Internship for SEMICONDUCTOR TECHNOLOGIES VIETNAM

Design and Implementation of 32bits RISC Processor

Instructor: Trung Nguyen-Duy
Student: Manh Tran-Duc

Contents

1 Overview

1.1 Introduction

MIPS (Microprocessor without Interlocked Pipeline Stages) processors are a family of RISC (Reduced Instruction Set Computer) architectures. The 32-bit MIPS processors are known for their simplicity and efficiency, making them a popular choice in both academic and commercial settings. Developed in the 1980s, MIPS processors have evolved over the years, maintaining their relevance due to their clean design and effective performance.

1.2 Application

32-bit MIPS processors are widely used in various applications, particularly in embedded systems. Some common applications include:

- **Networking Equipment:** : MIPS processors are commonly used in routers and switches due to their efficiency and reliability.
- **Consumer Electronics**: Devices like set-top boxes, digital TVs, and DVD players often incorporate MIPS processors.
- **Automotive Systems**: They are used in automotive control systems for their robustness and low power consumption.
- **Industrial Control**: Mips processor are found in industrial automation and control systems, where stability and efficiency are crucial.

1.3 MIPS Instruction

1.3.1 MIPS Instruction Formats

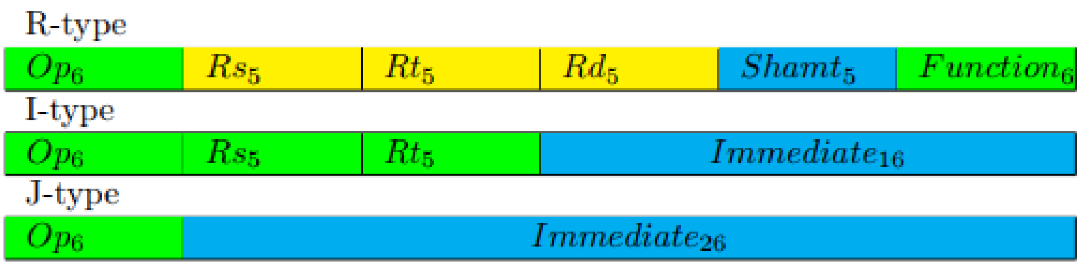

Figure 1: Mips Instruction Format

32 bits Mips include:

- **Op (opcode):** Instruction code, used to determine the execution instruction (for R-type, Op = 0).
- **Rs, Rt, Rd (register):** Identifies the registers (5-bit). For example, Rs = 4 means Rs is using register a0 or register 4.
- **Shamt (shift amount):** Specifies the number of bits to shift in shift instructions.
- **Immediate:** Represents a direct number, address, or offset.
- **Funct:** Specifies the exact operation to perform (e.g., add, subtract) in R-type instructions when Op = 0.

- **Op (opcode):** Instruction code, used to determine the execution instruction (for R-type, Op = 0).
- **Rs, Rt, Rd (register):** Identifies the registers (5-bit). For example, Rs = 4 means Rs is using register a0 or register 4.
- **Shamt (shift amount):** Specifies the number of bits to shift in shift instructions.
- **Immediate:** Represents a direct number, address, or offset.
- **Funct:** Specifies the exact operation to perform (e.g., add, subtract) in R-type instructions when Op = 0.

1.3.2 MIPS Instruction Execution Cycle (5 Stage)

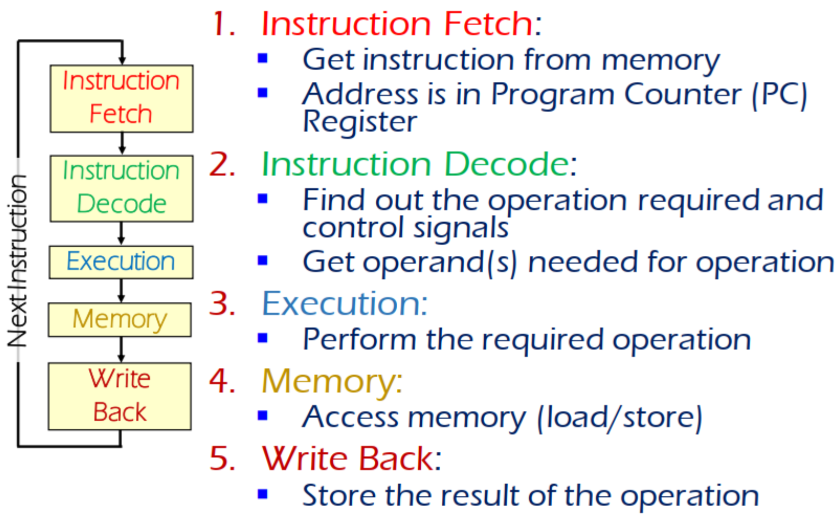

Figure 2: 5-Stage Pipeline

2 Architecture

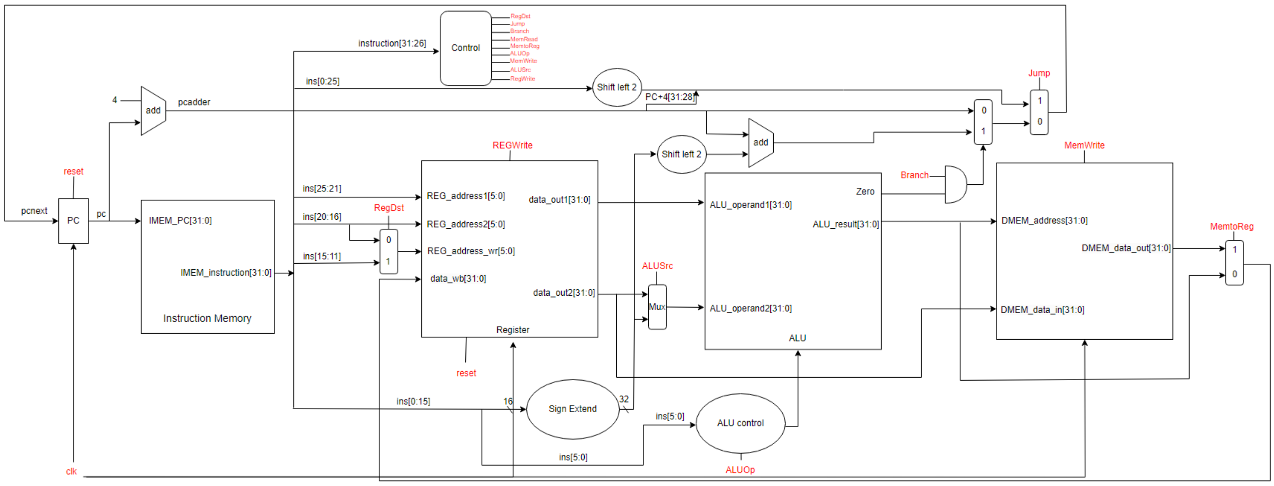

Figure 3: MIPS Single Clock Cycle Processor Architecture

The description of the modular design in the architecture:

- **Program Counter (PC)**: Points to the next instruction to be executed.
- **Instruction memory**: Contains the code, the program to be executed.
- **Register file**: Consists of 32 registers, therefore 5 bits are needed to identify the register. For detailed information on the registers.
- **Sign-extend**: Sign extension, extends a 16-bit signed number to a 32-bit signed number.
- **Multiplexer (MUX)**: Used to select the input for the corresponding output. The select signal determines the choice.
- **ALU**: Performs the necessary calculations.
- **Data memory**: This is the data memory area. Only the LOAD and STORE instructions can access the Data Memory.
- **Control**: The control unit, generates the control signals based on the Opcode of the instruction.

3 Functional implementation

- The functional implementation of the MIPS processor can be described as follows:

  1. The controller module decodes the instruction fetched from the external memory and generates the necessary control signals.
  2. The datapath module uses these control signals to perform the required operations, such as reading/writing registers, executing ALU operations, and accessing memory.
  3. The results of the datapath operations are then used to update the processor state (e.g., writing back to the register file, updating the PC) and/or interact with the external memory.

  The modular design of the MIPS processor allows for easier implementation, testing, and optimization. The controller and datapath modules can be developed and verified independently, and the external memory interface can be adapted to different memory architectures as needed.

4 Interface

The instruction and data memories are separated from the main processor and connected by address and data busses. This is more realistic, because most real processors have external memory. It also illustrates how the processor can communicate with the outside world.

The processor is composed of a datapath and a controller. The controller, in turn, is composed of the Control and the ALUControl. Figure 2 shows a block diagram of the single-cycle MIPS processor interfaced to external memories.

