# Keys

A table detailing the fields and descriptions of a computer instruction set. Fields include Opcode, rs, rt, rd, shamt, Funct, PC, and Immediate, which represent instruction opcode, source registers, destination registers, shift amount, function code, program counter, and immediate values for operations, respectively.

**Design and Implementation of a Microprocessor System with an HDL Coursework Report/Solution**

Code: [Solutions](https://livecoventryac-my.sharepoint.com/:f:/g/personal/aghanyan_uni_coventry_ac_uk/EqpDKDGY6GJGuoFhd6rTvhMBbmpD8PqmxcBEjH4dXwHlsw?e=SyBLGF) or

<https://livecoventryac-my.sharepoint.com/:f:/g/personal/aghanyan_uni_coventry_ac_uk/EqpDKDGY6GJGuoFhd6rTvhMBbmpD8PqmxcBEjH4dXwHlsw?e=hC0Vt2>

Contents

[Keys 1](#_Toc151740647)

[**Introduction** 3](#_Toc151740648)

[**Extension of processor.** 3](#_Toc151740649)

[**R-type instructions** 3](#_Toc151740650)

[SRLV – R-type 3](#_Toc151740651)

[JR R-type Instruction 6](#_Toc151740652)

[XOR R-type Instruction 8](#_Toc151740653)

[**I-type instructions** 11](#_Toc151740654)

[XORI I-type Instruction 11](#_Toc151740655)

[ANDI I-type Instruction 13](#_Toc151740656)

[ORI I-type Instruction 15](#_Toc151740657)

[**BGTZ** 17](#_Toc151740658)

[Description: 17](#_Toc151740659)

[Instructions Truth Table: 18](#_Toc151740660)

[Main Decoder Truth Table: 18](#_Toc151740661)

[Alu Function Table: 19](#_Toc151740662)

[Alu Decoder: 19](#_Toc151740663)

[Visualization: 19](#_Toc151740664)

[**BLTZ** 20](#_Toc151740665)

[Description: 20](#_Toc151740666)

[Instructions Truth Table: 20](#_Toc151740667)

[Main Decoder Truth Table: 21](#_Toc151740668)

[Alu Function Table: 22](#_Toc151740669)

[Alu Decoder: 22](#_Toc151740670)

[Visualization: 22](#_Toc151740671)

[**Final Truth Tables, Visualization.** 25](#_Toc151740672)

[Main Decoder 25](#_Toc151740673)

[Alu Function Table 25](#_Toc151740674)

[Alu Decoder 26](#_Toc151740675)

[Visualization: 27](#_Toc151740676)

[**Code:** 27](#_Toc151740677)

[Code Snippets Illustrating Key Changes: 29](#_Toc151740678)

[Test Bench and Visualisation: 31](#_Toc151740679)

[Memfile.dat update: 32](#_Toc151740680)

[Translating these instructions: 33](#_Toc151740681)

[Result: 33](#_Toc151740682)

[Conclusion 41](#_Toc151740683)

[References: 41](#_Toc151740684)

[Final Updated Code: 43](#_Toc151740685)

# **I****ntroduction**

The advancement in microprocessor design and Hardware Description Language (HDL) programming represents a critical frontier in the field of computer engineering (Harris & Harris, 2021; Wakerly, 2018). This coursework aims to demonstrate a comprehensive understanding and application of these concepts through the design, implementation, and testing of a single-cycle processor. The processor, based on the MIPS architecture, incorporates a specific set of basic instructions (Harris & Harris, 2021). The task extends beyond mere implementation; it involves enhancing the original processor design to support additional complex instructions, showcasing not only technical proficiency but also innovative design thinking and problem-solving skills (Wakerly, 2018). The coursework is structured into distinct parts, each focusing on different aspects of processor design and programming, including the extension of the processor to support new instructions, SystemVerilog coding, simulation of all instructions, and a critical report detailing each step of the process.

# **Extension of processor.**

The selection of the eight instructions for extending the MIPS-based single-cycle processor was a strategic decision aimed at enhancing the processor's capabilities and versatility – SRLV, JR, XOR, XORI, ANDI, ORI, BGTZ, BLTZ. Each chosen instruction brings a unique functionality to the processor, broadening its applicability in various computational scenarios.

Each of these instructions contributes to a more robust and functional processor. The careful consideration of their roles and impact on the existing design ensures a well-rounded enhancement of the processor's instruction set, making it more suitable for a wide range of applications in modern computing environments.

## **R-type instructions**

### SRLV – R-type

#### Description:

SRLV (Shift Right Logical Variable) performs a logical right shift on the contents of one register (rt) by the number of shift positions specified in the lower 5 bits of another register (rs), storing the result in a third register (rd). See Table 1.0 for the srlv instruction table.

#### Instructions Truth Table:

The instruction is encoded with an opcode of 000000, which is standard for R-type instructions in this ISA.

The funct field for SRLV is specified as 000110. See table below.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Opcode** | | **Funct** | | **Name** | | **Description** | | **Operation** | |
| 000000 | | 000110 | | SRLV rd, rt, rs | | Shift Right Logical Variable | | rd = rt >>> rs | |
| **Opcode** | **rs4:0** | | **rt4:0** | | **rd4:0** | | **shamt4:0** | | **Funct** |
| 000000 | xxxxx | | xxxxx | | Xxxxx | | 00000 | | 000110 |

**Table 1.0: Instruction Table for SRLV (Shift Right Logical Variable)**

Here, >>> represents a logical right shift. The value in register rt is shifted right by the number of bits specified in the lower 5 bits of the register rs, and the result is placed in the register rd.

rs, rt, rd can be any valid register numbers (from 00000 to 11111).

#### Main Decoder Truth Table:

For the SRLV instruction there is no update to the Main decoder truth table.

#### Alu Function Table:

The SRLV operation corresponds to an ALUControl code of 100. See table 1.1.

