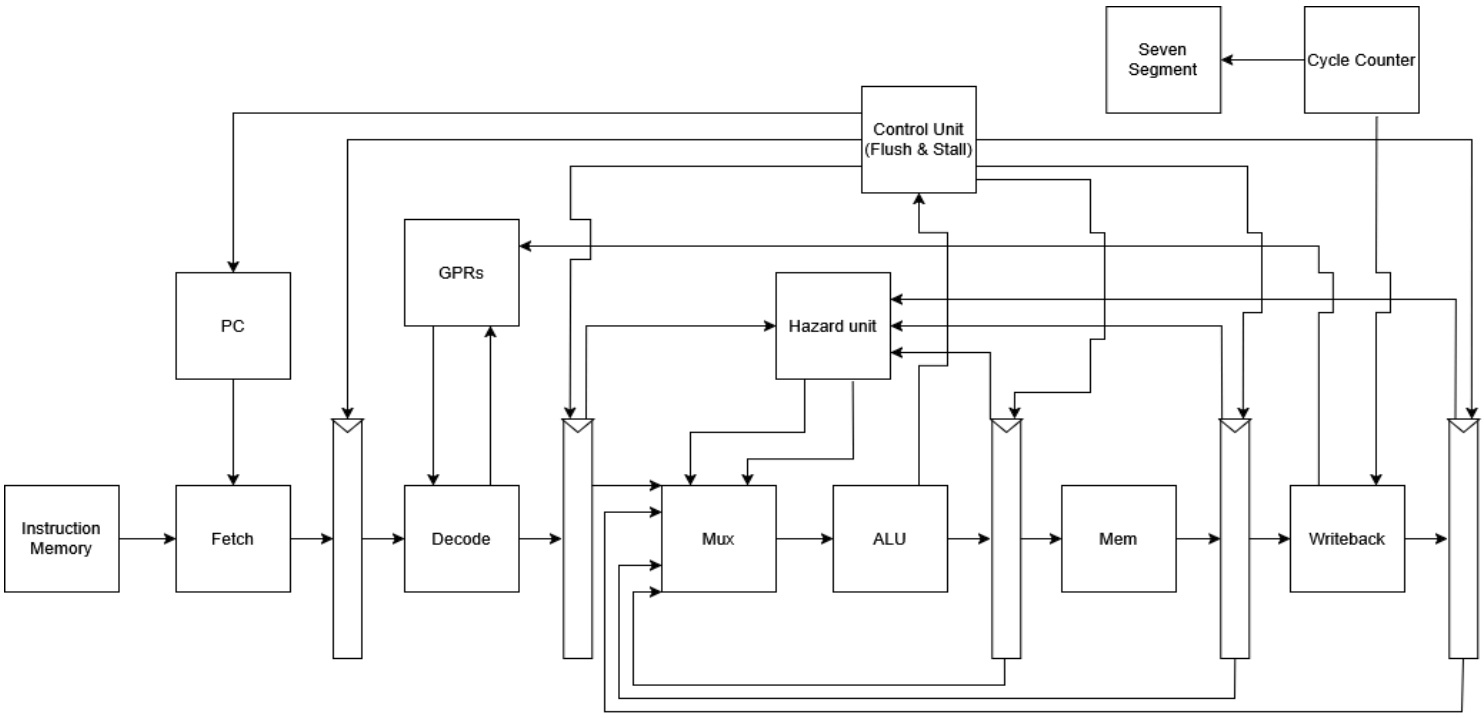

Fig 1. Processor Block Diagram

Custom Instruction Set Architecture (ISA)

The ISA used in this processor is similar to the RISC ISA but modified for this project scope. It supports fundamental arithmetic, memory, control and conditional operations. Initially, the ISA was 10 bits wide but the width was increased to 14 bits to accommodate immediate loads. With 4 GPRs, just two bits are used for addressing which leads to the default format.

Instruction Format:
{Opcode[13:10], Rd(Destination)[9:8], Rs1(Source 1)[7:6], Rs2(Source 2)[5:4], Reserved[3:0]}

- For register to register instructions (ADD, SUB, AND, OR, XOR, SLL, SRL), the default format is used with the reserved bits unused.
- For an immediate instruction (LDI), {Rs1, Rs2,Reserved} are concatenated to form the immediate 8 bit value to be operated on.
- For memory access instructions (LDM, ST):
    - In LDM, Rs1 indicates the register containing the base memory address and Reserved is the unsigned immediate offset with Rs2 unused.
      (GPR[Rd] = Memory[GPR[Rs1] + zeroextend(Reserved)]
    - In ST, Rs1 indicates the register containing the base memory address, Rs2 is the source register and Reserved is the unsigned immediate offset with Rd unused since there is no destination register.
      Memory[GPR[Rs1] + zeroextend(Reserved)] = GPR[Rs2]
- For control instruction(JMP), Rs1 indicates the register containing the target address with other fields unused.
- For conditional instructions(BEQ, BENQ), Rs1 and Rs2 are the register addresses for comparison with Reserved bits as offsets bits and GPR[2] as a reference register. Compare(Rs1, Rs2) then target address = GPR[2] + zeroextend(Reserved).