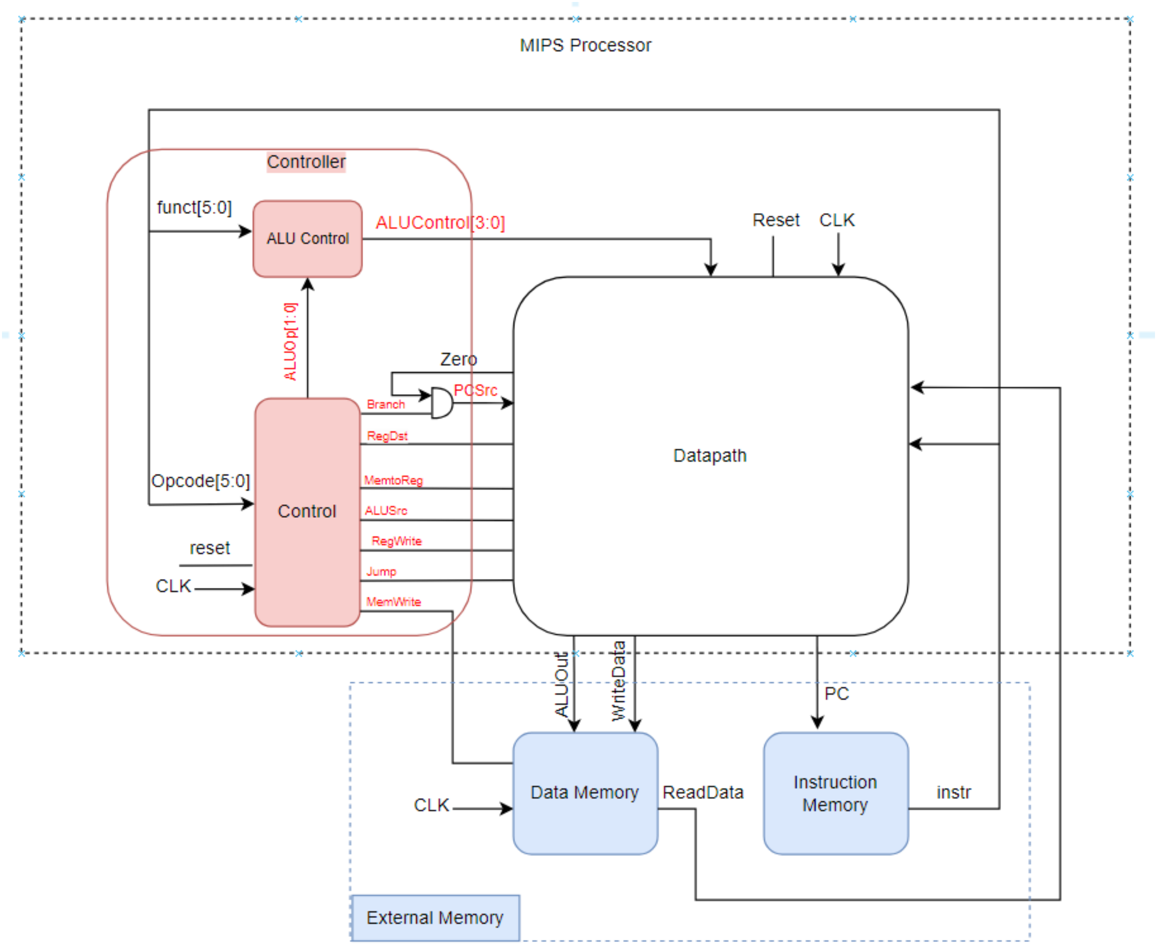

Figure 4: MIPS single-cycle processor interfaced to external memory

Description of signals in top:

Signal	Width	In/Out	Description
clk	1	Input	This is the clock signal for the module.
reset	1	Input	This typically sets the pc to a starting address, clears registers.
pc	32	Ouput	PC address, determining the address of the instruction to be fetched.
writedata	32	Ouput	The data that is to be written to the memory.
readdata	32	Ouput	Read the data from memory
aluout	32	Output	The result of the arithmetic or logical operation

4.1 Controller

4.1.1 ALUControl

The ALU Control is responsible for generating the appropriate control signals for the Arithmetic Logic Unit (ALU) based on the instruction being executed. Its main functions are:

- Decoding the ALU operation field from the instruction opcode.
- Generating the ALU control signals that specify the operation to be performed by the ALU (e.g., addition, subtraction, bitwise AND, bitwise OR).
- Handling special cases, such as determining the correct ALU operation for branch instructions.

The ALU Control unit takes the instruction opcode and the function field (for R-type instructions) as inputs, and outputs the ALU control signals that are used to configure the ALU's operation.

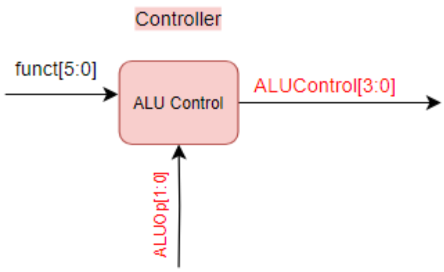

Figure 5: ALU Control module design

ALUOp meaning:

ALUOp	Meaning
00	add
01	subtract
10	Look at opcode bit
11	Look at funct fields

Table 1: ALUControl truth table:

Instruction	Opcode	ALUOp	Operation	Funct	ALU Function	ALU Control
lw	100011	00	load word	XXXXXX	add	0010
sw	101011	00	store word	XXXXXX	add	0010
beq	000100	01	branch equal	XXXXXX	subtract	0110
addi	001000	10	add immediate	XXXXXX	add	0010
andi	001100	10	AND immediate	XXXXXX	AND	0000
ori	001101	10	OR immediate	XXXXXX	OR	0001
slti	001010	10	set-on-less-than immediate	XXXXXX	subtract	0111
slti	001010	10	set-on-less-than immediate	XXXXXX	subtract	0111
R-type	000000	11	add	100000	add	0010
			subtract	100010	subtract	0110
			AND	100100	AND	0000
			OR	100101	OR	0001
			set-on-less-than	101010	set-on-less-than	0111

Description of signals in ALU Control:

Signal	Width	In/Out	Description
funct	6	In	function field from the instruction
opcode	6	Input	Instruction opcode
aluop	2	In	ALU operation control from the control unit
alucontrol	4	Out	Control signals to the ALU

4.1.2 Control

The Control unit is the main component of the controller that generates the control signals for the entire datapath based on the instruction being executed. Its key functions include:

- Decoding the instruction opcode to determine the type of instruction (e.g., R-type, I-type, branch, jump).
- Generating the control signals for the datapath components, such as:
  - Register file read/write signals
  - Memory read/write signals
  - ALU operation selection
  - PC update logic
- Handling control flow instructions (branches and jumps) by updating the Program Counter (PC) accordingly.

The Control unit takes the instruction opcode as input and generates a set of control signals that are used to control the operation of the datapath components throughout the instruction execution process.

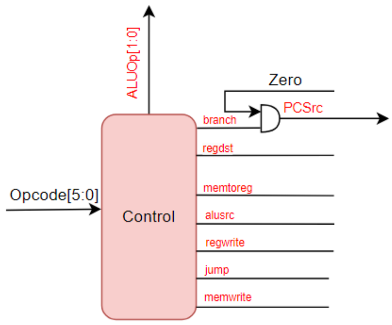

Figure 6: Control Unit module design

Table 2: Control Unit truth table

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
R-type	000000	1	1	0	0	0	0	11	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	0	1	0	1	0	00	0
beq	000100	0	0	0	1	0	0	01	0
addi	001000	1	0	1	0	0	0	10	0
andi	001100	1	0	1	0	0	0	10	0
ori	001101	1	0	1	0	0	0	10	0
slti	001010	1	0	1	0	0	0	10	0
j	000010	0	0	0	0	0	0	0	1

Description of signals in control unit:

Signal	Width	In/Out	Description
opcode	6	Input	Instruction opcode
branch	1	Output	Indicate branch operation
regdst	1	Output	Select destination register
memtoreg	1	Output	Select memory or ALU result for register file
alusrc	1	Output	Select ALU second operand source
regwrite	1	Output	Control register file write
jump	1	Output	Indicate jump operation
memwrite	1	Output	Control memory write operation
aluop	2	Output	Additional ALU control information

4.1.3 Controller

The controller (composed of the ALU Control and the Control unit) operates in sync with the datapath to ensure the correct execution of instructions. The typical sequence of operations is as follows:

- The instruction opcode is fetched from the Instruction Memory and passed to the controller.
- The ALU Control decodes the opcode and generates the appropriate ALU control signals.
- The Control unit decodes the opcode, generates the necessary control signals for the datapath, and coordinates the execution of the instruction.
- The control signals from the controller are used to configure the datapath components (e.g., register file, ALU, memory) to perform the required operations.
- The execution of the instruction is completed, and the controller prepares for the next instruction.

