Faculty of Engineering and Technology Electrical and Computer Engineering Department

Computer Architecture ENCS4370

Project #2: Project No. 2

Design and Verification of a Simple Pipelined RISC Processor in Verilog

Prepared by:

Student1 Name: Khaled Abo Lebdah **Student1 NO: 1220087**

Student2 Name: Tareq Nazzal **Student2 NO: 1221899**

Student3 Name: Mohammad Shamasneh **Student3 NO:**

Instructor: Ayman Hroup

Section: 2

Date: 10th June 2025

Team Member Name	Team Member ID	Contributions
Khaled Abu Lebdah	1220087	We worked together as a team, meeting in person or online and using a single laptop for development. This made it easier to manage the project, since even small changes in the code could affect how everything works together, making it harder to combine separate parts later.
Tareq Nazzal	1221899	
Mohammad Shamasneh	1220092	

Processor Implementation (Tick One)	
Single Cycle	
Multi Cycle	
Pipelined	✓

In case of pipeline implementation (Tick the correct answer)		
	Implemented in RTL Code?	**Verified and Correctly Worked?**
Data hazards detection	✓	✓
Control hazards detection	✓	✓
Structural hazards detection	✓	✓
Forwarding	✓	✓
Stalling	✓	✓

Tick the correct answer

Instruction	**Did you implement this instruction in the RTL?**		**Did you write the verification code for this instruction?**		**Did the instruction Work perfectly when it has been verified?**	
OR Rd, Rs, Rt	Yes	No	Yes	No	Yes	No
ADD Rd, Rs, Rt	✓		✓		✓	
SUB Rd, Rs, Rt	✓		✓		✓	
CMP Rd, Rs, Rt	✓		✓		✓	
ORI Rd, Rs, Imm	✓		✓		✓	
ADDI Rd, Rs, Imm	✓		✓		✓	
LW Rd, Imm(Rs)	✓		✓		✓	
LDW Rd, Imm(Rs)	✓		✓		✓	
SDW Rd, Imm(Rs)	✓		✓		✓	
BZ Rs, Label	✓		✓		✓	
BGZ Rs, Label	✓		✓		✓	
BLZ Rs, Label	✓		✓		✓	
JR Rs	✓		✓		✓	

J Label	✓		✓		✓	
CALL Label	✓		✓		✓	
Tick one of the following						
My processor can execute only test programs consisting of one instruction only						
My processor can execute complete programs (A simulation screenshot must be provided as evidence)						✓

Abstract

This project presents the design and verification of a five-stage pipelined 32-bit RISC processor implemented in Verilog. The processor adheres to a simplified custom instruction set architecture (ISA) comprising arithmetic, logical, memory, and control flow operations. Key architectural components include dedicated instruction and data memories, a register file with 16 general-purpose registers, and support for complex instructions such as load/store double word (LDW, SDW) and conditional branches (BZ, BGZ, BLZ).

The processor's pipeline stages—Fetch, Decode, Execute, Memory, and Write-Back—are interconnected through carefully designed intermediate buffers that propagate both data and control signals. Advanced features include data forwarding, hazard detection, and pipeline stalling mechanisms to manage dependencies and ensure correct instruction execution. The control logic, generated via truth tables and Boolean expressions, enables precise handling of various instruction types, including jumps (JR, J, CLL) and function calls with return address support.

Verification is conducted through a comprehensive testbench that simulates processor behavior using a memory-initialized instruction set. debug messages validate correct execution and pipeline dynamics. The modular design not only facilitates extensibility but also emphasizes clarity and maintainability, demonstrating an effective application of hardware description methodologies for pipelined processor architecture.

Datapath and components:

This processor is a 5-stage pipelined 32-bit RISC architecture consisting of the following pipeline stages:

1. **Instruction Fetch (IF)**
2. **Instruction Decode (ID)**
3. **Execute (EX)**
4. **Memory Access (MEM)**
5. **Write-Back (WB)**

Each stage handles a portion of the instruction execution process. Inter-stage pipeline registers separate the stages, enabling simultaneous instruction processing and maintaining proper data and control flow.

🔹 1. Instruction Fetch (IF) Stage

Modules Involved:

- FetchSegment
- InstructionMemory
- handlingPC

Functionality:

- The handlingPC module calculates both the **current** and **next** program counter (PC) values based on branching, jumping, and make a disable of PC.
- The InstructionMemory fetches the 32-bit instruction from the address specified by the current PC.
- The FetchSegment calls handlingPC and InstructionMemory and manages special cases, including handling **double-word instructions** (LDW/SDW) and injecting **bubbles** when needed based in INSTRUCTIONsrc signal .
- The stage inputs:

BTA, JRA ,JLA to manage PC using 4-1 mux, and control signals (PCsrc, instructionsrc, DisablePC).

- The stage outputs:
    - Instruction: fetched instruction
    - nextPC: updated PC value for the next cycle

🔹 2. IF/ID Pipeline Buffer

Module: IF_ID_Buffer

Functionality:

- Captures the instruction and nextPC values from the Fetch stage.
- Passes these to the Decode stage.
- Supports stalling via the DisableIR signal, holding the current instruction during a stall.
- Supports special case for LDW and SDW to execute them in two cycles using the mux controlled by DisableIR.

🔹 3. Instruction Decode (ID) Stage

Module: DecodeSegment

Functionality:

- Parses the 32-bit instruction into fields:
    - OPcode: 6-bits operation code
    - Rd, Rs, Rt: register operands, each one of them is 4-bits
    - Imm14: 14-bits immediate value
- Computes:
    - Imm32: 32-bit extended immediate value (zero- or sign-extended)
    - BTA: Branch Target Address (BTA = nextPC + Imm32)
    - JRA: Jump Register Address (value from Rs)

    - JLA: Jump Label Address (JLA = nextPC + Imm32)
    - NextDoubleInstruction: derived instruction for LDW/SDW, also detects the exception (if any) in them
    - The NextDoubleInstruction is the second cycle of double word instruction that we return it in instruction register and we choose it based at INSTRUCTIONsrc signal where we disable the PC based of DisablePC signal.
- Determines the destination register (Rd) if CLL : Rd = R14 , otherwise Rd=Rd .
- Determines the Second source register (Rt) if SDW : Rt = Rd2_in , otherwise Rt=Rt
- Outputs A and B data based on 4-1 mux controlled by forwarding unit, determining if the data came from the register file or from any other stage or when ForwardA = ForwardB = 4 (CLL) then A , B = nextPC , 0 .
- At this stage me make a compare at branch to find if branch taken or not (Branch_flag) if taken then = 1 , other = 0
- Calls the RegisterFile , control unit, forwarding unit and stall unit, and get the control signals from them, then pass these control signals and data to the ID_EX buffer.

Register File Integration

- **Module**: RegisterFile
- **Registers**: R0–R14 (R15 is used as PC, not directly writable)
- **Functionality**:
    - Reads source registers Rs and Rt via BusA and BusB
    - Writes data to register Rd4 using BusW from the WB stage
    - Controlled by the write enable (RegWrite) signal
- Fully synchronized with the clock and reset to ensure reliable register access.
- Supports simultaneous dual-read and single-write operations.
- There we make the R15 cant write at it

🔹 4. ID/EX Pipeline Buffer

Module: ID_EX_Buffer

Functionality:

- Holds data and signals to be used in the Execute stage:
    - A, B: register operands
    - Imm32: extended immediate value
    - Rd2: destination register index
    - Pass these signals: ALUOp, ALUsrc, RegWr, MemRd, MemWr, WBdata, LDW_SDW_Signal, ALUOp, ALUsrc,CLR, Stall
- Can reset or clear contents on control signal or stall condition using CLR signal.

🔹 5. Execute (EX) Stage

Modules: ExecuteSegment, ALU

Functionality:

- Selects the second operand for the ALU:
    - Either register B or immediate Imm32, based on the ALUsrc flag
- Performs arithmetic/logical operations using the ALU:
    - Supported operations include ADD, SUB, OR, and CMP with similar instructions for imm including ADDI, ORI.
- Outputs:
    - ALU_result: the result of the operation
    - zero_flag: indicates whether the ALU result is zero (used for branches)

🔹 6. EX/MEM Pipeline Buffer

Module: EX_MEM_Buffer

Functionality:

- Stores:
    - ALU_result: result of the EX stage
    - Data_in_mem: value from register B to be stored in memory (if applicable)
    - Destination register index and write-back control signals
    - Passes these control signals: RegWr, MemRd, MemWr, WBdata, CLR.
- Transfers this data to the Memory stage.
- Can reset or clear contents on control signal or stall condition using CLR signal.