This code is used by the Arithmetic Logic Unit (ALU) to recognize the operation type and execute the shift accordingly.

|  |  |
| --- | --- |
| **ALUControl2:0** | **Function** |
| 100 | SRLV |

**Table 1.1: Alu Function Table for SRLV (Shift Right Logical Variable)**

#### Alu Decoder:

For the SRLV instruction, when the ALUOp is 010 and the funct field from the instruction is 000110, the ALUControl signal is set to 100, triggering the ALU to perform the shift right logical variable operation. See table below.

|  |  |  |  |
| --- | --- | --- | --- |
| **ALUOp2:0** | **Function** | **ALUControl2:0** | **ALU Operation** |
| 010 | 000110 | 100 | ‘SRLV’ R-type instructions |

**Table 1.2: Alu Decoder Table for SRLV (Shift Right Logical Variable)**

#### Visualization:

The included figures (Fig 0.1.0 and Fig 0.1.1) represent the ALU schematic and Datapath where the SRLV operation takes place. The Datapath visualizes the flow of data and control signals that execute the SRLV instruction within the processor.

Alu:

A diagram of a machine

Description automatically generated

**Fig 0.1.0: Alu for SRLV (Shift Right Logical Variable)**

In the context of the Datapath, no changes are necessary for the implementation of SRLV since the existing ALUControl logic has unused or "spare" codes that can be allocated for new instructions like SRLV without affecting the main decoder's truth table.A diagram of a computer program

Description automatically generated

**Fig 0.1.1: Main Datapath for SRLV (Shift Right Logical Variable)**

### JR R-type Instruction

#### Description:

JR (Jump Register) is used to jump to the address contained in a register. It is typically used for function return operations in assembly language programs.

#### Instructions Truth Table:

The instruction is encoded with an opcode of 000000, which is standard for R-type instructions in this ISA.

The funct field for SRLV is specified as 001000. See table below.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Opcode** | | **Funct** | | **Name** | | **Description** | | **Operation** | |
| 000000 | | 001000 | | JR, rs | | Jump to address in register | | PC = [rs] | |
| **Opcode** | **rs4:0** | | **rt4:0** | | **rd4:0** | | **shamt4:0** | | **Funct** |
| 000000 | xxxxx | | 00000 | | 00000 | | 00000 | | 001000 |

**Table 2.0: Instruction Table for JR (Jump)**

rs: Register containing the jump address.

rt, rd, shamt: Not used in JR, typically 00000.

#### Main Decoder Truth Table:

Please note there is no update to the Main decoder truth table for the JR instruction.

#### Alu Function Table:

Since JR does not perform an arithmetic or logical operation, it does not have a corresponding entry in the ALU Function operation table. The operation of JR is primarily controlled by the instruction decoder and the program counter (PC).

#### Alu Decoder:

For JR, the operation is a jump to an address, and it does not involve the ALU for arithmetic or logical computation. Therefore, it is not represented in the ALU operation table.

#### Visualization:

Alu: No modifications to ALU

In the Datapath everything remains since we have spare codes in the ALUControl logic just need to change the MUX2 to a mux3 and make ScrA an input and that’s it. See Fig 0.2.1 below.

A diagram of a computer

Description automatically generated

**Fig 0.2.1: Main Datapath for JR (Jump)**

### XOR R-type Instruction

#### Description:

The XOR (Exclusive OR) instruction is a bitwise operation that compares corresponding bits of two register values (rs and rt). For each bit position, the resulting bit in register rd is set to 1 if and only if one of the corresponding bits is set (i.e., the bits are different). See Table 3.0

#### Instructions Truth Table:

The instruction is encoded with an opcode of 000000, which indicates an R-type instruction. The funct field is 100110, uniquely identifying the XOR operation. The operation performed is [rd] = [rs] XOR [rt], which stores the exclusive OR result of rs and rt into rd.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Opcode** | | **Funct** | | **Name** | | **Description** | | **Operation** | |
| 000000 | | 100110 | | XOR, rd, rs, rt | | Bitwise exclusive OR | | [rd] = [rs] XOR [rt] | |
| **Opcode** | **rs4:0** | | **rt4:0** | | **rd4:0** | | **shamt4:0** | | **Funct** |
| 000000 | Xxxxx | | xxxxx | | xxxxx | | 00000 | | 100110 |

**Table 3.0: Instruction Table for XOR (Exclusive OR)**

#### Main Decoder Truth Table:

The implementation of the XOR instruction does not require changes to the main decoder truth table. It utilizes the standard fields for an R-type instruction without necessitating new decoding logic.

#### Alu Function Table:

For the XOR operation, the ALUControl bits are set to 011. These control bits instruct the Arithmetic Logic Unit (ALU) to carry out the exclusive OR operation when the corresponding ALUOp bits from the instruction decoder are set.

|  |  |
| --- | --- |
| **ALUControl2:0** | **Function** |
| 011 | XOR |

**Table 3.1: Alu Function Table for XOR (Exclusive OR)**

#### Alu Decoder:

When the ALUOp bits are 010 and the function code extracted from the instruction is 100110, the ALU decoder outputs the control bits 011. This prompts the ALU to execute the XOR operation between the operands provided by the rs and rt registers.

|  |  |  |  |
| --- | --- | --- | --- |
| **ALUOp2:0** | **Function** | **ALUControl2:0** | **ALU Operation** |
| 010 | 100110 | 011 | ‘XOR’ R-type  instructions |

**Table 3.2: Alu Decoder Table for XOR (Exclusive OR)**

#### Visualization:

The visualization of the ALU would include the XOR gate schematic showing how the inputs from the rs and rt registers are compared to produce the output in rd.

Alu:

A diagram of a machine

Description automatically generated

**Fig 0.3: ALU for XOR (Exclusive OR)**

Within the Datapath, the XOR instruction uses available codes in the ALUControl logic. This means that the existing multiplexer configurations and control paths can be used without significant modifications. The XOR operation leverages the standard R-type instruction format and signal path within the ALU.

## **I-type instructions**

### XORI I-type Instruction

#### Description:

XORI (Exclusive OR Immediate) performs a bitwise exclusive OR operation on the contents of a register and an immediate constant. Each bit in the result is set to 1 if the corresponding bits of the register and the immediate value are different, and 0 if they are the same.