By understanding the functional design of the controller and its interaction with the datapath, you can gain a deeper understanding of the MIPS processor architecture and how it executes instructions.

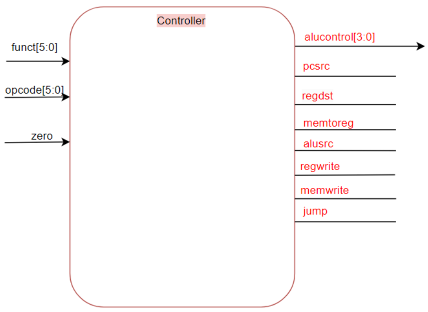

Figure 7: Controller Interface

Description of signals in Controller:

Signal	Width	In/Out	Description
funct	6	Input	function field from the instruction
op	6	Input	Instruction opcode
zero	1	Input	Comparison between 2 input
alucontrol	4	Output	Control signals to the ALU
pcsrc	1	Output	Select the next PC
regdst	1	Output	Select destination register
memtoreg	1	Output	Select memory or ALU result for register file
alusrc	1	Output	Select ALU second operand source
regwrite	1	Output	Control register file write
memwrite	1	Output	Control memory write operation
jump	1	Output	Indicate jump operation

4.2 Datapath

The datapath of the processor consists of four main components: the Program Counter (PC), the Arithmetic Logic Unit (ALU), the Register File, and the overall Datapath.

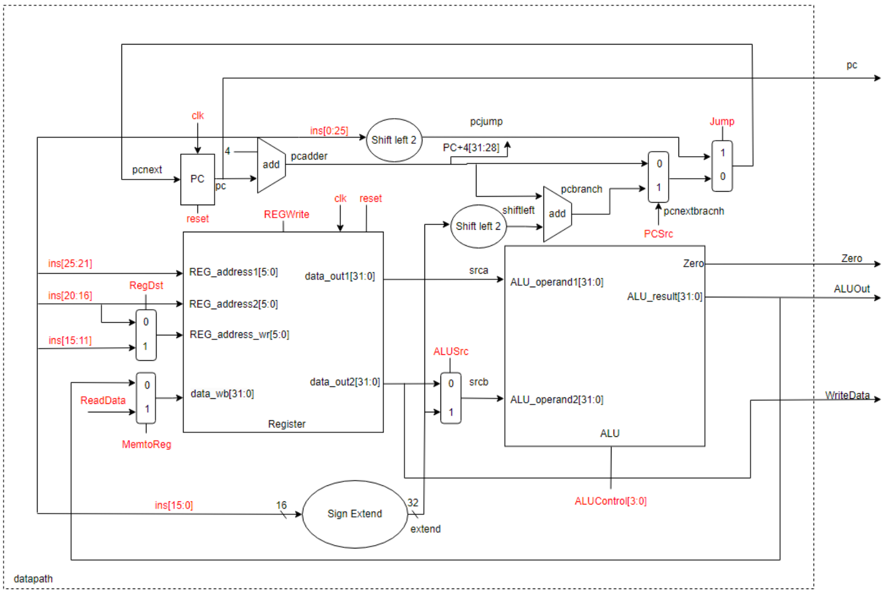

Figure 8: Datapath Block Diagram

4.2.1 PC module desgin

The Program Counter (PC) is a 32-bit register that holds the address of the current instruction being executed. The PC is updated during the instruction fetch stage based on the control signals from the controller. The PC can be updated in the following ways:

- Increment by 4 to fetch the next sequential instruction.
- Load a new value from the datapath for branch and jump instructions.
- Load a new value from the reset signal during processor initialization.

The PC provides the address to the Instruction Memory, which fetches the next instruction to be executed.

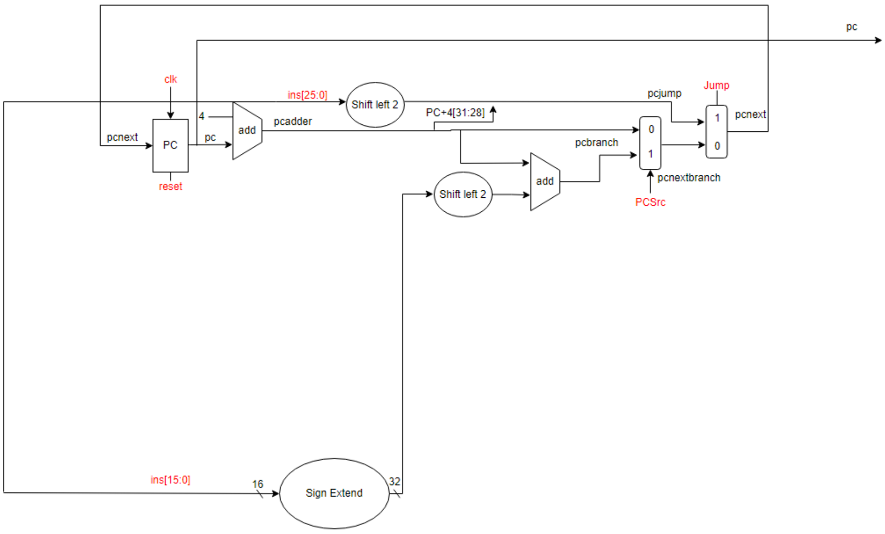

Figure 9: PC module Diagram

Description of signals in PC:

Signal	Width	In/Out	Description
clk	1	In	Clock signal
reset	1	In	Reset signal
jump	1	In	Jump control signal
pcsrc	1	In	Branch control signal
instr	26	In	Instruction address for jump
pc	32	Out	Current program counter

4.2.2 ALU module design

A simple ALU contains arithmetic and logical operations include:

- **Memory reference**: lw, sw
- **Arithmetic/logical**: add, sub, and, or, slt, nor, nand
- **Control transfer**: beq, j

The ALU takes two 32-bit operands and an operation control signal as inputs, and produces a 32-bit result. It also generates flags, such as zero, carry, and overflow, which are used by the controller for conditional branch instructions.

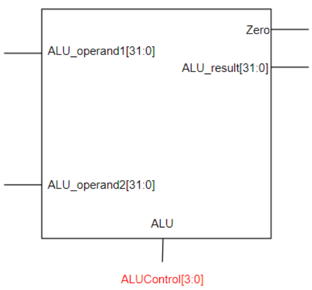

Figure 10: ALU module design

Description of signals in ALU:

Signal	Width	In/Out	Description
ALU_operand1	32	In	First input data
ALU_operand2	32	In	Second input data
Zero	1	Out	Comparison between 2 input
alucontrol	4	In	Select operation to perform
aluout	32	Out	Result after performing operations

4.2.3 Register module design

- The Register File is a collection of 32 general-purpose 32-bit registers. It provides two read ports and one write port to support the operand fetch and write-back stages of the instruction execution.
- The Register File is implemented using a 2-to-1 multiplexer for each read port, allowing the selection of the appropriate register based on the control signals. The write port is controlled by the control signals to update the destination register with the result of the instruction.

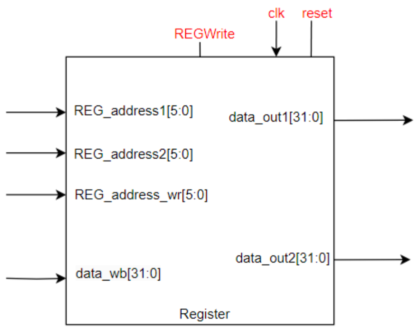

Figure 11: Register module design

Description of signals in Register:

Signal	Width	In/Out	Description
clk	1	In	This is the clock signal for the module.
reset	1	In	Clear registers
REG_address1	5	In	Address of register 1
REG_address2	5	In	Address of register 2
REG_address_wr	5	In	Address of the register to be written
regwrite	1	In	Enable write to the register
data_wb	32	In	Data to be written to the register
data_out1	32	Out	Value of register 1
data_out2	32	Out	Value of register 2

4.2.4 Datapath

- The Datapath is the overall structural component that connects the PC, ALU, and Register File, as well as other components, to facilitate the execution of instructions. The Datapath provides the necessary data paths, control signals, and communication channels to enable the processor to fetch, decode, execute, and write back the results of instructions.
- The Datapath works in conjunction with the Controller to coordinate the execution of instructions by providing the required operands, performing the necessary computations, and updating the relevant state elements (e.g., registers, memory) based on the control signals generated by the Controller.

By understanding the functionality of these datapath components, you can gain a better understanding of how the processor executes instructions and how the controller interacts with the datapath to achieve the desired execution flow.