🔹 7. Memory (MEM) Stage

Modules: MemorySegment, DataMemory

Functionality:

- Performs memory operations:
    - **Load**: read data from DataMemory using the ALU_result as the address
    - **Store**: write register data to memory
- Selects between memory output and ALU result to be forwarded to the WB stage by WBdata signal .
- Select between write and read from signal (MemRd, MemWr)
- Output: Data_wb for write-back

🔹 8. MEM/WB Pipeline Buffer

Module: MEM_WB_Buffer

Functionality:

- Buffers the final result (Data_wb) and destination register index.
- Transfers these to the Write-Back stage.

🔹 9. Write-Back (WB) Stage

Functionality:

- The final result is written back to the RegisterFile at the location specified by Rd4 by RegWr signal, completing the instruction's lifecycle.
- Controlled by the RegWrite signal from the MEM/WB buffer.

Assembly of Components

The modules are assembled within the top-level processor module in a **linear pipeline**, where outputs from one stage feed into buffers that serve as inputs to the next:

1. **FetchSegment → IF_ID_Buffer**
2. **IF_ID_Buffer → DecodeSegment**
3. **DecodeSegment → ID_EX_Buffer**
4. **ID_EX_Buffer → ExecuteSegment**
5. **ExecuteSegment → EX_MEM_Buffer**
6. **EX_MEM_Buffer → MemorySegment**
7. **MemorySegment → MEM_WB_Buffer**
8. **MEM_WB_Buffer → RegisterFile**

Each segment is connected through **control logic** and **data forwarding paths** to resolve hazards and manage instruction dependencies. Control decisions influence not just execution but also instruction flow and register behavior.

Control Signals from Main_Control_Signal Module

These signals determine how each instruction is executed by controlling the ALU, register file, memory access, and pipeline flow.

1. ALUOp ([2:0])

- **Purpose**: Selects the operation for the ALU (e.g., ADD, SUB, OR, CMP)
- **Values**:
    - 000: ADD
    - 001: SUB
    - 010: OR
    - 011: CMP
    - Others: Default/no-op

---

2. ALUsrc

- **Purpose**: Selects the source for the second ALU operand
- **Values**:
    - 0: Use register B (from Register File)
    - 1: Use immediate value (Imm32)

---

3. RegDst ([1:0])

- **Purpose**: Chooses the destination register
- **Values**:
    - 0: Use Rd from instruction
    - 1: Use R14 (for CLL)
- **Usage**: Impacts how Rd2_in is determined in Decode stage

---

4. RBsrc

- **Purpose**: Selects whether Rt or Rd2_in is used as register B's index
- **Values**:
    - 0: Use Rt
    - 1: Use Rd2_in
- **Usage**: Mainly relevant for SDW instruction where both Rd and Rd+1 are written

---

5. ExtOp

- **Purpose**: Chooses between sign-extension and zero-extension for the immediate field
- **Values**:
    - 0: Zero-extend (logical instructions)
    - 1: Sign-extend (arithmetic/branching)

---

6. RegWr

- **Purpose**: Enables writing data to the register file in the WB stage
- **Values**:
    - 1: Write is allowed
    - 0: No write

---

7. MemRd

- **Purpose**: Enables reading from data memory
- **Values**:
    - 1: Read from memory

- 0: No read

---

8. MemWr

- **Purpose**: Enables writing to data memory
- **Values**:
  - 1: Write to memory
  - 0: No write

---

9. WBdata

- **Purpose**: Selects the source of the write-back data
- **Values**:
  - 0: Use ALU result
  - 1: Use memory read data

---

10. PCSrc ([1:0])

- **Purpose**: Determines the next PC source
- **Values**:
  - 0: Normal sequential PC
  - 1: Jump Label Address (JLA)
  - 2: Jump Register Address (JRA)
  - 3: Branch Target Address (BTA)

---

11. INSTRUCTIONsrc ([1:0])

- **Purpose**: Controls the instruction loaded into the pipeline (normal, double-word, bubble)
- **Values**:
    - 0: Normal fetched instruction
    - 1: NextDoubleInstruction (second half of LDW/SDW)
    - 2: Bubble (NOP)

---

12. LDW_SDW_Signal

- **Purpose**: Indicates that a double-word instruction (LDW or SDW) is in the decode stage
- **Values**:
    - 1: First half of double-word instruction
    - 0: Second half or not a double-word instruction

Control Signals from Stall_Unit

These signals manage pipeline control during hazards or special instruction execution.

13. Stall

- **Purpose**: ignore the current instruction in the decode stage by passing zero control signals to the execution stage, so the coming 3 stages (EX, MEM and WB) will never affect on the register file or the data memory
- **Values**:
    - 1: Stall pipeline (Freeze ID/EX and hold PC)

- 0: Normal operation

---

14. DisableIR

- **Purpose**: Freezes the instruction register (IF/ID buffer) during a stall
- **Values**:
  - 1: Hold current instruction
  - 0: Update instruction normally

---

15. DisablePC

- **Purpose**: Prevents the PC from incrementing (halts instruction fetching)
- **Values**:
  - 1: Freeze PC
  - 0: Allow PC update

Control Signals from Forwarding_Unit

These manage data hazards by forwarding operands from later stages to earlier ones.

16. ForwardA ([2:0])

- **Purpose**: Chooses data source for operand A (Rs)
- **Values**:
  - 0: Use Register File
  - 1: Forward from EX stage (ALU result)
  - 2: Forward from MEM stage (Memory result)
  - 3: Forward from WB stage (Data_wb)
  - 4: Use PC + 1 (for CLL)

---

17. ForwardB ([2:0])

- **Purpose**: Chooses data source for operand B (Rt or Rd)
- **Values**: Same as ForwardA but the deferent when the instruction SW or SDW and the Rd :
    - 0: Use Register File
    - 1: Forward from EX stage (ALU result) OR Forward value of Rd from EX stage (ALU result)
    - 2: Forward from MEM stage (Memory result) OR Forward value of Rd from MEM stage (Memory result)
    - 3: Forward from WB stage (Data_wb) OR Forward value of Rd from WB stage (Data_wb)
    - 4: Use PC + 1 (for CLL)

Signal Name	Width	Purpose
ALUOp	3 bits	Selects ALU operation (ADD, SUB, OR, CMP, etc.)
ALUsrc	1 bit	Chooses ALU second operand: register (0) or immediate (1)
RegDst	2 bits	Selects destination register: Rd, Rd+1, or R14 (for CLL)
RBsrc	1 bit	Chooses second register input: Rt or Rd2
ExtOp	1 bit	Determines whether to sign-extend (1) or zero-extend (0) the immediate
RegWr	1 bit	Enables register file write (1 = write enabled)
MemRd	1 bit	Enables reading from data memory
MemWr	1 bit	Enables writing to data memory
WBdata	1 bit	Selects write-back data source: memory (1) or ALU result (0)

Signal Name	Width	Purpose
PCSrc	2 bits	Determines next PC: normal (0), JLA (1), JRA (2), BTA (3)
INSTRUCTIONsrc	2 bits	Instruction input source: normal, double-word, or bubble
LDW_SDW_Signal	1 bit	Indicates LDW/SDW instruction is in decode stage
Stall	1 bit	Stalls the pipeline if hazard or dependency is detected or
DisableIR	1 bit	Freezes instruction register (IF/ID buffer) during stall
DisablePC	1 bit	Freezes PC update during stalls or LDW/SDW handling
ForwardA	3 bits	Controls operand A forwarding from EX, MEM, or WB stages or nextINSTRUCTION for CLL.
ForwardB	3 bits	Controls operand B forwarding from EX, MEM, or WB stages or Zero for CLL.

