Marmara University - Faulty of Engineering

Department of Computer Engineering

CSE3038 Computer Organization and Architecture (Spring 2024)

Submit Date: 25/05/2024.

Enhanced MIPS Single Cycle Datapath

Student Number (ID)	Name	Surname
150120998	Abdelrahman	Zahran
150120997	Mohamed Nael	Ayoubi
150120012	Kadir	Bat
150121021	Feyzullah	Asıllıoğlu

Sections Of the Report: -

Section (1): Introduction.

Introduction

The purpose of this report is to document the design and implementation process of an enhanced single-cycle MIPS processor, as part of the CSE 3038 – Computer Organization course project for Spring 2024. This project required the extension of the basic MIPS single-cycle datapath to support six additional instructions selected from a predefined set of 22 instructions, spanning R-type, I-type, and J-type formats.

To achieve this, we meticulously designed a revised single-cycle datapath and control unit that integrate these new instructions without compromising the functionality of the existing MIPS instruction set. The project specifications mandated the use of the ModelSim simulator for developing and testing our implementation, and the enhanced processor was implemented in Verilog HDL.

Special Instruction Set

1. **brnv (R-type, funct=21)**

   - **Syntax: brnv $rs**
   - **Meaning:** Branches to the address found in register **$rs** if the **V** bit in the Status register is **0**.
   - **Details:** This instruction is used for conditional branching based on the overflow flag (**V**). If there is no overflow (**V=0**), the program counter (PC) is set to the address contained in register **$rs**.

2. **jmnor (R-type, funct=37)**

- **Syntax: jmnor $rs, $rt**
- **Meaning:** Jumps to the address found in memory location indexed by the NOR of registers **$rs** and **$rt**. The link address is stored in register **$31**.
- **Details:** This instruction calculates the NOR of the values in registers **$rs** and **$rt**, uses this result as a memory address, and jumps to the address stored at that memory location. The current PC + 4 (the address of the next instruction) is saved in the link register **$31**.

3. **nandi (I-type, opcode=16)**

- **Syntax: nandi $rt, $rs, Label**
- **Meaning:** Performs a logical NAND operation between the value in register **$rs** and a zero-extended immediate value (**Label**), and stores the result in register **$rt**.
- **Details:** This instruction takes the value in **$rs**, performs a NAND operation with the immediate value (extended to 32 bits with zeros), and stores the result in **$rt**.

4. **bgtzal (I-type, opcode=33)**

- **Syntax: bgtzal $rs, Label**
- **Meaning:** If the value in register **$rs** is greater than zero, branches to a PC-relative address. The link address is stored in register **$25**.
- **Details:** This conditional branch instruction checks if the value in **$rs** is greater than zero. If true, it calculates a target address using the PC-relative offset (**Label**), sets the PC to this address, and stores the link address (current PC + 4) in register **$25**.

5. **jrsal (I-type, opcode=17)**

- **Syntax: jrsal $rs**
- **Meaning:** Jumps to the address found in memory, where the memory address is stored in register **$rs**, and stores the link address in memory at the location indexed by **$rs**.
- **Details:** This instruction uses the value in **$rs** as a pointer to a memory location containing the jump target address. It jumps to this target address and saves the link address (current PC + 4) back into the memory location indexed by **$rs**.

6. **balv (J-type, opcode=32)**

- **Syntax: balv Target**
- **Meaning:** If the **V** bit in the Status register is **1**, branches to a pseudo-direct address (similar to the **jal** instruction), and stores the link address in register **$31**.
- **Details:** This conditional branch instruction checks if the overflow bit (**V**) is set. If true, it forms a target address using the pseudo-direct addressing method, sets the PC to this address, and stores the link address (current PC + 4) in the link register **$31**.

General Explanation

These instructions are part of a project to extend the MIPS single-cycle implementation. The instructions cover a range of operations including conditional branches (**brnv**, **bgtzal**), jumps (**jmnor**, **jrsal**, **balv**), and logical operations (**nandi**). Each instruction has specific syntax, opcode/funct values, and operational details, impacting how the processor's control unit and datapath are designed and implemented in Verilog HDL.

Requirements and Implementation

- **Design Requirements:** The project requires implementing a set of 6 specific instructions, maintaining compatibility with existing MIPS instructions, and enhancing the processor's control and datapath to support these new instructions.
- **Implementation:** The implementation involves modifying the single-cycle datapath, creating new control signals, and ensuring that each instruction performs as specified. The project is to be coded in Verilog using the ModelSim simulator, with a demo session for validation and grading.

Final Notes

These instructions are to be implemented and tested in a single cohesive processor design. Students must document their design process, provide simulations, and submit their code and reports by the specified deadline.

Enhanced Single-Cycle Datapath Design

The enhanced single-cycle datapath design incorporated several key modifications to support the new instructions:

- **Status Register**: A Status register was added to the MIPS datapath to monitor the zero (Z), negative (N), and overflow (V) flags from ALU operations, crucial for executing conditional branch instructions like brv and bgtzal.
- **Control Unit**: The control unit was revised to generate appropriate control signals for the new instructions, ensuring seamless integration with existing MIPS instructions.
- **Datapath Modifications**: Modifications were made to the datapath to handle new data paths required for instructions like jsp, which involves indirect addressing via the stack pointer register ($sp).

Verification and Simulation

We verified our design by simulating each new instruction in ModelSim, ensuring correct execution and adherence to MIPS conventions. Our simulation setup included a comprehensive testbench encompassing the register file, data memory, and instruction memory. Example runs for each instruction were documented to demonstrate the functionality and correctness of our implementation.

Section (2): Single Cycle Datapath Diagram.

This single-cycle Datapath diagram illustrates the architecture of a simple processor that executes one instruction per clock cycle. Here's a detailed description of each component and their functions:

Main Components:

1. **Program Counter (PC):**
    - The PC holds the address of the next instruction to be executed.
    - It is incremented by 4 after each instruction fetch, pointing to the next sequential instruction.
2. **Instruction Memory:**
    - This block stores the program instructions.
    - The instruction is fetched from the memory address pointed to by the PC.
3. **Control Unit:**
    - The control unit decodes the fetched instruction and generates control signals for the other components.
    - It interprets the opcode of the instruction to determine the operation to be performed and sets the control lines accordingly.
4. **Registers:**
    - This block contains the register file, a small, fast storage area containing a set of registers.
    - It has two read ports (Read Register 1 and Read Register 2) and one write port (Write Register).
    - The data from the registers are used as inputs to the ALU.
5. **ALU (Arithmetic Logic Unit):**
    - The ALU performs arithmetic and logical operations on the input data.
    - It takes two operands and a control signal that specifies the operation to be performed.
    - The result is then sent to either the data memory or the register file.
6. **Data Memory:**
    - This block is used for reading from or writing to memory locations during load and store instructions.
    - It receives the address from the ALU result and the data to be written from the register file.
7. **Sign Extension:**
    - This unit extends the 16-bit immediate value to a 32-bit value for use in arithmetic operations.
    - It is used for instructions that involve immediate values.
8. **Zero Extension:**
    - This unit extends a smaller bit value to a 32-bit value, filling the upper bits with zeros.
9. **Multiplexers (MUX):**
    - Several multiplexers are used throughout the Datapath to select between different input values based on control signals.
    - Key multiplexers include those that select:

- The source of the second ALU operand (either a register value or an immediate value).
- The data to be written to the register file (from the ALU result or from memory).
- The next value of the PC (for branching and jumping).

10. **Adder:**
    - There are two adders in the Datapath:
        - One for incrementing the PC by 4.
        - Another for calculating branch target addresses.
11. **Shift Left 2:**
    - This unit shifts the immediate value left by 2 bits.
    - Used in calculating branch target addresses.
12. **Status Register:**
    - This block holds flags for zero (Z), negative (N), overflow (V), and carry out (C) from the ALU.
    - These flags are used to evaluate conditions for conditional branching.
13. **Logical AND, OR, NOT Gates:**
    - These gates are used to determine the control logic for branching and jumping.
    - They combine the status flags with the control signals to make decisions.

Detailed Data Flow:

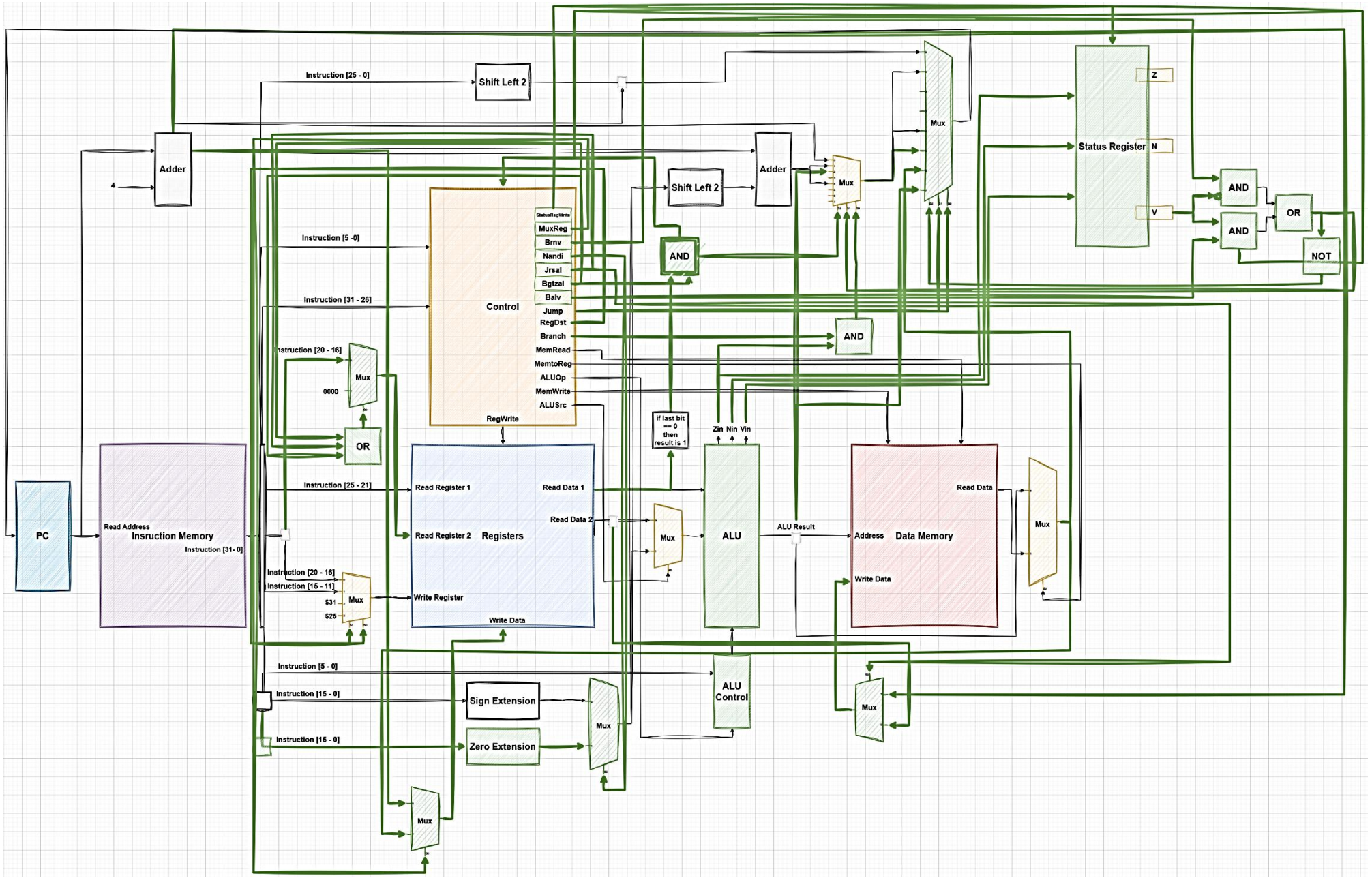
1. **Instruction Fetch:**
    - The PC provides the address to the instruction memory.
    - The instruction memory fetches the instruction located at that address.
    - The PC is incremented by 4 for the next instruction fetch.
2. **Instruction Decode:**
    - The fetched instruction is divided into fields (opcode, source registers, destination register, immediate value).
    - The control unit decodes the opcode and generates the necessary control signals.
3. **Register Read:**
    - The source registers specified in the instruction are read from the register file.
    - These values are provided to the ALU as inputs.
4. **Execution:**
    - The ALU performs the specified operation (addition, subtraction, logical operations, etc.) based on the control signals.
    - If the instruction uses an immediate value, it is sign-extended and provided as an input to the ALU.
5. **Memory Access:**
    - For load and store instructions, the ALU result provides the address for the data memory.
    - Load instructions read data from memory into the register file.
    - Store instructions write data from the register file to memory.
6. **Write Back:**

- ♠ The result from the ALU or data memory is written back to the register file, completing the instruction cycle.