#### Instructions Truth Table:

The instruction is encoded with an opcode of 001110. Unlike R-type instructions, I-type instructions such as XORI do not use the funct field. The immediate field contains the constant value to be XORed with the contents of rs. The operation [rt] = [rs] XOR immediate computes the exclusive OR of the register's content with the immediate value.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Opcode** | | **Funct** | | **Name** | | | **Description** | | **Operation** | |
| 001110 | | - | | XORI, rt, rs, immediate | | | Bitwise exclusive OR with immediate | | [rt] = [rs] XOR immediate | |
| **Opcode** | **rs4:0** | | **rt4:0** | | **rd4:0** | **shamt4:0** | | **Funct** | | **Immediate15:0** |
| 001110 | xxxxx | | xxxxx | | - | - | | - | | iiiiiiiiiiiiiiii |

**Table 4.0: Instruction Table for XORI (Exclusive OR Immediate)**

#### Main Decoder Truth Table:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Opcode** | **regwrite** | **regdst** | **alusrc** | **branch** | **memwrite** | **memtoreg** | **jump** | **Aluop2:0** |  |
| 000000 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 010 | RTYPE |
| 100011 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 000 | LW |
| 101011 | 0 | X | 1 | 0 | 1 | X | 0 | 000 | SW |
| 000100 | 0 | X | 0 | 1 | 0 | X | 0 | 001 | BEQ |
| 001000 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 000 | ADDI |
| 000010 | 0 | X | X | X | 0 | X | 1 | XXX | J |
| 001110 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 101 | XORI |

**Table 4.1: Main decoder for XORI (Exclusive OR Immediate)**

The main decoder uses the opcode (see Table 4.1) to determine the control signals for the Datapath. For XORI, the relevant control signals are set as follows:

RegWrite: Indicates that the instruction writes to a register.

RegDst: 0 as destination register is rt.

ALUSrc: 1 because the ALU operation uses an immediate value.

Branch, MemWrite, MemToReg, Jump: All 0 as this is neither a branch, memory, nor jump instruction.

#### Alu Function Table:

Since XORI is an immediate operation, the ALU will perform the XOR operation between the contents of the register specified by rs and the immediate value. The operation is like the XOR but uses an immediate value instead of a second register.

#### Alu Decoder:

|  |  |  |  |
| --- | --- | --- | --- |
| **ALUOp2:0** | **Function** | **ALUControl2:0** | **ALU Operation** |
| 101 | xxxxxx | 011 | XORI |

**Table 4.2: ALU decoder for XORI (Exclusive OR Immediate)**

#### Visualization:

Alu:

The ALU schematic for XORI would include an XOR gate where one input is the contents of the rs register and the other input is the immediate value, with the output being the rt register.

A diagram of a machine

Description automatically generated

**Fig 0.4: ALU for XORI (Exclusive OR Immediate)**

No additional changes are needed in the Datapath for the XORI instruction since it uses the standard immediate instruction format and signal path within the ALU.

### ANDI I-type Instruction

#### Description:

ANDI (AND Immediate) performs a bitwise AND operation between the contents of a register and a 16-bit immediate value. Each bit in the result is set to 1 only if both corresponding bits of the register and the immediate value are 1.

#### Instructions Truth Table:

The ANDI instruction is encoded with an opcode of 001100. It uses the rs register and an immediate value to perform the AND operation, storing the result in the rt register. The instruction's format is [rt] = [rs] AND immediate, which is depicted in Table 5.0.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Opcode** | | **Funct** | | **Name** | | | **Description** | | **Operation** | |
| 001100 | | - | | ANDI, rt, rs, immediate | | | Bitwise AND with immediate | | [rt] = [rs] AND immediate | |
| **Opcode** | **rs4:0** | | **rt4:0** | | **rd4:0** | **shamt4:0** | | **Funct** | | **Immediate15:0** |
| 001100 | xxxxx | | Xxxxx | | - | - | | - | | iiiiiiiiiiiiiiii |

**Table 5.0: ALU decoder for ANDI (Exclusive AND Immediate)**

#### Main Decoder Truth Table:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Opcode** | **regwrite** | **regdst** | **alusrc** | **branch** | **Memwrite** | **memtoreg** | **jump** | **Aluop2:0** |  |
| 000000 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 010 | RTYPE |
| 100011 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 000 | LW |
| 101011 | 0 | X | 1 | 0 | 1 | X | 0 | 000 | SW |
| 000100 | 0 | X | 0 | 1 | 0 | X | 0 | 001 | BEQ |
| 001000 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 000 | ADDI |
| 000010 | 0 | X | X | X | 0 | X | 1 | XXX | J |
| 001100 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 011 | ANDI |
| 001110 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 101 | XORI |

**Table 5.1: Main decoder for ANDI (Exclusive AND Immediate)**

The main decoder determines the control signals for executing the instruction based on the opcode. The control signals for ANDI, as described in Table 5.1, are as follows:

RegWrite: Set to 1 to indicate that the instruction will write to a register.

RegDst: Set to 0 because the destination for the result is rt, which is characteristic of I-type instructions.

ALUSrc: Set to 1, signifying that the second operand for the ALU is an immediate value.

Branch, MemWrite, MemToReg, Jump: All set to 0 as ANDI is solely a register-immediate operation and does not involve branching, memory access, or jumping.

#### Alu Function Table:

The ALU will perform the AND operation between the contents of the register specified by rs and the immediate value. This operation is straightforward, as it simply ANDs each bit of the register with the corresponding bit of the immediate value.

#### Alu Decoder:

The ALUOp code for ANDI is 011. When the ALU decoder receives this code, it sets the ALUControl to 000, which configures the ALU to execute the AND operation. This is outlined in Table 5.2.

|  |  |  |  |
| --- | --- | --- | --- |
| **ALUOp2:0** | **Function** | **ALUControl2:0** | **ALU Operation** |
| 011 | xxxxxx | 000 | ANDI |

**Table 5.2: ALU Function for ANDI (Exclusive AND Immediate)**

#### Visualization:

In terms of the ALU's schematic, no changes are necessary for the ANDI operation. The ALU's existing gates and logic circuits are well-suited to handle the AND operation with an immediate value.

No additional changes are needed in the Datapath for the ANDI instruction since it uses the standard immediate instruction format and signal path within the ALU.

### ORI I-type Instruction

#### Description:

ORI (OR Immediate) performs a bitwise OR operation between the contents of a register and a 16-bit immediate value. Each bit in the result is set to 1 if either corresponding bit of the register or the immediate value is 1.

