Iran University of Science and Technology

Pipelined MIPS Processor

Digital System Design Project

Fall 1402

Professor: Hajar Falahati

Zahra Alizadeh - Bahareh Kaousi nejad

Contents

0.1 Introduction

The 32-bit MIPS Processor is a key component in modern computing systems, widely used in various applications ranging from embedded systems to high-performance computing. This documentation presents the design and implementation of a MIPS Processor that incorporates hazard solving techniques using Forwarding. The processor is intended to be synthesized on an FPGA using the Verilog hardware description language, enabling its deployment in real-world hardware systems.

The primary objective of this project is to develop a high-performance MIPS Processor with efficient hazard handling mechanisms. Hazards, such as data hazards, control hazards, and structural hazards, can significantly impact the performance and correctness of pipelined processors. In this implementation, we leverage the concept of Forwarding to mitigate these hazards effectively, ensuring efficient data flow and reducing the number of pipeline stalls.

By utilizing Verilog as the hardware description language, our processor design can be synthesized into a hardware configuration suitable for implementation on an FPGA. This opens up possibilities for practical deployment in various domains, including embedded systems, digital signal processing, and computer architecture research.

In this documentation, we provide a comprehensive overview of the architecture, hazard handling techniques, and guidelines for FPGA synthesis. We also present performance evaluations and discuss potential areas for future enhancements and optimizations.

By documenting our implementation, we aim to contribute to the existing body of knowledge on MIPS Processor design, hazard handling techniques, and FPGA implementation. This documentation serves as a valuable resource for researchers, engineers, and enthusiasts interested in MIPS architecture, hazard solving mechanisms, and FPGA-based hardware implementations.

Now, let's delve into the details of our MIPS Processor, exploring its architecture, instruction set, hazard handling techniques, testing methodologies, and performance evaluations.

0.2 Architecture

The architecture of the 32-bit MIPS Processor with Hazard solving using Forwarding is designed to provide efficient and reliable execution of MIPS instructions while addressing hazards that can occur in a pipelined processor. This

section provides an overview of the major components and their functionalities, highlighting the key features that make our processor unique.

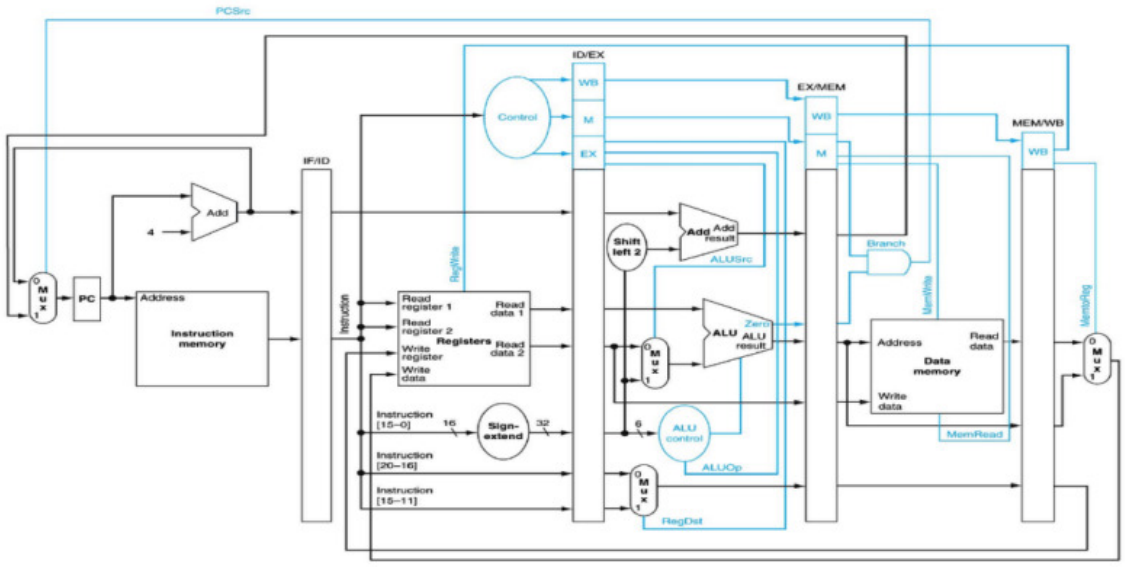

Figure 1: MIPS Basic Pipeline

At a high level, the processor consists of the following major components:

1. **Instruction Fetch (IF) Stage: The IF stage fetches instructions from memory based on the program counter (PC). It includes an instruction cache to minimize memory access latency and improve performance.**
2. **Instruction Decode (ID) Stage: The ID stage decodes the fetched instructions, extracting opcode, source and destination registers, immediate values, and control signals. It performs register file read operations and generates necessary control signals for subsequent stages.**
3. **Execution (EX) Stage: The EX stage performs arithmetic and logical operations specified by the instructions. It includes an Arithmetic Logic Unit (ALU) that supports a wide range of operations, such as addition, subtraction, logical AND/OR, and bit shifting.**
4. **Memory Access (MEM) Stage: The MEM stage handles memory-related operations, including data memory read and write operations. It also performs load and store instructions, ensuring correct memory access and data alignment.**
5. **Write Back (WB) Stage: The WB stage writes the results of the executed instructions back to the register file. It updates the destination registers with the computed values or the loaded data from memory.**

To handle hazards efficiently, our processor incorporates Forwarding mechanisms. Forwarding allows data to bypass certain pipeline stages and be directly

forwarded to the stages that require it, eliminating the need for pipeline stalls. This enables efficient data flow and minimizes the impact of hazards on the pipeline performance.

Additionally, our architecture includes hazard detection and control units that monitor the data dependencies between instructions and determine when forwarding is required. These units ensure correct instruction execution and maintain the program flow without introducing errors or stalls.

The architecture is designed to be modular and scalable, allowing for easy integration of additional components or extensions. It follows the MIPS instruction set architecture (ISA), with support for a wide range of instructions and addressing modes.

In the following sections, we will delve into each pipeline stage, discussing their functionalities, data flow, hazard handling techniques, and the interaction between stages. We will explore the implementation details of the Forwarding mechanism and its impact on performance.

Let's now explore the pipeline stages and their functionalities in detail.

0.3 Instruction Set and Pipeline Stages

Table 1: Opcode Name, Action, and Opcode bitfields

Opcode	**Name**	**Action**	**Opcode bitfields**
ADD rd,rs,rt	Add	rd = rs + rt	000000 rs rt rd 00000 100000
ADDI rt,rs,imm	Add Immediate	rt = rs + imm	001000 rs rt imm
AND rd,rs,rt	And	rd = rs & rt	000000 rs rt rd 00000 100100
ANDI rt,rs,imm	And Immediate	rt = rs & imm	001100 rs rt imm
OR rd,rs,rt	Or	rd = rs — rt	000000 rs rt rd 00000 100101
ORI rt,rs,imm	Or Immediate	rt = rs — imm	001101 rs rt imm
SUB rd,rs,rt	Subtract	rd = rs - rt	000000 rs rt rd 00000 100010
MULT rs,rt	Multiply	HI, LO = rs * rt	000000 rs rt 0000000000 011000
BEQ rs,rt,offset	Branch On Equal	if (rs == rt) pc += offset*4	000100 rs rt offset
BGEZ rs,offset	Branch On ≥ 0	if(rs ≥ 0) pc += offset*4	000001 rs 00001 offset
BGEZAL rs,offset	Branch On ≥ 0 And Link	r31 = pc; if(rs ≥ 0) pc += offset*4	000001 rs 10001 offset
BGTZ rs,offset	Branch On > 0	if (rs ¿ 0) pc += offset*4	000111 rs 00000 offset
BLEZ rs,offset	Branch On ≤ 0	if (rs ≤ 0) pc += offset*4	000110 rs 00000 offset
BNE rs,rt,offset	Branch On Not Equal	if (rs $\neq$ rt) pc += offset*4	000101 rs rt offset
J target	Jump	pc = pc_upper — (target¡¡2)	000010 target
JAL target	Jump And Link	r31 = pc; pc = target¡¡2	000011 target
JALR rs	Jump And Link Register	rd=pc; pc=rs	000000 rs 00000 rd 0 001001
JR rs	Jump Register	pc=rs	000000 rs 000000000000000 001000
LW rt,offset(rs)	Load Word	rt = *(int*)(offset+rs)	100011 rs rt offset
SW rt,offset(rs)	Store Word	*(int*)(offset+rs)=rt	101011 rs rt offset

0.3.1 ADD

Opcode Name: ADD
Action: rd = rs + rt
Opcode bitfields: 000000 rs rt rd 00000 100000

Explanation: The ADD instruction is an arithmetic instruction in MIPS that performs addition. It adds the values of registers rs and rt and stores the result in register rd. In the MIPS pipeline, the ADD instruction goes through several stages.
In the instruction fetch stage (IF stage), the instruction is fetched from memory using the program counter (PC) and stored in the instruction register (IR).
In the instruction decode stage (ID stage), the opcode and register operands (rs and rt) are extracted from the instruction.
In the execution stage (EX stage), the ALU (Arithmetic Logic Unit) performs the addition operation on the values in registers rs and rt.
In the memory stage (MEM stage), there is no memory access required for this instruction.
In the write-back stage (WB stage), the result of the addition operation is written back to register rd.