Branching and Jumping:

- The control unit and status register determine whether a branch or jump should occur based on the instruction and ALU flags.
- For branches, the target address is calculated by adding the shifted immediate value to the incremented PC.
- For jumps, the target address is specified directly by the instruction.

This single-cycle Datapath ensures that each instruction completes in one clock cycle, leading to a straightforward and efficient design suitable for simple processors. However, it may not be as efficient for more complex instructions, as every instruction takes the same amount of time regardless of complexity.


Instruction [25 - 0]
Shift Left 2
Adder
4
Instruction [5 -0]
Instruction [31 - 26]
Control
StatusRegWrite
MuxReg
Brnv
Nandi
Jrsal
Bgtzal
Balv
Jump
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Shift Left 2
Adder
Mux
AND
AND
Status Register
Z
N
V
AND
OR
NOT
Instruction [20 - 16]
0000
Mux
OR
PC
Read Address
Insruction Memory
Instruction [31- 0]
Instruction [25 - 21]
Read Register 1
Read Register 2
Registers
Write Register
Write Data
Read Data 1
Read Data 2
Instruction [20 - 16]
Instruction [15 - 11]
$31
$25
Mux
if last bit == 0 then result is 1
Zin Nin Vin
ALU
ALU Result
Address
Data Memory
Read Data
Write Data
Mux
Mux
Instruction [5 - 0]
Instruction [15 - 0]
Sign Extension
Zero Extension
Mux
ALU Control
Mux
Mux

Section (3): Instruction (1): **JRSAL**

Instruction "JRSAL," which is an I-type instruction with the opcode 17. The instruction "jrsal $rs" involves jumping to an address found in memory (Data Memory[$rs]) where the address is specified by the value in register $rs. Additionally, the current link address (PC + 4) is stored back in memory at the address specified by the value in register $rs. Here is an extensive explanation of the Datapath and the execution cycle order for this instruction.

Execution Cycle Order:

1. **Instruction Fetch:**
   - ♠ The PC provides the address to the Instruction Memory.
   - ♠ The Instruction Memory fetches the instruction at the PC address.
   - ♠ The fetched instruction is then passed to the Control Unit.
2. **Instruction Decode:**
   - ♠ The Control Unit decodes the opcode of the fetched instruction.
   - ♠ The jrsal instruction is identified by the opcode 17.
   - ♠ Control signals for Jrsal are asserted, which include setting the MemRead, MemWrite, and jump control signals appropriately.
3. **Register Read:**
   - ♠ The source register $rs is read from the Register File.
   - ♠ The value in $rs specifies the address from which to read the jump target in Data Memory.
4. **Address Calculation (if needed):**
   - ♠ In this case, no ALU operation is needed to modify the address, as it is directly used to access Data Memory.
5. **Memory Access:**
   - ♠ The Data Memory is accessed using the address from $rs.
   - ♠ The content at Data Memory[$rs] is fetched, which is the jump target address.
   - ♠ Simultaneously, the PC + 4 value (link address) is prepared to be written back to Data Memory at the same address.
6. **PC Update:**
   - ♠ The PC is updated with the new jump target address fetched from Data Memory[$rs].
7. **Write Back:**
   - ♠ The link address (PC + 4) is written back to Data Memory at the address specified by $rs.
   - ♠ This effectively stores the return address for the jump instruction.

Signal Flow in Datapath for JRSAL:

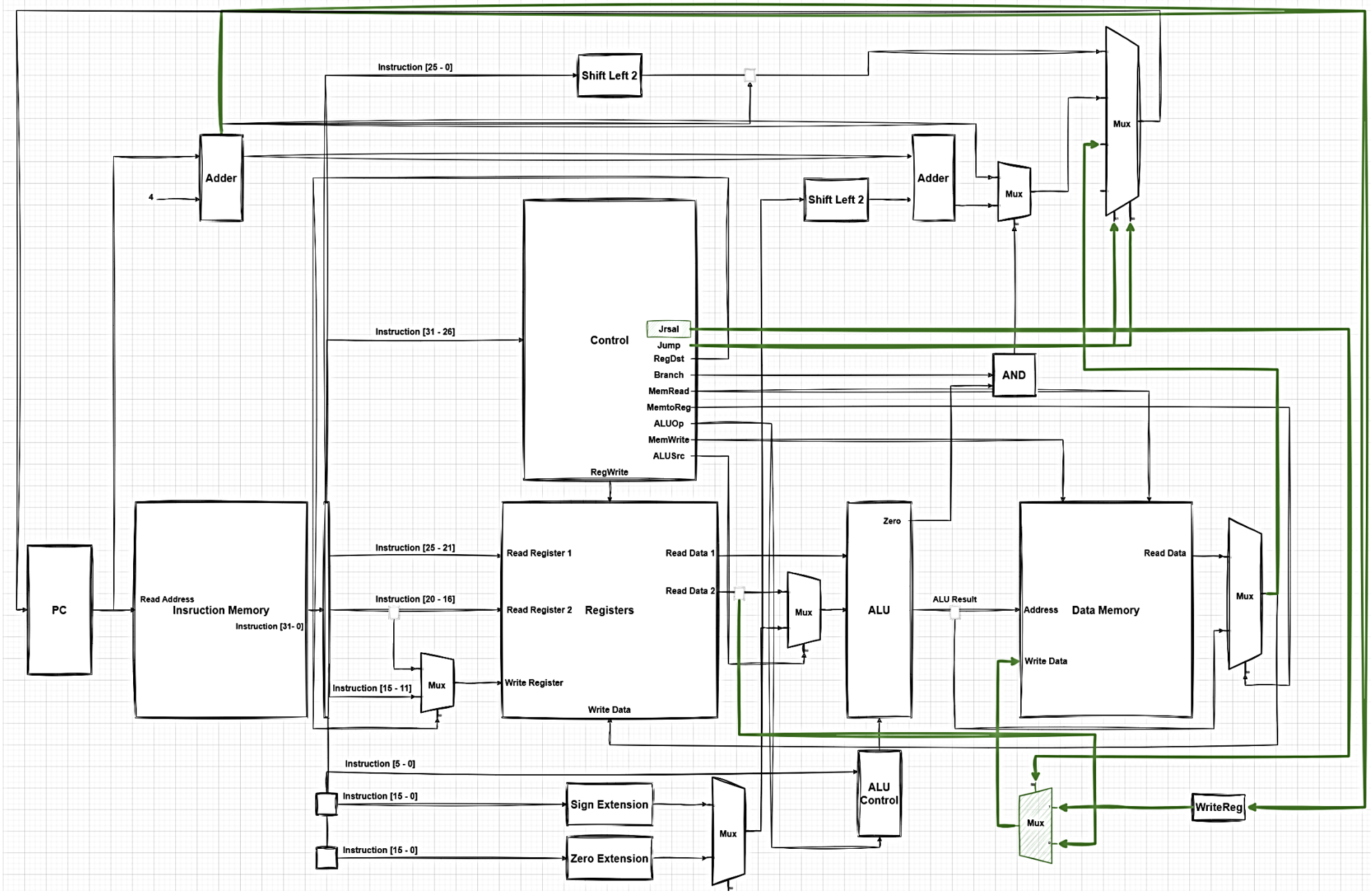
- **PC** → Instruction Memory (Fetches jrsal $rs instruction).
- **Instruction [25-21]** → Registers (Reads $rs).
- **Registers[$rs]** → Data Memory Address (Fetches jump target address).
- **PC + 4** → Data Memory (Writes link address back to Data Memory at the location specified by $rs).
- **Data Memory[$rs]** → MUX → PC (Updates the PC with the jump target address).

Control Signals Specific to JRSAL:

- **RegWrite**: Disabled.
- **MemRead**: Enabled (to read jump target from Data Memory).
- **MemWrite**: Enabled (to write the link address to Data Memory).
- **ALUSrc**: Not applicable.
- **ALUOp**: Not applicable.
- **Jrsal**: Specific control signal to handle the jrsal logic.
- **PCSrc**: Control signal to select the jump target address for updating the PC.

This execution cycle ensures that the processor jumps to the address found in memory at the location specified by $rs and saves the return address (PC + 4) back into the memory at the same location, achieving the desired behavior for the jrsal instruction.

Instruction: JRSAL I-type opcode=17 jrsal $rs jumps to address found in memory where the memory address is written in register $rs and link address is stored in memory (DataMemory[$rs]).



Section (4): Instruction (2): **NANDI**

Instruction "NANDI," which is an I-type instruction with the opcode 16. The instruction "nandi $rt, $rs" computes the logical NAND of the value in register $rs and a zero-extended immediate value, then stores the result in register $rt. Here's an extensive explanation of the Datapath and the execution cycle order for this instruction.

Execution Cycle Order:

1. **Instruction Fetch:**
   - The PC provides the address to the Instruction Memory.
   - The Instruction Memory fetches the instruction at the PC address.
   - The fetched instruction is then passed to the Control Unit.
2. **Instruction Decode:**
   - The Control Unit decodes the opcode of the fetched instruction.
   - The nandi instruction is identified by the opcode 16.
   - Control signals for Nandi are asserted, which include setting the ALUSrc, RegWrite, and ALU operation control signals appropriately.
3. **Register Read:**
   - The source register $rs is read from the Register File.
   - The value in $rs serves as one input to the ALU.
4. **Immediate Value Zero Extension:**
   - The 16-bit immediate value from the instruction is zero-extended to 32 bits.
   - This extended value serves as the second input to the ALU.
5. **ALU Operation:**
   - The ALU performs the logical NAND operation between the value read from $rs and the zero-extended immediate value.
   - The operation is controlled by the Nandi control signal, which configures the ALU accordingly.
6. **Write Back:**
   - The result from the ALU is written back to the destination register $rt in the Register File.
   - This completes the execution of the nandi instruction.