#### Instructions Truth Table:

Encoded with an opcode of 001101, the ORI instruction specifies the source register rs and the destination register rt, with the immediate value being sign-extended and used as the second operand. The operation performed by the instruction is [rt] = [rs] OR immediate, as presented in Table 6.0.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Opcode** | | **Funct** | | **Name** | | | **Description** | | **Operation** | |
| 001101 | | - | | ORI, rt, rs, immediate | | | Bitwise OR with immediate | | [rt] = [rs] OR immediate | |
| **Opcode** | **rs4:0** | | **rt4:0** | | **rd4:0** | **shamt4:0** | | **Funct** | | **Immediate15:0** |
| 001101 | xxxxx | | Xxxxx | | - | - | | - | | iiiiiiiiiiiiiiii |

**Table 6.0: Instruction Table for ORI (Exclusive OR Immediate)**

#### Main Decoder Truth Table:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Opcode** | **regwrite** | **regdst** | **alusrc** | **branch** | **Memwrite** | **memtoreg** | **jump** | **Aluop2:0** |  |
| 000000 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 010 | RTYPE |
| 100011 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 000 | LW |
| 101011 | 0 | X | 1 | 0 | 1 | X | 0 | 000 | SW |
| 000100 | 0 | X | 0 | 1 | 0 | X | 0 | 001 | BEQ |
| 001000 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 000 | ADDI |
| 000010 | 0 | X | X | X | 0 | X | 1 | XXX | J |
| 001100 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 011 | ANDI |
| 001101 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 100 | ORI |
| 001110 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 101 | XORI |

**Table 6.1: ALU Function for ORI (Exclusive OR Immediate)**

The main decoder uses the opcode to set the control signals for the Datapath execution of the instruction. The settings for the ORI instruction are detailed in Table 6.1:

RegWrite: Set to 1, allowing the result of the operation to be written to the destination register (rt).

RegDst: Set to 0 since the destination register for I-type instructions is rt.

ALUSrc: Set to 1, indicating the ALU should use an immediate value as one of its operands.

Branch, MemWrite, MemToReg, Jump: All set to 0, reflecting that ORI is not a control flow instruction and does not involve memory operations or jumps.

#### Alu Function Table:

The ALU performs the OR operation using the contents of the rs register and the immediate value. This operation simply combines each bit of the register with the corresponding bit of the immediate value using the OR logic.

#### Alu Decoder:

For the ORI instruction, when the ALUOp is set to 100, the ALUControl is directed to 001. This control signal configures the ALU to carry out the OR operation, as outlined in the ALU decoder.

|  |  |  |  |
| --- | --- | --- | --- |
| **ALUOp2:0** | **Function** | **ALUControl2:0** | **ALU Operation** |
| 100 | xxxxxx | 001 | ORI |

**Table 6.2: ALU Function for ORI (Exclusive OR Immediate)**

#### Visualization:

The ALU structure does not require modifications for the ORI operation. The existing logic gates within the ALU can perform the OR operation using the immediate value as specified.

The implementation of the ORI instruction within the Datapath utilizes existing components. The immediate value is sign-extended and serves as one input to the ALU, while the contents of the rs register provide the other input. The ALU then executes the OR operation, and the result is written to the rt register. The standard Datapath configuration supports this operation without any alterations.

The Datapath for the single-cycle MIPS processor already contains all the necessary components to support the ANDI, ORI and XORI instruction. This instruction is executed in the ALU. The instruction performs an its operation on the two source operands, the results of the instructions are written to the destination register.

To execute an any of these instructions the following steps are taken:

•The instruction is fetched from memory and decoded.

•The two source operands are read from the register file.

•The ALU performs the specified operation on the two source operands.

•The result of the operation is written to the destination register.

## **BGTZ**

### Description:

BGTZ (Branch on Greater Than Zero) causes a branch to a target address if the contents of a specified register are greater than zero.

### Instructions Truth Table:

The BGTZ instruction has an opcode of 000111. It uses the rs register to test if the value is greater than zero and an offset to determine the branch target. The rt, rd, and shamt fields are not used in this instruction, as shown in the truth table. The operation of the instruction is defined as: if [rs] > 0 then PC = PC + 4 + (offset << 2).

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Opcode** | | **Funct** | | **Name** | | | **Description** | | **Operation** | |
| 000111 | | - | | BGTZ, rs, offset | | | Branch if greater than zero | | if [rs] > 0 then PC = PC + 4 + (offset << 2) | |
| **Opcode** | **rs4:0** | | **rt4:0** | | **rd4:0** | **shamt4:0** | | **Funct** | | **Immediate15:0** |
| 000111 | Xxxxx | | 00000 | | - | - | | - | | iiiiiiiiiiiiiiii |

**Table 7.0: Instruction Table for BGTZ (Branch greater than zero)**

rs: Source register to be tested.

rt: Not used in BGTZ, typically 00000.

### Main Decoder Truth Table:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Opcode** | **regwrite** | **Regdst** | **alusrc** | **branch** | **Memwrite** | **Memtoreg** | **jump** | **Aluop2:0** |  |
| 000000 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 010 | RTYPE |
| 100011 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 000 | LW |
| 101011 | 0 | X | 1 | 0 | 1 | X | 0 | 000 | SW |
| 000100 | 0 | X | 0 | 1 | 0 | X | 0 | 001 | BEQ |
| 001000 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 000 | ADDI |
| 000010 | 0 | X | X | X | 0 | X | 1 | XXX | J |
| 001100 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 011 | ANDI |
| 001101 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 100 | ORI |
| 001110 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 101 | XORI |
| 000111 | 0 | X | 0 | 1 | 0 | X | 0 | 110 | BGTZ |

**Table 7.1: Main Decoder Table for BGTZ (Branch greater than zero)**

In the context of the main decoder, (See Table 7.1) the BGTZ instruction is associated with specific control signals:

RegWrite and MemWrite are set to 0, indicating that there is no register or memory write operation.

Branch is set to 1, which signifies that this instruction involves a conditional branch.

Jump is set to 0 as this instruction performs a conditional branch, not an unconditional jump.