0.3.2 ADDI

Opcode Name: ADDI
Action: rt = rs + immediate
Opcode bitfields: 001000 rs rt immediate
Explanation: The ADDI instruction is an immediate arithmetic instruction in MIPS that performs addition. It adds the value of register rs with the sign-extended immediate value and stores the result in register rt. In the MIPS pipeline, the ADDI instruction goes through several stages.
In the instruction fetch stage (IF stage), the instruction is fetched from memory using the program counter (PC) and stored in the instruction register (IR).
In the instruction decode stage (ID stage), the opcode, register operand rs, and the immediate value are extracted from the instruction.
In the execution stage (EX stage), the ALU (Arithmetic Logic Unit) performs the addition operation on the value in register rs and the sign-extended immediate value.
In the memory stage (MEM stage), there is no memory access required for this instruction.
In the write-back stage (WB stage), the result of the addition operation is written back to register rt.
Below is an example code using ADD and ADDI with the simulation result:

```
addi $9, $0, 0x0025
addi $4, $0, 0x1000
add  $5, $4, $9
```

Listing 1: Example Assembly Code for ADD and ADDI

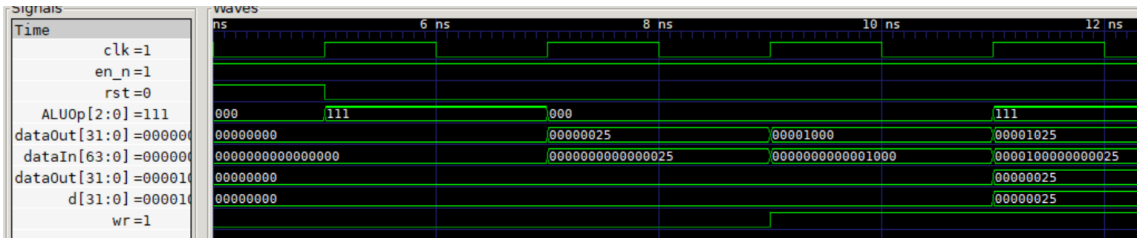

Figure 2: Signals for ADD Instruction - 1

Signal Descriptions:
ALU_out:

- Carries the result of the arithmetic or logical operation performed by the Arithmetic Logic Unit (ALU).
- Represents data that has undergone processing within the ALU.

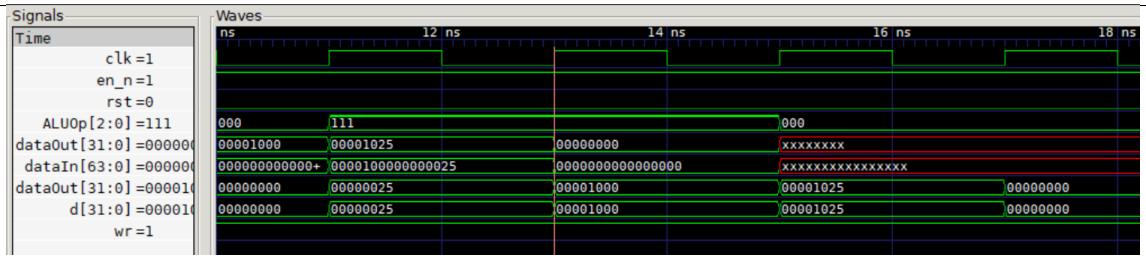

Figure 3: Signals for ADD Instruction - 2

- Serves as an input to other components, such as the register file or memory, for further utilization or storage.

Register File Inputs (concatenated):

- Combine multiple signals that provide data to be read from or written to the register file.

d (writedata):

- Carries the data to be written into a register within the register file. Activated when a write operation is initiated.

wr (writeenable):

- Control signal that enables or disables write operations to the register file.
- When wr is 1, writing to the specified register is permitted.
- When wr is 0, the register file maintains its current state, preventing data modifications.

Signal Behavior in Three Steps:

1. ALU Output Generation:
   - The ALU performs a calculation or logical operation based on its inputs.
   - The result is placed on the ALU_out signal.
2. Register File Input:
   - The register file receives necessary inputs for the intended operation.
   - These inputs include addresses for register selection, data to be written, and control signals.
3. Register Write:
   - The data on the d signal is written to the specified register within the register file.
   - The register file's contents are updated accordingly.

0.3.3 AND

Opcode Name: AND
Action: rd = rs AND rt
Opcode bitfields: 000000 rs rt rd 00000 100100
Explanation: The AND instruction is a logical instruction in MIPS that performs a bitwise AND operation. It performs a logical AND between the values of register rs and register rt and stores the result in register rd. In the MIPS pipeline, the AND instruction goes through several stages.

In the instruction fetch stage (IF stage), the instruction is fetched from memory using the program counter (PC) and stored in the instruction register (IR).
In the instruction decode stage (ID stage), the opcode and register operands (rs and rt) are extracted from the instruction.
In the execution stage (EX stage), the ALU (Arithmetic Logic Unit) performs the logical AND operation on the values in registers rs and rt.
In the memory stage (MEM stage), there is no memory access required for this instruction.
In the write-back stage (WB stage), the result of the logical AND operation is written back to register rd.

0.3.4 ANDI

Opcode Name: ANDI
Action: rt = rs AND immediate
Opcode bitfields: 001100 rs rt immediate
Explanation: The ANDI instruction is an immediate logical instruction in MIPS that performs a bitwise AND operation. It performs a logical AND between the value of register rs and the zero-extended immediate value and stores the result in register rt. In the MIPS pipeline, the ANDI instruction goes through several stages.
In the instruction fetch stage (IF stage), the instruction is fetched from memory using the program counter (PC) and stored in the instruction register (IR).
In the instruction decode stage (ID stage), the opcode, register operand rs, and the immediate value are extracted from the instruction.
In the execution stage (EX stage), the ALU (Arithmetic Logic Unit) performs the logical AND operation on the value in register rs and the zero-extended immediate value.
In the memory stage (MEM stage), there is no memory access required for this instruction.
In the write-back stage (WB stage), the result of the logical AND operation is written back to register rt.

0.3.5 OR

Opcode Name: OR
Action: rd = rs OR rt
Opcode bitfields: 000000 rs rt rd 00000 100101
Explanation: The OR instruction is a logical instruction in MIPS that performs a bitwise OR operation. It performs a logical OR between the values of register rs and register rt and stores the result in register rd. In the MIPS pipeline, the OR instruction goes through several stages.
In the instruction fetch stage (IF stage), the instruction is fetched from memory using the program counter (PC) and stored in the instruction register (IR).
In the instruction decode stage (ID stage), the opcode and register operands (rs and rt) are extracted from the instruction.
In the execution stage (EX stage), the ALU (Arithmetic Logic Unit) performs the logical OR operation on the values in registers rs and rt.
In the memory stage (MEM stage), there is no memory access required for this instruction.
In the write-back stage (WB stage), the result of the logical OR operation is written back to register rd.

0.3.6 ORI

Opcode Name: ORI
Action: rt = rs OR immediate
Opcode bitfields: 001101 rs rt immediate
Explanation: The ORI instruction is an immediate logical instruction in MIPS that performs a bitwise OR operation. It performs a logical OR between the value of register rs and the zero-extended immediate

value and stores the result in register rt. In the MIPS pipeline, the ORI instruction goes through several stages.
In the instruction fetch stage (IF stage), the instruction is fetched from memory using the program counter (PC) and stored in the instruction register (IR).
In the instruction decode stage (ID stage), the opcode, register operand rs, and the immediate value are extracted from the instruction.
In the execution stage (EX stage), the ALU (Arithmetic Logic Unit) performs the logical OR operation on the value in register rs and the zero-extended immediate value.
In the memory stage (MEM stage), there is no memory access required for this instruction.
In the write-back stage (WB stage), the result of the logical OR operation is written back to register rt.

0.3.7 SUB

Opcode Name: SUB
Action: rd = rs - rt
Opcode bitfields: 000000 rs rt rd 00000 100010
Explanation: The SUB instruction is an arithmetic instruction in MIPS that subtracts the value of register rt from the value of register rs and stores the result in register rd. In the MIPS pipeline, the SUB instruction goes through several stages.
In the instruction fetch stage (IF stage), the instruction is fetched from memory using the program counter (PC) and stored in the instruction register (IR).
In the instruction decode stage (ID stage), the opcode and register operands (rs and rt) are extracted from the instruction.
In the execution stage (EX stage), the ALU (Arithmetic Logic Unit) subtracts the value in register rt from the value in register rs.
In the memory stage (MEM stage), there is no memory access required for this instruction.
In the write-back stage (WB stage), the result of the subtraction operation is written back to register rd.

0.3.8 MULT

Opcode Name: MULT
Action: HI, LO = rs $\times rt$
Opcode bitfields: 000000 rs rt 00000 00000 011000
Explanation: The MULT instruction is a multiplication instruction in MIPS that multiplies the signed values of register rs and register rt and stores the 64-bit result in special registers HI (high-order 32 bits) and LO (low-order 32 bits). In the MIPS pipeline, the MULT instruction goes through several stages.
In the instruction fetch stage (IF stage), the instruction is fetched from memory using the program counter (PC) and stored in the instruction register (IR).
In the instruction decode stage (ID stage), the opcode and register operands (rs and rt) are extracted from the instruction.
In the execution stage (EX stage), the multiplication operation is performed by the hardware multiplier, and the 64-bit product is stored in the special registers HI and LO.
In the memory stage (MEM stage), there is no memory access required for this instruction.
In the write-back stage (WB stage), the result is not written to a general-purpose register but stored in the special registers HI and LO.

0.3.9 BEQ

Opcode Name: BEQ
Action: if (rs == rt) PC = PC + 4 + 4 * offset
Opcode bitfields: 000100 rs rt offset

Explanation: The BEQ instruction is a branch instruction in MIPS that performs a conditional branch based on the equality of the values in register rs and register rt. If the values are equal, the program counter (PC) is updated to the current PC plus 4 plus the sign-extended offset, which is shifted left by 2 bits to account for MIPS instruction alignment. This causes a branch to the target instruction. If the values are not equal, the branch is not taken, and the PC continues to the next sequential instruction.

0.3.10 BGEZ

Opcode Name: BGEZ
Action: if (rs ¿= 0) PC = PC + 4 + 4 * offset
Opcode bitfields: 000001 rs 00001 offset
Explanation: The BGEZ instruction is a branch instruction in MIPS that performs a conditional branch based on the sign of the value in register rs. If the value is greater than or equal to zero, the program counter (PC) is updated to the current PC plus 4 plus the sign-extended offset, which is shifted left by 2 bits to account for MIPS instruction alignment. This causes a branch to the target instruction. If the value is less than zero, the branch is not taken, and the PC continues to the next sequential instruction.