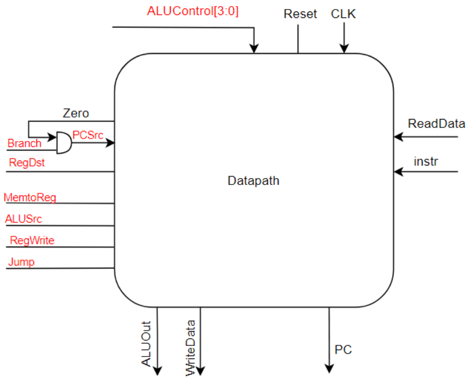

Figure 12: Datapath Interface

4.3 External Memory

4.3.1 Module Data Memory

- In computer architecture, the data memory is a component of the computer system responsible for storing and retrieving data.

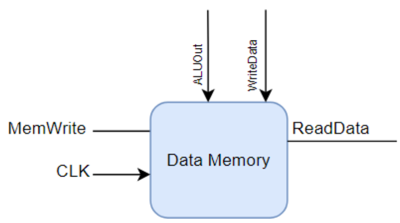

Figure 13: Data Memory module design

Signal	Width	In/Out	Description
memwrite	1	In	Write enable signal for memory
DMEM_address	32	In	Address of the memory location
DMEM_data_in	32	In	Input value to the memory
readdata	32	Out	Output value read from memory

4.3.2 Module Instruction Memory

- In computer architecture, instruction memory is a component of the computer system that stores the instructions of a program. In this module, a 32-bit instruction set is generated and stored in a RAM array. The instruction to be fetched is based on the input of the Program Counter (PC).

Signal	Width	In/Out	Description
pc	32	In	Program Counter address, determining the address of the instruction to be fetched.
instr	32	Out	Instruction fetched from the instruction memory.

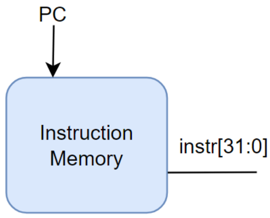

Figure 14: Instruction Memory module design

5 Design Verification

5.1 Controller Block

5.1.1 ALU_control Verification

- **Description:** The ALU_control_tb testbench is designed to comprehensively verify the functionality of the ALU control module. This module generates the appropriate control signals for the ALU based on the 6-bit function code (funct) and the 2-bit ALUOp input
- **Expected Result:** The testbench should validate that the ALU control module produces the correct output match with the output in truth table described in section 3.1.1 (see Figure 1).
- **Result:**

```
# Time: 10 | opcode: 100011 | funct: 000000 | aluop: 00 | alucontrol: 0010
# Test success: LW, SW
# Time: 20 | opcode: 000100 | funct: 000000 | aluop: 01 | alucontrol: 0110
# Test success: BEQ, BNE
# Time: 30 | opcode: 001000 | funct: 000000 | aluop: 10 | alucontrol: 0010
# Test success: ADDI
# Time: 40 | opcode: 001100 | funct: 000000 | aluop: 10 | alucontrol: 0000
# Test success: ANDI
# Time: 50 | opcode: 001101 | funct: 000000 | aluop: 10 | alucontrol: 0001
# Test success: ORI
# Time: 60 | opcode: 001010 | funct: 000000 | aluop: 10 | alucontrol: 0111
# Test success: SLTI
# Time: 70 | opcode: 000000 | funct: 100000 | aluop: 11 | alucontrol: 0010
# Test success: ADD
# Time: 80 | opcode: 000000 | funct: 100010 | aluop: 11 | alucontrol: 0110
# Test success: SUB
# Time: 90 | opcode: 000000 | funct: 100100 | aluop: 11 | alucontrol: 0000
# Test success: AND
# Time: 100 | opcode: 000000 | funct: 100101 | aluop: 11 | alucontrol: 0001
# Test success: OR
# Time: 110 | opcode: 000000 | funct: 101010 | aluop: 11 | alucontrol: 0111
# Test success: SLT
# All test cases passed
```

Figure 15: Output of ALU_control_tb

- **Exlaination:**
  - Case 1: LW, SW
  - Case 2: BEQ, BNE
  - Case 3: ADDI, ADD
  - Case 4: ORI, OR
  - Case 5: ANDI, AND
  - Case 6: SLTI, SLT

  The opcode (input) of different instructions such as R-type, lw, sw, beq, and jump generates outputs that match with the table in section 3.1.1 (see Figure 1).

5.1.2 Control Verification

- **Description:** The control_tb testbench is designed to thoroughly verify the functionality of the control module. This module is responsible for generating the various control signals (such as memwrite, branch, alusrc, regdst, regwrite, jump, etc.) based on the input opcode.
- **Expected Result:** The testbench should validate that the control module produces the correct output match with the output in truth table described in section 3.1.2 (see Figure 2).
- **Result:**

```
R-type: regwrite=1, regdst=1, alusrc=0, branch=0, memwrite=0, memtoreg=0, aluop=11, jump=0
lw: regwrite=1, regdst=0, alusrc=1, branch=0, memwrite=0, memtoreg=1, aluop=00, jump=0
sw: regwrite=0, regdst=0, alusrc=1, branch=0, memwrite=1, memtoreg=0, aluop=00, jump=0
reset: regwrite=0, regdst=0, alusrc=0, branch=0, memwrite=0, memtoreg=0, aluop=00, jump=0
beq: regwrite=0, regdst=0, alusrc=0, branch=1, memwrite=0, memtoreg=0, aluop=01, jump=0
addi: regwrite=1, regdst=0, alusrc=1, branch=0, memwrite=0, memtoreg=0, aluop=10, jump=0
ori: regwrite=1, regdst=0, alusrc=1, branch=0, memwrite=0, memtoreg=0, aluop=10, jump=0
slti: regwrite=1, regdst=0, alusrc=1, branch=0, memwrite=0, memtoreg=0, aluop=10, jump=0
jump: regwrite=0, regdst=0, alusrc=0, branch=0, memwrite=0, memtoreg=0, aluop=00, jump=1
default: regwrite=0, regdst=0, alusrc=0, branch=0, memwrite=0, memtoreg=0, aluop=00, jump=0
** Note: $finish    : D:/BACH KHOA/Internship/Single-Risc/control_tb.v(93)
   Time: 110 ps  Iteration: 0  Instance: /tb_control
1
Break in Module tb_control at D:/BACH KHOA/Internship/Single-Risc/control_tb.v line 93
```

Figure 16: Output of control_tb

- **Explaination:** Testing different instruction such as:
  - Case 1: R-type
  - Case 2: Load Word
  - Case 3: Store Word
  - Case 4: Reset
  - Case 5: BEQ
  - Case 6: ADDi
  - Case 7: Ori
  - Case 8: SLTI
  - Case 9: Jump

  The opcode (input) of different instructions such as R-type, lw, sw, beq, and jump generates outputs that match with the table in section 3.1.2 (see Figure 2).

5.1.3 Controller Verification

- **Description:** The controller_tb testbench is designed to thoroughly verify the combined functionality of the control module and the ALU control module. This testbench will check the output values when the two modules are integrated together, simulating the overall operation of the controller.

- **Expected Result:** The testbench should validate that the combined control and ALU control logic produces the expected output control signals for a comprehensive set of input conditions. This includes verifying the correct operation for common instruction types (e.g., R-type, load, store, branch, jump)

- **Result:**

```
# Time: 20 | R-type ADD (opcode=000000, funct=100000): memtoreg=0, memwrite=0, pcsrc=0, alusrc=0, regdst=1, regwrite=1, jump=0, alucontrol=0010
# Time: 30 | R-type SUB (opcode=000000, funct=100010): memtoreg=0, memwrite=0, pcsrc=0, alusrc=0, regdst=1, regwrite=1, jump=0, alucontrol=0110
# Time: 40 | R-type AND (opcode=000000, funct=100100): memtoreg=0, memwrite=0, pcsrc=0, alusrc=0, regdst=1, regwrite=1, jump=0, alucontrol=0000
# Time: 50 | R-type OR (opcode=000000, funct=100101): memtoreg=0, memwrite=0, pcsrc=0, alusrc=0, regdst=1, regwrite=1, jump=0, alucontrol=0001
# Time: 60 | R-type SLT (opcode=000000, funct=101010): memtoreg=0, memwrite=0, pcsrc=0, alusrc=0, regdst=1, regwrite=1, jump=0, alucontrol=0111
# Time: 70 | LW (opcode=100011): memtoreg=1, memwrite=0, pcsrc=0, alusrc=1, regdst=0, regwrite=1, jump=0, alucontrol=0010
# Time: 80 | SW (opcode=101011): memtoreg=0, memwrite=1, pcsrc=0, alusrc=1, regdst=0, regwrite=0, jump=0, alucontrol=0010
# Time: 90 | BEQ (opcode=000100, zero=1): memtoreg=0, memwrite=0, pcsrc=1, alusrc=0, regdst=0, regwrite=0, jump=0, alucontrol=0110
# Time: 100 | BEQ (opcode=000100, zero=0): memtoreg=0, memwrite=0, pcsrc=0, alusrc=0, regdst=0, regwrite=0, jump=0, alucontrol=0110
# Time: 110 | ADDI (opcode=001000): memtoreg=0, memwrite=0, pcsrc=0, alusrc=1, regdst=0, regwrite=1, jump=0, alucontrol=0010
# Time: 120 | ORI (opcode=001101): memtoreg=0, memwrite=0, pcsrc=0, alusrc=1, regdst=0, regwrite=1, jump=0, alucontrol=0001
# Time: 130 | SLTI (opcode=001010): memtoreg=0, memwrite=0, pcsrc=0, alusrc=1, regdst=0, regwrite=1, jump=0, alucontrol=0111
# Time: 140 | JUMP (opcode=000010): memtoreg=0, memwrite=0, pcsrc=0, alusrc=0, regdst=0, regwrite=0, jump=1, alucontrol=0010
# Time: 150 | default (opcode=111111): memtoreg=x, memwrite=x, pcsrc=0, alusrc=x, regdst=x, regwrite=x, jump=x, alucontrol=xxxx
```