Table 1: control signal description summary

	RedDst	RBsrc	Ext OP	INSsrc	PCsrc	ALUsrc	ALU op	LDW_SDW_Signal	RegWr	MemRd	MemWr	Wbdata
OR	0	0	x	0	0	0	2	0	1	0	0	0
ADD	0	0	x	0	0	0	0	0	1	0	0	0
SUB	0	0	x	0	0	0	1	0	1	0	0	0
CMP	0	0	x	0	0	0	3	0	1	0	0	0
ORI	0	0	0	0	0	1	2	0	1	0	0	0
ADDI	0	0	1	0	0	1	0	0	1	0	0	0
LW	0	x	1	0	0	1	0	0	1	1	0	1
SW	x	x	1	0	0	1	0	0	0	0	1	x
LDW	0	x	1	First cycle : 1 Second cycle: 0	0	1	0	First cycle : 1 Second cycle: 0	1	1	0	1
SDW	0	1	1	First cycle : 1 Second cycle: 0	0	1	0	First cycle : 1 Second cycle: 0	0	0	1	x
BZ	x	0	1	Taken:2 other: 0	Taken:3 other: 0	x	x	0	0	0	0	x
BGZ	x	0	1	Taken:2 other: 0	Taken:3 other: 0	x	x	0	0	0	0	x
BLZ	x	0	1	Taken:2 other: 0	Taken:3 other: 0	x	x	0	0	0	0	x
JR	x	x	x	2	2	x	x	0	0	0	0	x
J Label	x	x	1	2	1	x	x	0	0	0	0	x
CLL Label	1	x	1	2	1	0	0	0	1	0	0	0

Table 2: Control Signal Truth Table for Instruction Decoding

Datapath and control path diagrams

- **Stall :** execution (insert bubble to execution stage)
    1. The stall signal will be 1 at load delay :
       ((EX_MemRd==1) && (ForwardA==1 || ForwardB==1))

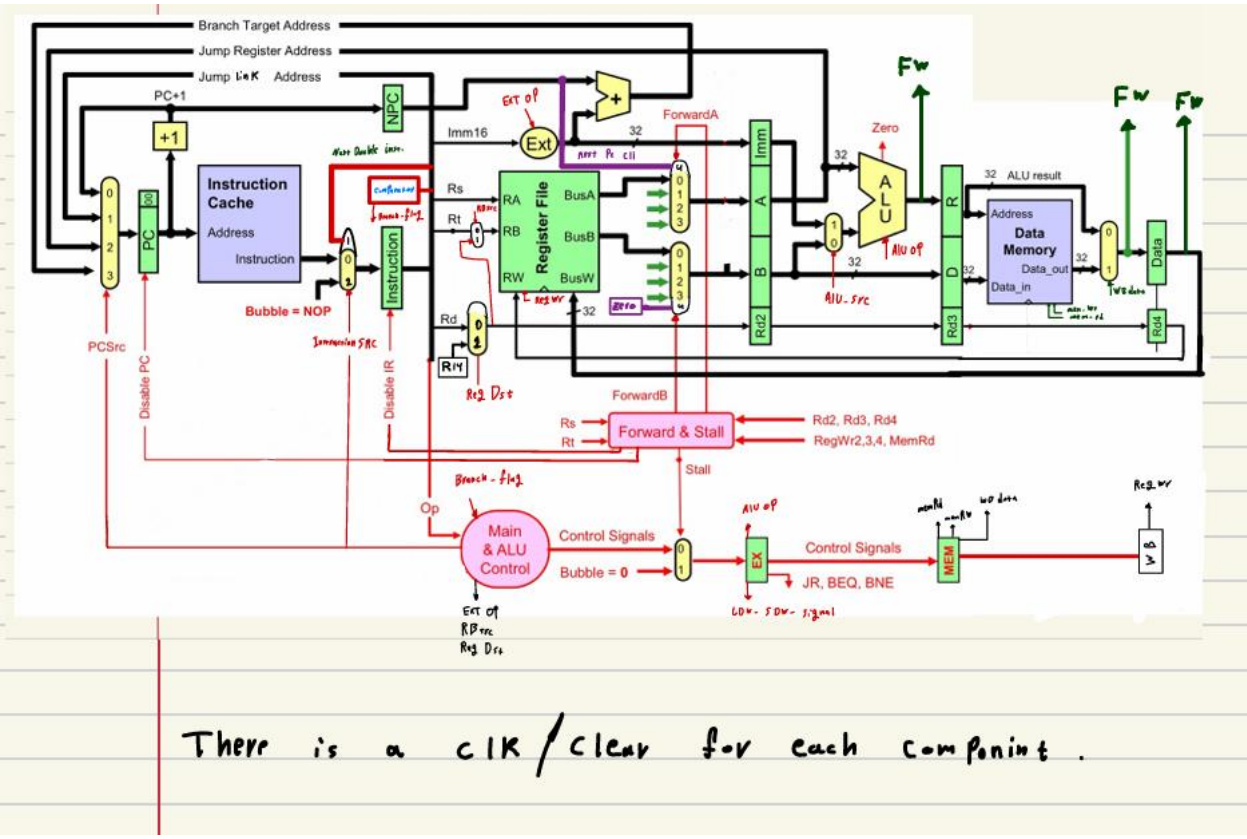
2. When the brach is taken :
   (((OPcode==10)||(OPcode==11)||(OPcode==12)) && Branch_flag)

3. At the JR and J lable :
   ((OPcode==13)||(OPcode==14))
   where the CLL don't stall because its will continue to store the return address in register R14

- **DisableIR** : this signal will disable the IR and don't pass any instruction , and its use for load delay.
- **DisablePC :** this signal will disable the Pc to don't pass the next instruction, its use for load delay and double word instruction.

Note : In jump and branch instructions (assume taken branches), the instruction in the fitch stage is ignored by passing zero instruction to the decode stage via controlling the instructionsSrc mux.

Data Path



There is a clk / clear for each component.

Test programs

Test case 1:

instruction	result	hex code
ADDI R4, R4, 1	R4 = 1	15100001
ADDI R1, R4, 2	R1 = 3	14500002
SW R1, 1(R4)	MEM[2] = 3	1C500001
LW R0, 1(R4)	R0 = 3	18100001
ADD R5, R0, R1	R5 = 6	05404000
ADD R3, R4, R1	R3 = 4	04D04000

Description:

In this test case we test some basic instructions like: ADDI, ADD with Lw and SW, all of instructions works correctly and the result are shown in the table.

Output Screenshots:

```
# KERNEL: PCsrc= x
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000000, (in1=00000000, in2=00000000, ALUop=0)
# KERNEL: [Fetch] PC = 00000000, Instruction = 15100001
# KERNEL: [Decode] Instruction = 00000000, Rs =  0, Rt =  0, Rd =  0, A = 00000000, B = 00000000, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000000
# KERNEL: [Memory] Data = 00000000, MemRead = 0, MemWrite = 0, Data_wb = 00000000
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000000, (in1=00000000, in2=00000000, ALUop=2)
# KERNEL: [Fetch] PC = 00000001, Instruction = 14500002
# KERNEL: [Decode] Instruction = 15100001, Rs =  4, Rt =  0, Rd =  4, A = 00000000, B = 00000000, Imm32 = 00000001
# KERNEL: [Execute]ALU_result= 00000000
# KERNEL: [Memory] Data = 00000000, MemRead = 0, MemWrite = 0, Data_wb = 00000000
# KERNEL: PCsrc= 0
```

```
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000000, (in1=00000000, in2=00000000, ALUop=2)
# KERNEL: [Fetch] PC = 00000001, Instruction = 14500002
# KERNEL: [Decode] Instruction = 15100001, Rs =  4, Rt =  0, Rd =  4, A = 00000000, B = 00000000, Imm32 = 00000001
# KERNEL: [Execute]ALU_result= 00000000
# KERNEL: [Memory] Data = 00000000, MemRead = 0, MemWrite = 0, Data_wb = 00000000
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000001, (in1=00000000, in2=00000001, ALUop=0)
# KERNEL: [Fetch] PC = 00000002, Instruction = 1c500001
# KERNEL: [Decode] Instruction = 14500002, Rs =  4, Rt =  0, Rd =  1, A = 00000001, B = 00000000, Imm32 = 00000002
# KERNEL: [Execute]ALU_result= 00000000
# KERNEL: [Memory] Data = 00000000, MemRead = 0, MemWrite = 0, Data_wb = 00000000
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000

# KERNEL: Testing ALU: result= 00000003, (in1=00000001, in2=00000002, ALUop=0)
# KERNEL: [Fetch] PC = 00000003, Instruction = 18100001
# KERNEL: [Decode] Instruction = 1c500001, Rs =  4, Rt =  0, Rd =  1, A = 00000001, B = 00000003, Imm32 = 00000001
# KERNEL: [Execute]ALU_result= 00000001
# KERNEL: [Memory] Data = 00000001, MemRead = 0, MemWrite = 0, Data_wb = 00000001
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000002, (in1=00000001, in2=00000001, ALUop=0)
# KERNEL: [Fetch] PC = 00000004, Instruction = 05404000
# KERNEL: [Decode] Instruction = 18100001, Rs =  4, Rt =  0, Rd =  0, A = 00000001, B = 00000000, Imm32 = 00000001
# KERNEL: [Execute]ALU_result= 00000003
# KERNEL: [Memory] Data = 00000003, MemRead = 0, MemWrite = 0, Data_wb = 00000003
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000001
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
```

```
# KERNEL: [Fetch] PC = 00000005, Instruction = 04d04000
# KERNEL: [Decode] Instruction = 05404000, Rs =  0, Rt =  1, Rd =  5, A = 00000002, B = 00000003, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000002
# KERNEL: [Memory] Data = 00000002, MemRead = 0, MemWrite = 1, Data_wb = 00000002
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000003
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000001
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: (Write on address= 00000002) - (Data= 00000003)
# KERNEL: Testing ALU: result= 00000000, (in1=00000000, in2=00000000, ALUop=0)
# KERNEL: [Fetch] PC = 00000005, Instruction = 04d04000
# KERNEL: [Decode] Instruction = 05404000, Rs =  0, Rt =  1, Rd =  5, A = 00000003, B = 00000003, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000002
# KERNEL: [Memory] Data = 00000003, MemRead = 1, MemWrite = 0, Data_wb = 00000003
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000003
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000001
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
```

```
# KERNEL: Testing ALU: result= 00000006, (in1=00000003, in2=00000003, ALUop=0)
# KERNEL: [Fetch] PC = 00000006, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = 04d04000, Rs =  4, Rt =  1, Rd =  3, A = 00000001, B = 00000003, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000000
# KERNEL: [Memory] Data = 00000000, MemRead = 0, MemWrite = 0, Data_wb = 00000000
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000003
# KERNEL: Reg[1] = 00000003
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000001
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000004, (in1=00000001, in2=00000003, ALUop=0)
# KERNEL: [Fetch] PC = 00000007, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = xxxxxxxx, Rs =  x, Rt =  x, Rd =  x, A = xxxxxxxx, B = xxxxxxxx, Imm32 = 0000Xxxx
# KERNEL: [Execute]ALU_result= 00000006
# KERNEL: [Memory] Data = 00000006, MemRead = 0, MemWrite = 0, Data_wb = 00000006
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000003
# KERNEL: Reg[1] = 00000003
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000001
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= xxxxxxxx, (in1=xxxxxxxx, in2=xxxxxxxx, ALUop=0)
```