The ALUOp bits are set to 110 for the BGTZ operation, as specified in the decoder truth table.

### Alu Function Table:

The ALU performs a comparison operation to check if the value in rs is greater than zero. This is not a typical arithmetic or logical function but a comparison, and it sets the condition codes accordingly.

### Alu Decoder:

When the ALUOp is 110, the ALUControl is set to 101, indicating that the ALU should perform the comparison operation necessary for the BGTZ instruction.

|  |  |  |  |
| --- | --- | --- | --- |
| **ALUOp2:0** | **Function** | **ALUControl2:0** | **ALU Operation** |
| 110 | Xxxxxx | 101 | BGTZ |

**Table 7.2: ALU Decoder Table for BGTZ (Branch greater than zero)**

### Visualization:

Alu:

The provided ALU schematic diagram illustrates the components within the ALU that contribute to the operation of the BGTZ instruction:

A diagram of a machine

Description automatically generated

**Fig 0.5: ALU for BGTZ (Branch greater than zero)**

## **BLTZ**

### Description:

BLTZ (Branch on Less Than Zero) causes a branch to a target address if the contents of a specified register are less than zero.

### Instructions Truth Table:

The BLTZ instruction is specified with an opcode of 000001. It utilizes the rs register to test the value and an offset to calculate the branch target. The fields rt, rd, and shamt are not used and are typically set to 00000. If [rs] < 0, then the branch is taken, and the PC is updated: PC = PC + 4 + (offset << 2).

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Opcode** | | **Funct** | | **Name** | | | **Description** | | **Operation** | |
| 000001 | | - | | BLTZ, rs, offset | | | Branch if less than zero | | if [rs] < 0 then PC = PC + 4 + (offset << 2) | |
| **Opcode** | **rs4:0** | | **rt4:0** | | **rd4:0** | **shamt4:0** | | **Funct** | | **Immediate15:0** |
| 000001 | Xxxxx | | 00000 | | - | - | | - | | iiiiiiiiiiiiiiii |

**Table 8.0: Instruction Table for BLTZ (Branch less than zero)**

### Main Decoder Truth Table:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Opcode** | **Regwrite** | **Regdst** | **Alusrc** | **branch** | **Memwrite** | **memtoreg** | **jump** | **Aluop2:0** |  |
| 000000 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 010 | RTYPE |
| 100011 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 000 | LW |
| 101011 | 0 | X | 1 | 0 | 1 | X | 0 | 000 | SW |
| 000100 | 0 | X | 0 | 1 | 0 | X | 0 | 001 | BEQ |
| 001000 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 000 | ADDI |
| 000010 | 0 | X | X | X | 0 | X | 1 | XXX | J |
| 001100 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 011 | ANDI |
| 001101 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 100 | ORI |
| 001110 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 101 | XORI |
| 000111 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 110 | BGTZ |
| 000001 | 0 | X | 0 | 1 | 0 | x | 0 | 110 | BLTZ |

**Table 8.1: Main Decoder Table for BLTZ (Branch less than zero)**

For the BLTZ instruction, the main decoder (see Table 8.1) sets the control signals as follows:

RegWrite and MemWrite are set to 0 since the instruction does not involve writing to a register or memory.

Branch is set to 1, indicating that this is a branch instruction.

Jump is set to 0, as BLTZ performs a conditional branch, not an unconditional jump.

The ALUOp bits for BLTZ are set to 110, indicating a branch instruction that requires a comparison.

### Alu Function Table:

Within the ALU, the operation performed for BLTZ is a comparison to determine if the value in rs is less than zero. This comparison is not a conventional arithmetic or logical operation, but it sets the necessary flags for branch decision-making.

### Alu Decoder:

For BLTZ, when the ALUOp is 110, the ALUControl is typically set to a code (such as 101) that triggers the ALU to perform the comparison operation necessary for BLTZ.

|  |  |  |  |
| --- | --- | --- | --- |
| **ALUOp2:0** | **Function** | **ALUControl2:0** | **ALU Operation** |
| 110 | Xxxxxx | 101 | BLTZ |

**Table 8.2: ALU Decoder Table for BLTZ (Branch less than zero)**

### Visualization:

Alu: The provided ALU schematic diagram illustrates the components within the ALU that contribute to the operation of the BGTZ instruction:

A diagram of a machine

Description automatically generated

**Fig 0.6.0: ALU for BLTZ (Branch less than zero)**

Datapath for BGTZ and BLTZ:

For BGTZ and BLTZ we get two extra outputs from alu name as ‘g’ and ‘l’ that two inputs are for detection of rs>0 and rs<0

This equation implements instead a AND gate, we used three and gates and an OR gate (See fig 0.6.1 and fig 0.6.2)

A diagram of a block diagram

Description automatically generated

**Fig 0.6.1: Branch Control Logic for PC Source Selection**

A diagram of a computer

Description automatically generated

**Fig 0.6.2: Datapath for BLTZ and BGTZ (Branch greater/less than zero)**

## **Final Truth Tables, Visualization.**

### Main Decoder

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Opcode** | **regwrite** | **regdst** | **alusrc** | **branch** | **memwrite** | **Memtoreg** | **jump** | **Aluop2:0** |  |
| 000000 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 010 | RTYPE |
| 100011 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 000 | LW |
| 101011 | 0 | X | 1 | 0 | 1 | X | 0 | 000 | SW |
| 000100 | 0 | X | 0 | 1 | 0 | X | 0 | 001 | BEQ |
| 001000 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 000 | ADDI |
| 000010 | 0 | X | X | X | 0 | X | 1 | XXX | J |
| 001100 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 011 | ANDI |
| 001101 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 100 | ORI |
| 001110 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 101 | XORI |
| 000111 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 110 | BGTZ/BLTZ |

**Table 9.0: Final Main Decoder Table**

### Alu Function Table

|  |  |
| --- | --- |
| **ALUControl2:0** | **Function** |
| 000 | AND |
| 001 | OR |
| 010 | ADD |
| 011 | XOR |
| 100 | SRLV |
| 101 | BGTZ, BLTZ |
| 110 | SUB |
| 111 | SLT |