Signal Flow in Datapath for NANDI:

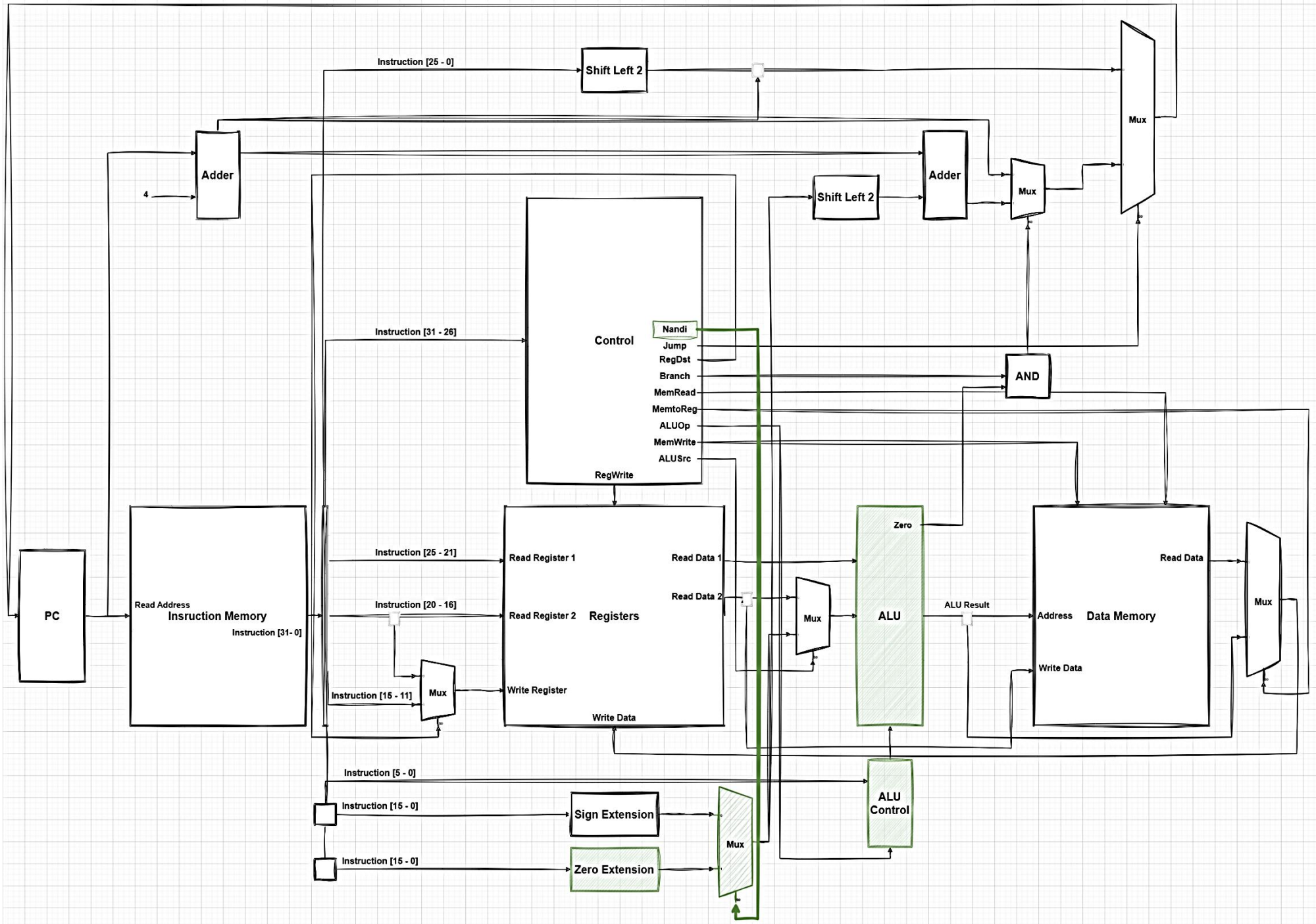
- **PC** → Instruction Memory (Fetches nandi $rt, $rs, immediate instruction).
- **Instruction [25-21]** → Registers (Reads $rs).
- **Instruction [20-16]** → Control Unit (Identifies $rt for the destination register).
- **Instruction [15-0]** → Zero Extension (Extends the immediate value to 32 bits).
- **Registers[$rs]** → ALU (First operand).
- **Zero Extension** → ALU (Second operand).
- **ALU** → Performs NAND operation between the operands.
- **ALU Result** → Registers[$rt] (Writes the result back to the destination register).

Control Signals Specific to NANDI:

- **RegWrite**: Enabled (to write the result back to the destination register).
- **ALUSrc**: Enabled (to use the zero-extended immediate value as the second ALU operand).
- **ALUOp**: Configured for NAND operation.
- **Nandi**: Specific control signal to handle the nandi logic.

By following this execution cycle, the nandi instruction successfully computes the logical NAND of the value in register $rs and the zero-extended immediate value, then stores the result in register $rt. This single-cycle Datapath ensures that each instruction completes in one clock cycle, maintaining efficiency and simplicity.

Instruction: nandi I-type opcode=16 nandi $rt, $rs, Put the logical NAND of register $rs and the zero - extended Label immediate into register $rt.



Section (5): Instruction (3): **BRNV**

The **brnv** instruction is a branch instruction used in a single-cycle datapath. It is an R-type instruction that causes the program to branch to the address contained in register **$rs** if the overflow flag (**V**) in the status register is not set (i.e., **V = 0**). Let's break down the detailed execution of the **brnv** instruction step by step.

Execution Cycle Order:

1. **Instruction Fetch**
   - ♠ The Program Counter (PC) provides the address to the Instruction Memory.
   - ♠ The Instruction Memory fetches the **brnv** instruction.
   - ♠ The fetched instruction is passed to the next stage in the datapath.
2. **Instruction Decode**
   - ♠ The instruction is decoded by the Control unit.
   - ♠ The instruction fields are extracted:
     - Opcode for **brnv** (R-type)
     - **rs** (source register)
     - **funct** (21 for **brnv**)
3. **Register Read**
   - ♠ The register file is accessed to read the contents of register **rs**.
   - ♠ The read data from **rs** contains the target address for the branch.
4. **Status Register Check**
   - ♠ The Status Register is checked, particularly the **V** (overflow) bit.
   - ♠ An AND gate is used to determine if **V** = 0 (no overflow).
5. **Branch Decision**
   - ♠ If **V** = 0, the branch condition is true, and the PC will be updated.
   - ♠ If **V** $\neq$ 0, the PC will continue to the next sequential instruction.
6. **PC Update**
   - ♠ If the branch is taken, the PC is updated with the address found in register **rs**.
   - ♠ If the branch is not taken, the PC increments to the address of the next instruction.

Detailed Signal Flow

1. **Control Unit**
   - Decodes the instruction and sets the control signals.
   - Ensures the **ALUOp** is set for the **brnv** operation.
   - Sets the branch control signal based on the instruction.
2. **Register File**
   - **Read Register 1** is set to **rs** to read the branch target address.
   - **Read Data 1** outputs the value of **rs**, which is the target address for branching.

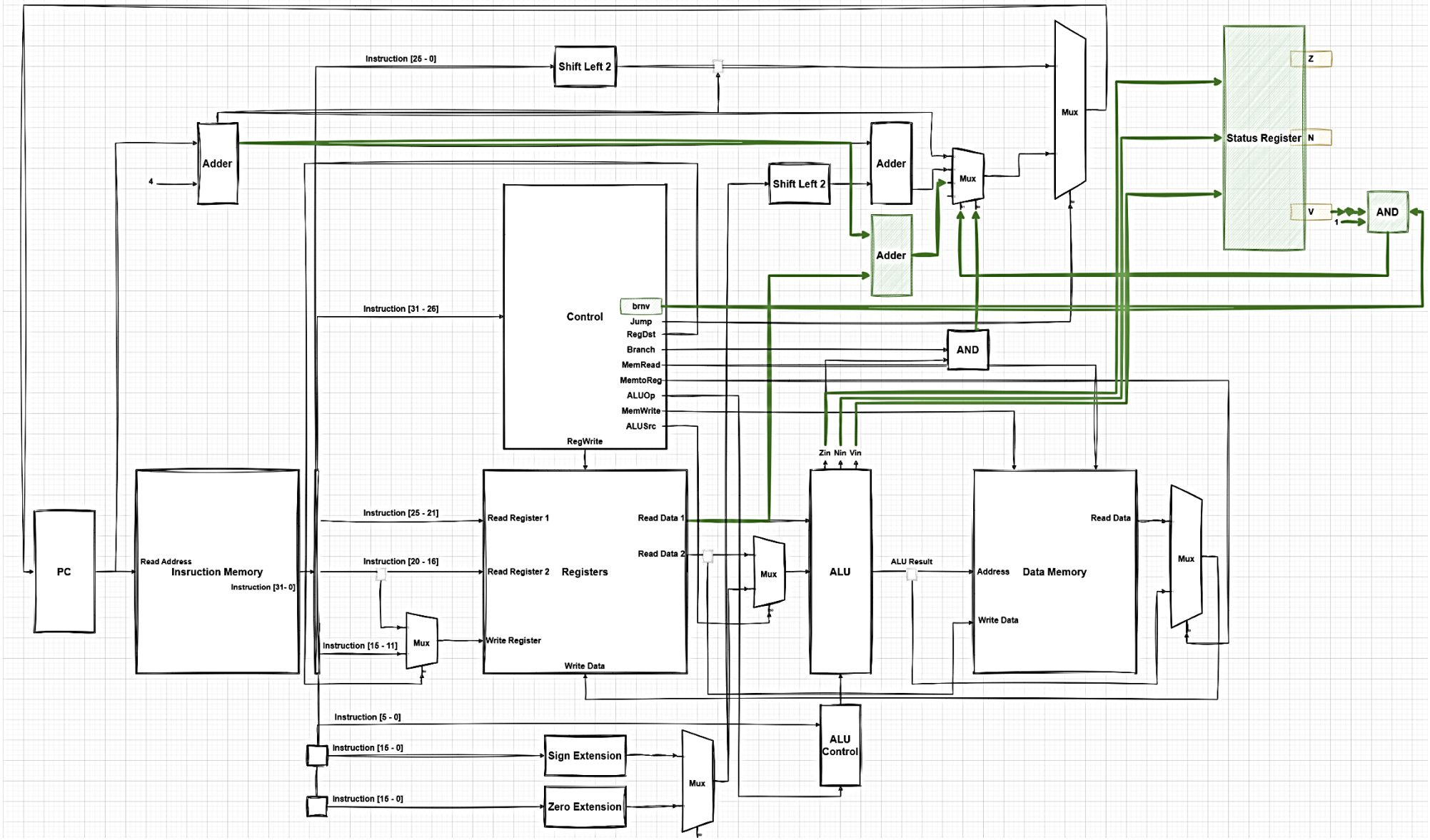
3. **Status Register**
    - The **V** bit is examined.
    - An AND gate checks if **V** is 0, indicating no overflow.
4. **ALU**
    - Receives the value from **rs** and prepares for potential branching.
    - The actual branching logic (whether to update the PC) is handled by the control logic outside the ALU.
5. **Program Counter (PC)**
    - The PC is conditionally updated based on the result of the AND gate check.
    - If **V** = 0, the new address from **rs** is loaded into the PC.
    - If **V** ≠ 0, the PC increments to the next instruction address.

Summary of Single Cycle Execution

1. **Fetch:** Instruction is fetched from memory.
2. **Decode:** Instruction is decoded, control signals set, and registers read.
3. **Check Status:** Status register's **V** bit is checked.
4. **Branch Decision:** If **V** = 0, branch to address in **rs**; otherwise, continue to next instruction.
5. **Update PC:** Based on the branch decision, the PC is updated appropriately.