instruction	result	hex code
ADDI R0, R0, 1	R0 = 1	14000001
ADDI R1, R1, 2	R1 = 2	14440002
ADDI R2, R2, 3	R2 = 3	14880003
ADDI R3, R3, 4	R3 = 4	14CC0004
SDW R0, 1(R0)	MEM[2] = 1	24000001
ORI R3, R3, 0	R3 = 4	10CC0000
LDW R2, 1(R0)	R2 = 1, R3=2	20800001
ADD R4, R2, R3	R4 = 3	0508C000

Description:

this test case tests some other instructions like ORI and LDW and SDW, all of them works as specified, the result are shown in the table and screenshots.

Output Screenshots:

```
# KERNEL: PCsrc= x
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000000, (in1=00000000, in2=00000000, ALUop=0)
# KERNEL: [Fetch] PC = 00000000, Instruction = 15100001
# KERNEL: [Decode] Instruction = 00000000, Rs =  0, Rt =  0, Rd =  0, A = 00000000, B = 00000000, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000000
# KERNEL: [Memory] Data = 00000000, MemRead = 0, MemWrite = 0, Data_wb = 00000000
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000000, (in1=00000000, in2=00000000, ALUop=2)
# KERNEL: [Fetch] PC = 00000001, Instruction = 14500002
# KERNEL: [Decode] Instruction = 15100001, Rs =  4, Rt =  0, Rd =  4, A = 00000000, B = 00000000, Imm32 = 00000001
# KERNEL: [Execute]ALU_result= 00000000
# KERNEL: [Memory] Data = 00000000, MemRead = 0, MemWrite = 0, Data_wb = 00000000
# KERNEL: PCsrc= 0
```

```
# KERNEL: Testing ALU: result= 00000000, (in1=00000000, in2=00000000, ALUop=2)
# KERNEL: [Fetch] PC = 00000001, Instruction = 14500002
# KERNEL: [Decode] Instruction = 15100001, Rs =  4, Rt =  0, Rd =  4, A = 00000000, B = 00000000, Imm32 = 00000001
# KERNEL: [Execute]ALU_result= 00000000
# KERNEL: [Memory] Data = 00000000, MemRead = 0, MemWrite = 0, Data_wb = 00000000
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000001, (in1=00000000, in2=00000001, ALUop=0)
# KERNEL: [Fetch] PC = 00000002, Instruction = 1c500001
# KERNEL: [Decode] Instruction = 14500002, Rs =  4, Rt =  0, Rd =  1, A = 00000001, B = 00000000, Imm32 = 00000002
# KERNEL: [Execute]ALU_result= 00000000
# KERNEL: [Memory] Data = 00000000, MemRead = 0, MemWrite = 0, Data_wb = 00000000
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000003, (in1=00000001, in2=00000002, ALUop=0)
```

```
# KERNEL: [Fetch] PC = 00000003, Instruction = 18100001
# KERNEL: [Decode] Instruction = 1c500001, Rs =  4, Rt =  0, Rd =  1, A = 00000001, B = 00000003, Imm32 = 00000001
# KERNEL: [Execute]ALU_result= 00000001
# KERNEL: [Memory] Data = 00000001, MemRead = 0, MemWrite = 0, Data_wb = 00000001
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000002, (in1=00000001, in2=00000001, ALUop=0)
# KERNEL: [Fetch] PC = 00000004, Instruction = 05404000
# KERNEL: [Decode] Instruction = 18100001, Rs =  4, Rt =  0, Rd =  0, A = 00000001, B = 00000000, Imm32 = 00000001
# KERNEL: [Execute]ALU_result= 00000003
# KERNEL: [Memory] Data = 00000003, MemRead = 0, MemWrite = 0, Data_wb = 00000003
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000001
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
```

```
# KERNEL: [Fetch] PC = 00000005, Instruction = 04d04000
# KERNEL: [Decode] Instruction = 05404000, Rs =  0, Rt =  1, Rd =  5, A = 00000002, B = 00000003, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000002
# KERNEL: [Memory] Data = 00000002, MemRead = 0, MemWrite = 1, Data_wb = 00000002
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000003
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000001
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: (Write on address= 00000002) - (Data= 00000003)
# KERNEL: Testing ALU: result= 00000000, (in1=00000000, in2=00000000, ALUop=0)
# KERNEL: [Fetch] PC = 00000005, Instruction = 04d04000
# KERNEL: [Decode] Instruction = 05404000, Rs =  0, Rt =  1, Rd =  5, A = 00000003, B = 00000003, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000002
# KERNEL: [Memory] Data = 00000003, MemRead = 1, MemWrite = 0, Data_wb = 00000003
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000003
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000001
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000006, (in1=00000003, in2=00000003, ALUop=0)
```

```
# KERNEL: [Fetch] PC = 00000006, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = 04d04000, Rs =  4, Rt =  1, Rd =  3, A = 00000001, B = 00000003, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000000
# KERNEL: [Memory] Data = 00000000, MemRead = 0, MemWrite = 0, Data_wb = 00000000
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000003
# KERNEL: Reg[1] = 00000003
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000001
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000004, (in1=00000001, in2=00000003, ALUop=0)
# KERNEL: [Fetch] PC = 00000007, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = xxxxxxxx, Rs =  x, Rt =  x, Rd =  x, A = xxxxxxxx, B = xxxxxxxx, Imm32 = 0000Xxxx
# KERNEL: [Execute]ALU_result= 00000006
# KERNEL: [Memory] Data = 00000006, MemRead = 0, MemWrite = 0, Data_wb = 00000006
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000003
# KERNEL: Reg[1] = 00000003
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000001
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= xxxxxxxx, (in1=xxxxxxxx, in2=xxxxxxxx, ALUop=0)
# KERNEL: [Fetch] PC = 00000008, Instruction = xxxxxxxx
```

```
# KERNEL: [Fetch] PC = 00000008, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = xxxxxxxx, Rs =  x, Rt =  x, Rd =  x, A = xxxxxxxx, B = xxxxxxxx, Imm32 = 0000Xxxx
# KERNEL: [Execute]ALU_result= 00000004
# KERNEL: [Memory] Data = 00000004, MemRead = 0, MemWrite = 0, Data_wb = 00000004
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000003
# KERNEL: Reg[1] = 00000003
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000001
# KERNEL: Reg[5] = 00000006
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: [Fetch] PC = 00000009, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = xxxxxxxx, Rs =  x, Rt =  x, Rd =  x, A = xxxxxxxx, B = xxxxxxxx, Imm32 = 0000Xxxx
# KERNEL: [Execute]ALU_result= xxxxxxxx
# KERNEL: [Memory] Data = xxxxxxxx, MemRead = 0, MemWrite = 0, Data_wb = xxxxxxxx
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000003
# KERNEL: Reg[1] = 00000003
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000004