0.3.11 BGEZAL

Opcode Name: BGEZAL
Action: if (rs ¿= 0) { $31 = PC + 8; PC = PC + 4 + 4 * offset$ }
Opcode bitfields: 000001 rs 10001 offset
Explanation: The BGEZAL instruction is a branchinstruction in MIPS that performs a conditional branch based on the sign of the value in register rs. If the value is greater than or equal to zero, the program counter (PC) is updated to the current PC plus 4 plus the sign-extended offset, which is shifted left by 2 bits to account for MIPS instruction alignment. Additionally, the return address is stored in register 31(ra) before branching. If the value is less than zero, the branch is not taken, and the PC continues to the next sequential instruction.

0.3.12 BGTZ

Opcode Name: BGTZ
Action: if (rs ¿ 0) PC = PC + 4 + 4 * offset
Opcode bitfields: 000111 rs 00000 offset
Explanation: The BGTZ instruction is a branch instruction in MIPS that performs a conditional branch based on the value in register rs. If the value is greater than zero, the program counter (PC) is updated to the current PC plus 4 plus the sign-extended offset, which is shifted left by 2 bits to account for MIPS instruction alignment. This causes a branch to the target instruction. If the value is less than or equal to zero, the branch is not taken, and the PC continues to the next sequential instruction.

0.3.13 BLEZ

Opcode Name: BLEZ
Action: if (rs ¡= 0) PC = PC + 4 + 4 * offset
Opcode bitfields: 000110 rs 00000 offset
Explanation: The BLEZ instruction is a branch instruction in MIPS that performs a conditional branch based on the value in register rs. If the value is less than or equal to zero, the program counter (PC) is updated to the current PC plus 4 plus the sign-extended offset, which is shifted left by 2 bits to account for MIPS instruction alignment. This causes a branch to the target instruction. If the value is greater than zero, the branch is not taken, and the PC continues to the next sequential instruction.

0.3.14 BNE

Opcode Name: BNE
Action: if (rs != rt) PC = PC + 4 + 4 * offset
Opcode bitfields: 000101 rs rt offset
Explanation: The BNE instruction is a branch instruction in MIPS that performs a conditional branch based on the inequality of the values in register rs and register rt. If the values are not equal, the program counter (PC) is updated to the current PC plus 4 plus the sign-extended offset, which is shifted left by 2 bits to account for MIPS instruction alignment. This causes a branch to the target instruction. If the values are equal, the branch is not taken, and the PC continues to the next sequential instruction.

0.3.15 J

Opcode Name: J
Action: $PC = (PC\&0xf0000000) \| (target << 2)$
Opcode bitfields: 000010 target
Explanation: The J instruction is a jump instruction in MIPS that performs an unconditional jump to a target address. The target address is obtained by concatenating the upper 4 bits of the current PC with the 26-bit target field, shifted left by 2 bits. The current PC's upper 4 bits are preserved by the AND operation with the bit mask 0xf0000000. This allows for a jump within the same 256 MB region.
Below is an example code using BEQ and J instructions with the simulation result and terminal outputs:

```
addi $9, $0, 0x0003
addi $7, $0, 0x0000
loop:
beq $7, $9, end
addi $7, $7, 0x0001
j loop
end:
j end
```

Listing 2: Example Assembly Code for BEQ and J

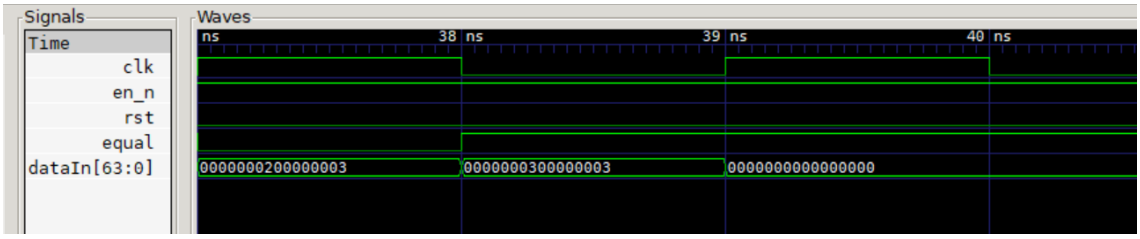

Figure 4: Signals for BEQ and J Instructions

Description of the Code Behavior:
Initialization:

- The code begins by setting a register to an initial value of 0. This register will act as a counter to control the loop's execution.

Loop Execution:

1. Increment: The register's value is increased by 1 in each iteration of the loop.
2. Comparison: A comparator circuit compares the updated register value with the target value of 3.

Figure 5: Terminal Results for BEQ and J Instructions

3. Flag Setting: If the register value matches the target value (3), the comparator's output, a flag register, is set to 1.
4. Branch Decision: The BEQ (branch if equal) instruction checks the state of the flag register.
5. Loop Termination: If the flag register is 1, the BEQ instruction causes a jump to the "end" label, ending the loop and program execution.
6. Loop Continuation: If the flag register is 0, the loop continues with the next iteration, starting again from step 1.

Program Termination:

- When the register value reaches 3, the flag register triggers the BEQ instruction, leading to a jump to the "end" label.
- Program execution ceases at the "end" label.

Key Points:

- The loop relies on a register as a counter to track the number of iterations.
- A comparator circuit determines when the target value is reached.
- A flag register signals loop completion.
- The BEQ instruction controls the program flow, enabling conditional branching.

0.3.16 JAL

Opcode Name: JAL
Action: $ra = PC + 4; PC = (PC\&0xf0000000) \| (target << 2)$
Opcode bitfields: 000011 target
Explanation: The JAL instruction is a jump-and-link instruction in MIPS that performs an unconditional jump to a target address and stores the return address in register ra (register 31). Similar to the J instruction, the target address is obtained by concatenating the upper 4 bits of the current PC with the 26-bit target field, shifted left by 2 bits. The current PC's upper 4 bits are preserved by the AND operation with the bit mask 0xf0000000. The return address is the address of the instruction following the JAL instruction.

0.3.17 JALR

Opcode Name: JALR
Action: rd = PC + 4; PC = rs
Opcode bitfields: 000000 rs rd 00000 001001
Explanation: The JALR instruction is a jump-and-link-register instruction in MIPS that performs an unconditional jump to the address stored in register rs and stores the return address in register rd. The return address is the address of the instruction following the JALR instruction. The contents of rs are loaded into the PC

0.3.18 JR

Opcode Name: JR
Action: PC = rs
Opcode bitfields: 000000 rs 00000 00000 000000
Explanation: The JR instruction is a jump-register instruction in MIPS that performs an unconditional jump to the address stored in register rs. The contents of rs are loaded into the PC, effectively changing the program flow to the target address.

0.3.19 LW

Opcode Name: LW
Action: $rt = Memory$[rs + offset]
Opcode bitfields: 100011 base rt offset
Explanation: The LW instruction is used to load a word from memory into a register. It takes the value stored in memory at the address formed by adding the contents of register rs and the sign-extended offset, and stores it in register rt. The offset is a 16-bit signed value, which is sign-extended to 32 bits before being added to the contents of rs.

0.3.20 SW

Opcode Name: SW
Action: Memory[rs + offset] = rt
Opcode bitfields: 101011 base rt offset
Explanation: The SW instruction is used to store a word from a register into memory. It takes the value stored in register rt and stores it in memory at the address formed by adding the contents of register rs and the sign-extended offset. The offset is a 16-bit signed value, which is sign-extended to 32 bits before being added to the contents of rs.

0.4 Hazard Handling

Efficient hazard handling is crucial in pipelined processors to ensure correct instruction execution and maintain high performance. Our MIPS Processor incorporates several hazard handling techniques, including the widely-used Forwarding mechanism, to mitigate hazards effectively. These techniques minimize pipeline stalls and maintain a smooth and uninterrupted flow of instructions through the pipeline. Let's explore these techniques in more detail:

1. **Forwarding: Forwarding, also known as data bypassing, allows data to be directly forwarded from the output of one pipeline stage to the input of another, bypassing intermediate stages. This technique eliminates the need for pipeline stalls caused by data hazards, where an instruction depends on the output of a previous instruction that is not yet available. By forwarding the data, instructions can proceed without waiting for the data to be written back to the register file. Our processor incorporates both forwarding from the EX stage to the ID and MEM stages, as well as forwarding from the MEM stage to the ID stage. This enables instructions in the ID stage to access the most up-to-date data, avoiding pipeline stalls and improving overall performance.**
2. **Hazard Detection Unit: To detect hazards, our processor includes a dedicated hazard detection unit. This unit examines the control and data dependencies between instructions in the pipeline and identifies potential hazards that may arise. It analyzes instructions in the ID stage and compares their**

register dependencies with instructions in later pipeline stages. The hazard detection unit detects hazards such as data hazards, control hazards, and structural hazards. It generates control signals that determine when forwarding is required and when pipeline stalls need to be inserted to resolve hazards.

3. Pipeline Stalls: Although our processor leverages forwarding to minimize pipeline stalls, some hazards may still require inserting pipeline stalls for correct instruction execution. For example, in the case of a control hazard, where a branch instruction changes the program counter (PC), pipeline stalls are necessary to ensure that the correct branch target instruction is fetched.

Our processor employs a stall control mechanism that inserts the appropriate number of pipeline stalls when required, ensuring proper instruction flow and maintaining data integrity.

It is important to note that while our hazard handling techniques significantly reduce the number of stalls, they do not eliminate all hazards. In certain cases, hazards may still result in pipeline stalls to ensure correctness and data consistency.

By implementing these hazard handling techniques, our MIPS Processor effectively mitigates hazards that could affect the performance and correctness of the pipeline. The combination of forwarding, hazard detection, and pipeline stalls ensures that our processor maintains a smooth and efficient execution of instructions, delivering high-performance computing capabilities.