**Table 9.1: Final ALU Function Table**

### Alu Decoder

|  |  |  |  |
| --- | --- | --- | --- |
| **Aluop2:0** | **Funct** | **ALUControl** | **Description** |
| 000 | XXXXXX | 010 | add (for lw/sw/addi) |
| 001 | XXXXXX | 110 | sub (for beq) |
| 010 | 000110 | 100 | ‘SRLV’ R-type instructions |
| 010 | 100000 | 010 | ‘add’ R-type instructions |
| 010 | 100010 | 110 | ‘sub’ R-type instructions |
| 010 | 100100 | 000 | ‘and’ R-type instructions |
| 010 | 100101 | 001 | ‘or’ R-type instructions |
| 010 | 100110 | 011 | ‘XOR’ R-type instructions |
| 010 | 101010 | 111 | ‘slt’ R-type instructions |
| 011 | XXXXXX | 000 | ANDI |
| 100 | XXXXXX | 001 | ORI |
| 101 | XXXXXX | 011 | XORI |
| 110 | XXXXXX | 101 | BGTZ, BLTZ |

**Table 9.2: Final ALU Decoder Table**

### Visualization:

Datapath:

A diagram of a computer

Description automatically generated

**Table 9.3: Final Datapath**

## **Code:**

Key Changes to the code Summary:

1. Controller Module Modifications:

Addition of g and l Inputs and Outputs: Two new input and output logic signals, g and l, are added to the controller module. These are likely used for additional branching conditions such as BGTZ and BLTZ.

Expanded aluop: The aluop signal in the controller module is expanded from [1:0] to [2:0]. This expansion supports a broader range of ALU operations.

New Instructions Added: Instructions like andi, ori, xori, BGTZ, and BLTZ are included in the maindec module's case statement. See Fig 1.

1. ALU Decoder (aludec) Enhancements:

Additional ALU Operations: Logic for andi, ori, xori, and the branch instructions BGTZ and BLTZ are included. A new output, jump, is introduced for handling the JR instruction. As Seen in Fig 2.

1. Datapath Module Expansion:

New Outputs g and l: These outputs are used for branching logic, likely corresponding to BGTZ and BLTZ.

ALU Logic Enhancement: The ALU now handles additional operations like SRLV (3'b100) and updated branch conditions. See Fig 3.

1. Introduction of mux3 Module:

Purpose: This 3-way multiplexer module is a new addition in the updated code. It's designed to handle three data inputs and select one based on a 2-bit selector.

Usage in Datapath: The mux3 is used in the Datapath module for determining the next program counter (pcnext). This allows for more complex branching logic, particularly to handle jump instructions effectively. See Fig 4.

1. Incorporation of g and l Signals:

Functionality: These signals are likely used to facilitate new branching conditions. g and l might stand for "greater" and "less", respectively, and are used in branch instructions like BGTZ and BLTZ.

Implementation: The g and l signals are calculated in the ALU and used in the controller module to determine the pcsrc signal, influencing the branching logic. See Fig 5.

1. Complex Branching Logic in pcsrc:

Enhanced Branching Conditions: The pcsrc signal now considers not only the zero flag for BEQ instructions but also incorporates conditions for BGTZ and BLTZ using the g and l flags. This addition significantly enhances the MIPS processor's branching capabilities. See Fig 6.

1. Enhanced Branching Conditions: The pcsrc signal now considers not only the zero flag for BEQ instructions but also incorporates conditions for BGTZ and BLTZ using the g and l flags. This addition significantly enhances the MIPS processor's branching capabilities. See Fig 7.

### Code Snippets Illustrating Key Changes:

**Fig 1. Controller Module:**

Controller Module Code Update 

**Fig 2. ALU Decoder (aludec) Module:**

ALU decoder module code update.


**Fig 3. Datapath Module:**

Datapath Module Code.


**Fig 4. Adjustments to BEQ (Branch on Equal) and J (Jump):**

Branch on Equal and Jump Code.


**Fig 5. Mux3 Module:**

MUX3 Module Code.

**Fig 6. G and L signals:**

G and L signals Code introduction.

**Fig 7. Branching Logic in pcsrc:**

Branching Logic Code.


## Test Bench and Visualisation:

I designed the testbench in the provided mipstest.sv code to validate the functionality of a MIPS processor. My testbench is responsible for stimulating the MIPS processor, which is the design under test (DUT), and analysing its outputs to ensure it operates correctly.

Since the testbench is functional in nature, concentrating on the processor's operational behaviour under various test scenarios. It's more about verifying the correctness of the processor's actions rather than assessing its performance or timing aspects. The incorporation of the $readmemh function in the imem module to load a set of instructions from an external file tells me that this approach simulates a real-world scenario where the processor executes a program, enhancing the realism and relevance of my testbench.

So, I took the liberty and made no changes to the original `mipstest.cv` file whereas in fact most of the changes occurred in the `memfile.dat` file (see Fig 8). In this file I notice a sequence of hexadecimal instructions, which are crucial for the functionality of the MIPS processor program. In the initial version, the instructions seem to focus on basic arithmetic and logical operations, memory access, and control flow.

However, in the updated version, there are notable additions and changes. New instructions, introduce more complex operations and expand the program's capability. setting up new data in registers, modifying conditions for branch instructions, or preparing for bitwise operations. The extended sequence also allows more intricate control flows and possibly the introduction of loops or additional decision-making logic.

Overall, the updates to memfile.dat imply an enhancement in the program's complexity, introducing more sophisticated operations and control structures. This aligns with the enhancements I have been implementing in the MIPS processor's functionality in my project.

### Memfile.dat update:

memfile.dat update


**Fig 8. Memfile.dat Updated.**

### Translating these instructions:

How to translate Hexadecimal.

