Faculty of Engineering and Technology
Electrical and Computer Engineering Department

ENCS4370 - Computer Architecture

Project #2

Prepared by:

Qasim Batrawi	ID:1220204	Section: 1
Taleed Hamadneh	ID:1220006	Section: 1
Laith Nimer	ID:1213046	Section: 2

Instructor: Ayman Hroub

Date: Jan.14.25

Abstract

In this project, we designed and verified a simple 16-bit pipelined RISC processor using Verilog. The processor includes a five-stage pipeline: fetch, decode, ALU, memory access, and write-back. It supports a custom instruction set architecture (ISA) with three types of instructions: R-type, I-type, and J-type. The design also includes eight 16-bit general-purpose registers, a 16-bit program counter, and separate instruction and data memories. This report explains the design and implementation of the processor's datapath and control path, covering details like control signals, truth tables, and Boolean equations. To verify the design, we built a testbench and ran custom binary programs to test various instructions. Through testing and performance analysis, we confirmed the processor's ability to execute instructions correctly and handle pipeline stalls efficiently.

Table of Contents

Table of Figures

List of Tables

Design and Implementation

Designing a successful processor requires careful planning of the RTL, particularly the datapath, before coding begins. Initially, we focused on the RTL design and created a single-cycle datapath that supported all required instruction formats (I-type, R-type, and J-type). Once this was complete, we introduced buffers between stages to facilitate pipelining. The design evolved step by step as we added pipeline stages, addressing data hazards through techniques such as forwarding and stalling. We then tackled control hazards to ensure correct handling of branches and jumps. After finalizing the datapath structure, we began implementing the Verilog code, developing each stage individually while ensuring all necessary components were incorporated. This process included coding the datapath elements, designing the control unit, and creating the clock module. In the end, we integrated all the stages into a datapath module.

RTL Design

R-type:

AND Rd , Rs , Rt

* Fetch Instruction: Instruction ← MEM [PC]
* Fetch Operands: data1 ← Reg (Rs) , data2 ← Reg (Rt)
* Execute Operation: ALU_Result ← data1 + data2
* Write ALU Result: Reg (Rd) ← ALU_Result
* Next PC Address: PC ← PC + 1

ADD Rd , Rs , Rt

* Fetch Instruction: Instruction ← MEM [PC]
* Fetch Operands: data1 ← Reg (Rs) , data2 ← Reg (Rt)
* Execute Operation: ALU_Result ← data1 & data2
* Write ALU Result: Reg (Rd) ← ALU_Result
* Next PC Address: PC ← PC + 1

SUB Rd , Rs , Rt

* Fetch Instruction: Instruction ← MEM [PC]
* Fetch Operands: data1 ← Reg (Rs) , data2 ← Reg (Rt)
* Execute Operation: ALU_Result ← data1 - data2
* Write ALU Result: Reg (Rd) ← ALU_Result
* Next PC Address: PC ← PC + 1

SLL Rd , Rs , Rt

* Fetch Instruction: Instruction ← MEM [PC]
* Fetch Operands: data1 ← Reg (Rs) , data2 ← Reg (Rt)
* Execute Operation: ALU_Result ← data1 << data2
* Write ALU Result: Reg (Rd) ← ALU_Result
* Next PC Address: PC ← PC + 1

SRL Rd , Rs , Rt

* Fetch Instruction: Instruction ← MEM [PC]
* Fetch Operands: data1 ← Reg (Rs) , data2 ← Reg (Rt)
* Execute Operation: ALU_Result ← data1 >> data2
* Write ALU Result: Reg (Rd) ← ALU_Result
* Next PC Address: PC ← PC + 1

I-Type:

ANDI Rt , Rs , Imm

* Fetch Instruction: Instruction ← MEM [PC]
* Fetch Operands: data ← Reg (Rs) , Immediate ← Extend (Imm)
* Execute Operation: ALU_Result ← data & Immediate
* Write ALU Result: Reg (Rt) ← ALU_Result
* Next PC Address: PC ← PC + 1

ADDI Rt , Rs , Imm

* Fetch Instruction: Instruction ← MEM [PC]
* Fetch Operands: data ← Reg (Rs) , Immediate ← Extend (Imm)
* Execute Operation: ALU_Result ← data + Immediate
* Write ALU Result: Reg (Rt) ← ALU_Result
* Next PC Address: PC ← PC + 1

LW Rt , Imm(Rs)

* Fetch Instruction: Instruction ← MEM [PC]
* Fetch Base Register: address ← Reg (Rs) , Immediate ← Extend (Imm)
* Calculate address: data ← MEM[address]
* Read Memory: ALU_Result ← data + Immediate
* Write Register Rt: Reg (Rd) ← data
* Next PC Address: PC ← PC + 1

SW Rt , Imm(Rs)

* Fetch Instruction: Instruction ← MEM [PC]
* Fetch Register: Base ← Reg (Rs) , Data ← Reg(Rt)
* Calculate address: Address ← address + Extend (imm)
* Write Memory: MEM [Address] ← data
* Next PC Address: PC ← PC + 1

BEQ Rs , Rt , Imm

* Fetch Instruction: Instruction ← MEM [PC]
* Fetch Operands: data1 ← Reg(Rs) , data2 ← Reg(Rt)
* Branch decision: If (data1 == data2) then PC ← PC + Sign_ext(imm) else PC ← PC + 1

BNE Rs , Rt , Imm

* Fetch Instruction: Instruction ← MEM [PC]
* Fetch Operands: data1 ← Reg(Rs) , data2 ← Reg(Rt)
* Branch decision: If (data1 != data2) then PC ← PC + Sign_ext(imm) else PC ← PC + 1

FOR Rs , Rt

* Fetch Instruction: Instruction ← MEM [PC]
* Fetch Operands: data1 ← Reg(Rs) , data2 ← Reg(Rt)
* For Decision: If (data2 != 0) then PC ← data1 else PC ← PC + 1
    * Decrement Rt : Reg (Rt) ← Reg (Rt) - 1

J-Type

JMP Offset

* Fetch Instruction: Instruction: ← MEM [PC]
* Calculate Target Address: Target ← { PC [15:9] , Offset }
* Jump: PC ← Target

CALL Offset

* Fetch Instruction: Instruction: ← MEM [PC]
* Calculate Target Address: Target ← { PC [15:9] , Offset }
* Save Return Address: Reg(RR) ← PC + 1
* Jump: PC ← Target