Figure 17: Output of controller_tb

- **Explaination:** Testing different instruction such as:
  - Case 1: R-type
  - Case 2: LW/SW
  - Case 3: Store Word
  - Case 4: Branch
  - Case 5: Immediate-instruction
  - Case 6: Jump
  - Case 7: Default
  - Case 8: SLTI
  - Case 9: Jump

  The opcode (input) of different instructions such as R-type, lw, sw, beq, and jump generates outputs that match with the table in section 3.1.2 and 3.1.1 (see Figure 1 and 2).

5.2 Datapath Block

5.2.1 PC Verification

- **Description:** The pc_tb testbench is designed to thoroughly test the functionality of the pc module. This module is responsible for keeping track of the current instruction address and updating it based on control signals, such as branch, jump, and reset.
- **Expected Result:** The testbench should validate the PC module's correct operation in the following scenarios:
  - Normal instructions: Ensure the PC is incremented by 4 (the size of an instruction) after each instruction.
  - Branch instructions: Check that the PC is updated correctly based on the branch condition and target address.
  - Jump instructions: Ensure the PC is updated to the correct jump target address (If the condition is not met, increment the program counter by 4, If the condition is met, jump to the target address)
  - Reset signal: Validate that the PC is correctly reset to the initial address when the reset signal is 1(pc = 0)
- **Result:**

```
# Testcase 0[Success]: expected pc = 00000000, got pc = 00000000
# Testcase 1[Success]: expected pc = 00000004, got pc = 00000004
# Testcase 2[Success]: expected pc = 00000008, got pc = 00000008
# Testcase 3[Success]: expected pc = 0000000c, got pc = 0000000c
# Testcase 4[Success]: expected pc = 00000060, got pc = 00000060
# Testcase 5[Success]: expected pc = 00000064, got pc = 00000064
# Testcase 6[Success]: expected pc = 00800080, got pc = 00800080
# Testcase 7[Success]: expected pc = 00800084, got pc = 00800084
# Testcase 8[Success]: expected pc = 00800138, got pc = 00800138
# Testcase 9[Success]: expected pc = 0080013c, got pc = 0080013c
# Testcase 10[Success]: expected pc = 008000e0, got pc = 008000e0
# Testcase 11[Success]: expected pc = 00000000, got pc = 00000000
# Testcase 12[Success]: expected pc = 00000004, got pc = 00000004
# Testcase 13[Success]: expected pc = 00000008, got pc = 00000008
# Testcase 14[Success]: expected pc = 0000004c, got pc = 0000004c
# Testcase 15[Success]: expected pc = 00000050, got pc = 00000050
# Testcase 16[Success]: expected pc = 00800070, got pc = 00800070
# Testcase 17[Success]: expected pc = 00800074, got pc = 00800074
# Testcase 18[Success]: expected pc = 00800118, got pc = 00800118
# Testcase 19[Success]: expected pc = 0080011c, got pc = 0080011c
# Testcase 20[Success]: expected pc = 008000d0, got pc = 008000d0
# /n==============================
# TEST SUCCESS
# ==============================
```

Figure 18: Output of pc_tb

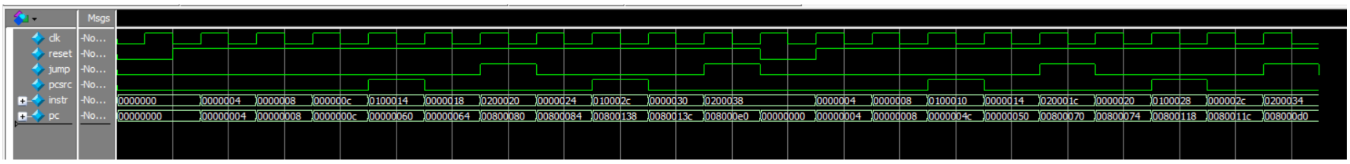

Figure 19: Waveform of pc_tb

- **Explaination:** Testing 20 testcase with 4 different cases: Normal instruction, branch, jump and reset.
  - Normal PC increment: 1, 2, 3, 5, 7, 9, 12, 13, 15, 17, 19 (with PC = PC + 4)
  - Branch: 4, 8, 14, 18 (With PC = PC + 4 + BranchAddr)
  - Jump: 6, 10, 16, 20 (With PC = {PC+4[31:28], address, 2'b0})
  - Reset: 11 (with PC = 0)

5.2.2 ALU Verification

- **Description:** The ALU_tb testbench is designed to verify that the ALU performs basic arithmetic (add, sub) and logical (and, or) operations correctly.
- **Expected Result:** The testbench should validate that the ALU module correctly performs basic arithmetic operations (addition, subtraction) and logical operations (AND, OR, NOT). Specific expectations may include:
  - Correctly perform the addition operation for instructions such as add, lw, and sw when receiving two input values.
  - Correctly perform the subtraction operation for comparison instructions such as beq and slt when receiving two input values.
  - Correctly perform logical operations (AND, OR) when receiving two input values.
- **Result:**

```
# ADD: In1 = 4, In2 = 1, aluout = 5, Zero = 0, Expected: aluout = 5, Zero = 0
# OR: In1 = 9, In2 = 3, aluout = 11, Zero = 0, Expected: aluout = 11, Zero = 0
# SUB: In1 = 13, In2 = 5, aluout = 8, Zero = 0, Expected: aluout = 8, Zero = 0
# AND: In1 = 13, In2 = 5, aluout = 5, Zero = 0, Expected: aluout = 5, Zero = 0
# ADD: In1 = 18, In2 = 1, aluout = 19, Zero = 0, Expected: aluout = 19, Zero = 0
# SLT: In1 = 13, In2 = 22, aluout = 1, Zero = 0, Expected: aluout = 1, Zero = 0
# SUB: In1 = 29, In2 = 13, aluout = 16, Zero = 0, Expected: aluout = 16, Zero = 0
# BEQ(equal): In1 = 25, In2 = 25, aluout = 0, Zero = 1, Expected: aluout = 0, Zero = 1
# BEQ(not equal): In1 = 12, In2 = 25, aluout = 19, Zero = 0, Expected: aluout = 19, Zero = 0
# AND: In1 = 6, In2 = 5, aluout = 4, Zero = 0, Expected: aluout = 4, Zero = 0
# OR: In1 = 10, In2 = 5, aluout = 15, Zero = 0, Expected: aluout = 15, Zero = 0
# SLT: In1 = 23, In2 = 18, aluout = 0, Zero = 0, Expected: aluout = 0, Zero = 0
# SUB: In1 = 15, In2 = 18, aluout = 29, Zero = 0, Expected: aluout = 29, Zero = 0
#
# ===============================================
# TEST SUCCESS
# ===============================================
```

Figure 20: Output of ALU_tb

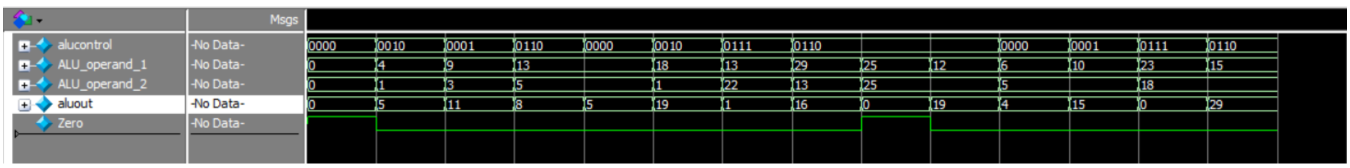

Figure 21: Waveform of ALU_tb

- **Explaination:** ALU will execute different arithemetic and logic function include: Add, Subtract, AND, OR, SLT. I also implement **expected_aluout** and **expected_zero** to compare with aluout and zero. This module pass all testcase.
  - Case 1: ADD (alucontrol: 0010)
  - Case 2: OR (alucontrol: 0001)
  - Case 3: Subtract (alucontrol: 0110)
  - Case 4: AND (alucontrol: 0000)
  - Case 5: SLT (alucontrol: 0111)