This detailed explanation shows how the **brnv** instruction executes in one clock cycle in a single-cycle datapath. The key operations—fetching the instruction, decoding it, reading the necessary registers, checking the status register, and updating the PC—are all completed within one clock cycle.

Instruction: brnv R-type funct=21 brnv $rs if Status [V] = 0, branches to address found in register $rs.



Section (6): Instruction (4): **BALV**

The **balv** instruction is a hypothetical J-type instruction that branches to a pseudo-direct address if a specific condition is met, storing the link address (the address of the next instruction) in register 31.

Execution Cycle Order:

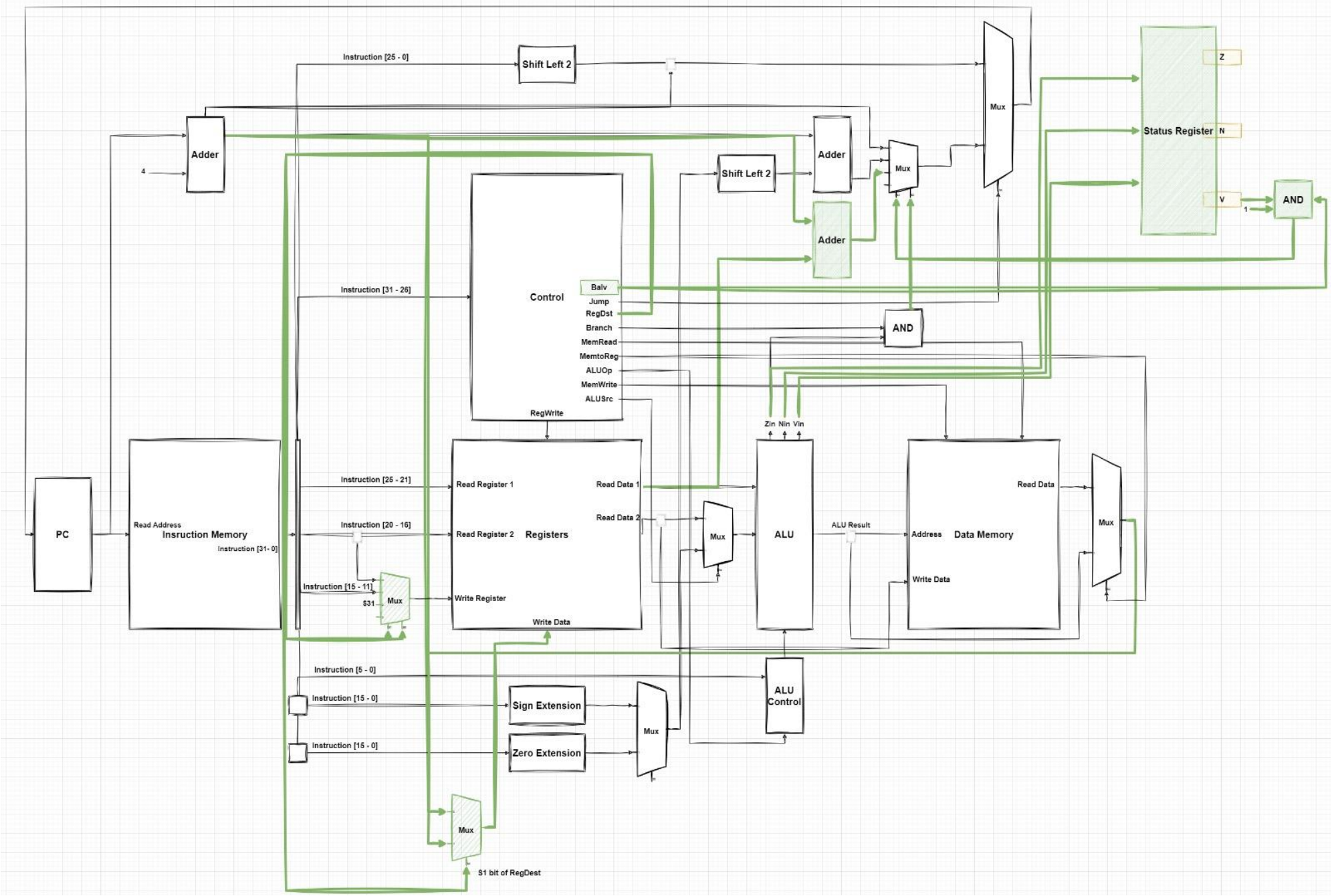
1. **Instruction Fetch (IF):**
    - The Program Counter (PC) provides the address to the Instruction Memory.
    - The Instruction Memory fetches the instruction based on the PC address and sends it to the instruction register.
    - The PC is incremented by 4 using the Adder, preparing for the next instruction fetch.
2. **Instruction Decode (ID):**
    - The fetched instruction is divided into its respective fields.
    - For **balv**, the opcode (32) is identified.
    - The target address (pseudo-direct) is extracted from the instruction bits **[25-0]**.
3. **Control Signal Generation:**
    - The Control Unit decodes the opcode and generates the necessary control signals.
    - For **balv**, the relevant control signals are activated: **Balv**, **Jump**, **RegDst**, **ALUSrc**, **MemtoReg**, **RegWrite**, and others as required.
4. **Read Registers:**
    - Although not used in **balv**, the register file reads the values of the source registers specified in the instruction.
    - Read Register 1 and Read Register 2 are fetched from the Registers block.
5. **Status Check (V bit):**
    - The status register value is checked. If the overflow bit **V** is set (**V = 1**), the branching will occur.
    - This value influences the **AND** gate to determine the branch condition.
6. **ALU Operations:**
    - ALU is not actively used for computation in **balv** but could be involved for address calculation or condition checking.
    - The result from ALU is used for the next instruction.
7. **Branch Address Calculation:**
    - The target address is shifted left by 2 (to account for word alignment) using the Shift Left 2 unit.
    - The target address is then combined with the upper bits of the PC using an Adder to form the pseudo-direct jump address.
8. **Write Back (Register 31):**
    - The link address (PC + 4) is stored in register 31.
    - This ensures that the return address after the jump is saved for future use.
9. **PC Update:**
    - If the status bit **V** is set, the PC is updated to the target address (branch occurs).
    - If **V** is not set, the PC remains unchanged and proceeds to the next sequential instruction.

Detailed Signals and Data Paths:

- **Instruction [31-26]** goes to the Control Unit to generate control signals.
- **Instruction [25-0]** is used to calculate the target address.
- The Control Unit sets the **Balv** signal, indicating a branch instruction.
- The Adder and Shift Left 2 components handle address calculation for the branch target.
- The ALU and Mux blocks assist in handling various data paths and selection based on control signals.
- **RegWrite** and **Write Register** signals ensure the link address is written into register 31.

This execution order ensures that the **balv** instruction correctly branches to the target address if the overflow condition is met, and saves the return address for future reference.

Instruction: balv J-type opcode=32 balv Target if Status [V] = 1, branches to pseudo-direct address (formed as jal does), link address is stored in register 31



Section (7): Instruction (5): **JMNOR**

The **jmnor** instruction, which is an R-type instruction with a specific function code (**funct=37**). This instruction performs a NOR operation on the contents of two registers (**$rs** and **$rt**), uses the result as an address to jump to, and stores the link address (the return address) in register **$31**.

Execution Cycle Order:

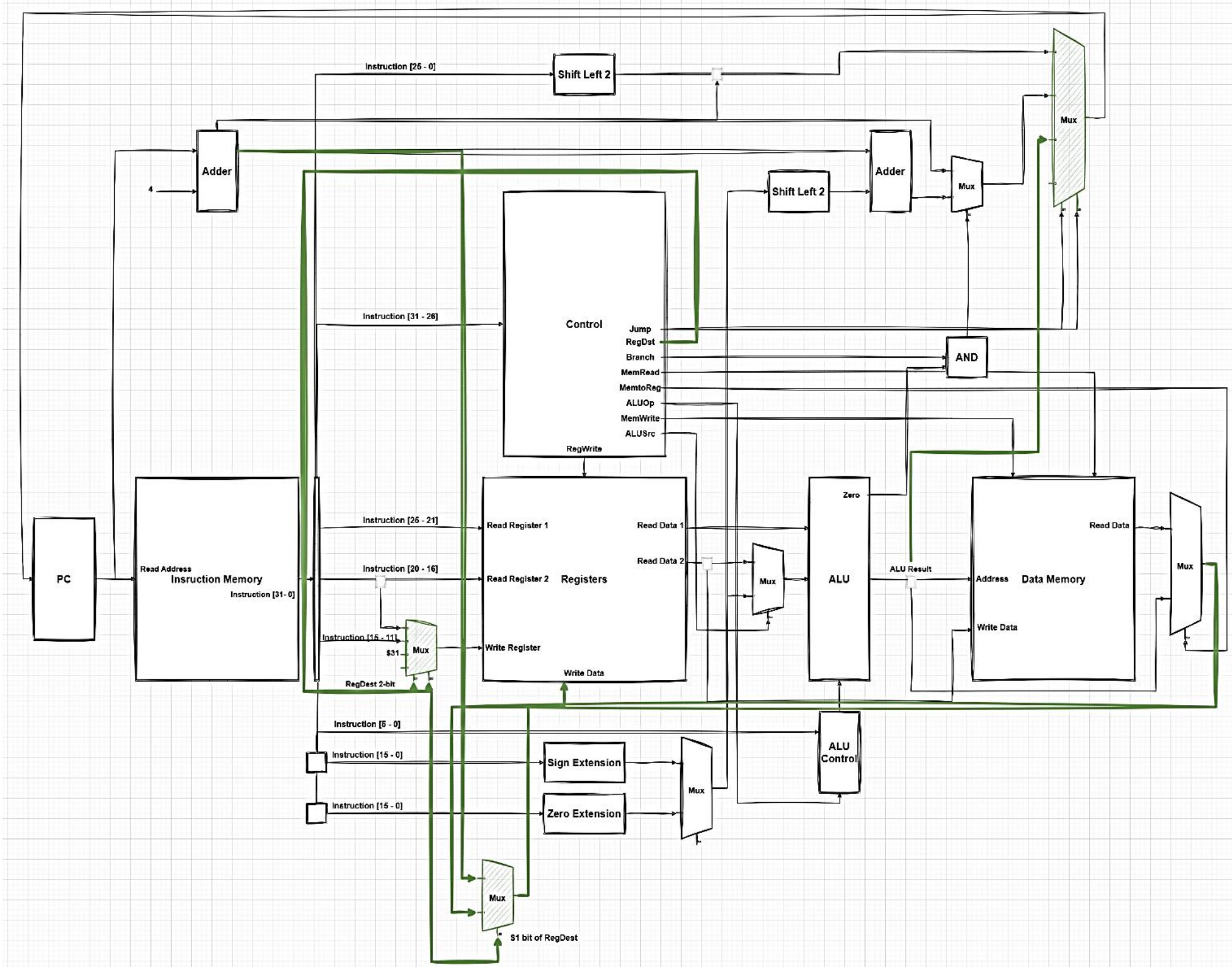
1. **Instruction Fetch (IF):**
    - The Program Counter (PC) provides the address to the Instruction Memory.
    - The Instruction Memory fetches the instruction based on the PC address and sends it to the instruction register.
    - The PC is incremented by 4 using the Adder, preparing for the next instruction fetch.
2. **Instruction Decode (ID):**
    - The fetched instruction is divided into its respective fields.
    - For **jmnor**, the opcode identifies it as an R-type instruction, and the function code (**funct=37**) specifies the **jmnor** operation.
    - The source registers (**$rs** and **$rt**) are identified from the instruction fields **[25-21]** and **[20-16]**.
3. **Control Signal Generation:**
    - The Control Unit decodes the opcode and function code to generate the necessary control signals.
    - For **jmnor**, the relevant control signals are activated: **Jump**, **RegDst**, **ALUSrc**, **MemRead**, **MemtoReg**, **RegWrite**, and others as required.
4. **Read Registers:**
    - The register file reads the values of the source registers **$rs** and **$rt**.
    - These values are fetched from the Registers block as Read Data 1 and Read Data 2.
5. **ALU Operation (NOR):**
    - The values from Read Data 1 and Read Data 2 are sent to the ALU.
    - The ALU Control unit sets the operation to NOR based on the function code (**funct=37**).
    - The ALU performs the NOR operation on the values of **$rs** and **$rt**, producing the result.
6. **Memory Address Calculation:**
    - The result of the NOR operation is used as the memory address to fetch the jump target address.
    - This result is sent to the Data Memory as the address for reading.
7. **Data Memory Read:**
    - The Data Memory reads the value at the address computed by the NOR operation.
    - This read value is the target address for the jump.
8. **Write Back (Register 31):**
    - The link address (PC + 4) is stored in register 31 to save the return address.
    - The Control Unit ensures that the link address is written into **$31** through the appropriate control signals (**RegWrite**, **Write Register**, etc.).
9. **PC Update:**

- The PC is updated to the address read from the Data Memory.
- This causes the jump to the target address specified by the memory contents.