RET

* Fetch Instruction: Instruction ← MEM [PC]
* Return: PC ← Reg (RR)

DataPath implementation

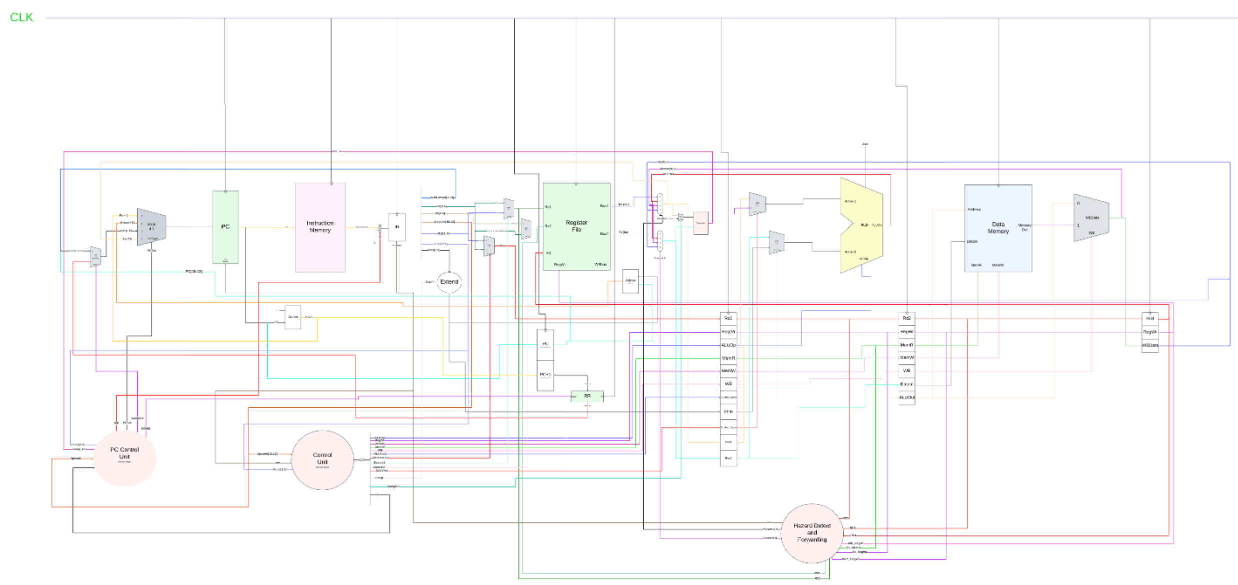


Figure 1 DataPath implementation

Datapath Overview

The datapath is the hardware block responsible for performing all operations required to execute instructions in a processor. For a pipelined processor, the datapath is divided into **five stages**, with each stage handling a specific part of the instruction execution. These stages are:

*** Instruction Fetch (IF)**
*** Instruction Decode (ID)**
*** Execute (EX)**
*** Memory Access (MEM)**
*** Write Back (WB)**

Each stage operates concurrently in a pipeline, with intermediate results stored in **pipeline registers** (buffers) to ensure proper data flow and timing.

Datapath Components

* **Mux 2-1 (decode)**

This multiplexer has two inputs and one output, and it selects between the inputs based on the **JumpSrc** control signal. The inputs are **{PC[15:9] || 9-bit offset}** for jump instructions and **RR** (Return Register), which stores **PC+1**, for return instructions. If **JumpSrc** is 0, the output is **{PC[15:9] || 9-bit offset}**, used during a jump. If **JumpSrc** is 1, the output is **RR**, which is selected during a return instruction.

*** MUX 4-1 (PC)**

This multiplexer has four inputs and one output, and it selects between the inputs based on the **PCSignal** control signal. The inputs are **PC+1**, **Branch TA**, **Jump TA**, and **For TA**. If the control signal is 0, the output is **PC+1**, which is used for normal sequential execution of the next instruction. If the control signal is 1, the output is **Branch TA** (calculated by the adder), used for branch instructions like **BEQ** or **BNE**. If the control signal is 2, the output is **Jump TA** (the result of a previously explained 2-to-1 multiplexer), used for instructions like **CALL** and **JMP**. Finally, if the control signal is 3, the output is **For TA**, which is used for **FOR** loop instructions.

* **Program Counter (PC)**

The Program Counter (PC) keeps track of the address of the current instruction to be executed. At the beginning of each clock cycle, the PC sends its value to the instruction memory to fetch the next instruction. At the end of the clock cycle, the PC is updated with either PC+1, a branch target address, a jump target address, or a For TA value, depending on the selection from the 4-to-1 multiplexer explained earlier. The output of the PC is connected to the instruction memory.

* Instruction Memory

The instruction memory stores all the program's instructions, and the PC provides the address to fetch the current instruction. The PC value is used as an address to retrieve the corresponding 16-bit instruction, which is then split into several parts: the **opcode [15:12]** determines the instruction type (R-type, I-type, J-type), **R1 [11:9]** is the source register 1, **R2 [8:6]** is the source register 2 or destination for I-type instructions, **R3 [5:3]** is the destination register for R-type instructions, **9-bit offset [11:3]** is used for address calculations, **Func [2:0]** specifies the operation for R-type instructions, and **immediate [5:0]** provides the immediate value for I-type or jump address for J-type instructions. The fetched instruction is then sent to the Instruction Register (IR).

* Instruction Register (IR)

The Instruction Register (IR) temporarily holds the fetched instruction for decoding and further processing. The instruction, which is fetched from memory, might sometimes be a "zero" instruction if a "kill" happens, depending on the output of the multiplexer. This ensures that the instruction remains stable as it moves through the fetch and decode stages of the pipeline. The IR then sends the instruction fields to be used in the decode stage.

* Mux 2-1 (destination)

This multiplexer has two inputs and one output, allowing selection between the two inputs based on the **Destination** control signal. The two inputs are either **Rd** or **Rt**. If the control signal is 0, the output will be **Rd**, which happens for R-type instructions. If the control signal is 1, the output will be **Rt**, which occurs for I-type instructions.

* **Mux 2-1 (source1)**

This multiplexer has two inputs and one output, allowing selection between the inputs based on the **Source1** control signal. The two inputs are either **Rs** from an R-type instruction or **Rs** from an I-type instruction. If the control signal is 0, the output will be **Rs** from the R-type instruction. If the control signal is 1, the output will be **Rs** from the I-type instruction.