# KERNEL: Reg[4] = 00000001
# KERNEL: Reg[5] = 00000006
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: [Fetch] PC = 0000000a, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = xxxxxxxx, Rs =  x, Rt =  x, Rd =  x, A = xxxxxxxx, B = xxxxxxxx, Imm32 = 0000Xxxx
# KERNEL: [Execute]ALU_result= xxxxxxxx

# KERNEL: [Fetch] PC = 0000000a, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = xxxxxxxx, Rs =  x, Rt =  x, Rd =  x, A = xxxxxxxx, B = xxxxxxxx, Imm32 = 0000Xxxx
# KERNEL: [Execute]ALU_result= xxxxxxxx
# KERNEL: [Memory] Data = xxxxxxxx, MemRead = 0, MemWrite = 0, Data_wb = xxxxxxxx
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000003
# KERNEL: Reg[1] = 00000003
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000004
# KERNEL: Reg[4] = 00000001
# KERNEL: Reg[5] = 00000006
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: [Fetch] PC = 0000000b, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = xxxxxxxx, Rs =  x, Rt =  x, Rd =  x, A = xxxxxxxx, B = xxxxxxxx, Imm32 = 0000Xxxx
# KERNEL: [Execute]ALU_result= xxxxxxxx
# KERNEL: [Memory] Data = xxxxxxxx, MemRead = 0, MemWrite = 0, Data_wb = xxxxxxxx
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000003
# KERNEL: Reg[1] = 00000003
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000004
# KERNEL: Reg[4] = 00000001
# KERNEL: Reg[5] = 00000006
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
```

```
# KERNEL: [Fetch] PC = 0000000c, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = xxxxxxxx, Rs =  x, Rt =  x, Rd =  x, A = xxxxxxxx, B = xxxxxxxx, Imm32 = 0000Xxxx
# KERNEL: [Execute]ALU_result= xxxxxxxx
# KERNEL: [Memory] Data = xxxxxxxx, MemRead = 0, MemWrite = 0, Data_wb = xxxxxxxx
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000003
# KERNEL: Reg[1] = 00000003
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000004
# KERNEL: Reg[4] = 00000001
# KERNEL: Reg[5] = 00000006
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: [Fetch] PC = 0000000d, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = xxxxxxxx, Rs =  x, Rt =  x, Rd =  x, A = xxxxxxxx, B = xxxxxxxx, Imm32 = 0000Xxxx
# KERNEL: [Execute]ALU_result= xxxxxxxx
# KERNEL: [Memory] Data = xxxxxxxx, MemRead = 0, MemWrite = 0, Data_wb = xxxxxxxx
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000003
# KERNEL: Reg[1] = 00000003
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000004
# KERNEL: Reg[4] = 00000001
# KERNEL: Reg[5] = 00000006
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: [Fetch] PC = 0000000e, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = xxxxxxxx, Rs =  x, Rt =  x, Rd =  x, A = xxxxxxxx, B = xxxxxxxx, Imm32 = 0000Xxxx
# KERNEL: [Execute]ALU_result= xxxxxxxx
```

```
# KERNEL: [Fetch] PC = 0000000e, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = xxxxxxxx, Rs =  x, Rt =  x, Rd =  x, A = xxxxxxxx, B = xxxxxxxx, Imm32 = 0000Xxxx
# KERNEL: [Execute]ALU_result= xxxxxxxx
# KERNEL: [Memory] Data = xxxxxxxx, MemRead = 0, MemWrite = 0, Data_wb = xxxxxxxx
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000003
# KERNEL: Reg[1] = 00000003
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000004
# KERNEL: Reg[4] = 00000001
# KERNEL: Reg[5] = 00000006
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: [Fetch] PC = 0000000f, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = xxxxxxxx, Rs =  x, Rt =  x, Rd =  x, A = xxxxxxxx, B = xxxxxxxx, Imm32 = 0000Xxxx
# KERNEL: [Execute]ALU_result= xxxxxxxx
# KERNEL: [Memory] Data = xxxxxxxx, MemRead = 0, MemWrite = 0, Data_wb = xxxxxxxx
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000003
# KERNEL: Reg[1] = 00000003
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000004
# KERNEL: Reg[4] = 00000001
# KERNEL: Reg[5] = 00000006
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: [Fetch] PC = 00000010, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = xxxxxxxx, Rs =  x, Rt =  x, Rd =  x, A = xxxxxxxx, B = xxxxxxxx, Imm32 = 0000Xxxx
# KERNEL: [Execute]ALU_result= xxxxxxxx
```

```
# KERNEL: [Fetch] PC = 00000010, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = xxxxxxxx, Rs =  x, Rt =  x, Rd =  x, A = xxxxxxxx, B = xxxxxxxx, Imm32 = 0000Xxxx
# KERNEL: [Execute]ALU_result= xxxxxxxx
# KERNEL: [Memory] Data = xxxxxxxx, MemRead = 0, MemWrite = 0, Data_wb = xxxxxxxx
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000003
# KERNEL: Reg[1] = 00000003
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000004
# KERNEL: Reg[4] = 00000001
# KERNEL: Reg[5] = 00000006
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: [Fetch] PC = 00000011, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = xxxxxxxx, Rs =  x, Rt =  x, Rd =  x, A = xxxxxxxx, B = xxxxxxxx, Imm32 = 0000Xxxx
# KERNEL: [Execute]ALU_result= xxxxxxxx
# KERNEL: [Memory] Data = xxxxxxxx, MemRead = 0, MemWrite = 0, Data_wb = xxxxxxxx
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000003
# KERNEL: Reg[1] = 00000003
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000004
# KERNEL: Reg[4] = 00000001
# KERNEL: Reg[5] = 00000006
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: [Fetch] PC = 00000012, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = xxxxxxxx, Rs =  x, Rt =  x, Rd =  x, A = xxxxxxxx, B = xxxxxxxx, Imm32 = 0000Xxxx
# KERNEL: [Execute]ALU_result= xxxxxxxx
```

Test case 3

Program counter (PC)	Instruction	result	Hexadecimal
0	ADDI R0, R0, 5	R0 = 5	14000005
1	ADDI R1, R1, 7	R1 = 7	14440007
2	ORI R2, R0, 0	R2 = 5	10800000
3	BLZ R1, 2	Condition not happened	28040002
4	BGZ R2, 2	PC = 7	2C020002
5	OR R4, R3, R1	Passed	010C4000
6	SUB R5, R4, R2	Passed	09508000
7	J 2	Jump to PC = 10	38000002
8	CMP R6, R2, R5	Passed	0D894000
9	OR R8, R0, R1	Passed	02004000
10	ADD R7, R1, R0	R7 = 12	05C04000

Description:

This test case is very important, it tests OR, BLZ, BGZ and J instruction, the results show the following:

1- assign values to R0 and R1 to use them later
2- Assign R2 = R0 using ORI instruction
3- Check the first branch instruction BLZ R1, 2 , the condition of the branch did not happen, so the program continues.
4- Check the next branch BGZ R2, 2, this branch condition satisfied, so the program will go to instruction 7
5- Instruction 7 is a normal jump instruction so the program will jump to instruction 10 directly
6- Finally execute the final instruction, R7 will be 12.