Detailed Signals and Data Paths:

- **Instruction [31-26]** goes to the Control Unit to generate control signals.
- **Instruction [25-21]** and **Instruction [20-16]** specify the source registers **$rs** and **$rt**.
- The Control Unit sets the necessary signals for the **jmnor** operation.
- The Register file reads the values of **$rs** and **$rt**.
- **Read Data 1** and **Read Data 2** are sent to the ALU.
- The ALU performs a NOR operation, and the result is used as the memory address.
- The Data Memory reads the value from the calculated address.
- The link address (PC + 4) is stored in register 31 through the appropriate control signals.
- The PC is updated to the target address read from memory.

This execution order ensures that the **jmnor** instruction correctly jumps to the address found in memory after performing the NOR operation on the specified registers, and saves the return address in register 31.


Instruction [25 - 0]
Shift Left 2
Mux
Adder
4
Adder
Shift Left 2
Adder
Mux
Instruction [31 - 26]
Control
Jump
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
AND
PC
Read Address
Insruction Memory
Instruction [31 - 0]
Instruction [25 - 21]
Instruction [20 - 16]
Instruction [15 - 11]
$31
Mux
RegDest 2-bit
Read Register 1
Read Register 2
Registers
Write Register
Write Data
Read Data 1
Read Data 2
Mux
Zero
ALU
ALU Result
Address
Data Memory
Write Data
Read Data
Mux
Instruction [5 - 0]
ALU Control
Instruction [15 - 0]
Sign Extension
Instruction [15 - 0]
Zero Extension
Mux
Mux
$1 bit of RegDest

Section (8): Instruction (6): **BGTZAL**

The **bgtzal** instruction is an I-type branch instruction that checks if the value in register **$rs** is greater than zero. If true, it branches to a PC-relative address, and the link address (return address) is stored in register **$25**. Here is the step-by-step execution order in this single-cycle datapath for the **bgtzal** instruction:

Execution Cycle Order:

1. **Instruction Fetch (IF):**
    - The Program Counter (PC) provides the address to the Instruction Memory.
    - The Instruction Memory fetches the instruction based on the PC address and sends it to the instruction register.
    - The PC is incremented by 4 using the Adder, preparing for the next instruction fetch.
2. **Instruction Decode (ID):**
    - The fetched instruction is divided into its respective fields.
    - The opcode (33) identifies it as an **bgtzal** instruction.
    - The source register (**$rs**) is identified from the instruction field **[25-21]**.
    - The immediate value (offset for PC-relative address) is extracted from the instruction field **[15-0]**.
3. **Control Signal Generation:**
    - The Control Unit decodes the opcode and generates the necessary control signals.
    - For **bgtzal**, the relevant control signals are activated: **Bgtzal**, **Branch**, **RegDst**, **ALUSrc**, **RegWrite**, and others as required.
4. **Read Registers:**
    - The register file reads the value of the source register **$rs**.
    - This value is fetched from the Registers block as Read Data 1.
5. **Sign-Extend Immediate:**
    - The 16-bit immediate value is sign-extended to 32 bits to form the branch offset.
    - The sign-extended value is then shifted left by 2 (to account for word alignment) using the Shift Left 2 unit.
6. **Branch Address Calculation:**
    - The branch address is calculated by adding the shifted offset to the incremented PC (PC + 4).
    - This calculation is done using the Adder.
7. **Condition Check:**
    - The value of **$rs** (Read Data 1) is compared to zero using the ALU.
    - The ALU performs a subtraction to check if the value is greater than zero.
    - The Zero, Negative, and Overflow flags from the ALU output are used to determine the condition.
8. **Link Address Write Back:**
    - The link address (PC + 4) is stored in register 25.
    - The Control Unit ensures that the link address is written into **$25** through the appropriate control signals (**RegWrite**, **Write Register**, etc.).
9. **PC Update:**

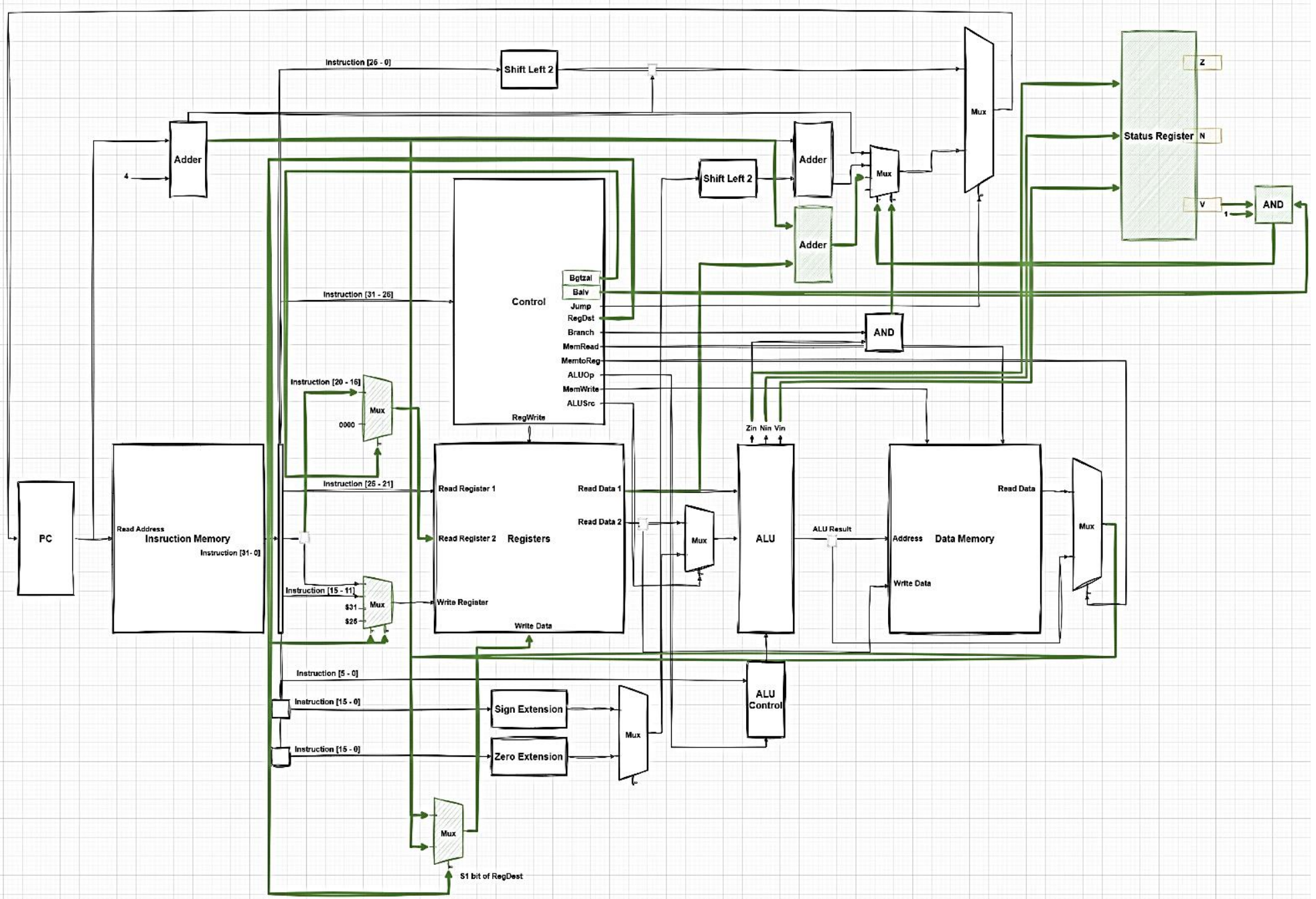
- If the condition (**$rs** > 0) is met, the PC is updated to the branch address calculated earlier.
- If the condition is not met, the PC remains unchanged and proceeds to the next sequential instruction.

Detailed Signals and Data Paths:

- **Instruction [31-26]** goes to the Control Unit to generate control signals.
- **Instruction [25-21]** specifies the source register **$rs**.
- **Instruction [15-0]** provides the immediate value for the branch offset.
- The Control Unit sets the necessary signals for the **bgtzal** operation.
- The Register file reads the value of **$rs**.
- **Read Data 1** is compared with zero using the ALU.
- The immediate value is sign-extended and shifted left by 2.
- The Adder calculates the branch target address.
- If the condition is true (**$rs** > 0), the PC is updated to the branch address.
- The link address (PC + 4) is stored in register 25 through the appropriate control signals.

This execution order ensures that the **bgtzal** instruction correctly checks if the value in **$rs** is greater than zero, branches to the PC-relative address if true, and saves the return address in register 25.

Instruction: bgtzal I-type opcode=33 bgtzal $rs, if R[rs] >0, branch to PC-relative address (for example as beq & bne Label do), link address is stored in register 25.