* **Mux 2-1 (source2)**

This multiplexer has two inputs and one output, allowing selection between the inputs based on the **Source2** control signal. The two inputs are either **Rt** from an R-type instruction or **Rt** used as a source in **For**, **BEQ**, and **BNE** instructions. If the control signal is 0, the output will be **Rt** from the R-type instruction. If the control signal is 1, the output will be **Rt** used in **For**, **BEQ**, or **BNE** instructions.

*** Register File**

The register file holds 16 general-purpose registers used by instructions. It reads two registers (**Rs** and **Rt**) based on the control signals from the multiplexers, which are determined by the instruction fields. During the write-back stage, it writes results to a register (**Rd** or **Rt**). The **RegWr** control signal determines whether data is written to the registers or not. The inputs are the register addresses (**Rs**, **Rt**, and **Rd** or **Rt**) and the data to be written back (from the WB stage). The outputs are the data from the source registers (**Bus1** and **Bus2**), which are used by the ALU or for memory access.

*** MUX 4-1 (forward A)**

This multiplexer has four inputs and one output, allowing selection between the inputs based on the **forwardA** control signal. The four inputs are **Rs**, **ALUOut**, **MemoryOut**, and **WBOut**. If the control signal is 0, the output will be **Rs**, which happens for **For** or R-type instructions. If the control signal is 1, the output will be **ALUOut**, which occurs when there is an instruction with a 2-cycle stall. If the control signal is 2, the output will be **MemoryOut**, which happens for an instruction with a 1-cycle stall. If the control signal is 3, the output will be **WBOut**, which also happens for an instruction with a 1-cycle stall. All these conditions occur when forwarding happens on **Rs**.

*** MUX 4-1 (forward B)**

This multiplexer has four inputs and one output, allowing selection between the inputs based on the **forwardB** control signal. The four inputs are **Rt**, **ALUOut**, **MemoryOut**, and **WBOut**. If the control signal is 0, the output will be **Rt**, which happens for **For** or R-type instructions. If the control signal is 1, the output will be **ALUOut**, which occurs when there is an instruction with a 2-cycle stall. If the control signal is 2, the output will be **MemoryOut**, which happens for an instruction with a 1-cycle stall. If the control signal is 3, the output will be **WBOut**, which also occurs for an instruction with a 1-cycle stall. These conditions occur when forwarding happens on **Rt**.

*** Comparator**

The **Comparator** is used to compare two values, typically for branch instructions, to determine if a branch should be taken. It receives two inputs, often from registers or the ALU, and compares them based on the condition specified by the instruction (e.g., equality for **BEQ** or inequality for **BNE**). The comparator generates a result, such as a **Zero** flag or a branch condition signal, which is used to control the next instruction flow. The result from the comparator is then passed to the **PC control unit** to decide whether the PC should be updated with the branch target address or continue with the next sequential instruction. This component is essential for implementing conditional branching and controlling program flow in the processor.

* Mux 2-1 (ALUSrc1)

This multiplexer has two inputs and one output, allowing selection between the inputs based on the **ALUSrc1_signal** control signal. The two inputs are **Bus1** (from the register bus1 in the ID/E registers) or **1** (used to decrement 1 from **Rt** in the **For** instruction). If the control signal is 0, the output will be **Bus1**, which occurs for instructions like R-type, I-type, or For. If the control signal is 1, the output will be **1**, which happens specifically for the **For** instruction.

* Mux 2-1 (ALUSrc2)

This multiplexer has two inputs and one output, allowing selection between the inputs based on the **ALUSrc2_signal** control signal. The two inputs are **Bus2** (from the register bus2 in the ID/E registers) or **Imm** (from the immediate value in the ID/E registers). If the control signal is 0, the output will be **Bus2**, which happens for instructions like R-type, I-type, or For. If the control signal is 1, the output will be **Imm**, which occurs for **lw** and **sw** instructions.

* **Arithmetic Logic Unit (ALU)**

The ALU performs arithmetic operations like addition and subtraction, as well as logical operations like AND and OR. It has two inputs, ALUsrc1 and ALUsrc2, which are taken from the outputs of two multiplexers we explained earlier. The **ALUOp** control signal specifies the operation to perform. The ALU outputs the result of the operation (ALUOut) and a zeroFlag, which is used for making branch decisions. The operations performed include:
Add (for instructions like add, addi, lw, and sw), **Subtract** (for instructions like sub and BEQ), **AND** and **OR** (for logical operations like and and or).

* **Data Memory**

It stores and retrieves data for load and store instructions. It has two inputs: **Address** (from the **ALUOut** register) and **DataIn** (from **Bus2 →** [DataIn Register]). There are also two control signals: **MemR** (which allows reading from memory) and **MemW** (which allows writing to memory). The output is **MemoryOut**, which is the data fetched from memory when **MemR** is enabled.

* **Mux 2-1 (WBData)**

This multiplexer has two inputs and one output, allowing selection between the two inputs based on the **WB** control signal. The two inputs are either **MemoryOut** or **ALUOut** (from the **ALUOut** register in the M/WB registers). If the control signal is 0, the output will be **ALUOut**, which happens for instructions that are not **lw** or **sw**. If the control signal is 1, the output will be **MemoryOut**, which occurs for **lw** and **sw** instructions.

* Immediate Extension Unit

The extension unit increases the immediate value from 16 bits to 32 bits. It either **sign-extends** or **zero-extends** the immediate value based on the **ExOp** control signal. For example, in an **addi** instruction, it performs sign-extension, which fills the upper bits with the most significant bit. In zero-extension, it fills the upper bits with zeros. The output of the extension unit is then added either to the PC or to a register. For instructions like **lw** and **sw**, the immediate value is added to the base address for memory access. For branch instructions like **BNE** or **BEQ**, the immediate value is added to the PC using an adder.

* Control Unit

The control unit generates control signals based on the opcode of the instruction. It has two inputs: **Opcode[15:12]** and **Func[2:0]**, which help determine the type of instruction (R-type, J-type) based on the opcode. For R-type instructions, the opcode is 0000, and for J-type instructions, it's 0001. The control unit produces 13 control signals:

ALUOp signal: Specifies the ALU operation (e.g., add, subtract, or) based on the opcode and func inputs.
RegWr signal: Enables writing to the register file.
MemR signal: Enables reading from data memory.
MemW signal: Enables writing to data memory.
WB signal: Determines whether to store data from MemoryOut or ALUOut in the WB register, depending on whether the instruction is lw or sw.
ALUSrc1 signal: Selects the first input for the ALU, either from Bus1 or 1, based on the instruction opcode (e.g., For instruction or others).
Destination signal: Determines the value of the Rd register, either from R1[11:9] or R2[8:6], depending on whether the instruction is R-type or I-type.