Screenshots:

```
# KERNEL: ++++++++++++++++++Starting++++++++++++++++++++++++++
# KERNEL: PCsrc= x
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000000, (in1=00000000, in2=00000000, ALUop=0)
# KERNEL: [Fetch] PC = 00000000, Instruction = 14000005
# KERNEL: [Decode] Instruction = 00000000, Rs =  0, Rt =  0, Rd =  0, A = 00000000, B = 00000000, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000000
# KERNEL: [Memory] Data = 00000000, MemRead = 0, MemWrite = 0, Data_wb = 00000000
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000000, (in1=00000000, in2=00000000, ALUop=2)
# KERNEL: [Fetch] PC = 00000001, Instruction = 14440007
# KERNEL: [Decode] Instruction = 14000005, Rs =  0, Rt =  0, Rd =  0, A = 00000000, B = 00000000, Imm32 = 00000005
# KERNEL: [Execute]ALU_result= 00000000
# KERNEL: [Memory] Data = 00000000, MemRead = 0, MemWrite = 0, Data_wb = 00000000
```

```
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000005, (in1=00000000, in2=00000005, ALUop=0)
# KERNEL: [Fetch] PC = 00000002, Instruction = 10800000
# KERNEL: [Decode] Instruction = 14440007, Rs =  1, Rt =  0, Rd =  1, A = 00000000, B = 00000005, Imm32 = 00000007
# KERNEL: [Execute]ALU_result= 00000000
# KERNEL: [Memory] Data = 00000000, MemRead = 0, MemWrite = 0, Data_wb = 00000000
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000007, (in1=00000000, in2=00000007, ALUop=0)
# KERNEL: [Fetch] PC = 00000003, Instruction = 28040002
# KERNEL: [Decode] Instruction = 10800000, Rs =  0, Rt =  0, Rd =  2, A = 00000005, B = 00000005, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000005
# KERNEL: [Memory] Data = 00000005, MemRead = 0, MemWrite = 0, Data_wb = 00000005
# KERNEL: PCsrc= 0
```

```
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000005, (in1=00000005, in2=00000000, ALUop=2)
# KERNEL: [Fetch] PC = 00000004, Instruction = 2c020002
# KERNEL: [Decode] Instruction = 28040002, Rs =  1, Rt =  0, Rd =  0, A = 00000007, B = 00000005, Imm32 = 00000002
# KERNEL: [Execute]ALU_result= 00000007
# KERNEL: [Memory] Data = 00000007, MemRead = 0, MemWrite = 0, Data_wb = 00000007
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000005
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 0000000c, (in1=00000007, in2=00000005, ALUop=0)
# KERNEL: [Fetch] PC = 00000005, Instruction = 00000000
# KERNEL: [Decode] Instruction = 2c020002, Rs =  0, Rt =  8, Rd =  0, A = 00000005, B = 00000000, Imm32 = 00000002
# KERNEL: [Execute]ALU_result= 00000005
# KERNEL: [Memory] Data = 00000005, MemRead = 0, MemWrite = 0, Data_wb = 00000005
# KERNEL: PCsrc= 3
```

```
# KERNEL: Reg[0] = 00000005
# KERNEL: Reg[1] = 00000007
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000000, (in1=00000000, in2=00000000, ALUop=0)
# KERNEL: [Fetch] PC = 00000007, Instruction = 38000002
# KERNEL: [Decode] Instruction = 00000000, Rs =  0, Rt =  0, Rd =  0, A = 00000005, B = 00000005, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 0000000c
# KERNEL: [Memory] Data = 0000000c, MemRead = 0, MemWrite = 0, Data_wb = 0000000c
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000005
# KERNEL: Reg[1] = 00000007
# KERNEL: Reg[2] = 00000005
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000005, (in1=00000005, in2=00000005, ALUop=2)
# KERNEL: [Fetch] PC = 00000008, Instruction = 00000000
# KERNEL: [Decode] Instruction = 38000002, Rs =  0, Rt =  0, Rd =  0, A = 00000005, B = 00000005, Imm32 = 00000002
# KERNEL: [Execute]ALU_result= 00000000
# KERNEL: [Memory] Data = 00000000, MemRead = 0, MemWrite = 0, Data_wb = 00000000
# KERNEL: PCsrc= 1
# KERNEL: Reg[0] = 00000005
```

```
# KERNEL: PCsrc= 1
# KERNEL: Reg[0] = 00000005
# KERNEL: Reg[1] = 00000007
# KERNEL: Reg[2] = 00000005
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000000, (in1=00000000, in2=00000000, ALUop=0)
# KERNEL: [Fetch] PC = 0000000a, Instruction = 05c04000
# KERNEL: [Decode] Instruction = 00000000, Rs =  0, Rt =  0, Rd =  0, A = 00000005, B = 00000005, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000005
# KERNEL: [Memory] Data = 00000005, MemRead = 0, MemWrite = 0, Data_wb = 00000005
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000005
# KERNEL: Reg[1] = 00000007
# KERNEL: Reg[2] = 00000005
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000005, (in1=00000005, in2=00000005, ALUop=2)
# KERNEL: [Fetch] PC = 0000000b, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = 05c04000, Rs =  0, Rt =  1, Rd =  7, A = 00000005, B = 00000007, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000000
# KERNEL: [Memory] Data = 00000000, MemRead = 0, MemWrite = 0, Data_wb = 00000000
# KERNEL: PCsrc= 0
```

Test case 4

instruction number	Instruction	result	Hexadecimal
0	addI R0, R0, 5	R0 = 5	14000005
1	addI R1, R1, 7	R1 = 7	14440007
2	ORI R2, R0, 0	R2 = 5	10800000
3	BLZ R1, 2	Branch not taken	28040002
4	BGZ R2, 2	PC = 7	2C020002
5	OR R4, R3, R1	Passed	010C4000
6	SUB R5, R4, R2	Passed	09508000
7	J 2	Passed	38000002
8	CMP R6, R2, R5	Passed	0D894000
9	OR R8, R0, R1	Passed	02004000
10	ADD R7, R1, R0	R7 = 3	05C04000
11	OR R3, R2, R1	R3 = 15	00C84000
12	ADD R3, R2, R1	R3 = 12	04C84000
13	SUB R3, R2, R1	R3 = -2	08C84000
14	CMP R3, R2, R1	R3 = -1	0CC84000
15	ORI R3, R2, 15	R3 = 15	10C8000F
16	ADDI R3, R2, 15	R3 = 20	14C8000F

Note: numbers in Registers are in decimal not hex

Description:

This test case is extended from the previous one to test the rest of Arithmetic and logic instructions such as SUB and CMP.

The program will change the Register R3 several times to check all Arithmetic and logic instructions together.