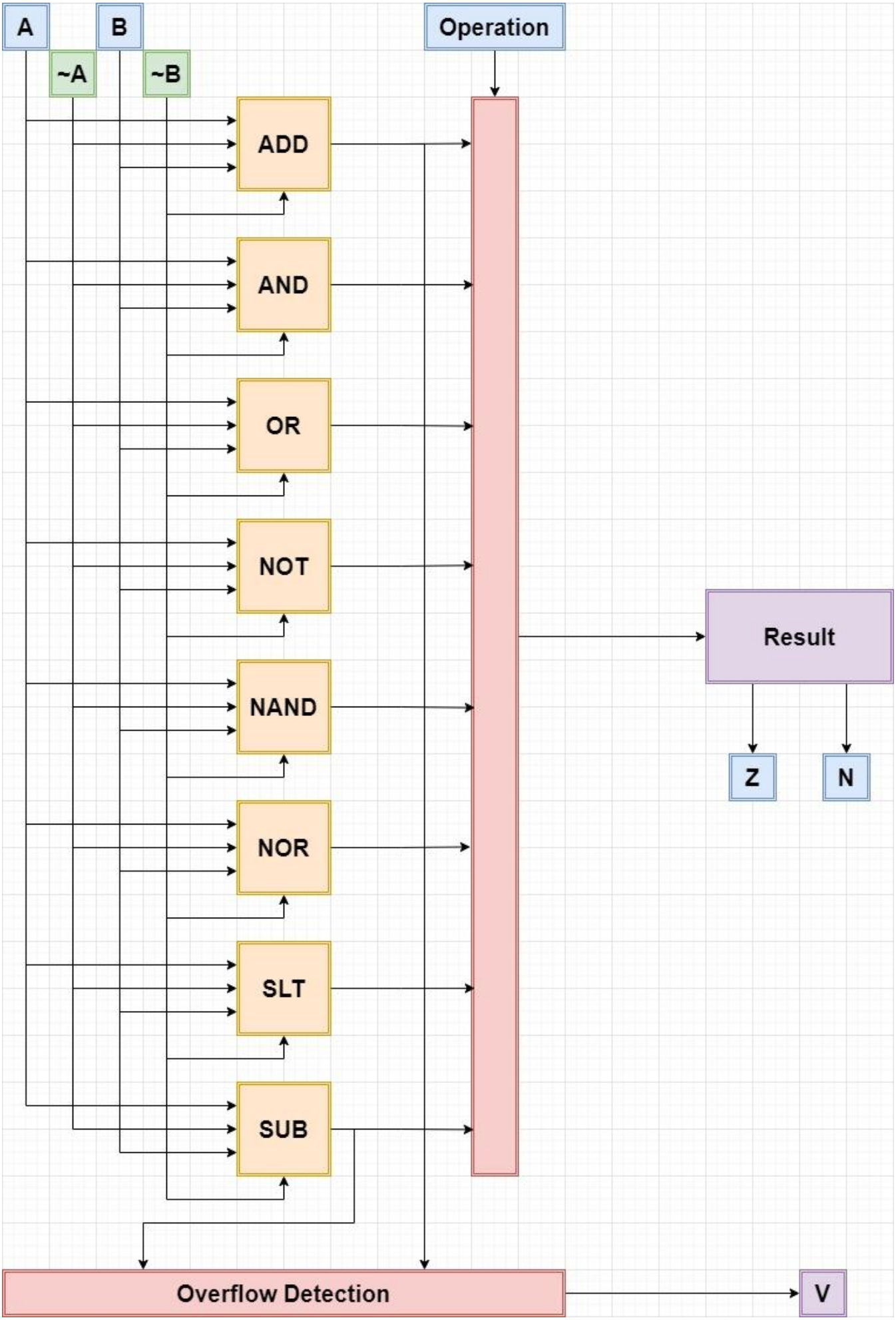
Section (9): Control Lines + ALU Lines (Internal Design).

ALU Internal Design

Arithmetic Logic Unit (ALU) used in a single-cycle Datapath of a processor. This ALU performs a variety of arithmetic and logical operations and provides additional status outputs to indicate specific conditions of the result. Here's a detailed breakdown of its components and functionality:

1. **Inputs (A and B):**
    - These are the primary inputs to the ALU.
    - **~A** and **~B** represent the bitwise negations of inputs A and B, respectively.
2. **Operations:**
    - The ALU supports multiple operations, each represented by a functional block:
        - **ADD:** Adds inputs A and B.
        - **AND:** Performs a bitwise AND on inputs A and B.
        - **OR:** Performs a bitwise OR on inputs A and B.
        - **NOT:** Performs a bitwise NOT on input A.
        - **NAND:** Performs a bitwise NAND on inputs A and B.
        - **NOR:** Performs a bitwise NOR on inputs A and B.
        - **SLT (Set Less Than):** Sets the result to 1 if A is less than B, otherwise sets it to 0.
        - **SUB:** Subtracts input B from input A.
3. **Operation Selector:**
    - The operation selector (labelled as "Operation") determines which operation the ALU will perform.
    - This selector controls the multiplexers that route the appropriate output from the corresponding functional block to the final result.
4. **Result:**
    - The final result of the selected operation is outputted to the "Result" line.
5. **Status Bits:**
    - **Z (Zero):** This status bit is set if the ALU result is zero.
    - **N (Negative):** This status bit is set if the ALU result is negative.
    - **V (Overflow):** This status bit is set if the ALU result causes an overflow, particularly for arithmetic operations like addition and subtraction.
6. **Overflow Detection:**
    - The overflow detection block checks for arithmetic overflow conditions. If an overflow is detected, it sets the overflow status bit (V).
    - This block ensures that arithmetic errors are properly flagged.

In summary, this ALU can perform a variety of arithmetic and logical operations based on the inputs and the selected operation. It includes overflow detection and status bits to indicate if the result is zero, negative, or has caused an overflow. These features make it a critical component in a single-cycle Datapath, enabling the processor to execute instructions that require mathematical and logical computations while providing essential status information about the results.


A
B
Operation
~A
~B
ADD
AND
OR
NOT
NAND
NOR
SLT
SUB
Result
Z
N
Overflow Detection
V

- Control Unit

Instruction	RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOpO	Jump	Brnv	Nandi	MuxReg	Balv	Bgtzal	Jrsal	StatusRegWrite
R-format	01	0	0	1	0	0	0	1	0	001	0	0	1	0	0	0	1
lw	00	1	1	1	1	0	0	0	0	001	0	0	1	0	0	0	0
SW	XX	1	X	0	0	1	0	0	0	001	0	0	X	0	0	0	0
beq	XX	0	X	0	0	0	1	0	1	001	0	0	X	0	0	0	0
JMNOR (R-F)	10	0	0	1	0	0	0	1	0	111	0	0	0	0	0	0	0
BRNV (R-F)	XX	0	0	0	0	0	0	1	0	01	1	0	X	0	0	0	0
Nandi	00	1	0	1	0	0	0	1	1	101	0	1	1	0	0	0	1
BGTZAL	11	X	X	0	0	0	0	X	X	00	0	0	X	0	1	0	0
BALV	10	X	X	(CU)	0	0	0	X	X	00	0	0	0	1	0	0	0
JRSAL	XX	0	1	0	1	1	0	0	0	10	0	0	0	0	0	1	0

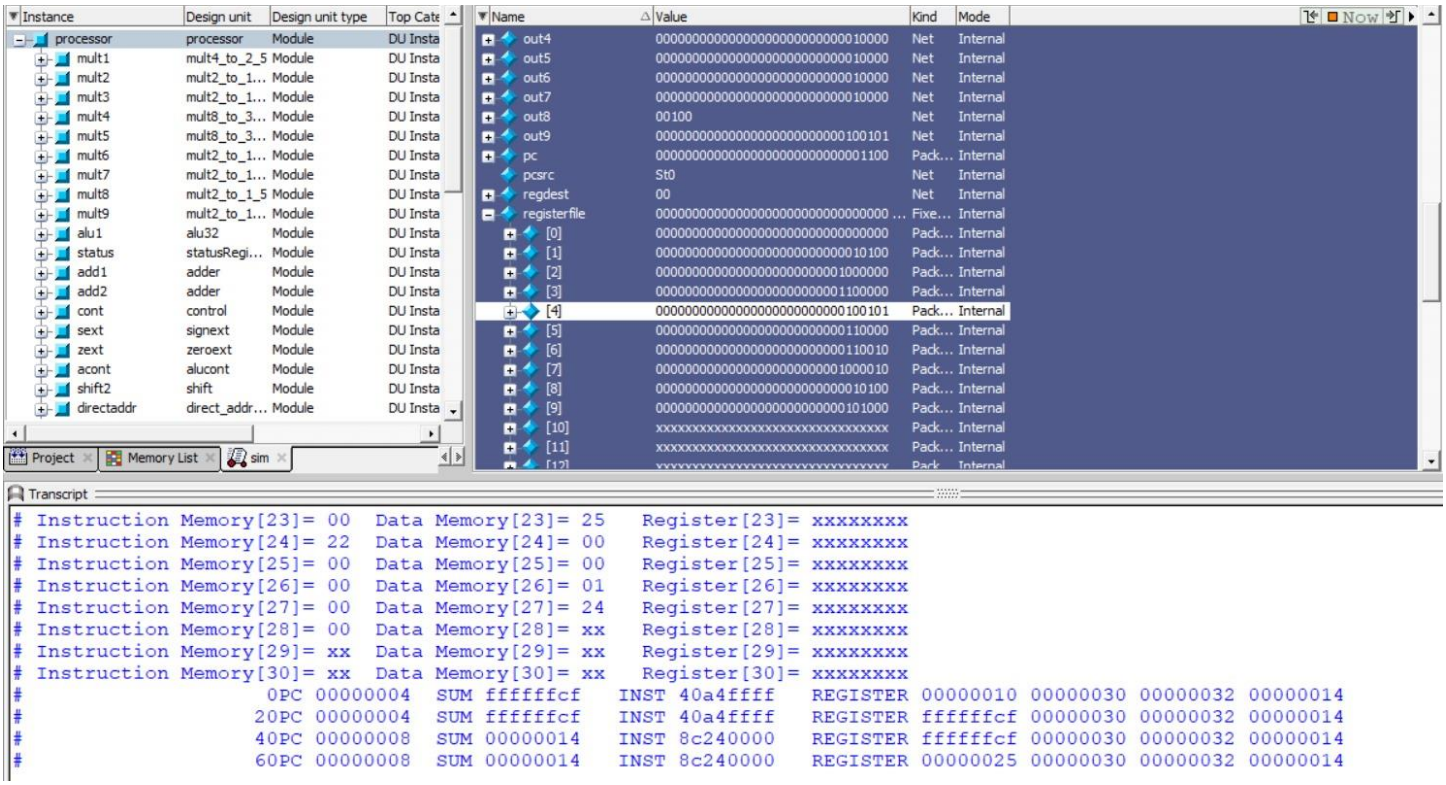
- ALU Control

opcode	ALUOp	Operation	funct	ALU function	ALU control
Iw	00	load word	XXXXXX	add	010
SW	00	store word	XXXXXX	add	010
beq	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
		subtract	100010	subtract	110
		AND	100100	AND	000
		OR	100101	OR	001
		set-on-less-than	101010	set-on-less-than	111
		NOR	100111	NOR	100
		BRNV	010101	add	010
Nandi (16)	11	Nand	XXXXXX	Nand	011

Section (10): Verilog + Simulation Results.