0.5 Code Explanation

0.5.1 add r0

```
module add_r0 #(
    parameter DATA_WIDTH = 32
)(
    input [DATA_WIDTH - 1:0] input1,
    input [DATA_WIDTH - 1:0] input2,
    output[DATA_WIDTH - 1:0] dataOut,
    output C,
    output Z,
    output V,
    output S
);
reg [DATA_WIDTH:0] tmpAdd;
reg Ctmp, Ztmp, Vtmp, Stmp;
 always @(input1, input2) begin
    Ctmp = 0;
    Ztmp = 0;
    Vtmp = 0;
    Stmp = 0;

```

```
20      tmpAdd = input1 + input2;
21
22      Ctmp = tmpAdd[DATA_WIDTH];  // Carry Flag
23
24      if(tmpAdd[DATA_WIDTH-1:0] == {(DATA_WIDTH){1'b0}}) begin
25          Ztmp = 1;
26      end
27
28      if((input1[DATA_WIDTH - 1] == input2[DATA_WIDTH - 1]) &&
            (tmpAdd[DATA_WIDTH - 1] != input1[DATA_WIDTH - 1])) begin
29          Vtmp = 1;
30      end
31
32      Stmp = tmpAdd[DATA_WIDTH - 1];
33  end
34
35  assign dataOut = tmpAdd[DATA_WIDTH - 1:0];
36  assign C = Ctmp;
37  assign Z = Ztmp;
38  assign V = Vtmp;
39  assign S = Stmp;
40
41  endmodule
```

This module is an adder circuit that takes two input values and produces their sum. It also outputs flags for carry, zero, overflow, and sign.

0.5.2 alu controller r0

```
1   module alu_controller_r0 (
2       input [2:0] ALUOp,      // ALUOp from the main controller
3       input [5:0] funcode,    // Function code from instruction
4       output [4:0] ALUCtrl,   // ALU function
5       output JumpReg
6   );
7
8    // ALUOp Signals
9    localparam ALUadd = 3'b000;
10   localparam ALUsub = 3'b001;
11   localparam ALUand = 3'b010;
12   localparam ALUor  = 3'b011;
13   localparam ALUxor = 3'b100;
14   localparam ALUslt = 3'b101;
15   localparam ALURtp = 3'b111;
16
17   // Function codes
18   localparam fun_sll     = 6'h00;
19   localparam fun_srl     = 6'h02;
20   localparam fun_sra     = 6'h03;
21   localparam fun_sllv    = 6'h04;
22   localparam fun_srlv    = 6'h06;
23   localparam fun_srav    = 6'h07;
24   localparam fun_jr      = 6'h08;
25   //localparam fun_jalr  = 6'h09;
26   localparam fun_mfhi    = 6'h10;
27   localparam fun_mthi    = 6'h11;
```

```
28  localparam fun_mflo    = 6'h12;
29  localparam fun_mtlo    = 6'h13;
30  localparam fun_mult    = 6'h18;
31  localparam fun_multu   = 6'h19;
32  localparam fun_div     = 6'h1A;
33  localparam fun_divu    = 6'h1B;
34  localparam fun_add     = 6'h20;
35  localparam fun_addu    = 6'h21;
36  localparam fun_sub     = 6'h22;
37  localparam fun_subu    = 6'h23;
38  localparam fun_and     = 6'h24;
39  localparam fun_or      = 6'h25;
40  localparam fun_xor     = 6'h26;
41  localparam fun_nor     = 6'h27;
42  localparam fun_slt     = 6'h2A;
43  localparam fun_sltu    = 6'h2B;
44
45  always @(ALUOp, funcode) begin
46     JumpReg_tmp = 0;
47
48     //----------------------- NOT R-Type ---------------------------//
49     if(ALUOp == ALUadd) begin
50         ALUCtrl_tmp <= 5'b00000;
51     end else if(ALUOp == ALUsub) begin
52         ALUCtrl_tmp <= 5'b00001;
53     end else if(ALUOp == ALUand) begin
54         ALUCtrl_tmp <= 5'b01101;
55     end else if(ALUOp == ALUor) begin
56         ALUCtrl_tmp <= 5'b01110;
57     end else if(ALUOp == ALUxor) begin
58         ALUCtrl_tmp <= 5'b01111;
59     end else if(ALUOp == ALUslt) begin
60         ALUCtrl_tmp <= 5'b10001;
61
62     //------------------------ If R-Type --------------------------//
63     end else if((funcode == fun_add) || (funcode == fun_addu)) begin
64         ALUCtrl_tmp <= 5'b00000;
65     end else if((funcode == fun_sub) || (funcode == fun_subu)) begin
66         ALUCtrl_tmp <= 5'b00001;
67     end else if((funcode == fun_mult) || (funcode == fun_multu)) begin
68         ALUCtrl_tmp <= 5'b00010;
69     end else if(funcode == fun_sll) begin
70         ALUCtrl_tmp <= 5'b00011;
71     end else if(funcode == fun_sllv) begin
72         ALUCtrl_tmp <= 5'b00100;
73     end else if(funcode == fun_srl) begin
74         ALUCtrl_tmp <= 5'b00101;
75     end else if(funcode == fun_srlv) begin
76         ALUCtrl_tmp <= 5'b00110;
77     end else if(funcode == fun_sra) begin
78         ALUCtrl_tmp <= 5'b00111;
79     end else if(funcode == fun_srav) begin
80         ALUCtrl_tmp <= 5'b01000;
81     end else if(funcode == fun_mfhi) begin
82         ALUCtrl_tmp <= 5'b01001;
83     end else if(funcode == fun_mflo) begin
```

```
            ALUCtrl_tmp <= 5'b01010;
        end else if(funcode == fun_mthi) begin
            ALUCtrl_tmp <= 5'b01011;
        end else if(funcode == fun_mtlo) begin
            ALUCtrl_tmp <= 5'b01100;
        end else if(funcode == fun_and) begin
            ALUCtrl_tmp <= 5'b01101;
        end else if(funcode == fun_or) begin
            ALUCtrl_tmp <= 5'b01110;
        end else if(funcode == fun_xor) begin
            ALUCtrl_tmp <= 5'b01111;
        end else if(funcode == fun_nor) begin
            ALUCtrl_tmp <= 5'b10000;
        end else if(funcode == fun_slt) begin
            ALUCtrl_tmp <= 5'b10001;
        end else if(funcode == fun_sltu) begin
            ALUCtrl_tmp <= 5'b10010;
        end else if(funcode == fun_jr) begin
            ALUCtrl_tmp <= 5'b00000;
            JumpReg_tmp = 1;
        end
    end

    assign ALUCtrl = ALUCtrl_tmp;
    assign JumpReg = JumpReg_tmp;

endmodule
```

This module is an ALU (Arithmetic Logic Unit) controller that determines the ALU function code and jump signals based on input values. It provides the ALU function code and indicates whether a jump instruction is executed.