Source1 signal: Selects the value of the Rs1 register from R1[11:9] or R2[8:6], based on whether the instruction is R-type or I-type.
Source2 signal: Selects the value of the Rs2 register from R3[5:3] or R2[8:6], based on the instruction type (R-type, branch, or For).
ALUSrc2 signal: Selects the second input for the ALU, either from Bus2 or the immediate value (Imm), depending on the instruction type (R-type) and whether forwarding is used.
ExOp signal: Determines whether the extension is signed or unsigned based on the instruction type (e.g., addi requires signed extension).
CompSrc signal: Selects the input for the comparator when the instruction is a branch or For loop.
J signal: Used as an input to the PC control unit, which decides the JumpSrc based on whether a jump instruction is present.
This set of control signals ensures that the processor correctly executes different instruction types by selecting the right data paths and operations.

*** Pipeline Buffers**

The pipeline is divided into different stages, and intermediate results are stored in buffers to prevent data from being overwritten. There are four groups of buffers: **IF/ID**, **ID/EX**, **EX/MEM**, and **MEM/WB**.
In the **IF/ID** buffer, the **PC register** and **PC+1 register** store the fetched instruction and the value of **PC + 4**. In the **ID/EX** buffer, there are 11 registers that store decoded values such as **Rd2 (Rd)**, **Bus1 (Rs)**, **Bus2 (Rt)**, and control signals like **Imm** (immediate), **RegWr**, **WB**, etc.
The **EX/MEM** buffer stores the ALU result in the **ALUOut** register, **DataIn (Bus2)**, and control signals like the **WB** signal in the **WB** register. Finally, in the **MEM/WB** buffer, the data read from memory is stored in the **WBdata** register, and the **Rd4 (Rd)** register is used for forwarding, along with the **RegWr** signal for writing back to the register file. These buffers help manage the flow of data through the pipeline, ensuring that results are correctly passed from one stage to the next.

* Hazard Detect and forwarding Unit

The **Hazard Detection and Forwarding Unit** is crucial for managing data and control hazards in the pipeline. It detects situations where data dependencies exist between instructions, such as when a later instruction needs data that is still being processed by a previous instruction. The unit can insert **stalls** to delay the execution of certain instructions, allowing the necessary data to become available. It also implements **forwarding** (or bypassing), which allows data to be passed directly from later stages of the pipeline (e.g., the EX/MEM or MEM/WB buffers) to earlier stages (e.g., the ID/EX or EX/MEM stages), avoiding unnecessary delays. The forwarding unit helps resolve read-after-write hazards by forwarding values from the pipeline registers instead of waiting for them to be written back to the register file. This unit ensures the processor operates efficiently by minimizing pipeline stalls and improving instruction throughput.

* PC Control Unit

The **PC Control Unit** is responsible for managing the program counter (PC) and determining the next instruction address during program execution. It takes input from various control signals, such as the **branch condition** (from the comparator), **jump instructions**, and **return instructions**, to decide whether the PC should simply increment to the next instruction (**PC + 1**), jump to a target address, or branch to a different part of the program. The unit selects the next PC value by using multiplexers, which are controlled by signals like **PCSrc** (indicating whether to branch or jump). For example, if a branch instruction like **BEQ** or **BNE** is executed, the PC is updated with the branch target address based on the comparator's result. Similarly, for jump instructions like **JMP**, the PC is updated with the jump target address. This unit ensures the correct flow of control in the pipeline, enabling conditional branching, function calls, and returns.

Assembly of Components:

The datapath is assembled by connecting all these components to support instruction execution. Here's how it all comes together: Instruction Fetch Stage (IF): The instruction is fetched from memory by the PC. The new value of the PC and the fetched instruction are buffered in IF/ID buffers. Instruction Decode Stage (ID): The instruction is decoded into its components, which include opcode, Rs, Rt, Rd, immediate, offset, function. Register values are fetched, ReadData1 and ReadData2. Immediate is zero-extended or sign-extended. Control signals are generated in this stage. All the outputs are saved into the ID/EX buffer. Execution Stage (EX): The ALU performs the operation of arithmetic, logical, and calculation of an address. Results are saved into the EX/MEM buffer. Memory Access Stage (MEM): Depending on the control signal, data is either read from or written into memory. Outputs are saved into the MEM/WB buffer. Write-Back Stage (WB): Results from memory or ALU are written back to the register file.

Control Units and control signals generation

Main Control unit

Insrtuction	opcode	Function	ALU Op	RegWr	MemR	MemW	WB	ALU Src1	Destination	Source1	Source2	ALU Src2	ExtOp	ComSrc	J
And	0000	000	000	1	0	0	1	0	0	0	0	0	X	X	X
Add	0000	001	001	1	0	0	1	0	0	0	0	0	X	X	X
Sub	0000	010	010	1	0	0	1	0	0	0	0	0	X	X	X
SLL	0000	011	011	1	0	0	1	0	0	0	0	0	X	X	X
SRL	0000	100	100	1	0	0	1	0	0	0	0	0	X	X	X
Andi	0010	NA	000	1	0	0	1	0	1	1	X	1	0	X	X
Addi	0011	NA	001	1	0	0	1	0	1	1	X	1	1	X	X
LW	0100	NA	001	1	1	0	1	0	1	1	X	1	1	X	X
SW	0101	NA	001	0	0	1	0	0	X	1	1	1	1	X	X
BEQ	0110	NA	X	0	0	0	0	0	X	1	1	X	1	0	X
BNE	0111	NA	X	0	0	0	0	0	X	1	1	X	1	0	X
For	1000	NA	101	1	0	0	1	1	1	1	1	0	X	1	X
Jmp	0001	000	XXX	0	0	0	0	X	X	X	X	X	X	X	1
Call	0001	001	XXX	0	0	0	0	X	X	X	X	X	X	X	1
Ret	0001	010	XXX	0	0	0	0	X	X	X	X	X	X	X	1