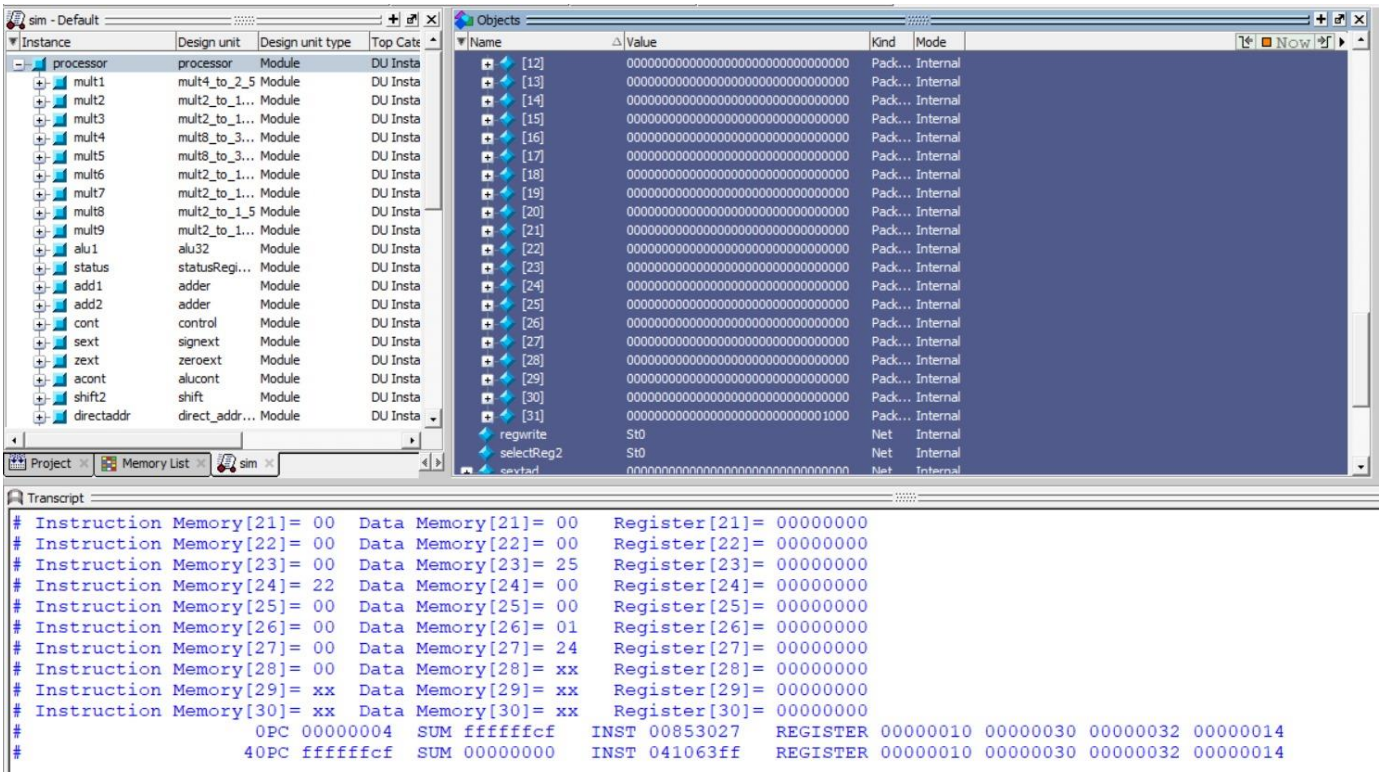
Test (1): Nandi

Nandi a1, a0, 1111 1111 1111 1111 - 010000 00101 00100 1111 1111 1111 1111



Test (2): Jmnor

000000 00100 00101 00110 00000 100111 - jmnor $4, $5 - 00 85 30 27

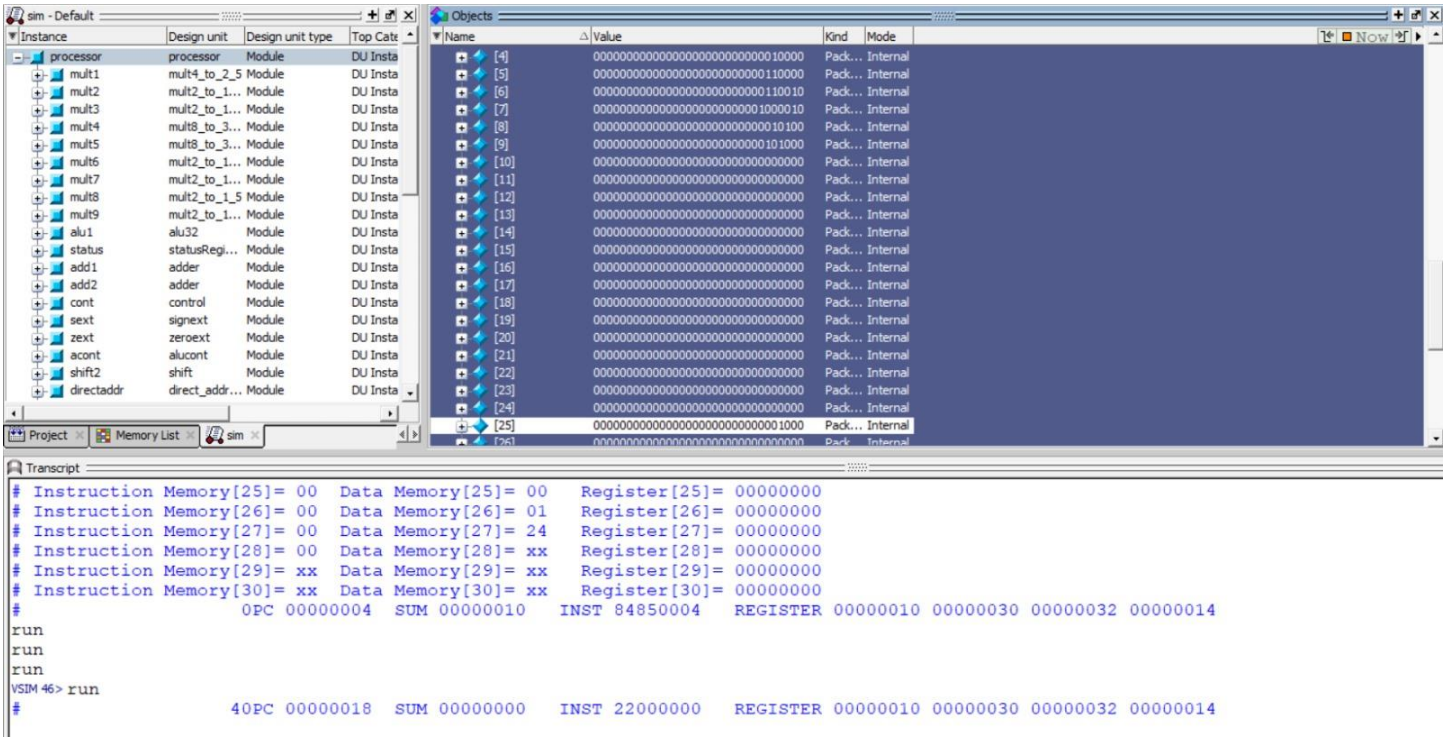


Test (3): Bgtzal

100001 00100 00101 0000 0000 0000 0100

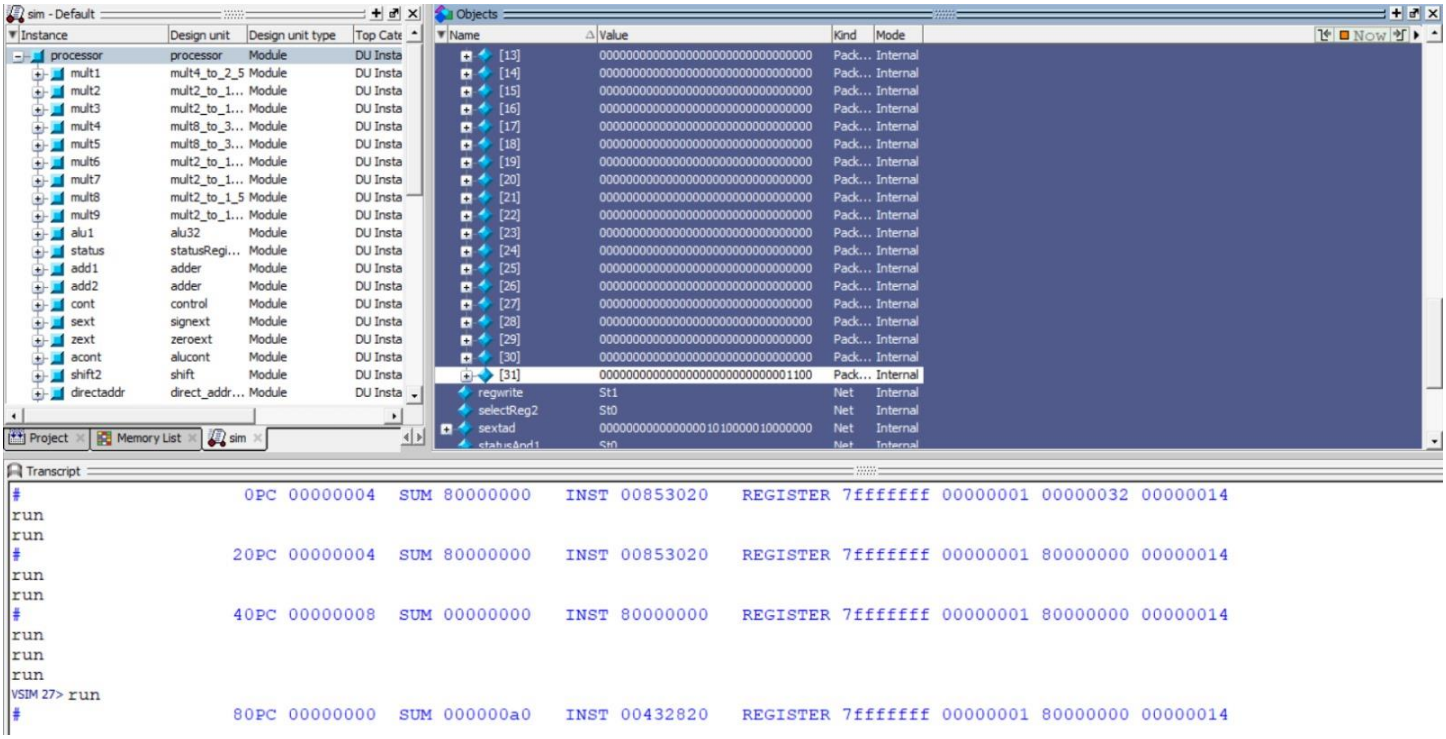
bgtzal $4, 0000 0000 0000 0000

84 85 00 04



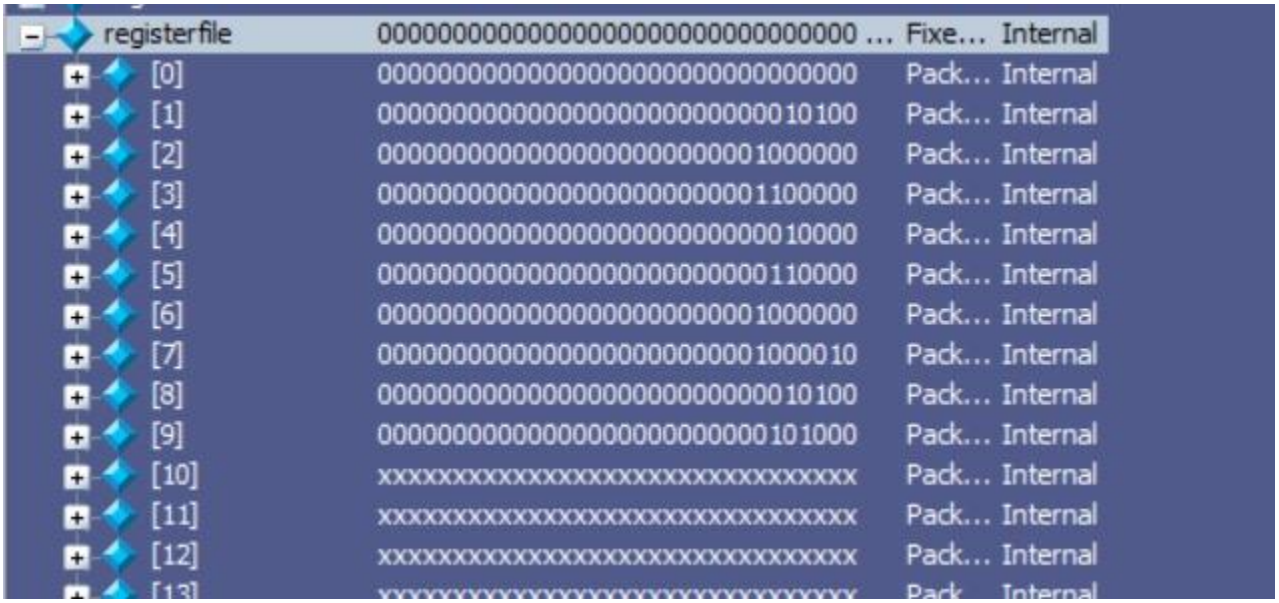
Test (4): Balv

100000 0000 0000 0000 0000 0000 0000 00 - balv 0 - 80 00 00 00

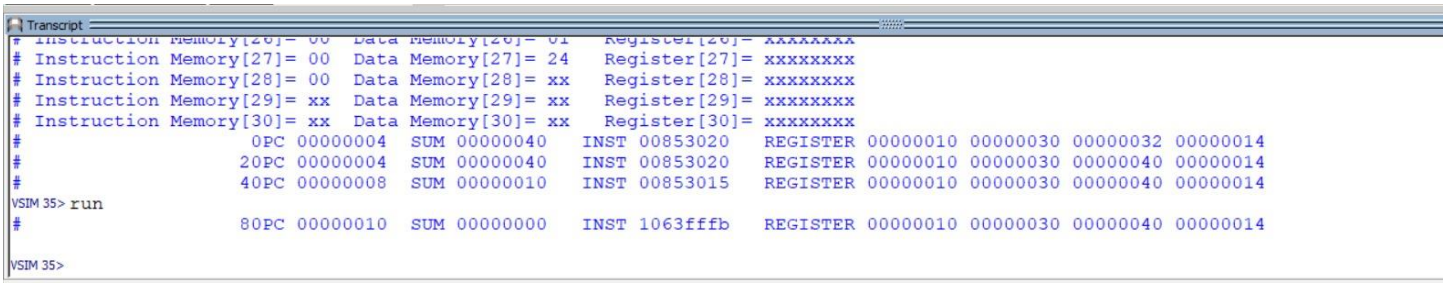


Test (5): Brnv

add before it - add $6, $4, $5 - 00 85 30 20



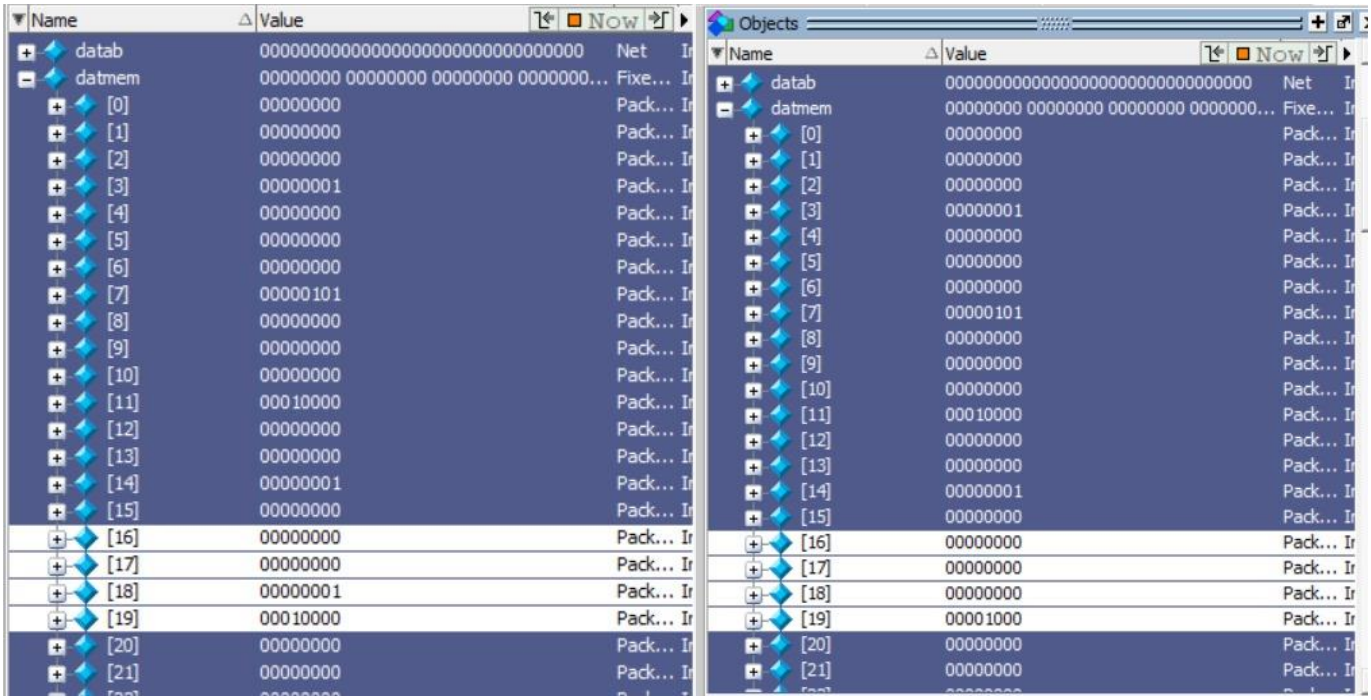
000000 00100 00101 00110 00000 010101 - brnv $4 - 00 85 30 15



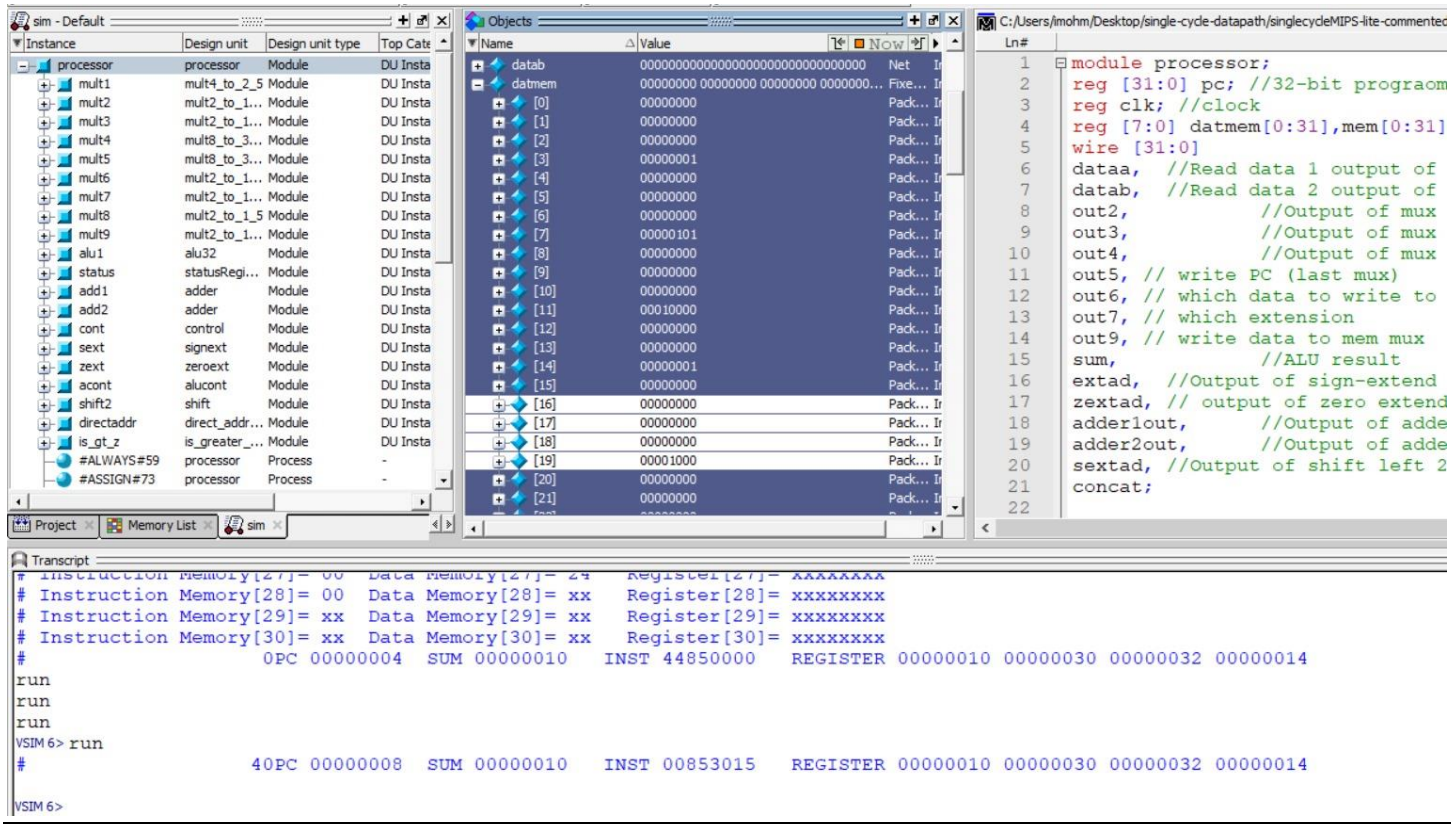
Test (6): Jrsal

010001 00100 00101 0000 0000 0000 0000 - jrsal $4 - 44 85 00 00

datamem before linking values inside address rs


Name
Value
Now
datab
datmem
Objects
[0] 00000000
[1] 00000000
[2] 00000000
[3] 00000001
[4] 00000000
[5] 00000000
[6] 00000000
[7] 00000101
[8] 00000000
[9] 00000000
[10] 00000000
[11] 00010000
[12] 00000000
[13] 00000000
[14] 00000001
[15] 00000000
[16] 00000000
[17] 00000000
[18] 00000001
[19] 00010000
[20] 00000000
[21] 00000000
[19] 00001000

After