### Result:

#### **Simulation Succeeded**

A screenshot of a computer

Description automatically generated

**Fig 9.0: Test Succeed**

A screenshot of a computer

Description automatically generated

**Fig 9.1: Test Succeeded**

#### **Full Translation:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **PC** | **Inst** | **Op code** | **rs** | **Rt** | **Immediate** | | | | | | **Operation** | **Instruction** | **Fig** |
| 00 | 20020005 | 001000 | 00000 | 00010 | 0005 | | | | | | ADDI | R2=R0+0005 | Fig 10.1 |
| 04 | 2003000c | 001000 | 00000 | 00011 | 000c | | | | | | ADDI | R3=R0+000c | Fig 10.2 |
| 08 | 2067fff7 | 001000 | 00011 | 00111 | fff7 | | | | | | ADDI | R7=R3+fff7 | Fig 10.3 |
| 0C | 00e22025 | 000000 | 00111 | 00010 | 00100 | 00000 | | | 100101 | | OR | R4=R7|R2 | Fig 10.4 |
| 10 | 00642824 | 000000 | 00011 | 00100 | 00101 | 00000 | | | 100100 | | AND | R5=R4&R3 | Fig 10.5 |
| 14 | 00a42820 | 000000 | 00101 | 00100 | 00101 | 00000 | | | 100000 | | ADD | R5=R4+R5 | Fig 10.6 |
| 18 | 10a7000a | 000100 | 00101 | 00111 | 000a | | | | | | BEQ | R5==R7 |  |
| 1c | 0064202a | 000000 | 00011 | 00100 | 00100 | 00000 | | | 101010 | | SLT | R4=msb[R3-R4] | Fig 10.7 |
| 20 | 10800001 | 000100 | 00100 | 00000 | 0001 | | | | | | BEQ | R4==R0 🡪 1 | Fig 10.8 |
| 24 | 20050000 | 001000 | 00000 | 00101 | 0000 | | | | | | ADDI | R5=R0+0000 | Fig 10.9 |
| 28 | 00e2202a | 000000 | 00111 | 00010 | 00100 | 00000 | | | 101010 | | SLT | R4=msb[R7-R2] | Fig 10.10 |
| 2C | 00853820 | 000000 | 00100 | 00101 | 00111 | 00000 | | | 100000 | | ADD | R7=R4+R5 | Fig 10.11 |
| 30 | 00e23822 | 000000 | 00111 | 00010 | 00111 | 00000 | | | 100010 | | SUB | R7=R7-R2 | Fig 10.12 |
| 34 | ac670044 | 101011 | 00011 | 00111 | 0044 | | | | | | SW | [R3+0044] =R7 | Fig 10.13 |
| 38 | 8c020050 | 100011 | 00000 | 00010 | 0050 | | | | | | LW | R2=[R0+0050] | Fig 10.14 |
| 3c | 08000011 | 000010 | 0000011 | | | | | | | | J | J 3 |  |
| 40 | 200a0008 | 001000 | 00000 | 01010 | 0008 | | | | | | ADDI | R10=R0+0008 | Fig 10.15 |
| 44 | 200a0005 | 001000 | 00000 | 01010 | 0005 | | | | | | ADDI | R10=R0+0005 | Fig 10.16 |
| 48 | 200b0006 | 001000 | 00000 | 01011 | 0006 | | | | | | ADDI | R11=R0+0006 | Fig 10.17 |
| 4c | 314c0001 | 001100 | 01010 | 01100 | 0001 | | | | | | Andi | R12=R10&0001 | Fig 10.18 |
| 50 | 356d0004 | 001101 | 01011 | 01101 | 0004 | | | | | | Ori | R13=R11|0004 | Fig 10.19 |
| 54 | 394e000a | 001110 | 01010 | 01110 | 000a | | | | | | Xori | R14=R10^000a | Fig 10.20 |
| 58 | 1c800001 | 000111 | 00100 | 00000 | 0001 | | | | | | BGTZ | R4>0 | Fig 10.21 |
| 5c | 04800001 | 000001 | 00100 | 00000 | 0001 | | | | | | BLTZ | R4<0 | Fig 10.22 |
| 60 | 2002006c | 001000 | 00000 | 00010 | 006c | | | | | | ADDI | R2=R0+0068 | Fig 10.23 |
| 64 | 00400008 | 000000 | 00010 | 00000 | 00000 | | | 00000 | | 001000 | Jr | Jr R2 |  |
| 68 | 200a000a | 001000 | 00000 | 01010 | 000a | | | | | | ADDI | R10=R0+0008 | Fig 10.24 |
| 6c | 008a7806 | 000000 | 00100 | 01010 | 01111 | | 00000 | | | 000110 | Srlv | R15=R10>>R4 | Fig 10.25 |
| 70 | 004a8026 | 000000 | 00010 | 01010 | 10000 | | 00000 | | | 100110 | Xor | R16=R2^R10 | Fig 10.26 |
| 74 | 8c020050 | 100011 | 00000 | 00010 | 0050 | | | | | | LW | R2=[R0+0050] | Fig 10.27 |
| 78 | ac020054 | 101011 | 00000 | 00010 | 0054 | | | | | | SW | [R0+0054] =R2 | Fig 10.28 |