0.5.3 alu r0

```
module alu_r0 #(
    parameter DATA_WIDTH = 32,
    parameter CTRL_WIDTH = 5,
    parameter STATUS_WIDTH = 4,
    parameter SHAMT_WIDTH = 5,
    parameter DELAY = 0
)(
    input clk,
    input rst,
    input en_n,
    input [DATA_WIDTH*2-1:0] dataIn,
    input [CTRL_WIDTH-1:0] ctrl,
    input [SHAMT_WIDTH-1:0] shamt,
    output [DATA_WIDTH-1:0] dataOut,
    output [STATUS_WIDTH-1:0] status
);

`define PACK_ARRAY(PK_WIDTH,PK_DEPTH,PK_SRC,PK_DEST, BLOCK_ID, GEN_VAR)
    genvar GEN_VAR; generate for (GEN_VAR=0; GEN_VAR<(PK_DEPTH);
    GEN_VAR=GEN_VAR+1) begin: BLOCK_ID assign
    PK_DEST[((PK_WIDTH)*GEN_VAR+((PK_WIDTH)-1)):((PK_WIDTH)*GEN_VAR)] =
    PK_SRC[GEN_VAR][((PK_WIDTH)-1):0]; end endgenerate
```

```
19 `define UNPACK_ARRAY(PK_WIDTH,PK_DEPTH,PK_DEST,PK_SRC, BLOCK_ID, GEN_VAR)
       genvar GEN_VAR; generate for (GEN_VAR=0; GEN_VAR<(PK_DEPTH);
       GEN_VAR=GEN_VAR+1) begin: BLOCK_ID assign
       PK_DEST[GEN_VAR][((PK_WIDTH)-1):0] =
       PK_SRC[((PK_WIDTH)*GEN_VAR+(PK_WIDTH-1)):((PK_WIDTH)*GEN_VAR)]; end
       endgenerate
20
21 wire [DATA_WIDTH - 1:0] tmpIn [1:0];
22 reg [DATA_WIDTH - 1:0] outtmp;
23 wire [DATA_WIDTH - 1:0] addOut;
24 wire [DATA_WIDTH - 1:0] subOut;
25 wire [2*DATA_WIDTH - 1:0] multOut;
26 reg [STATUS_WIDTH - 1:0] statusTmp;
27 reg [DATA_WIDTH - 1:0] hi;
28 reg [DATA_WIDTH - 1:0] lo;
29 wire Cadd, Zadd, Vadd, Sadd, Csub, Zsub, Vsub, Ssub, Cmult, Zmult, Vmult,
       Smult;
30
31  `UNPACK_ARRAY(DATA_WIDTH,2,tmpIn,dataIn, U_BLK_0, idx_0)
32
33   delay #(
34     .BIT_WIDTH(DATA_WIDTH),
35     .DEPTH(1),
36     .DELAY(DELAY)
37  ) U_DEL(
38     .clk(clk),
39     .rst(rst),
40     .en_n(en_n),
41     .dataIn(outtmp),
42     .dataOut(dataOut)
43  );
44
45  delay #(
46     .BIT_WIDTH(STATUS_WIDTH),
47     .DEPTH(1),
48     .DELAY(DELAY)
49  ) U_DEL2(
50     .clk(clk),
51     .rst(rst),
52     .en_n(en_n),
53     .dataIn(statusTmp),
54     .dataOut(status)
55  );
56
57  add_r0 #(
58     .DATA_WIDTH(DATA_WIDTH)
59  )U_ADD(
60     .input1(tmpIn[1]),
61     .input2(tmpIn[0]),
62     .dataOut(addOut),
63     .C(Cadd),
64     .Z(Zadd),
65     .V(Vadd),
66     .S(Sadd)
67  );
68
```

```
69  sub_r0 #(
70      .DATA_WIDTH(DATA_WIDTH)
71  )U_SUB(
72      .input1(tmpIn[1]),
73      .input2(tmpIn[0]),
74      .dataOut(subOut),
75      .C(Csub),
76      .Z(Zsub),
77      .V(Vsub),
78      .S(Ssub)
79  );
80
81  mult_r0 #(
82      .DATA_WIDTH(DATA_WIDTH)
83  )U_MULT(
84      .input1(tmpIn[1]),
85      .input2(tmpIn[0]),
86      .dataOut(multOut),
87      .C(Cmult),
88      .Z(Zmult),
89      .V(Vmult),
90      .S(Smult)
91  );
92
93  always @(posedge clk) begin
94      if(rst == 1'b1) begin
95          hi <= {(DATA_WIDTH){1'b0}};
96          lo <= {(DATA_WIDTH){1'b0}};
97      end else if(ctrl == 5'b00010) begin
98          hi <= multOut[2*DATA_WIDTH - 1:DATA_WIDTH];
99          lo <= multOut[DATA_WIDTH-1:0];
100     end else if(ctrl == 5'b01011) begin // mthi
101         hi <= tmpIn[0];
102     end else if(ctrl == 5'b01100) begin // mtlo
103         lo <= tmpIn[0];
104     end
105 end
106
107 always @(dataIn, ctrl, shamt, addOut, subOut, multOut) begin
108     statusTmp = {(STATUS_WIDTH){1'b0}};
109
110     if(ctrl == 5'b00000) begin  // add, addu, addi, addiu
111         outtmp = addOut;
112         statusTmp[3] = Cadd;
113         statusTmp[2] = Zadd;
114         statusTmp[1] = Vadd;
115         statusTmp[0] = Sadd;
116     end else if(ctrl == 5'b00001) begin // sub, subu, subi, subiu
117         outtmp = subOut;
118         statusTmp[3] = Csub;
119         statusTmp[2] = Zsub;
120         statusTmp[1] = Vsub;
121         statusTmp[0] = Ssub;
122     end else if(ctrl == 5'b00010) begin // mult, multu
123         outtmp = multOut[DATA_WIDTH-1:0];
124         statusTmp[3] = Cmult;
```

```
125             statusTmp[2] = Zmult;
126             statusTmp[1] = Vmult;
127             statusTmp[0] = Smult;
128         end else if(ctrl == 5'b00011) begin // sll
129             outtmp = tmpIn[0] << shamt;
130         end else if(ctrl == 5'b00100) begin // sllv
131             outtmp = tmpIn[0] << tmpIn[1];
132         end else if(ctrl == 5'b00101) begin // srl
133             outtmp = tmpIn[0] >> shamt;
134         end else if(ctrl == 5'b00110) begin // srlv
135             outtmp = tmpIn[0] >> tmpIn[1];
136         end else if(ctrl == 5'b00111) begin // sra
137             outtmp = $signed(tmpIn[0]) >>> shamt;
138             //outtmp[DATA_WIDTH-1:DATA_WIDTH-shamt] = tmpIn[0][DATA_WIDTH-1];
139         end else if(ctrl == 5'b01000) begin // srav
140             outtmp = $signed(tmpIn[0]) >>> tmpIn[1];
141             //outtmp[DATA_WIDTH-1:DATA_WIDTH-shamt] = tmpIn[0][DATA_WIDTH-1];
142         end else if(ctrl == 5'b01001) begin // mfhi
143             outtmp = hi;
144         end else if(ctrl == 5'b01010) begin // mflo
145             outtmp = lo;
146         end else if(ctrl == 5'b01011) begin // mthi
147             //hi <= tmpIn[0];
148             outtmp = tmpIn[0];
149         end else if(ctrl == 5'b01100) begin // mtlo
150             //lo <= tmpIn[0];
151             outtmp = tmpIn[0];
152         end else if(ctrl == 5'b01101) begin
153             outtmp = tmpIn[0] & tmpIn[1];   // and, andi
154         end else if(ctrl == 5'b01110) begin
155             outtmp = tmpIn[0] | tmpIn[1];   // or, ori
156         end else if(ctrl == 5'b01111) begin
157             outtmp = tmpIn[0] ^ tmpIn[1];   // xor, xori
158         end else if(ctrl == 5'b10000) begin
159             outtmp = ~(tmpIn[0]) & ~(tmpIn[1]); // nor
160         end else if(ctrl == 5'b10001) begin
161             if($signed(tmpIn[1]) < $signed(tmpIn[0])) begin // slt
162                 outtmp = 32'h00000001;
163             end else begin
164                 outtmp = 32'h00000000;
165             end
166         end else if(ctrl == 5'b10010) begin
167             if($unsigned(tmpIn[1]) < $unsigned(tmpIn[0])) begin // sltu
168                 outtmp = 32'h00000001;
169             end else begin
170                 outtmp = 32'h00000000;
171             end
172         end
173 
174         if((ctrl != 5'b00000) && (ctrl != 5'b00001) && (ctrl != 5'b00010)) begin
175             if(outtmp[DATA_WIDTH-1:0] == {(DATA_WIDTH){1'b0}}) begin
176                 statusTmp[2] = 1;
177             end
178 
179             statusTmp[0] = outtmp[DATA_WIDTH-1];
180         end
```

```
181  end
182 endmodule
```

The ‘alu r0‘ module is a Verilog implementation of an Arithmetic Logic Unit (ALU). It performs various operations such as addition, subtraction, multiplication, shifting, and bitwise operations based on the control signal and input data. It provides the output result and status information, including carry, zero, overflow, and sign flags.

0.5.4 comparator r0

```
1  module comparator_r0 #(
2      parameter BIT_WIDTH = 32
3  )(
4      input [2*BIT_WIDTH - 1:0] dataIn,
5      output equal
6  );
7      always @(dataIn) begin
8          if(dataIn[2*BIT_WIDTH - 1:BIT_WIDTH] == dataIn[BIT_WIDTH - 1:0])
               begin
9              equal_tmp <= 1'b1;
10         end else begin
11             equal_tmp <= 1'b0;
12         end
13     end
14
15     assign equal = equal_tmp;
16
17 endmodule
```

The ‘comparator r0‘ module is a Verilog implementation of a comparator. It compares the input data to check if they are equivalent and sets the ‘equal‘ flag accordingly. The module has an internal signal ‘equal tmp‘ that holds the temporary value of the ‘equal‘ flag. The comparison is performed in the combinational logic block using an ‘always‘ block, and the ‘equal tmp‘ value is updated based on the comparison result. The ‘equal‘ output is assigned the value of ‘equal tmp‘.