Table 1 Main control unit truth table

PC Control unit

Instruction	opcode	Func	J	comp_res	Kill	PCSrc	RRWE	JumpSrc
R type	0000	X	X	X	0	0	0	0
Andi	0010	NA	X	X	0	0	0	0
Addi	0011	NA	X	X	0	0	0	0
LW	0100	NA	X	X	0	0	0	0
SW	0101	NA	X	X	0	0	0	0
BEQ	0110	NA	X	0 if not taken 1 if taken	0 if not taken 1 if taken	1	0	0
BNE	0111	NA	X	1 if not taken 0 if taken	0 if not taken 1 if taken	1	0	0
For	1000	NA	X	1 if finish 0 if not finish	0 if finish 1 if not finish	3	0	0
Jmp	0001	000	1	X	1	2	0	0
Call	0001	001	1	X	1	2	1	0
Ret	0001	010	1	X	1	2	0	1

Table 2 PC control unit truth table

Boolean equations

ALUOp[2] = O[3] & ~O[2] & ~O[1] & ~O[0] & F[2]
ALUOp[1] = ~O[3] & ~O[2] & ~O[1] & ~O[0] & F[1] | O[3] & (O[2] | O[1])
ALUOp[0] = ~O[3] & ~O[2] & ~O[1] & ~O[0] & F[0]
RegWr = ~O[3] & ~O[2] & ~O[1] & ~O[0] | O[3] & ~O[2] & (F[2] | F[1] | F[0])
MemR = O[3] & ~O[2] & ~O[1] & ~O[0]
MemW = O[3] & ~O[2] & ~O[1] & O[0]
WB = ~O[3] & ~O[2] & ~O[1] & ~O[0] | O[3] & ~O[2] & ~O[1] & ~O[0]
ALUSrc1 = O[3] & ~O[2] & ~O[1] & ~O[0]
Destination = O[3] & ~O[2] & ~O[1] & ~O[0] | O[3] & O[2] & O[1] & ~O[0]
Source1 = O[3] & ~O[2] & ~O[1] & ~O[0] | ~O[3] & ~O[2] & ~O[1] & ~O[0]
Source2 = O[3] & ~O[2] & ~O[1] & ~O[0] | O[3] & O[2] & O[1] & ~O[0]
ALUSrc2 = ~O[3] & ~O[2] & ~O[1] & ~O[0] | O[3] & ~O[2] & ~O[1] & ~O[0]
ExOp = O[3] & O[2] & O[1] & ~O[0] | O[3] & O[2] & ~O[1] & ~O[0]
CompSrc = O[3] & O[2] & ~O[1] & ~O[0]
J = O[3] & O[2] & ~O[1] & ~O[0]
Kill = O[3] & ~O[2] & O[1] & ~O[0] | O[3] & O[2] & ~O[1] & ~O[0]
PCSrc = (O[3] & ~O[2] & ~O[1] & ~O[0]) ? 01: 00
RRWE = (O[3] & ~O[2] & ~O[1] & O[0]) & (F[2] | F[1] | F[0])
JumpSrc = O[3] & ~O[2] & ~O[1] & O[0] & F[2]

Simulation and Testing

First Program:

Alu instructions with data dependencies & Forwarding:

<u>The Register file content:</u>

```
initial begin
    registers[0] <= 16'h0000;
    registers[1] <= 16'h0001;
    registers[2] <= 16'h0002;
    registers[3] <= 16'h0003;
    registers[4] <= 16'h0004;
    registers[5] <= 16'h0005;
    registers[6] <= 16'h0006;
    registers[7] <= 16'h0007;
end
```

Figure 2 Register file content for program 1

<u>The Instruction Memory content:</u>

```
//Alu instructions with data dependencies & Forwarding

instMemory[0] = {4'b0000, 3'b001, 3'b010, 3'b011, 3'b001}; // R1 = R2 + R3 = 2 + 3 = 5
instMemory[1] = {4'b0000, 3'b111, 3'b001, 3'b011, 3'b001}; // R7 = R1 + R3 = 5 + 3 = 8
instMemory[2] = {4'b0000, 3'b110, 3'b001, 3'b101, 3'b010}; // R6 = R1 - R5 = 5 - 5 = 0
instMemory[3] = {4'b0000, 3'b101, 3'b001, 3'b010, 3'b010}; // R5 = R1 - R2 = 5 - 2 = 3
instMemory[4] = {4'b0000, 3'b100, 3'b001, 3'b010, 3'b000}; // R4 = R1 & R2 = 101 & 010 = 0
```

Figure 3 Instruction memory content for program 1

<u>Performance Registers output:</u>

```
run
# KERNEL: Total number of executed instructions: 5
# KERNEL: Total number of load instructions: 0
# KERNEL: Total number of store instructions: 0
# KERNEL: Total number of alu instructions: 5
# KERNEL: Total number of control instructions: 0
# KERNEL: Total number of stall cycles: 0
# KERNEL: Total number of cycles: 9
# RUNTIME: Info: RUNTIME_0068 DataPath.v (13): $finish called.
```

Figure 4 Performance Registers output for program 1

Waveform output:

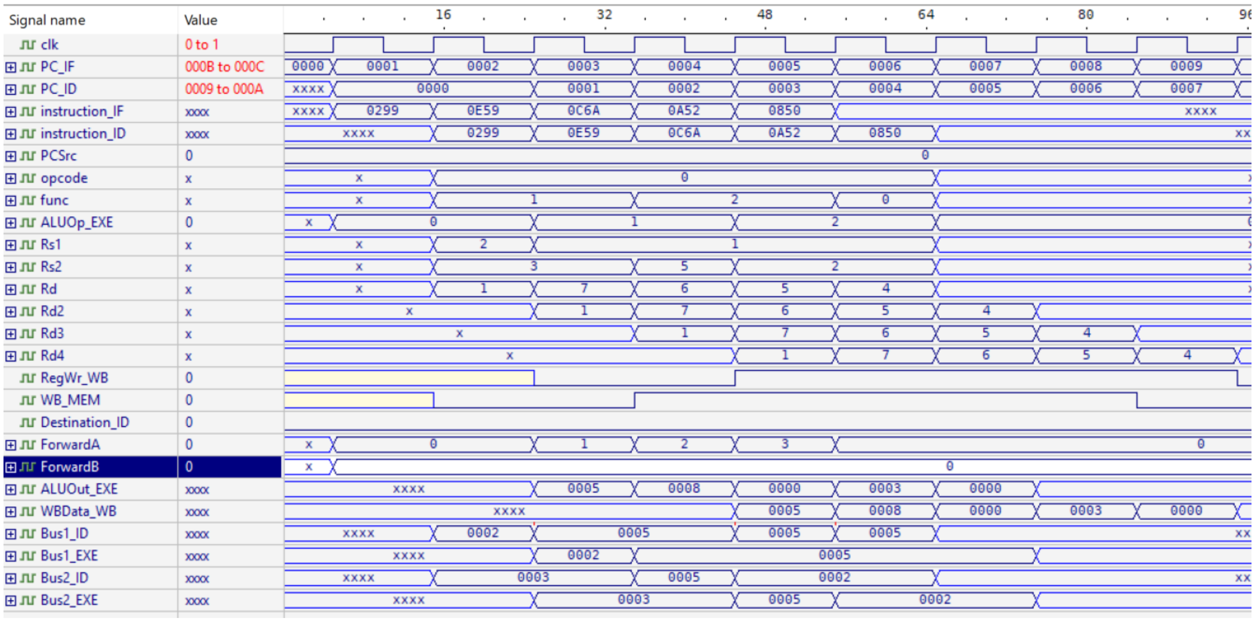


Figure 5 Waveform for program 1

Discussion:

The first instruction calculates **R1 = R2 + R3**, where R2 = 2 and R3 = 3, resulting in **R1 = 5**. There is no dependency yet since this is the first operation. The second instruction performs **R7 = R1 + R3**, where R1 has been updated to 5 from the first instruction and R3 is still 3. This instruction depends on the result of the first one (R1), so forwarding is used to pass the value of R1 directly from the **EXE** stage of the first instruction. The third instruction computes **R6 = R1 - R5**, where R1 is 5 (from the first instruction) and R5 is 5 (previously unaffected). This instruction also depends on the result of R1 from the first instruction, so forwarding is used again from the **MEM** stage of the first instruction. The fourth instruction calculates **R5 = R1 - R2**, where R1 is 5 (from the first instruction) and R2 is 2. As in the previous instructions, , so forwarding is used again from the **WB** stage of the first instruction. The fifth instruction performs **R4 = R1 & R2** without forwarding.

The waveform shows the Alu output in each stage: 5, 8, 0, 3, 0
Also, it shows the Forwarding Mux Selection (1 for EXE, 2 for MEM, 3 for WB).

Second Program:

For instruction with Data dependency between Load and Alu instruction.

The Register file content:

```
initial begin
    registers[0] <= 16'h0000;
    registers[1] <= 16'h0001;
    registers[2] <= 16'h0002;
    registers[3] <= 16'h0003;
    registers[4] <= 16'h0004;
    registers[5] <= 16'h0005;
    registers[6] <= 16'h0006;
    registers[7] <= 16'h0007;
end
```

Figure 6 Register file content for program 2

The Instruction Memory content:

```
// For instruction with data dependincies between load and alu instruction (1 stall cycle), (1 kill after for at each iteration except the last one)

        instMemory[0] = {4'b0000, 3'b101, 3'b100, 3'b000, 3'b001}; // R5 = R4 + R0 = 4 + 0 = 4
        instMemory[1] = {4'b0100, 3'b010, 3'b001, 3'b000, 3'b001}; // R1 = Mem[Rs=R2=2 + Imm=3] = Mem[5] = 5
        instMemory[2] = {4'b0000, 3'b111, 3'b001, 3'b100, 3'b001}; // R7 = R1 + R4 = 5 + 4 = 9
        instMemory[3] = {4'b1000, 3'b001, 3'b010, 3'b011, 3'b001}; // For: Rs=R1=1, Rt=R2=2
        instMemory[4] = {4'b0000, 3'b111, 3'b100, 3'b101, 3'b011}; // R7 = R4 - R5 = 4 - 5 = 1
```

Figure 7 Instruction memory content for program 2

Performance Registers output:

```
run
# KERNEL: Total number of executed instructions: 13
# KERNEL: Total number of load instructions: 3
# KERNEL: Total number of store instructions: 0
# KERNEL: Total number of alu instructions: 5
# KERNEL: Total number of control instructions: 0
# KERNEL: Total number of stall cycles: 3
# KERNEL: Total number of cycles: 22
# RUNTIME: Info: RUNTIME_0068 DataPath.v (53): $finish called.
```

Figure 8 Performance Registers output for program 2t:

Waveform output:

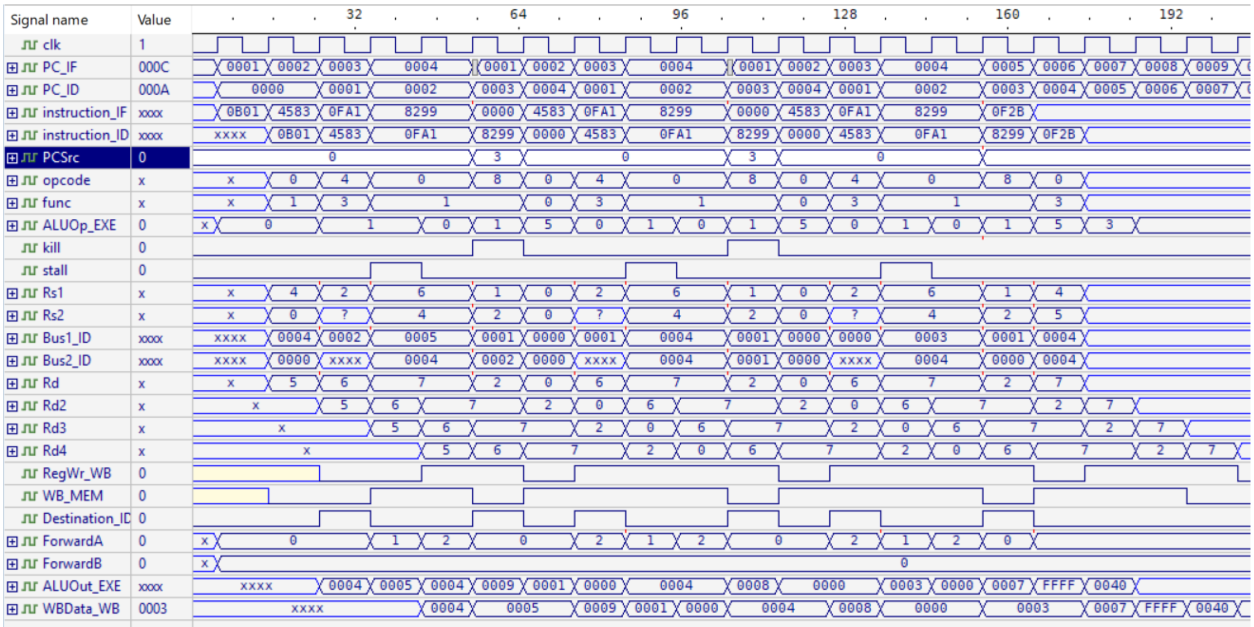


Figure 9 Waveform for program 2

Discussion:

In these instructions, data dependencies arise between the load and ALU operations, requiring the use of one stall cycle and one kill after each iteration except the last one. The first instruction performs an addition of R4 and R0, storing the result in R5. The second instruction loads data from memory into R6, which causes a dependency with the third instruction, which adds R6 and R4. Since R6 isn't available until the load completes, one stall cycle is introduced to wait for the load to finish. The fourth instruction checks R1 and R2 for branching, and if the condition is met, it loops back to the first instruction, effectively restarting the sequence. During this loop, one kill after each iteration (except the last one). Additionally, the Rt field has a counter of 2, which helps track the loop iterations.

The waveform above shows the kill and stall signals [3 stalls and 2 kills].

Third Program:

Control instructions

The Register file content:

```
initial begin
    registers[0] <= 16'h0000;
    registers[1] <= 16'h0001;
    registers[2] <= 16'h0002;
    registers[3] <= 16'h0003;
    registers[4] <= 16'h0004;
    registers[5] <= 16'h0005;
    registers[6] <= 16'h0006;
    registers[7] <= 16'h0007;
end
```

Figure 10 Register file content for program 3

The Instruction Memory content:

```
// Control instructions [Branch if equal and jump], containing one kill for the BEQ , and one or the jump , and store

    instMemory[0] = {4'b0110, 3'b001, 3'b001, 3'b000, 3'b010};  // BEQ: (taken) -> PC = 0 + 2 = 2
    instMemory[1] = {4'b0000, 3'b111, 3'b001, 3'b001, 3'b001};  // killed
    instMemory[2] = {4'b0000, 3'b001, 3'b010, 3'b110, 3'b011};  // R1 = R2 + R6 = 2 + 6 = 8
    instMemory[3] = {4'b0001, 3'b000, 3'b001, 3'b100, 3'b000};  // jump to address 12
    instMemory[2] = {4'b0000, 3'b001, 3'b010, 3'b110, 3'b011}; // killed
    instMemory[12] = {4'b0101, 3'b001, 3'b010, 3'b000, 3'b001}; // SW: Mem[Rs=R1=1 + Imm=1 = 2] = R2 = 2
```

Figure 11 Instruction memory content for program 3

Performance Registers output:

```
run
# KERNEL: Total number of executed instructions: 13
# KERNEL: Total number of load instructions: 0
# KERNEL: Total number of store instructions: 0
# KERNEL: Total number of alu instructions: 8
# KERNEL: Total number of control instructions: 5
# KERNEL: Total number of stall cycles: 0
# KERNEL: Total number of cycles: 17
# RUNTIME: Info: RUNTIME_0068 DataPath.v (53): $finish called.
```

Figure 12 Performance registers output of program 3

Waveform output:

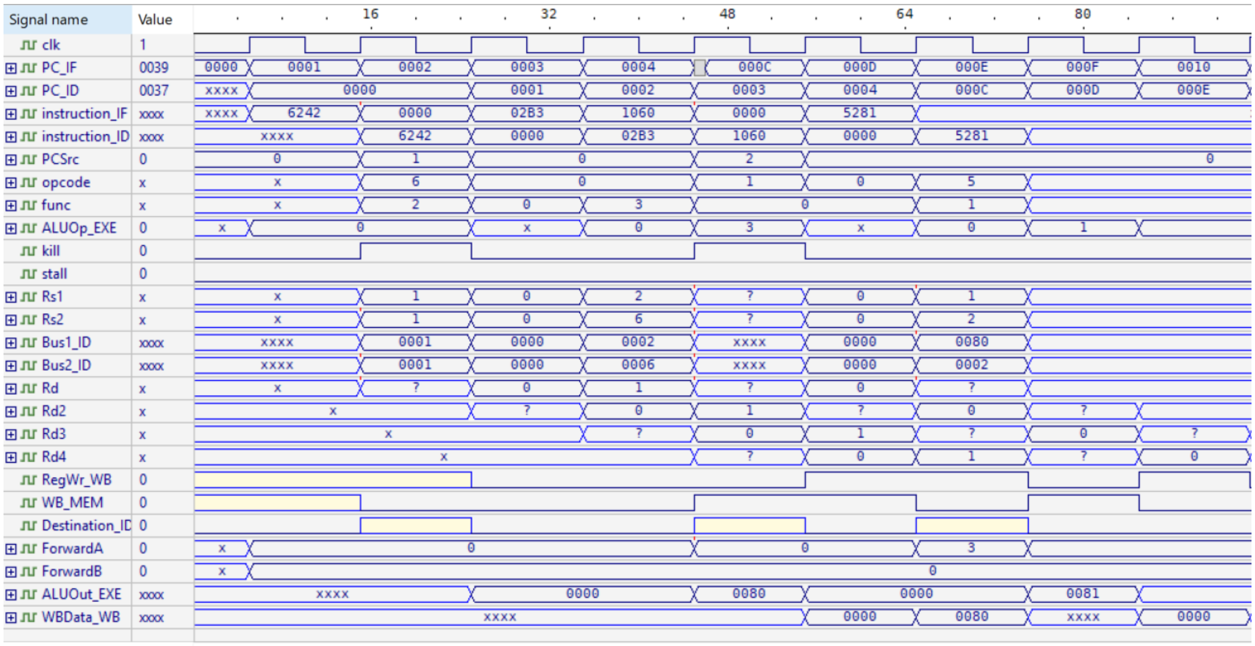


Figure 13 Waveform for program 3

Discussion:

In these instructions, control operations like Branch if Equal (BEQ) and Jump are used, which require handling with kills to ensure correct program flow. The first instruction is a BEQ that checks if R1 is equal to R2; if true, it takes the branch and updates the Program Counter (PC) to 2 (PC = 0 + 2). This instruction is followed by a kill for the next instruction. The third instruction performs an addition, R1 = R2 + R6, resulting in R1 = 8, but it's killed because of the branch. The fourth instruction is a Jump, which redirects the program flow to address 12. After the jump, the next instruction that performs a store operation SW stores the value of R2 at memory address 2 (Mem[Rs=R1 + Imm] = R2).