Screenshots:

```
# KERNEL: Testing ALU: result= 00000005, (in1=00000005, in2=00000005, ALUop=2)
# KERNEL: [Fetch] PC = 0000000b, Instruction = 00c84000
# KERNEL: [Decode] Instruction = 05c04000, Rs =  0, Rt =  1, Rd =  7, A = 00000005, B = 00000007, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000000
# KERNEL: [Memory] Data = 00000000, MemRead = 0, MemWrite = 0, Data_wb = 00000000
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000005
# KERNEL: Reg[1] = 00000007
# KERNEL: Reg[2] = 00000005
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 0000000c, (in1=00000005, in2=00000007, ALUop=0)
# KERNEL: [Fetch] PC = 0000000c, Instruction = 04c84000
# KERNEL: [Decode] Instruction = 00c84000, Rs =  2, Rt =  1, Rd =  3, A = 00000005, B = 00000007, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000005
# KERNEL: [Memory] Data = 00000005, MemRead = 0, MemWrite = 0, Data_wb = 00000005
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000005
# KERNEL: Reg[1] = 00000007
# KERNEL: Reg[2] = 00000005
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000007, (in1=00000005, in2=00000007, ALUop=2)
```

```
# KERNEL: [Fetch] PC = 0000000d, Instruction = 08c84000
# KERNEL: [Decode] Instruction = 04c84000, Rs =  2, Rt =  1, Rd =  3, A = 00000005, B = 00000007, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 0000000c
# KERNEL: [Memory] Data = 0000000c, MemRead = 0, MemWrite = 0, Data_wb = 0000000c
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000005
# KERNEL: Reg[1] = 00000007
# KERNEL: Reg[2] = 00000005
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 0000000c, (in1=00000005, in2=00000007, ALUop=0)
# KERNEL: [Fetch] PC = 0000000e, Instruction = 0cc84000
# KERNEL: [Decode] Instruction = 08c84000, Rs =  2, Rt =  1, Rd =  3, A = 00000005, B = 00000007, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000007
# KERNEL: [Memory] Data = 00000007, MemRead = 0, MemWrite = 0, Data_wb = 00000007
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000005
# KERNEL: Reg[1] = 00000007
# KERNEL: Reg[2] = 00000005
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 0000000c
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= fffffffe, (in1=00000005, in2=00000007, ALUop=1)
# KERNEL: [Fetch] PC = 0000000f, Instruction = 10c8000f
```

```
# KERNEL: Testing ALU: result= fffffffe, (in1=00000005, in2=00000007, ALUop=1)
# KERNEL: [Fetch] PC = 0000000f, Instruction = 10c8000f
# KERNEL: [Decode] Instruction = 0cc84000, Rs =  2, Rt =  1, Rd =  3, A = 00000005, B = 00000007, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 0000000c
# KERNEL: [Memory] Data = 0000000c, MemRead = 0, MemWrite = 0, Data_wb = 0000000c
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000005
# KERNEL: Reg[1] = 00000007
# KERNEL: Reg[2] = 00000005
# KERNEL: Reg[3] = 00000007
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 0000000c
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= ffffffff, (in1=00000005, in2=00000007, ALUop=3)
# KERNEL: [Fetch] PC = 00000010, Instruction = 14c8000f
# KERNEL: [Decode] Instruction = 10c8000f, Rs =  2, Rt =  0, Rd =  3, A = 00000005, B = 00000005, Imm32 = 0000000f
# KERNEL: [Execute]ALU_result= fffffffe
# KERNEL: [Memory] Data = fffffffe, MemRead = 0, MemWrite = 0, Data_wb = fffffffe
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000005
# KERNEL: Reg[1] = 00000007
# KERNEL: Reg[2] = 00000005
# KERNEL: Reg[3] = 0000000c
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 0000000c
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 0000000f, (in1=00000005, in2=0000000f, ALUop=2)
```

```
# KERNEL: Testing ALU: result= 0000000f, (in1=00000005, in2=0000000f, ALUop=2)
# KERNEL: [Fetch] PC = 00000011, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = 14c8000f, Rs =  2, Rt =  0, Rd =  3, A = 00000005, B = 00000005, Imm32 = 0000000f
# KERNEL: [Execute]ALU_result= ffffffff
# KERNEL: [Memory] Data = ffffffff, MemRead = 0, MemWrite = 0, Data_wb = ffffffff
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000005
# KERNEL: Reg[1] = 00000007
# KERNEL: Reg[2] = 00000005
# KERNEL: Reg[3] = fffffffe
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 0000000c
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000014, (in1=00000005, in2=0000000f, ALUop=0)
# KERNEL: [Fetch] PC = 00000012, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = xxxxxxxx, Rs =  x, Rt =  x, Rd =  x, A = xxxxxxxx, B = xxxxxxxx, Imm32 = xxxxxxxx
# KERNEL: [Execute]ALU_result= 0000000f
# KERNEL: [Memory] Data = 0000000f, MemRead = 0, MemWrite = 0, Data_wb = 0000000f
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000005
# KERNEL: Reg[1] = 00000007
# KERNEL: Reg[2] = 00000005
# KERNEL: Reg[3] = ffffffff
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 0000000c
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= xxxxxxxx, (in1=xxxxxxxx, in2=xxxxxxxx, ALUop=0)
```

Test case 5

PC	Instruction	Effect / Result	Hexadecimal
0	ADDI R0, R0, 4	R0 = 4	14000004
1	ADDI R1, R1, 5	R1 = 5	14440005
2	CLL 3	R14 = 3, jump to PC = 2 + 3 + 1 = 6	3C000003
3	ADDI R1, R1, 5	*Skipped due to jump*	14440005
4	ORI R7, R7, 3	*Skipped due to jump*	11DC0003
5	J 3	*Skipped due to jump*	38000003
6	ORI R3, R3, 1	R3 = 1	10CC0001
7	ADD R4, R0, R1	R4 = R0 + R1 = 4 + 5 = 9	05004000
8	JR R14	Jump to PC = R14 = 3	34380000
9	ADDI R0, R0, 3	R0 = 4 + 3 = 7 (executed after return)	14000003

Description:

This case cover the rest of the instructions such as CLL and JRA, with this final test case we have tested all the instructions in this ISA.

Program Execution Steps

1. **Initialize Registers**
    - R0 ← 4 using ADDI R0, R0, 4
    - R1 ← 5 using ADDI R1, R1, 5
2. **Function Call**
    - Execute CLL 3
        - Save return address PC = 3 in R14
        - Jump to PC = 2 + 3 + 1 = 6
3. **Function Execution (at PC = 6)**
    - R3 ← 1 using ORI R3, R3, 1

- R4 ← R0 + R1 = 4 + 5 = 9 using ADD R4, R0, R1

4. **Return from Function**
    - Execute JR R14
        - Jump back to PC = R14 = 3
5. **Resume Main Program**
    - R1 ← R1 + 5 = 10 using ADDI R1, R1, 5
    - R7 ← 3 using ORI R7, R7, 3
6. **Jump Forward**
    - Execute J 3
        - Jump to PC = current + 3 + 1 = 9
7. **Final Operation**
    - R0 ← R0 + 3 = 7 using ADDI R0, R0, 3