**Table 10: Processor Instruction Execution Table**

#### **Waveform each important step:**

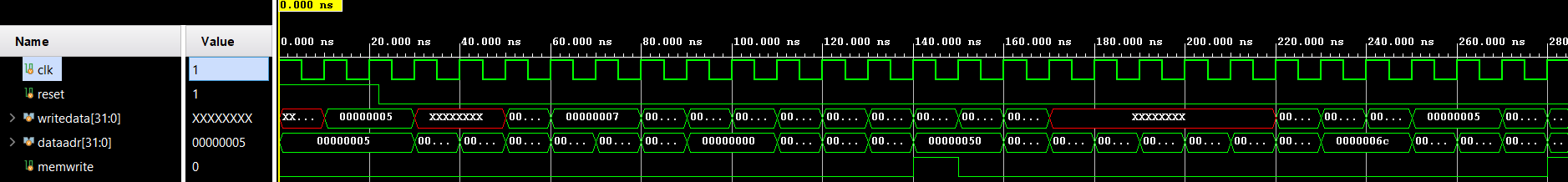


Fig 10.1: Wave form instruction

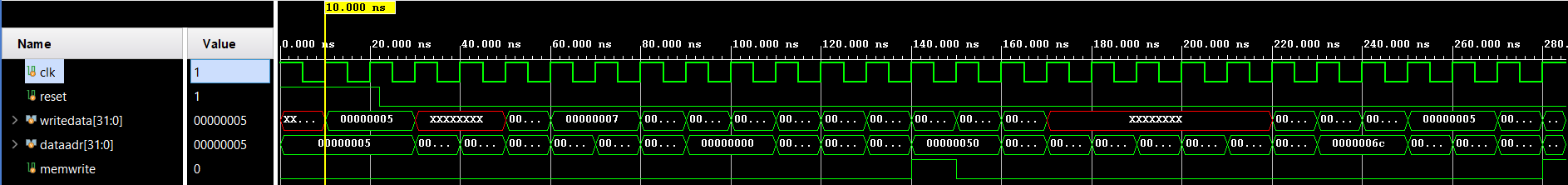


Fig 10.2: Wave form instruction

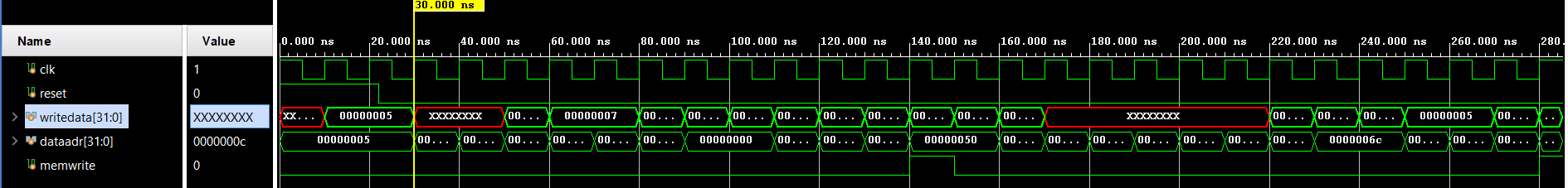


Fig 10.3: Wave form instruction

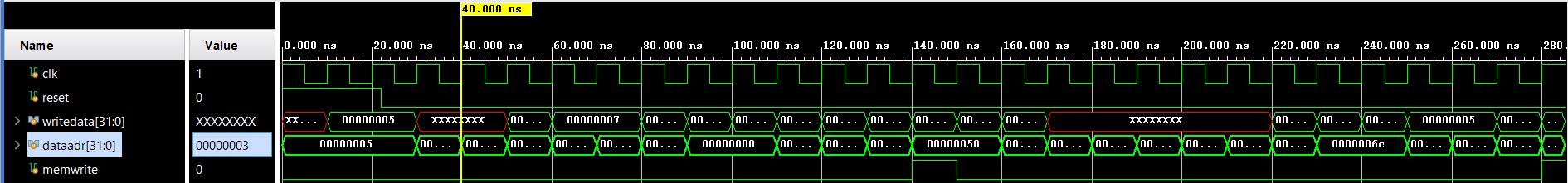


Fig 10.4: Wave form instruction

A black screen with green text

Description automatically generated

Fig 10.5: Wave form instruction

A black and green keyboard

Description automatically generated

Fig 10.6: Wave form instruction

A screen shot of a computer

Description automatically generated

Fig 10.7: Wave form instruction

A screenshot of a computer

Description automatically generated

Fig 10.8: Wave form instruction

A screen shot of a video game

Description automatically generated

Fig 10.9: Wave form instruction

A black screen with green lines

Description automatically generated

Fig 10.10: Wave form instruction

A screenshot of a computer

Description automatically generated

Fig 10.11: Wave form instruction

A screen shot of a computer

Description automatically generated

Fig 10.12: Wave form instruction

A screenshot of a computer

Description automatically generated

Fig 10.13: Wave form instruction

A screen shot of a computer

Description automatically generated

Fig 10.14: Wave form instruction

A black and green screen with numbers

Description automatically generated

Fig 10.15: Wave form instruction

A screenshot of a computer

Description automatically generated

Fig 10.16: Wave form instruction

A black screen with green lines

Description automatically generated

Fig 10.17: Wave form instruction

A black background with green lines

Description automatically generated

Fig 10.18: Wave form instruction

A black screen with green lines

Description automatically generated

Fig 10.19: Wave form instruction

A computer screen shot

Description automatically generated

Fig 10.20: Wave form instruction

A black keyboard with green lights

Description automatically generated

Fig 10.21: Wave form instruction

A computer screen shot

Description automatically generated

Fig 10.22: Wave form instruction

A black background with green text

Description automatically generated Fig 10.23: Wave form instruction

A black keyboard with green lines

Description automatically generated Fig 10.24: Wave form instruction

A black screen with green lines

Description automatically generated Fig 10.25: Wave form instruction

A computer screen shot

Description automatically generated Fig 10.26: Wave form instruction

A computer screen with green lines

Description automatically generated Fig 10.27: Wave form instruction

A black and green screen with green lines

Description automatically generated with medium confidence Fig 10.28: Wave form instruction

# Conclusion

The successful completion of this coursework underscores the importance of a deep understanding of microprocessor architecture and HDL programming in the realm of computer engineering (Harris & Harris, 2021). The extension of a MIPS-based single-cycle processor to support additional instructions, and the subsequent implementation and simulation using SystemVerilog, demonstrate not only a mastery of theoretical concepts but also the ability to apply these in a practical setting (Wakerly, 2018). The comprehensive nature of the tasks, from design modification to simulation, provided a holistic experience in processor development. The critical evaluation of each step, accompanied by appropriate illustrations and code segments, highlighted the significance of each design choice and its impact on processor performance and functionality. This coursework not only solidifies foundational knowledge in microprocessor design but also enhances skills in problem-solving, critical thinking, and effective communication, essential for future endeavours in the field of computer engineering (Harris & Harris, 2021; Wakerly, 2018).

# References:

Harris, D. & Harris, S. (2021). Digital Design and Computer Architecture: RISC-V Edition. Morgan Kaufmann.

Hyun, G. (2017, March 20). Computer Organization and Design 5th Edition Patterson Hennessy. Academia.edu. <https://www.academia.edu/31944631/Computer_Organization_and_Design_5th_Edition_Patterson_Hennessy>

MIPS Instruction Reference. (2023). Goucher.edu. https://phoenix.goucher.edu/~kelliher/f2009/cs220/mipsir.html

‌Wakerly, J. F. (2018). Digital Design: Principles and Practices. Pearson.

# Final Updated Code:

|  |
| --- |
| **Fig 11. Final Code** |