0.5.5 controller r0

```
1  module controller_r0 (
2      input [5:0] opcode,
3      input [5:0] funcode,
4      output [1:0] RegDst,                  // 0 = 20:16, 1 = 15:11, 2 = $31
5      output ALUSrc,
6      output MemtoReg,
7      output RegWrite,
8      output [1:0] RegWriteSrc,    // 0 = output from memory, 1 = 16-bit
           left-shifted value for lui, 2 = PC + 4 for JAL
9      output MemRead,
10     output Jump,
11     output JumpRegID,
12     output BranchBEQ,
13     output BranchBNE,
14     output [2:0] ALUOp,
15     output isSigned
16 );
17
```

```
18  reg [1:0] RegDst_tmp;
19  reg ALUSrc_tmp;
20  reg MemtoReg_tmp;
21  reg RegWrite_tmp;
22  reg [1:0] RegWriteSrc_tmp;
23  reg MemRead_tmp;
24  reg BranchBEQ_tmp;
25  reg BranchBNE_tmp;
26  reg Jump_tmp;
27  reg JumpRegID_tmp;
28  reg [2:0] ALUOp_tmp;
29  reg isSigned_tmp;
30
31  localparam R_Type  = 6'h00;
32  localparam j       = 6'h02;
33  localparam jal     = 6'h03;
34  localparam beq     = 6'h04;
35  localparam bne     = 6'h05;
36  localparam blez    = 6'h06;
37  localparam bgtz    = 6'h07;
38  localparam addi    = 6'h08;
39  localparam addiu   = 6'h09;
40  localparam slti    = 6'h0A;
41  localparam sltiu   = 6'h0B;
42  localparam andi    = 6'h0C;
43  localparam ori     = 6'h0D;
44  localparam xori    = 6'h0E;
45  localparam lui     = 6'h0F;
46  localparam lw      = 6'h23;
47  localparam lbu     = 6'h24;
48  localparam lhu     = 6'h25;
49  //localparam lwr   = 6'h26;
50  localparam sb      = 6'h28;
51  localparam sh      = 6'h29;
52  //localparam swl   = 6'h2A;
53  localparam sw      = 6'h2B;
54  //localparam swr   = 6'h2E;
55
56  localparam ALUadd = 3'b000;    // for addi, addiu, lw, lbu, lhu, sb, sh, sw
57  localparam ALUsub = 3'b001;    // for beq, bne
58  localparam ALUand = 3'b010;    // for andi
59  localparam ALUor  = 3'b011;    // for ori
60  localparam ALUxor = 3'b100;    // for xori
61  localparam ALUslt = 3'b101;    // for slti, sltiu
62  localparam ALURtp = 3'b111;    // for all R-Type
63
64  localparam fun_jr      = 6'h08;
65  always @(opcode, funcode) begin
66     RegDst_tmp = 2'b00;
67     ALUSrc_tmp = 0;
68     MemtoReg_tmp = 0;
69     RegWrite_tmp = 0;
70     RegWriteSrc_tmp = 2'b00;
71     MemRead_tmp = 0;
72     BranchBEQ_tmp = 0;
73     BranchBNE_tmp = 0;
```

```
74      Jump_tmp = 0;
75      ALUOp_tmp = 3'b000;
76      isSigned_tmp = 0;
77
78      JumpRegID_tmp = 0;
79
80      case(opcode)
81          R_Type: begin
82              RegDst_tmp = 2'b01;
83              RegWrite_tmp = 1;
84              ALUOp_tmp = ALURtp;
85
86              if(funcode == fun_jr) begin
87                  JumpRegID_tmp = 1;
88              end
89          end
90
91          addi: begin
92              ALUSrc_tmp = 1;
93              RegWrite_tmp = 1;
94              ALUOp_tmp = ALUadd;
95              isSigned_tmp = 1;
96          end
97
98          addiu: begin
99              ALUSrc_tmp = 1;
100             RegWrite_tmp = 1;
101             ALUOp_tmp = ALUadd;
102         end
103
104         slti: begin
105             ALUSrc_tmp = 1;
106             ALUOp_tmp = ALUslt;
107             RegWrite_tmp = 1;
108             isSigned_tmp = 1;
109         end
110
111         sltiu: begin
112             ALUSrc_tmp = 1;
113             ALUOp_tmp = ALUslt;
114             RegWrite_tmp = 1;
115         end
116
117         andi: begin
118             ALUSrc_tmp = 1;
119             RegWrite_tmp = 1;
120             ALUOp_tmp = ALUand;
121         end
122
123         ori: begin
124             ALUSrc_tmp = 1;
125             RegWrite_tmp = 1;
126             ALUOp_tmp = ALUor;
127         end
128
129         xori: begin
```

```
130             ALUSrc_tmp = 1;
131             RegWrite_tmp = 1;
132             ALUOp_tmp = ALUxor;
133         end
134
135         lui: begin
136             RegWrite_tmp = 1;
137             RegWriteSrc_tmp = 2'b01;
138             isSigned_tmp = 1;
139         end
140
141         lbu: begin
142             ALUSrc_tmp = 1;
143             MemtoReg_tmp = 1;
144             RegWrite_tmp = 1;
145             MemRead_tmp = 1;
146             ALUOp_tmp = ALUadd;
147         end
148
149         lhu: begin
150             ALUSrc_tmp = 1;
151             MemtoReg_tmp = 1;
152             RegWrite_tmp = 1;
153             MemRead_tmp = 1;
154             ALUOp_tmp = ALUadd;
155         end
156
157         lw: begin
158             ALUSrc_tmp = 1;
159             MemtoReg_tmp = 1;
160             RegWrite_tmp = 1;
161             MemRead_tmp = 1;
162             ALUOp_tmp = ALUadd;
163             isSigned_tmp = 1;
164         end
165
166         sb: begin
167             ALUSrc_tmp = 1;
168             ALUOp_tmp = ALUadd;
169             isSigned_tmp = 1;
170         end
171
172         sh: begin
173             ALUSrc_tmp = 1;
174             ALUOp_tmp = ALUadd;
175             isSigned_tmp = 1;
176         end
177
178         sw: begin
179             ALUSrc_tmp = 1;
180             ALUOp_tmp = ALUadd;
181             isSigned_tmp = 1;
182         end
183
184         beq: begin
185             BranchBEQ_tmp = 1;
```

```
186             ALUOp_tmp = ALUsub;
187             isSigned_tmp = 1;
188         end
189 
190         bne: begin
191             BranchBNE_tmp = 1;
192             ALUOp_tmp = ALUsub;
193             isSigned_tmp = 1;
194         end
195 
196         j: begin
197             Jump_tmp = 1;
198             isSigned_tmp = 1;
199         end
200 
201         jal: begin
202             Jump_tmp = 1;
203             RegDst_tmp = 2'b10;
204             RegWrite_tmp = 1;
205             RegWriteSrc_tmp = 2'b10;
206             isSigned_tmp = 1;
207         end
208     endcase
209 
210 end
211 
212 assign RegDst = RegDst_tmp;
213 assign ALUSrc = ALUSrc_tmp;
214 assign MemtoReg = MemtoReg_tmp;
215 assign RegWrite = RegWrite_tmp;
216 assign RegWriteSrc = RegWriteSrc_tmp;
217 assign MemRead = MemRead_tmp;
218 assign BranchBEQ = BranchBEQ_tmp;
219 assign BranchBNE = BranchBNE_tmp;
220 assign Jump = Jump_tmp;
221 assign JumpRegID = JumpRegID_tmp;
222 assign ALUOp = ALUOp_tmp;
223 assign isSigned = isSigned_tmp;
224 
225 endmodule
```

The controller r0 module is a Verilog implementation of a main controller for a processor. It takes opcode and funcode inputs and generates control signals for various components of the processor. The module has several output signals that control different parts of the processor. Here is a brief description of the output signals:

- 'RegDst': Specifies the destination register for the result of an instruction.
- 'ALUSrc': Determines whether the second operand of the ALU should come from a register or an immediate value.
- 'MemtoReg': Specifies whether the data to be written to a register should come from memory.
- 'RegWrite': Enables or disables the write operation to a register.
- 'RegWriteSrc': Specifies the source for the data to be written to a register.
- 'MemRead': Enables or disables the read operation from memory.

- ‘Jump‘: Indicates whether a jump instruction is being executed.
- ‘JumpRegID‘: Specifies the type of jump instruction.
- ‘BranchBEQ‘: Indicates whether a branch on equal instruction is being executed.
- ‘BranchBNE‘: Indicates whether a branch on not equal instruction is being executed.
- ‘ALUOp‘: Specifies the operation to be performed by the ALU.
- ‘isSigned‘: Indicates whether the instruction involves signed data.

The module uses a combinational logic block to determine the values of these signals based on the input opcode and funcode. It assigns the calculated values to the corresponding output signals.

0.5.6 counter r0

```
module counter_r0 #(
    parameter MAX_COUNT = 4,
    parameter COUNT_WIDTH = log2(MAX_COUNT) + 1,
    parameter DELAY = 0
)(
    input clk,
    input rst,
    input load,
    input pause,
    input [COUNT_WIDTH - 1:0] countIn,
    output [COUNT_WIDTH - 1:0] countOut,

    // Delay
    input en_n

);

function integer log2; //This is a macro function (no hardware created) which finds the log2, returns log2
   input [31:0] val; //input to the function
   integer  i;
   begin
      log2 = 0;
      for(i = 0; 2**i < val; i = i + 1)
        log2 = i + 1;
   end
endfunction

reg [COUNT_WIDTH - 1:0] countValue; // Register to hold the count value
wire [COUNT_WIDTH - 1:0] countMax = MAX_COUNT;
 always @(posedge clk) begin
    if(rst) begin
        countValue <= {(COUNT_WIDTH){1'b0}};        // Reset the count to zero
    end else begin
        if(load) begin
            countValue <= countIn;                  // If load = 1 then set count value to the input value
        end else if(pause) begin
```

```
            countValue <= countValue;                // If pause = 1 hold the
                value
        end else if(countValue^countMax) begin
            countValue <= countValue + 1;            // Increase count value
                by 1
        end else begin
            countValue <= {(COUNT_WIDTH){1'b0}};     // If at MAX_COUNT next
                value is 0
        end
    end
 end

  delay #(
    .BIT_WIDTH(COUNT_WIDTH),
    .DEPTH(1),
    .DELAY(DELAY)
 ) U_IP(
    .clk(clk),
    .rst(rst),
    .en_n(en_n),
    .dataIn(countValue),
    .dataOut(countOut)
 );

 //assign countOut = countValue;         // Assign output

endmodule
```

This Verilog code implements a counter module with configurable parameters for maximum count value, count width, and delay. The counter increments on each clock cycle unless it reaches the maximum count value, in which case it wraps around to zero. The count value can be loaded from an input and can be paused based on the pause input. The final count value is passed through a delay component before being outputted as countOut.