5.2.3 REG Verification

- **Description:** The REG_tb testbench is designed to thoroughly verify the functionality of the REG (Register File) module. This module is responsible for storing and retrieving data from the registers
- **Expected Result:** The testbench should validate that the register file can correctly store and retrieve data for all possible register addresses. Specific expectation mays include:
  - Correct data is read from the register address (Provide register address to REG module)
  - Correct data is writen to the register address (Provide register address and data to REG module)
  - When the reset signal is positive (active), the data stored in the registers reset to 0.
- **Result:**

```
: PASSED: data_out_1 = deadbeef
: PASSED: data_out_2 = cafebabe
: PASSED: data_out_1 = 12345678
: PASSED: data_out_2 = 87654321
: PASSED: data_out_1 = abcdef01
: PASSED: data_out_2 = 0101fedc
: PASSED: data_out_1 = 00110011
: PASSED: data_out_2 = 11001100
: PASSED: data_out_1 = ff00ff00
: PASSED: data_out_2 = 00ff00ff
: PASSED: data_out_1 = aaaaaaaa
: PASSED: data_out_2 = 55555555
: PASSED: data_out_1 = 12341234
: PASSED: data_out_2 = 56785678
: PASSED: data_out_1 = 9abc9abc
: PASSED: data_out_2 = 00000000
: PASSED: data_out_1 = 00000000
: PASSED: data_out_2 = 00000000
: PASSED: data_out_1 = 00000000
: PASSED: data_out_2 = 00000000
: PASSED: data_out_1 = 00000000
: PASSED: data_out_2 = 00000000
```

Figure 22: Output of REG_tb

- **Explaination:**
  - The testbench first initializes and writes specific data to various register addresses. The goal is to ensure that the REG (Register File) module correctly stores data at the given addresses. For each address, specific data is written, such as deadbeef at address_1 and cafebabe at address_2.

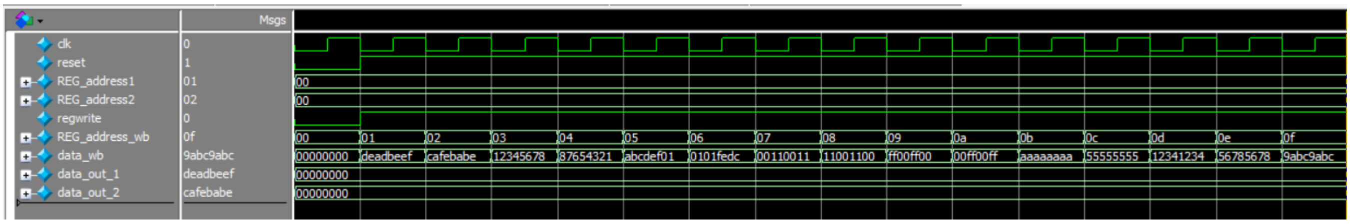

Figure 23: Waveform of writing data to Register

– After writing data to the register file, the testbench reads back the data from the same addresses to verify correctness. For example, after writing deadbeef to address_1, the testbench reads from address_1 to ensure the data deadbeef is correctly stored and retrieved.

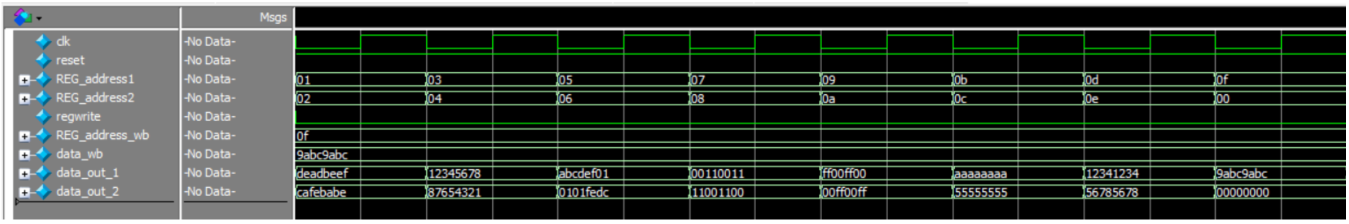

Figure 24: Waveform of reading data to Register

– The reset functionality is tested to ensure that when the reset signal is activated (positive/active), all registers are reset to 0. This is validated by writing specific values to the registers, asserting the reset signal, and then checking that all registers return 0 values.

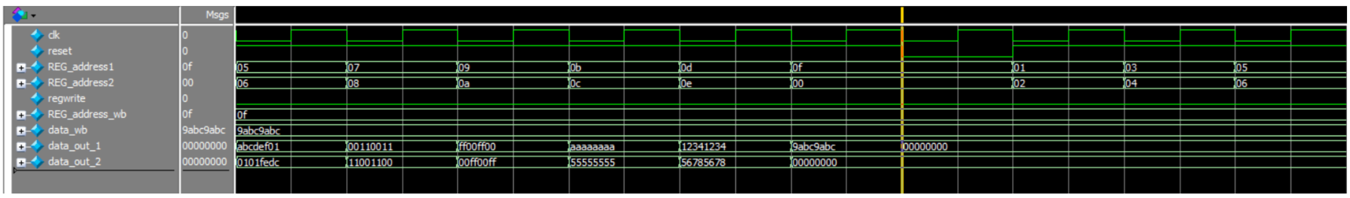

Figure 25: Waveform of reseting

5.2.4 Datapath Verification

- **Description:** The testbench datapath_tb is designed to verify the functionality of the entire datapath. It checks if the datapath correctly processes various types of instructions, including memory instructions, arithmetic and immediate instructions, jump, and branch instructions.
- **Expected Result:** The datapath should correctly output the expected values for each type of instruction. Specific expectation may include:
  – For arithmetic and immediate instructions, read correctly data from the Register the ALU should produce the correct result and write the result to the register.
  – For memory instructions, the correct memory address should be generated and write the data to REG module (for lw instruction)
  – For branch instructions, the correct branch target should be generated based on the branch condition.
  – For jump instructions, the correct jump address should be generated.
- **Result:**

```
# ADDI Instruction: aluout = 00000002 (Expected: R1 + 2)
# ADDI Instruction: aluout = 00000004 (Expected: R1 + 4)
# ANDI Instruction: aluout = 00000000 (Expected: R1 & 15)
# ORI Instruction: aluout = 0000000a (Expected: R1 | 10)
# SLTI Instruction: aluout = 00000001 (Expected: R1 < 3 ? 1 : 0)
# ADDI Instruction: aluout = 00000003 (Expected: R2 + 1)
# ANDI Instruction: aluout = 00000002 (Expected: R2 & 14)
# ORI Instruction: aluout = 0000000b (Expected: R2 | 11)
# SLTI Instruction: aluout = 00000000 (Expected: R2 < 2 ? 1 : 0)
# ADDI Instruction: aluout = 00000009 (Expected: R3 + 5)
# LW Instruction: data_wb = 0000000f (Expected: F)
# SW Instruction: Address = 00000004, Value = 0000000f (Expected: store R2 to memory locati
# Jump Instruction: pc = 00000004 (Expected: 4)
# BEQ Instruction: pc = 00000010 (Expected: branch target address)
```

Figure 26: Output of data_path

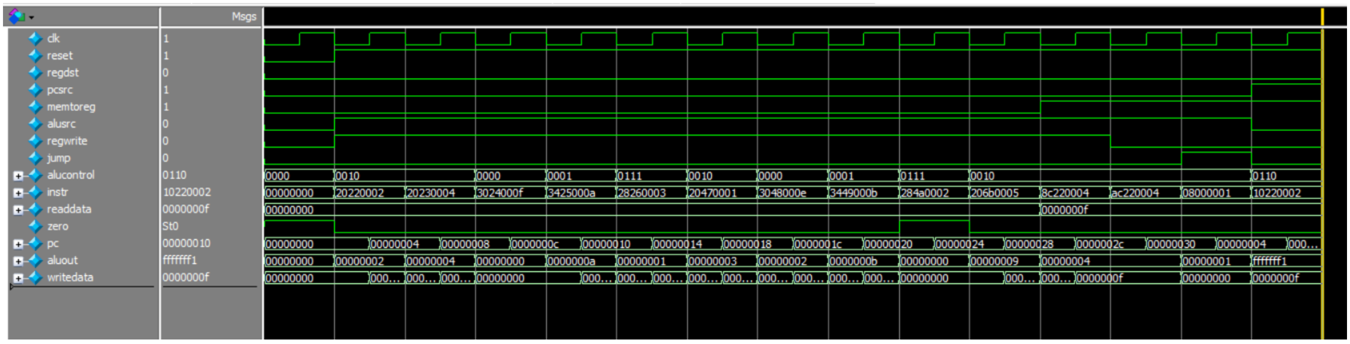

Figure 27: Waveform of data_path

- **Explaination:**
  - In the first 10 test cases, the arithmetic and immediate instructions have their respective aluout values matching the expected results described in the table below.
  - The next two test cases involve memory instructions. The LW instruction loads the value 0x0F from memory at the address in R1 (4) into R2. The SW instruction stores the value in R2 at the address R1 + 4, confirming the memory operations work as expected.
  - The next test case involves a jump instruction. The jump is to the address 1, which corresponds to PC = 1 * 4 = 4, indicating that the program counter has been updated correctly for the jump operation.
  - The next test case involves a branch instruction. In this case, pcsrc is set to 1, indicating that the branch will execute. This means the program counter will be updated to PC + 4 + BranchAddr, confirming the correct behavior of the branch operation.