The waveform above shows the kill signal and outputs.

Fourth Program:

Call and return instructions

The Register file content:

```
initial begin
    registers[0] <= 16'h0000;
    registers[1] <= 16'h0001;
    registers[2] <= 16'h0002;
    registers[3] <= 16'h0003;
    registers[4] <= 16'h0004;
    registers[5] <= 16'h0005;
    registers[6] <= 16'h0006;
    registers[7] <= 16'h0007;
end
```

Figure 14 Register file content for program 4

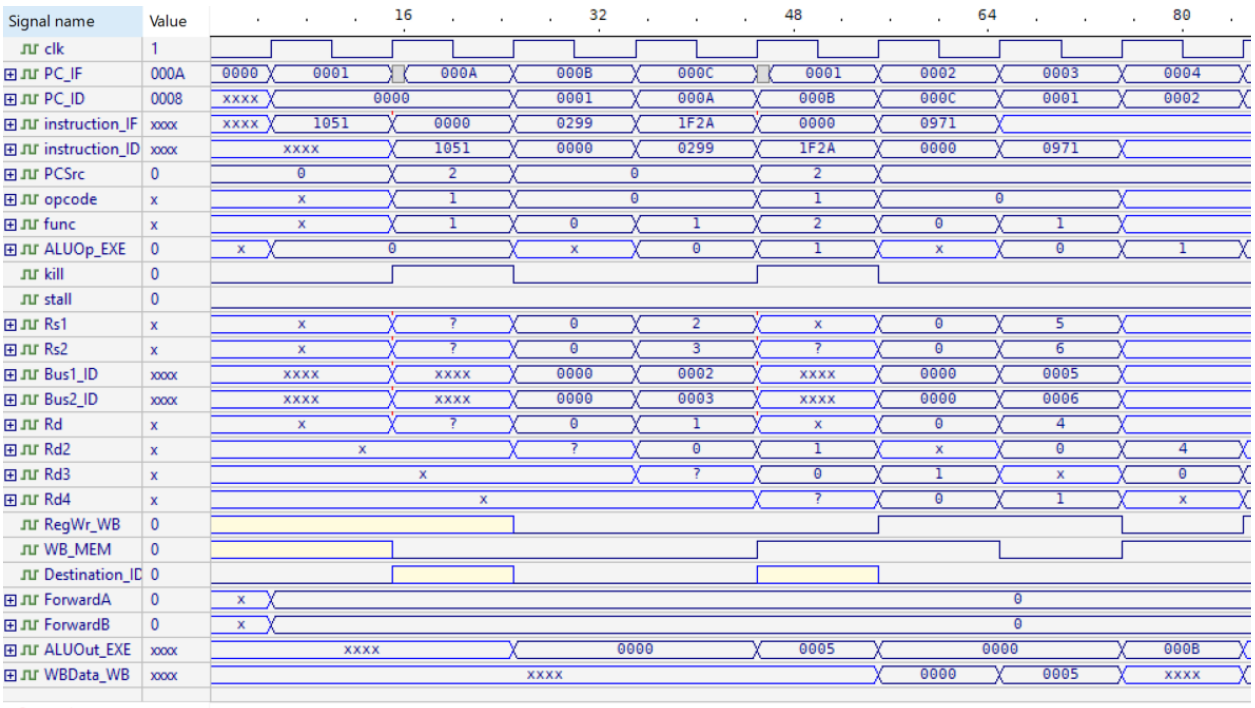
The Instruction Memory content:

```
// Call and Return Instructions with kill

    instMemory[0] = {4'b0001, 3'b000, 3'b001, 3'b010, 3'b001};  // jump to 10 and link
    instMemory[1] = {4'b0000, 3'b100, 3'b101, 3'b110, 3'b001};  // R4 = R5 + R6 = 5 + 6 = 11
    instMemory[10] = {4'b0000, 3'b001, 3'b010, 3'b011, 3'b001}; // R1 = R2 + R3 = 2 + 3 = 5
    instMemory[11] = {4'b0001, 3'b111, 3'b100, 3'b101, 3'b010}; // back to inst.1
    instMemory[12] = {4'b0000, 3'b011, 3'b100, 3'b101, 3'b010};
```

Figure 15 Instruction memory content for program 4

Waveform output:



Discussion:

In this set of instructions, we are using **Call and Return** operations, with kills to manage the flow of the program. The first instruction is a **jump to address 10 and link**, which means the program jumps to address 10 and saves the return address (address 1). The next instruction adds **R5 and R6**, and stores the result in **R4**. At address 10, the program adds **R2 and R3**, and stores the result in **R1**. After that, the program uses a **jump back to address 1**, which means it returns to where it left off. However, the instruction at address 1 is **killed** after the return.

Teamwork

In this project, we worked closely as a team, each contributing to different parts of the processor design and implementation. Laith was responsible for implementing the J-type instructions in the datapath, while Taleed handled the I-type instructions, and Qasim focused on the R-type instructions. We collaborated on implementing the forwarding units and buffers in Verilog. Taleed worked on the register file and data memory, Laith took charge of the instruction memory and ALU, and Qasim handled the control unit, along with other components such as multiplexers and the extender. Each of us also contributed to integrating the stages and combining our work into the final processor design. Our teamwork extended beyond just coding, as we also worked together on testing, simulation, and writing the report.

Conclusion

In this project, we built a simple 5-stage pipelined RISC processor using Verilog. The five stages of the processor are Fetch, Decode, Execute, Memory, and Write-back. We have used three different types of instruction set, namely R-type, I-type, and J-type, with 8 general-purpose registers and byte-addressable memory. The 5-stage pipeline allows the processor to work on multiple instructions at the same time, improving speed. We also handled various problems such as data and control hazards using different techniques, including forwarding and branching. This project helped us to understand how processors work and gave us hands-on experience in designing a functional and efficient processor.