Screenshots:

```
# KERNEL: ++++++++++++++++++Starting++++++++++++++++++++++++++
# KERNEL: PCsrc= x
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000000, (in1=00000000, in2=00000000, ALUop=0)
# KERNEL: [Fetch] PC = 00000000, Instruction = 14000004
# KERNEL: [Decode] Instruction = 00000000, Rs =  0, Rt =  0, Rd =  0, A = 00000000, B = 00000000, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000000
# KERNEL: [Memory] Data = 00000000, MemRead = 0, MemWrite = 0, Data_wb = 00000000
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000000, (in1=00000000, in2=00000000, ALUop=2)
# KERNEL: [Fetch] PC = 00000001, Instruction = 14440005
# KERNEL: [Decode] Instruction = 14000004, Rs =  0, Rt =  0, Rd =  0, A = 00000000, B = 00000000, Imm32 = 00000004
# KERNEL: [Execute]ALU_result= 00000000
# KERNEL: [Memory] Data = 00000000, MemRead = 0, MemWrite = 0, Data_wb = 00000000
```

```
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000004, (in1=00000000, in2=00000004, ALUop=0)
# KERNEL: [Fetch] PC = 00000002, Instruction = 3c000003
# KERNEL: [Decode] Instruction = 14440005, Rs =  1, Rt =  0, Rd =  1, A = 00000000, B = 00000004, Imm32 = 00000005
# KERNEL: [Execute]ALU_result= 00000000
# KERNEL: [Memory] Data = 00000000, MemRead = 0, MemWrite = 0, Data_wb = 00000000
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000005, (in1=00000000, in2=00000005, ALUop=0)
# KERNEL: [Fetch] PC = 00000003, Instruction = 00000000
# KERNEL: [Decode] Instruction = 3c000003, Rs =  0, Rt =  0, Rd = 14, A = 00000003, B = 00000000, Imm32 = 00000003
# KERNEL: [Execute]ALU_result= 00000004
# KERNEL: [Memory] Data = 00000004, MemRead = 0, MemWrite = 0, Data_wb = 00000004
# KERNEL: PCsrc= 1
```

```
# KERNEL: Reg[0] = 00000000
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000003, (in1=00000003, in2=00000000, ALUop=0)
# KERNEL: [Fetch] PC = 00000006, Instruction = 10cc0001
# KERNEL: [Decode] Instruction = 00000000, Rs =  0, Rt =  0, Rd =  0, A = 00000004, B = 00000004, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000005
# KERNEL: [Memory] Data = 00000005, MemRead = 0, MemWrite = 0, Data_wb = 00000005
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000004
# KERNEL: Reg[1] = 00000000
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000004, (in1=00000004, in2=00000004, ALUop=2)
# KERNEL: [Fetch] PC = 00000007, Instruction = 05004000
# KERNEL: [Decode] Instruction = 10cc0001, Rs =  3, Rt =  0, Rd =  3, A = 00000000, B = 00000004, Imm32 = 00000001
# KERNEL: [Execute]ALU_result= 00000003
# KERNEL: [Memory] Data = 00000003, MemRead = 0, MemWrite = 0, Data_wb = 00000003
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000004
```

```
# KERNEL: [Memory] Data = 00000003, MemRead = 0, MemWrite = 0, Data_wb = 00000003
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000004
# KERNEL: Reg[1] = 00000005
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000000
# KERNEL: Testing ALU: result= 00000001, (in1=00000000, in2=00000001, ALUop=2)
# KERNEL: [Fetch] PC = 00000008, Instruction = 34380000
# KERNEL: [Decode] Instruction = 05004000, Rs =  0, Rt =  1, Rd =  4, A = 00000004, B = 00000005, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000004
# KERNEL: [Memory] Data = 00000004, MemRead = 0, MemWrite = 0, Data_wb = 00000004
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000004
# KERNEL: Reg[1] = 00000005
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000003
# KERNEL: Testing ALU: result= 00000009, (in1=00000004, in2=00000005, ALUop=0)
# KERNEL: [Fetch] PC = 00000009, Instruction = 00000000
# KERNEL: [Decode] Instruction = 34380000, Rs = 14, Rt =  0, Rd =  0, A = 00000003, B = 00000004, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000001
# KERNEL: [Memory] Data = 00000001, MemRead = 0, MemWrite = 0, Data_wb = 00000001
# KERNEL: Testing ALU: result= 00000009, (in1=00000004, in2=00000005, ALUop=0)
# KERNEL: [Fetch] PC = 00000009, Instruction = 00000000
# KERNEL: [Decode] Instruction = 34380000, Rs = 14, Rt =  0, Rd =  0, A = 00000003, B = 00000004, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000001
# KERNEL: [Memory] Data = 00000001, MemRead = 0, MemWrite = 0, Data_wb = 00000001
# KERNEL: PCsrc= 2
# KERNEL: Reg[0] = 00000004
# KERNEL: Reg[1] = 00000005
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000000
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000003
# KERNEL: Testing ALU: result= 00000000, (in1=00000000, in2=00000000, ALUop=0)
# KERNEL: [Fetch] PC = 00000003, Instruction = 14440005
# KERNEL: [Decode] Instruction = 00000000, Rs =  0, Rt =  0, Rd =  0, A = 00000004, B = 00000004, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000009
# KERNEL: [Memory] Data = 00000009, MemRead = 0, MemWrite = 0, Data_wb = 00000009
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000004
# KERNEL: Reg[1] = 00000005
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000001
# KERNEL: Reg[4] = 00000000
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000003
# KERNEL: Testing ALU: result= 00000004, (in1=00000004, in2=00000004, ALUop=2)
```

```
# KERNEL: Testing ALU: result= 00000004, (in1=00000004, in2=00000004, ALUop=2)
# KERNEL: [Fetch] PC = 00000004, Instruction = 11dc0003
# KERNEL: [Decode] Instruction = 14440005, Rs =  1, Rt =  0, Rd =  1, A = 00000005, B = 00000004, Imm32 = 00000005
# KERNEL: [Execute]ALU_result= 00000000
# KERNEL: [Memory] Data = 00000000, MemRead = 0, MemWrite = 0, Data_wb = 00000000
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000004
# KERNEL: Reg[1] = 00000005
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000001
# KERNEL: Reg[4] = 00000009
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000003
# KERNEL: Testing ALU: result= 0000000a, (in1=00000005, in2=00000005, ALUop=0)
# KERNEL: [Fetch] PC = 00000005, Instruction = 38000003
# KERNEL: [Decode] Instruction = 11dc0003, Rs =  7, Rt =  0, Rd =  7, A = 00000000, B = 00000004, Imm32 = 00000003
# KERNEL: [Execute]ALU_result= 00000004
# KERNEL: [Memory] Data = 00000004, MemRead = 0, MemWrite = 0, Data_wb = 00000004
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000004
# KERNEL: Reg[1] = 00000005
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000001
# KERNEL: Reg[4] = 00000009
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000003
# KERNEL: Testing ALU: result= 00000003, (in1=00000000, in2=00000003, ALUop=2)
# KERNEL: [Fetch] PC = 00000006, Instruction = 00000000
# KERNEL: [Decode] Instruction = 38000003, Rs =  0, Rt =  0, Rd =  0, A = 00000004, B = 00000004, I
# KERNEL: [Execute]ALU_result= 0000000a
# KERNEL: [Memory] Data = 0000000a, MemRead = 0, MemWrite = 0, Data_wb = 0000000a
# KERNEL: PCsrc= 1
# KERNEL: Reg[0] = 00000004
# KERNEL: Reg[1] = 00000005
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000001
# KERNEL: Reg[4] = 00000009
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000003
# KERNEL: Testing ALU: result= 00000000, (in1=00000000, in2=00000000, ALUop=0)
# KERNEL: [Fetch] PC = 00000009, Instruction = 14000003
# KERNEL: [Decode] Instruction = 00000000, Rs =  0, Rt =  0, Rd =  0, A = 00000004, B = 00000004, I
# KERNEL: [Execute]ALU_result= 00000003
```

```
# KERNEL: Testing ALU: result= 00000003, (in1=00000000, in2=00000003, ALUop=2)
# KERNEL: [Fetch] PC = 00000006, Instruction = 00000000
# KERNEL: [Decode] Instruction = 38000003, Rs =  0, Rt =  0, Rd =  0, A = 00000004, B = 00000004, Imm32 = 00000003
# KERNEL: [Execute]ALU_result= 0000000a
# KERNEL: [Memory] Data = 0000000a, MemRead = 0, MemWrite = 0, Data_wb = 0000000a
# KERNEL: PCsrc= 1
# KERNEL: Reg[0] = 00000004
# KERNEL: Reg[1] = 00000005
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000001
# KERNEL: Reg[4] = 00000009
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000003
# KERNEL: Testing ALU: result= 00000000, (in1=00000000, in2=00000000, ALUop=0)
# KERNEL: [Fetch] PC = 00000009, Instruction = 14000003
# KERNEL: [Decode] Instruction = 00000000, Rs =  0, Rt =  0, Rd =  0, A = 00000004, B = 00000004, Imm32 = 00000000
# KERNEL: [Execute]ALU_result= 00000003
# KERNEL: [Memory] Data = 00000003, MemRead = 0, MemWrite = 0, Data_wb = 00000003
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000004
# KERNEL: Reg[1] = 0000000a
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000001
# KERNEL: Reg[4] = 00000009
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000000
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000003
# KERNEL: Testing ALU: result= 00000004, (in1=00000004, in2=00000004, ALUop=2)
```

```
# KERNEL: Reg[14] = 00000003
# KERNEL: Testing ALU: result= 00000004, (in1=00000004, in2=00000004, ALUop=2)
# KERNEL: [Fetch] PC = 0000000a, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = 14000003, Rs =  0, Rt =  0, Rd =  0, A = 00000004, B = 00000004, Imm32 = 00000003
# KERNEL: [Execute]ALU_result= 00000000
# KERNEL: [Memory] Data = 00000000, MemRead = 0, MemWrite = 0, Data_wb = 00000000
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000004
# KERNEL: Reg[1] = 0000000a
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000001
# KERNEL: Reg[4] = 00000009
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000003
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000003
# KERNEL: Testing ALU: result= 00000007, (in1=00000004, in2=00000003, ALUop=0)
# KERNEL: [Fetch] PC = 0000000b, Instruction = xxxxxxxx
# KERNEL: [Decode] Instruction = xxxxxxxx, Rs =  x, Rt =  x, Rd =  x, A = xxxxxxxx, B = xxxxxxxx, Imm32 = xxxxxxxx
# KERNEL: [Execute]ALU_result= 00000004
# KERNEL: [Memory] Data = 00000004, MemRead = 0, MemWrite = 0, Data_wb = 00000004
# KERNEL: PCsrc= 0
# KERNEL: Reg[0] = 00000004
# KERNEL: Reg[1] = 0000000a
# KERNEL: Reg[2] = 00000000
# KERNEL: Reg[3] = 00000001
# KERNEL: Reg[4] = 00000009
# KERNEL: Reg[5] = 00000000
# KERNEL: Reg[6] = 00000000
# KERNEL: Reg[7] = 00000003
# KERNEL: Reg[8] = 00000000
# KERNEL: Reg[9] = 00000000
# KERNEL: Reg[10] = 00000000
# KERNEL: Reg[11] = 00000000
# KERNEL: Reg[12] = 00000000
# KERNEL: Reg[13] = 00000000
# KERNEL: Reg[14] = 00000003
```

Conclusion

This project successfully demonstrates the design, implementation, and verification of a 32-bit pipelined RISC processor using Verilog. Through a modular and systematic approach, the processor supports a defined instruction set with arithmetic, logical, memory access, and control flow operations, including subroutine handling via CLL and JR.

Each pipeline stage—Fetch, Decode, Execute, Memory, and Write-Back—was implemented with careful attention to data and control hazards, pipeline stalls, and forwarding mechanisms to ensure correct and efficient instruction execution. Simulation results confirm the processor's correct behavior across various instruction scenarios, including function calls, branching, and memory operations.

Overall, the project provided a deep understanding of processor microarchitecture, pipelining concepts, and hardware description in Verilog. It also emphasized the importance of verification through testbenches and waveform analysis to ensure functional correctness and performance. This foundation serves as a critical step toward more advanced CPU designs and architectural exploration.