Table 1. ISA Details

Instruction	Opcode (Binary)	Description
NOP	0000	No operation
ADD Rd, Rs1, Rs2	0001	GPR[Rd] = GPR[Rs1] + GPR[Rs2]
SUB Rd, Rs1, Rs2	0010	GPR[Rd] = GPR[Rs1] - GPR[Rs2]
LDI Rd, 0x11	0011	GPR[Rd] = 0x11
AND Rd, Rs1, Rs2	0100	GPR[Rd] = GPR[Rs1] & GPR[Rs2]
OR Rd, Rs1, Rs2	0101	GPR[Rd] = GPR[Rs1] | GPR[Rs2]
XOR Rd, Rs1, Rs2	0110	GPR[Rd] = GPR[Rs1] ^ GPR[Rs2]
NOT Rd, Rs1	0111	GPR[Rd] = ~GPR[Rs1]
SLL Rd, Rs1, Rs2	1000	GPR[Rd] = GPR[Rs1] << GPR[Rs2]
SRL Rd, Rs1, Rs2	1001	GPR[Rd] = GPR[Rs1] >> GPR[Rs2]
LDM Rd, 0x1(Rs1)	1010	GPR[Rd] = Mem[GPR[Rs1] + zeroextend(0x1)]
ST Rs1, 0x1(Rs2)	1011	Mem[GPR[Rs2] + zeroextend(0x1)] = GPR[Rs1]
JMP Rs1	1100	PC = GPR[Rs1]
BEQ Rs1,Rs2,0x1	1101	if(Rs1 == Rs2), PC = zeroextend(0x1)
BENQ Rs1,Rs2,0x1	1110	if(Rs1 != Rs2), PC = zeroextend(0x1)
HLT	1111	Processor halt

Pipeline Architecture

The project follows a standard 5 stage pipeline processor design: Instruction Fetch(IF), Instruction Decode(ID), Execute (EX),Memory (MEM) and Writeback (WB).

- IF: Fetches the 14 bit instruction using the current PC from the instruction memory(ROM initiated with .COE file).
- ID: Decodes the instruction and acquires values from source GPRs.
- EX: ALU performs calculations for all operations when required.
- MEM: Data memory access for memory instructions implemented by Vivado IP catalog Read Access Memory(RAM).
- WB: Results are written back to the GPRs.

All 5 stages have pipeline registers(IF/ID, ID/EX, EX/MEM, MEM/WB) between them for adequate forwarding and stalling.

Custom WB_OUT pipeline register:
An additional pipeline register was added to temporarily store the results from the WB stage for one clock cycle. This was required to ensure accurate forwarding in simulated edge cases. Due to the GPRs having sequential writes(takes a clock cycle), there is a sequence of events when data is not available in the pipeline or in the GPR for one clock cycle. The custom WB_out register latches on to the data value which can be bypassed ensuring sequential execution without additional stalling.

Hazard Handling
The scope of this project was to primarily deal with two types of hazards: control and data hazards.

- Control Hazards (Jumps / Branches):
  These hazards are resolved after the EX stage based on the jmp and branch taken signals. Once activated, all operations in previous stages (IF, ID, ALU) are flushed effectively removing the wrong path chosen.
- Data Hazards (Read After Write):
  These Read After Write (RAW) hazards are handled by a dedicated hazard unit. This unit monitors the source register addresses and opcode from the ID stage and compares it to the destination registers in further stages checking for potential data dependencies based on the opcode. Through choosing the multiple Mux's select signal, the hazard unit can then adequately forward data values when needed. Additionally, it can stall parts of the processor to resolve specific load-use hazards.

Cycle Counter
For performance benchmarking, a processor cycle counter was implemented. This measures how many clock cycles it takes to execute a program ending with a HLT instruction. The counter is then displayed on the Basys 3 board's seven segment display for a simple Cycles per Instruction (CPI) calculation given the number of instructions.

Conclusion
Overall, the project was a rewarding practical learning experience about computer architecture. Even with its limited functionality, it involved difficult tradeoffs in ISA decisions, modularization of processor stages, simulation with testbenches, debugging with waveforms and verification on FPGA hardware. Future next steps could be creating a basic python assembler to convert assembly instructions into a .coe style format. Another potential project would be to use python scripting to automate the testing and validation covering a wider array of valid instructions and edge cases.