Assembly	Description	Address
ADDI R2, R1, 2	R2 = R1 + 2 = 2	0x0
ADDI R3, R1, 4	R3 = R1 + 4 = 4	0x4
ANDI R4, R1, 15	R4 = R1 & 15 = 0	0x8
ORI R5, R1, 10	R5 = R1 — 10 = a	0x0C
SLTI R6, R1, 3	R6 = (R1 ¡ 3) ? 1 : 0 = 1	0x10
ADDI R7, R2, 1	R7 = R2 + 1 = 3	0x14
ANDI R8, R2, 14	R8 = R2 & 14 = 2	0x18
ORI R9, R2, 11	R9 = R2 — 11 = b	0x1C
SLTI R10, R2, 2	R10 = (R2 ¡ 2) ? 1 : 0 = 0	0x20
ADDI R11, R3, 5	R11 = R3 + 5 = 9	0x24
LW R2, 4(R1)	R2 = MEM[R1 + 4] = 0x0F(Readdata = 0x0F)	0x28
SW R2, 4(R1)	MEM[4 + R1] = R2 = 0x0F	0x2C
J 1	Jump to address 1 (PC = {PC+4[31:28], address, 2'b0})	0x30
BEQ R1, R2, offset 2	If (R1 == R2) PC = PC + 4 + BranchAddr = 0x10	0x34

Table 3: Testcase for datapath_tb.v

5.3 Mips Processor

5.3.1 Mips Verification 0

- **Description:** The testbench mips_verification_0_tb verifies that the MIPS processor correctly handles the reset signal and the execution of basic instructions in the mips module including ADD, SUB, OR, AND.
- **Expected Result:** The MIPS processor should correctly execute the basic instructions, producing the expected results in the ALU and updating the registers appropriately and upon asserting the reset signal, the processor should:
  - When reset is active set the PC to 0 and clear all register to 0.
  - ADD instruction should produce the correct sum and store it in the specified register.
  - SUB instruction should produce the correct difference and store it in the specified register.
  - AND instruction should perform bitwise AND and store the result in the specified register.
  - OR instruction should perform bitwise OR and store the result in the specified register.
- **Result:**

```
# ADDI Instruction: aluout = 00000002 (Expected: R1 + 2)
# ORI Instruction: aluout = 00000004 (Expected: R1 | 4)
# SLTI Instruction: aluout = 00000001 (Expected: R1 < 3)
# ADD Instruction: aluout = 00000003 (Expected: R4 + R2)
# SUB Instruction: aluout = 00000002 (Expected: R3 - R4)
# PC reset:          0
# OR Instruction: aluout = 00000000 (Expected: R1 | R2)
# AND Instruction: aluout = 00000000 (Expected: R1 & R2)
# SLT Instruction: aluout = 00000000 (Expected: R1 < R2)
```

Figure 28: Output of mips_0_tb

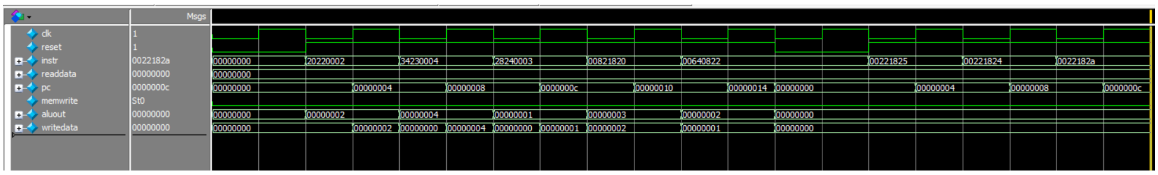

Figure 29: Output of mips_0_tb

- **Explaination:**
  - The first three test cases verify immediate instructions, which calculate and set the correct values in the target registers.
  - The next two test cases check R-type instructions, ensuring the registers hold the expected values set by the previous immediate instructions.
  - The reset signal is then activated, and the program counter (PC) is observed to reset to 0.
  - Finally, additional R-type instructions are tested to confirm that all relevant values are correctly reset to 0.

Assembly	Description	Address
ADDI R2, R1, 2	R2 = R1 + 2 = 2	0x0
ORI R3, R1, 4	R3 = R1 — 4 = 4	0x4
SLTI R4, R1, 3	R4 = (R1 ¡ 15) ? 1 : 0 = 1	0x8
ADD R3, R4, R2	R3 = R4 + R2 = 3	0x0c
SUB R1, R3, R4	R1 = R3 - R4 = 2	0x10
Reset = 0	PC = 4, Register = 0	0x00
OR R3, R1, R2	R3 = R1 — R2 = 0	0x04
AND R3, R1, R2	R3 = R1 & R2 = 0	0x08
SLT R3, R1, R2	R3 = (R1 ¡ R2) ? 1 : 0 = 0	0x0c

Table 4: Testcase for mips_0_tb.v

5.3.2 Mips Verification 1

- **Description:** The testbench mips_verification_2_tb verifies the execution of memory instructions, branch and jump instruction.
- **Expected Result:** The correct address for accessing memory to load or store values between the register and memory. The MIPS processor should correctly execute branch and jump instructions, updating the program counter (PC) to the correct target address when a jump instruction is received, or when the branch condition is true.
  - BEQ (branch if equal) should update the PC to the target address if the specified registers are equal.(PC = PC + 4 + BranchAddr)
  - Jump should update the PC to jump target address. (PC = {PC+4[31:28], address, 2'b0})
- **Result:**

```
# Test Case 1 - LW Instruction: writedata = deadbeef (Expected: DEADBEEF)
# Test Case 2 - SW Instruction: writedata = deadbeef (Expected: R2 value)
# Test Case 3 - LW Instruction: writedata = cafebabe (Expected: CAFEBABE)
# Test Case 4 - SW Instruction: writedata = cafebabe (Expected: R4 value)
# Test Case 5 - LW Instruction: writedata = b16b00b5 (Expected: B16B00B5)
# Test Case 6 - SW Instruction: writedata = b16b00b5 (Expected: R6 value)
# Test Case 7 - BEQ Instruction: pc = 00000024 (Expected: pc + 8 if R2 == R2)
# Test Case 8 - BNE Instruction: pc = 0X0X00X8 (Expected: pc + 8 if R2 != R3)
# Test Case 9 - J Instruction: pc = x0000004 (Expected: jump to address 1)
```

Figure 30: Output of mips_1_tb

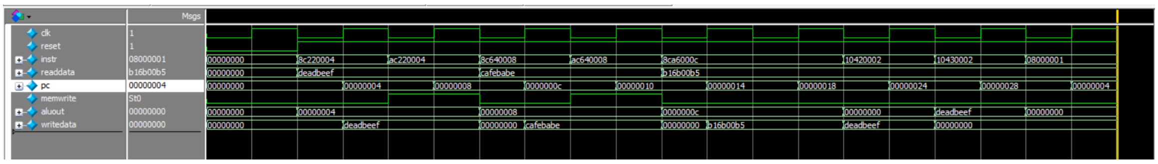

Figure 31: Waveform of mips_1_tb

- **Explaination:**
  - The first six test cases check the functionality of loading data from memory into registers (readdata) and writing data from registers to memory (writedata). These test cases confirm that the data is being written to and read from the correct register addresses.
  - The next two test cases check the branch instructions, including equal and not-equal conditions. The expected values for the program counter (PC) are observed to match the values specified in the table below, confirming the correct behavior of the branch operations.
  - The final test case checks the jump instruction. The observed value of the program counter (PC) is verified to match the expected value specified in the table below, confirming the correct execution of the jump operation.