sim - Default
Instance
Design unit
Design unit type
processor
mult1
mult2
mult3
mult4
mult5
mult6
mult7
mult8
mult9
alu1
status
add1
add2
cont
sext
zext
acont
shift2
directaddr
is_gt_z
#ALWAYS#59
#ASSIGN#73
Project
Memory List
sim
Objects
C:/Users/mohm/Desktop/single-cycle-datapath/singlecycleMIPS-lite-commented
module processor;
reg [31:0] pc; //32-bit programm
reg clk; //clock
reg [7:0] datmem[0:31],mem[0:31]
wire [31:0]
dataa, //Read data 1 output of
datab, //Read data 2 output of
out2, //Output of mux
out3, //Output of mux
out4, //Output of mux
out5, // write PC (last mux)
out6, // which data to write to
out7, // which extension
out9, // write data to mem mux
sum, //ALU result
extad, //Output of sign-extend
zextad, // output of zero extend
adder1out, //Output of adde
adder2out, //Output of adde
sextad, //Output of shift left 2
concat;
Transcript
Instruction Memory[28]= 00 Data Memory[28]= xx Register[28]= xxxxxxxx
Instruction Memory[29]= xx Data Memory[29]= xx Register[29]= xxxxxxxx
Instruction Memory[30]= xx Data Memory[30]= xx Register[30]= xxxxxxxx
0PC 00000004 SUM 00000010 INST 44850000 REGISTER 00000010 00000030 00000032 00000014
run
run
run
VSIM 6> run
40PC 00000008 SUM 00000010 INST 00853015 REGISTER 00000010 00000030 00000032 00000014
VSIM 6>