0.5.7 datapath r0

```
module datapath_r0 #(
    parameter DATA_WIDTH = 32,
    parameter ADDR_WIDTH = 5
)(
    input clk,
    input rst,
    input en_n
);

wire [4:0] WriteReg;                    // Register to write to
wire JumpReg;

// IF
wire [DATA_WIDTH - 1:0] BranchOut;      // Output of Branch Mux
wire [DATA_WIDTH - 1:0] JumpOut;        // Output of Jump Mux
wire [DATA_WIDTH - 1:0] JumpRegOut;

reg  [DATA_WIDTH - 1:0] PC;          // PC
wire [DATA_WIDTH - 1:0] address;
wire [DATA_WIDTH - 1:0] PCPlus4;        // $PC + 4
```

```
21  wire [DATA_WIDTH - 1:0] instruction;
22
23  // IF/ID
24  wire [DATA_WIDTH - 1:0] IF_ID_PC;
25  wire [DATA_WIDTH - 1:0] IF_ID_PCPlus4;
26  wire [DATA_WIDTH - 1:0] IF_ID_Instruction;
27
28  // ID
29  wire isSigned;
30  wire Jump;
31  wire JumpRegID;
32  wire [1:0] RegDst;  // 0 = 20:16, 1 = 15:11, 2 = $31
33  wire ALUSrc;
34  wire [2:0] ALUOp;   // Input to the ALU Controller
35  wire [4:0] ALUCtrl; // Input to the ALU
36  wire BranchBEQ;
37  wire BranchBNE;
38  wire MemtoReg;
39  wire ID_MemRead;
40  wire RegWrite;
41  wire [1:0] RegWriteSrc; // 0 = output from memory, 1 = 16-bit left-shifted
        value for lui, 2 = PC + 4 for JAL
42
43  wire [2*DATA_WIDTH - 1:0] RegFileOut;   // 2 Outputs from the Reg file
44  wire equal;
45  wire [DATA_WIDTH - 1:0] SignExtOut; // Output of Sign Extender
46
47  // Hazard Detection Unit
48  wire PCWrite;
49  wire ID_EX_CtrlFlush;
50  wire IF_ID_Flush;
51  wire IF_ID_Hold;
52
53  // ID/EX
54  wire ID_EX_ALUSrc;              // ALU Source Select
55  wire [2:0] ID_EX_ALUOp;             // ALU Operation
56  wire [1:0] ID_EX_RegDst;                // Destination Reg Select
57
58  wire ID_EX_BranchBEQ;           // Pass through to MEM
59  wire ID_EX_BranchBNE;           // Pass through to MEM
60  wire [5:0] ID_EX_Opcode;        // Pass through to MEM
61
62  wire ID_EX_MemtoReg;            // Pass through to MEM
63  wire ID_EX_MemRead;
64  wire ID_EX_RegWrite;            // Pass through to MEM
65  wire [1:0] ID_EX_RegWriteSrc;   // Pass through to MEM
66
67
68  wire [DATA_WIDTH - 1:0] ID_EX_PCPlus4;
69  wire [2*DATA_WIDTH - 1:0] ID_EX_RegFileOut;
70  wire [DATA_WIDTH - 1:0] ID_EX_SignExtOut;
71  wire [4:0] ID_EX_Instruction25to21;
72  wire [4:0] ID_EX_Instruction20to16;
73  wire [4:0] ID_EX_Instruction15to11;
74
75  // EX
```

```
76  wire [DATA_WIDTH - 1:0] ALUSrcOut;  // Output of ALU Source Mux
77  wire [DATA_WIDTH - 1:0] ALUOut; // Output of ALU
78  wire [3:0] StatusReg;           // Status Register from ALU
79
80  // EX/MEM
81  wire EX_MEM_BranchBEQ;          // if BEQ
82  wire EX_MEM_BranchBNE;          // if BNE
83  wire [3:0] EX_MEM_StatusReg;            // status reg for use with branch,
        etc...
84  wire [5:0] EX_MEM_Opcode;       // Load or store opcode for Memory Controller
85
86  wire EX_MEM_MemtoReg;           // Pass through to WB
87  wire EX_MEM_RegWrite;           // Pass through to WB
88  wire [1:0] EX_MEM_RegWriteSrc;  // Pass through to WB
89
90  wire [DATA_WIDTH - 1:0] EX_MEM_BranchADD;           // Result of addition in
        EX stage
91  wire [DATA_WIDTH - 1:0] EX_MEM_PCPlus4;
92  wire [DATA_WIDTH - 1:0] EX_MEM_SignExtOut;
93  wire [DATA_WIDTH - 1:0] EX_MEM_ALUOut;
94  wire [DATA_WIDTH - 1:0] EX_MEM_ReadData2;
95  wire [ADDR_WIDTH - 1:0] EX_MEM_WriteReg;
96
97  // MEM
98  wire [5:0] ALUaddress;
99  wire [1:0] MemSelect;
100
101 wire [3:0] MemRead;
102 wire [3:0] MemWrite;
103 wire MemMux1Sel;
104 wire [1:0] MemMux2Sel;
105 wire [1:0] MemMux3Sel;
106 wire [7:0] MemMux1Out;
107 wire [7:0] MemMux2Out;
108 wire [7:0] MemMux3Out;
109
110 wire [DATA_WIDTH - 1:0] DataMemOut;     // Output from Data Memory
111 wire [DATA_WIDTH - 1:0] MemOut;
112
113 // MEM/WB
114 wire MEM_WB_MemtoReg;           // Choose between Memory Out or ALU Result
115 wire MEM_WB_RegWrite;           // Write enable for Reg File
116 wire [1:0] MEM_WB_RegWriteSrc;  // Choose between output of MemtoReg,
        Immediate Value, or $PC+4
117
118 wire [DATA_WIDTH - 1:0] MEM_WB_PCPlus4;
119 wire [DATA_WIDTH - 1:0] MEM_WB_SignExtOut;
120 wire [DATA_WIDTH - 1:0] MEM_WB_DataMemOut;
121 wire [DATA_WIDTH - 1:0] MEM_WB_MemOut;
122 wire [DATA_WIDTH - 1:0] MEM_WB_ALUOut;
123 wire [ADDR_WIDTH - 1:0] MEM_WB_WriteReg;
124
125 // WB
126 wire [DATA_WIDTH - 1:0] MemtoRegOut;    // Out of MemtoReg Mux
127 wire [DATA_WIDTH - 1:0] WriteData;      // Data Written to Reg file
128
```

```
129 // Forwarding Unit
130 wire [1:0] ForwardA;
131 wire [1:0] ForwardB;
132 wire [DATA_WIDTH - 1:0] ForwardAOut;
133 wire [DATA_WIDTH - 1:0] ForwardBOut;
134
135  // ------- Forwarding Components ------ //
136  forwardingUnit_r0 #(
137     .BIT_WIDTH(ADDR_WIDTH)
138  ) U_FWDUNIT (
139     .ID_EX_Rs(ID_EX_Instruction25to21),
140     .ID_EX_Rt(ID_EX_Instruction20to16),
141     .EX_MEM_Rd(EX_MEM_WriteReg),
142     .MEM_WB_Rd(MEM_WB_WriteReg),
143     .EX_MEM_RegWrite(EX_MEM_RegWrite),
144     .MEM_WB_RegWrite(MEM_WB_RegWrite),
145     .ForwardA(ForwardA),
146     .ForwardB(ForwardB)
147  );
148
149  mux #(
150     .BIT_WIDTH(DATA_WIDTH),
151     .DEPTH(3)
152  ) U_FWDA (
153     .dataIn({EX_MEM_ALUOut, MemtoRegOut, ID_EX_RegFileOut[2*DATA_WIDTH -
           1:DATA_WIDTH]}),
154     .sel(ForwardA),
155     .dataOut(ForwardAOut)
156  );
157
158  mux #(
159     .BIT_WIDTH(DATA_WIDTH),
160     .DEPTH(3)
161  ) U_FWDB (
162     .dataIn({EX_MEM_ALUOut, MemtoRegOut, ID_EX_RegFileOut[DATA_WIDTH -
           1:0]}),
163     .sel(ForwardB),
164     .dataOut(ForwardBOut)
165  );
166
167  // ----------- Hazard Detection Unit ------- //
168  hazardDetectionUnit_r0 U_HDU(
169     .IF_ID_Opcode(IF_ID_Instruction[31:26]),
170     .IF_ID_Funcode(IF_ID_Instruction[5:0]),
171     .IF_ID_Rs(IF_ID_Instruction[25:21]),
172     .IF_ID_Rt(IF_ID_Instruction[20:16]),
173     .ID_EX_MemRead(ID_EX_MemRead),
174     .ID_EX_Rt(ID_EX_Instruction20to16),
175     .equal(equal),
176     .ID_EX_Rd(WriteReg),
177     .EX_MEM_Rd(EX_MEM_WriteReg),
178     .ID_EX_RegWrite(ID_EX_RegWrite),
179     .EX_MEM_RegWrite(EX_MEM_RegWrite),
180     .PCWrite(PCWrite),
181     .ID_EX_CtrlFlush(ID_EX_CtrlFlush),
182     .IF_ID_Flush(IF_ID_Flush),
```

```
183     .IF_ID_Hold(IF_ID_Hold)
184 );
185 
186 // --------- PIPELINE REGS ------------ //
187 // IF/ID Register
188 delay #(
189     .BIT_WIDTH(DATA_WIDTH),
190     .DEPTH(3),
191     .DELAY(1)
192 ) U_IF_ID_REG (
193     .clk(clk),
194     .rst(IF_ID_Flush | rst),
195     .en_n(IF_ID_Hold),
196     .dataIn({PC, PCPlus4, instruction}),
197     .dataOut({IF_ID_PC, IF_ID_PCPlus4, IF_ID_Instruction})
198 );
199 
200 // ID/EX Register
201 // --- CONTROL PIPELINE REGS --- //
202 delay #(
203     .BIT_WIDTH(1),
204     .DEPTH(6),
205     .DELAY(1)
206 ) U_ID_EX_REG0 (
207     .clk(clk),
208     .rst(ID_EX_CtrlFlush | rst),
209     .en_n(1'b0),
210     .dataIn({ALUSrc, BranchBEQ, BranchBNE, MemtoReg, ID_MemRead, RegWrite}),
211     .dataOut({ID_EX_ALUSrc, ID_EX_BranchBEQ, ID_EX_BranchBNE,
            ID_EX_MemtoReg, ID_EX_MemRead, ID_EX_RegWrite})
212 );
213 
214  delay #(
215     .BIT_WIDTH(2),
216     .DEPTH(2),
217     .DELAY(1)
218 ) U_ID_EX_REG1 (
219     .clk(clk),
220     .rst(ID_EX_CtrlFlush | rst),
221     .en_n(1'b0),
222     .dataIn({RegDst, RegWriteSrc}),
223     .dataOut({ID_EX_RegDst, ID_EX_RegWriteSrc})
224 );
225 
226  delay #(
227     .BIT_WIDTH(3),
228     .DEPTH(1),
229     .DELAY(1)
230 ) U_ID_EX_REG2 (
231     .clk(clk),
232     .rst(ID_EX_CtrlFlush | rst),
233     .en_n(1'b0),
234     .dataIn(ALUOp),
235     .dataOut(ID_EX_ALUOp)
236 );
237 
```

```
238 // --- END CONTROL PIPELINE REGS --- //