Assembly	Description	Address
lw R2, 4(R1)	R2 = MEM[4 + R1] = 32'hDEADBEEF	0x0
sw R2, 4(R1)	MEM[4 + R1] = R2	0x4
lw R4, 8(R3)	R4 = MEM[8 + R3] = 32'hCAFEBABE	0x8
sw R4, 8(R3)	MEM[8 + R3] = R4	0x0c
lw R6, 12(R5)	R6 = MEM[12 + R5] = 32'hB16B00B5	0x10
sw R6, 12(R5)	MEM[12 + R5] = R6	0x14
beq R2, R2, offset 2	Equal so PC = PC + 4 + 2 * 4 = 0x24	0x18
beq R2, R3, offset 2	Not Equal so PC = PC + 4 = 0x28	0x1c
j address 1	PC = {PC+4[31:28], address, 2'b0} = 0x04	0x20

Table 5: Testcase for mips_1_tb.v

5.3.3 Mips Verification 2

- **Description:** The testbench mips_verification_2_tb verifies the system with all the instruction.
- **Expected Result:** The aluout, writedata, and PC values should match the table provided below.
- **Result:**

```
# ADDI Instruction: R2 =          8 (Expected: 8)
# ADDI Instruction: R3 =         16 (Expected: 16)
# ADDI Instruction: R6 =          5 (Expected: 5)
# SUB Instruction: R4 =         11 (Expected: 11)
# OR Instruction: R5 =         24 (Expected: 24)
# LW Instruction: R7 =         35 (Expected: 35)
# SW Instruction: Memory[6+R1] =         35 (Expected: 35)
# SLT Instruction: R1 =          0 (Expected: 0)
# ADD Instruction: R8 =         35 (Expected: 35)
# BEQ (Equal) Instruction: PC = 00000030 (Expected: PC + 8)
# ADD Instruction: R2 =         27 (Expected: 27)
# SUB Instruction: R9 =         30 (Expected: 30)
# Jump Instruction: PC = 00000010 (Expected: 0x10)
```

Figure 32: Output of mips_2_tb

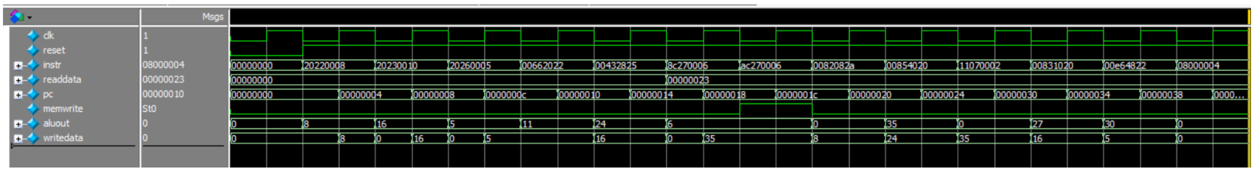

Figure 33: Waveform of mips_2_tb

- **Explaination:**

Assembly	Description	Address
ADDI R2, R1, 8	Initialize R2 = 8	0x0
ADDI R3, R1, 16	Initialize R3 = 16	0x4
ADDI R6, R1, 5	Initialize R6 = 5	0x8
SUB R4, R3, R6	R4 = R3 - R6 = 11	0xC
OR R5, R2, R3	R5 = R2 — R3 = 24	0x10
LW R7, 6(R1)	R7 = MEM[6 + R1] = 35	0x14
SW R7, 6(R1)	MEM[6 + R1] = R7 = 35	0x18
SLT R1, R4, R2	R1 = (R4 ¡ R2) ? 1 : 0 = 0	0x1C
ADD R8, R4, R5	R8 = R4 + R5 = 35	0x20
BEQ R8, R7, 2	Should be taken, PC = 48	0x24
ADD R2, R4, R3	R2 = R4 + R3 = 27	0x28
SUB R6, R7, R6	R6 = R7 - R6 = 30	0x2C
J 2	PC = 0x10	0x2C

Table 6: Testcase for mips_2_tb.v

5.4 Top Block

- **Description:** The testbench top_tb verifies the execution of arithmetic, logical, memory, branch, and jump instructions in the MIPS processor implemented in the top module. It checks if the processor correctly performs arithmetic and logical operations, handles memory access operations (including loading from and storing to memory), correctly evaluates branch conditions, and updates the program counter (PC) as required for both branch and jump instructions.
- **Expected Result:** The final PC when the jump instruction executes should be 0x44 (68 in decimal). The ALUout should be 84, which is the address to write to memory. And the writedata should be 7.
- **Result:**

```
# PC: 00000000, aluout:          5
# PC: 00000004, aluout:         12
# PC: 00000008, aluout:          3
# PC: 0000000c, aluout:          7
# PC: 00000010, aluout:          4
# PC: 00000014, aluout:         11
# PC: 00000018, aluout:          8
# PC: 0000001c, aluout:          0
# PC: 00000020, aluout:          0
# PC: 00000028, aluout:          1
# PC: 0000002c, aluout:         12
# PC: 00000030, aluout:          7
# PC: 00000034, aluout:         80
# PC: 00000038, aluout:         80
# PC: 0000003c, aluout:          0
# PC: 00000044, aluout:         84
# Test Success!!!
```

Figure 34: Output of `top_tb`

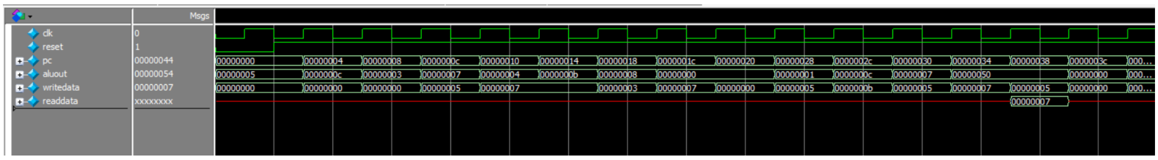

Figure 35: Waveform of top_tb

- **Explanation:**
  - The first three test cases use the addi instruction to add data to the register. This checks that the register file is properly updated with the correct values.
  - The later R-type instructions (e.g., add, sub) check that the register write is happening to the correct address and that the module is executing the right arithmetic and logical functions.
  - The beq (Branch Equal) instruction in test case 7 checks that the Program Counter (PC) is updated correctly based on the branch condition.
  - The sw (Store Word) and lw (Load Word) instructions in cases 14 and 15 confirm that the module can correctly write data to memory and read data from memory at the right addresses.
  - The j (Jump) instruction checks that the PC is updated correctly when jumping to a new address.

#	Assembly	Description	Address
main:	addi $2, $0, 5	# initialize $2 = 5	0
	addi $3, $0, 12	# initialize $3 = 12	4
	addi $7, $3, -9	# initialize $7 = 3	8
	or $4, $7, $9	# $4 = 3 or 5 = 7	c
	and $5, $3, $4	# $5 = 12 and 7 = 4	10
	add $5, $4, $7	# $5 = 4 + 7 = 11	14
	beq $5, $7, end	# shouldn't be taken	18
	slt $4, $3, $4	# $4 = 12 ¡ 7 = 0	1c
	beq $4, $0, around	# should be taken	20
	addi $5, $0, 0	# shouldn't happen	24
around:	slt $4, $7, $2	# $4 = 3 ¡ 5 = 1	28
	add $7, $4, $5	# $7 = 1 + 11 = 12	2c
	sub $7, $7, $2	# $7 = 12 - 5 = 7	30
	sw $7, 68($3)	# [80] = 7	34
	lw $2, 80($0)	# $2 = [80] = 7	38
	j end	# should be taken	3c
	addi $2, $0, 1	# shouldn't happen	40
end:	sw $2, 84($0)	# write address 84 = 7	44

Table 7: Test cases for the top module

6 Conclusion and Future Development

6.1 Conclusion

1. The processor has been designed and tested to perform the basic functionalities of a MIPS processor, including:

- Performing basic arithmetic and logical operations
- Reading/writing registers
- Accessing memory through load/store instructions
- Executing control flow instructions such as branch and jump

2. The test cases have ensured that the values of ALUout, WriteData, Readdata, and PC are all correct as expected, confirming the validity of the design.

6.2 Future Development

- **Instruction set expansion**: Currently, only a basic instruction set is supported. Additional instructions such as multiply, divide, logical shifts, etc., should be added.
- **Performance improvement**: Research and implement multi-cycle or pipelined architectures to increase the processing speed.
- **External Memory Interface**: Develop an IP (Intellectual Property) module to interface the processor with an external memory, such as SRAM or DRAM. This would allow the processor to access a larger memory space beyond the limited on-chip memory.
- **Architecture extension**: Investigate enhancements like adding cache, supporting multithreading, etc., to improve overall performance.

With these improvements, the single-cycle MIPS processor will become more complete and better serve embedded applications and computer architecture research.