239
240  delay #(
241     .BIT_WIDTH(6),
242     .DEPTH(1),
243     .DELAY(1)
244 ) U_ID_EX_REG3 (
245     .clk(clk),
246     .rst(rst),
247     .en_n(1'b0),
248     .dataIn(IF_ID_Instruction[31:26]),
249     .dataOut(ID_EX_Opcode)
250 );
251
252 delay #(
253     .BIT_WIDTH(DATA_WIDTH),
254     .DEPTH(4),
255     .DELAY(1)
256 ) U_ID_EX_REG4 (
257     .clk(clk),
258     .rst(rst),
259     .en_n(1'b0),
260     .dataIn({IF_ID_PCPlus4, RegFileOut, SignExtOut}),
261     .dataOut({ID_EX_PCPlus4, ID_EX_RegFileOut, ID_EX_SignExtOut})
262 );
263
264 delay #(
265     .BIT_WIDTH(ADDR_WIDTH),
266     .DEPTH(3),
267     .DELAY(1)
268 ) U_ID_EX_REG5 (
269     .clk(clk),
270     .rst(rst),
271     .en_n(1'b0),
272     .dataIn({IF_ID_Instruction[25:21], IF_ID_Instruction[20:16],
            IF_ID_Instruction[15:11]}),
273     .dataOut({ID_EX_Instruction25to21, ID_EX_Instruction20to16,
            ID_EX_Instruction15to11})
274 );
275
276 // EX/MEM Register
277 delay #(
278     .BIT_WIDTH(1),
279     .DEPTH(4),
280     .DELAY(1)
281 ) U_EX_MEM_REG0 (
282     .clk(clk),
283     .rst(rst),
284     .en_n(1'b0),
285     .dataIn({ID_EX_BranchBEQ, ID_EX_BranchBNE, ID_EX_MemtoReg,
            ID_EX_RegWrite}),
286     .dataOut({EX_MEM_BranchBEQ, EX_MEM_BranchBNE, EX_MEM_MemtoReg,
            EX_MEM_RegWrite})
287 );
288
289 delay #(
```

```
290     .BIT_WIDTH(2),
291     .DEPTH(1),
292     .DELAY(1)
293  ) U_EX_MEM_REG1 (
294     .clk(clk),
295     .rst(rst),
296     .en_n(1'b0),
297     .dataIn(ID_EX_RegWriteSrc),
298     .dataOut(EX_MEM_RegWriteSrc)
299  );
300
301   delay #(
302     .BIT_WIDTH(4),
303     .DEPTH(1),
304     .DELAY(1)
305  ) U_EX_MEM_REG2 (
306     .clk(clk),
307     .rst(rst),
308     .en_n(1'b0),
309     .dataIn(StatusReg),
310     .dataOut(EX_MEM_StatusReg)
311  );
312
313   delay #(
314     .BIT_WIDTH(6),
315     .DEPTH(1),
316     .DELAY(1)
317  ) U_EX_MEM_REG3 (
318     .clk(clk),
319     .rst(rst),
320     .en_n(1'b0),
321     .dataIn(ID_EX_Opcode),
322     .dataOut(EX_MEM_Opcode)
323  );
324
325   delay #(
326     .BIT_WIDTH(DATA_WIDTH),
327     .DEPTH(5),
328     .DELAY(1)
329  ) U_EX_MEM_REG4 (
330     .clk(clk),
331     .rst(rst),
332     .en_n(1'b0),
333     .dataIn({{ID_EX_SignExtOut[29:0], 2'b00} + ID_EX_PCPlus4, ID_EX_PCPlus4,
           ID_EX_SignExtOut, ALUOut, ForwardBOut}),
334     .dataOut({EX_MEM_BranchADD, EX_MEM_PCPlus4, EX_MEM_SignExtOut,
           EX_MEM_ALUOut, EX_MEM_ReadData2})
335  );
336
337    delay #(
338     .BIT_WIDTH(ADDR_WIDTH),
339     .DEPTH(1),
340     .DELAY(1)
341  ) U_EX_MEM_REG5 (
342     .clk(clk),
343     .rst(rst),
```

```
344     .en_n(1'b0),
345     .dataIn(WriteReg),
346     .dataOut(EX_MEM_WriteReg)
347 );
348 
349 // MEM/WB Register
350 delay #(
351     .BIT_WIDTH(1),
352     .DEPTH(2),
353     .DELAY(1)
354 ) U_MEM_WB_REG0 (
355     .clk(clk),
356     .rst(rst),
357     .en_n(1'b0),
358     .dataIn({EX_MEM_MemtoReg, EX_MEM_RegWrite}),
359     .dataOut({MEM_WB_MemtoReg, MEM_WB_RegWrite})
360 );
361 
362  delay #(
363     .BIT_WIDTH(2),
364     .DEPTH(1),
365     .DELAY(1)
366 ) U_MEM_WB_REG1 (
367     .clk(clk),
368     .rst(rst),
369     .en_n(1'b0),
370     .dataIn(EX_MEM_RegWriteSrc),
371     .dataOut(MEM_WB_RegWriteSrc)
372 );
373 
374 delay #(
375     .BIT_WIDTH(DATA_WIDTH),
376     .DEPTH(5),
377     .DELAY(1)
378 ) U_MEM_WB_REG2 (
379     .clk(clk),
380     .rst(rst),
381     .en_n(1'b0),
382     .dataIn({EX_MEM_PCPlus4, EX_MEM_SignExtOut, DataMemOut, MemOut,
            EX_MEM_ALUOut}),
383     .dataOut({MEM_WB_PCPlus4, MEM_WB_SignExtOut, MEM_WB_DataMemOut,
            MEM_WB_MemOut, MEM_WB_ALUOut})
384 );
385 
386 delay #(
387     .BIT_WIDTH(ADDR_WIDTH),
388     .DEPTH(1),
389     .DELAY(1)
390 ) U_MEM_WB_REG3 (
391     .clk(clk),
392     .rst(rst),
393     .en_n(1'b0),
394     .dataIn(EX_MEM_WriteReg),
395     .dataOut(MEM_WB_WriteReg)
396 );
397 
```

```
398
399 // ----- INSTRUCTION FETCH (IF) ----- //
400  mux #(
401    .BIT_WIDTH(DATA_WIDTH),
402    .DEPTH(2)
403 )U_BRANCHMUX(
404    .dataIn({{SignExtOut[29:0], 2'b00} + IF_ID_PCPlus4, PCPlus4}),
405    .sel((BranchBEQ & equal) | (BranchBNE & ~equal)),
406    .dataOut(BranchOut)
407 );
408
409  mux #(
410    .BIT_WIDTH(DATA_WIDTH),
411    .DEPTH(2)
412 )U_JUMPMUX(
413    .dataIn({{IF_ID_PCPlus4[31:28], {IF_ID_Instruction[25:0], 2'b00}},
       BranchOut}),
414    .sel(Jump),
415    .dataOut(JumpOut)
416 );
417
418  mux #(
419    .BIT_WIDTH(DATA_WIDTH),
420    .DEPTH(2)
421 )U_JUMPREGMUX(
422    .dataIn({RegFileOut[2*DATA_WIDTH - 1:DATA_WIDTH], JumpOut}),
423    .sel(JumpRegID),
424    .dataOut(JumpRegOut)
425 );
426
427  rom U_rom(
428    .q(instruction),
429    .a(address[6:0])
430 );
431
432 // ----- INSTRUCTION DECODE (ID) ----- //
433  controller_r0 U_CONTROLLER(
434    .opcode(IF_ID_Instruction[31:26]),
435    .funcode(IF_ID_Instruction[5:0]),
436    .RegDst(RegDst),
437    .ALUSrc(ALUSrc),
438    .MemtoReg(MemtoReg),
439    .MemRead(ID_MemRead),
440    .RegWrite(RegWrite),
441    .RegWriteSrc(RegWriteSrc),
442    .Jump(Jump),
443    .JumpRegID(JumpRegID),
444    .BranchBEQ(BranchBEQ),
445    .BranchBNE(BranchBNE),
446    .ALUOp(ALUOp),
447    .isSigned(isSigned)
448 );
449
450  registerFile #(
451    .DATA_WIDTH(DATA_WIDTH),
452    .RD_DEPTH(2),
```

```
453     .REG_DEPTH(32),
454     .ADDR_WIDTH(ADDR_WIDTH)
455 )U_REGFILE(
456     .clk(clk),
457     .rst(rst),
458     .wr(MEM_WB_RegWrite),
459     .rr({IF_ID_Instruction[25:21], IF_ID_Instruction[20:16]}),
460     .rw(MEM_WB_WriteReg),
461     .d(WriteData),
462     .q(RegFileOut)
463 );
464
465 comparator_r0 #(
466     .BIT_WIDTH(DATA_WIDTH)
467 ) U_COMPARE(
468     .dataIn(RegFileOut),
469     .equal(equal)
470 );
471
472  signextender_r0 #(
473     .IN_WIDTH(16),
474     .OUT_WIDTH(DATA_WIDTH),
475     .DEPTH(1),
476     .DELAY(0)
477 )U_SIGNEXTENDER(
478     .clk(clk),
479     .rst(rst),
480     .en_n(en_n),
481     .dataIn(IF_ID_Instruction[15:0]),
482     .dataOut(SignExtOut),
483     .isSigned(isSigned)
484 );
485
486 // ----- EXECUTE (EX) ----- //
487  mux #(
488     .BIT_WIDTH(DATA_WIDTH),
489     .DEPTH(2)
490 )U_ALUSRCMUX(
491     //.dataIn({ID_EX_SignExtOut, ID_EX_RegFileOut[DATA_WIDTH - 1:0]}),
492     .dataIn({ID_EX_SignExtOut, ForwardBOut}),
493     .sel(ID_EX_ALUSrc),
494     .dataOut(ALUSrcOut)
495 );
496
497  alu_controller_r0 U_ALUCONTROLLER(
498     .ALUOp(ID_EX_ALUOp),
499     .funcode(ID_EX_SignExtOut[5:0]),
500     .ALUCtrl(ALUCtrl),
501     .JumpReg(JumpReg)
502 );
503
504  alu_r0 #(
505     .DATA_WIDTH(DATA_WIDTH),
506     .CTRL_WIDTH(5),
507     .STATUS_WIDTH(4),
508     .SHAMT_WIDTH(5),
```

```
509     .DELAY(0)
510 )U_ALU(
511     .clk(clk),
512     .rst(rst),
513     .en_n(en_n),
514     .dataIn({ForwardAOut, ALUSrcOut}),
515     .ctrl(ALUCtrl),
516     .shamt(ID_EX_SignExtOut[10:6]),
517     .dataOut(ALUOut),
518     .status(StatusReg)
519 );
520 
521  mux #(
522     .BIT_WIDTH(ADDR_WIDTH),
523     .DEPTH(3)
524 )U_REGDSTMUX(
525     .dataIn({5'b11111, ID_EX_Instruction15to11, ID_EX_Instruction20to16}),
526     .sel(ID_EX_RegDst),
527     .dataOut(WriteReg)
528 );
529 
530 // ----- MEMORY (MEM) ----- //
531 // ----- Memory Muxes ----- //
532 mux #(
533     .BIT_WIDTH(8),
534     .DEPTH(2)
535 ) U_MEMMUX1(
536     .dataIn({EX_MEM_ReadData2[15:8], EX_MEM_ReadData2[7:0]}),
537     .sel(MemMux1Sel),
538     .dataOut(MemMux1Out)
539 );
540 
541 mux #(
542     .BIT_WIDTH(8),
543     .DEPTH(3)
544 ) U_MEMMUX2(
545     .dataIn({EX_MEM_ReadData2[23:16], EX_MEM_ReadData2[15:8],
            EX_MEM_ReadData2[7:0]}),
546     .sel(MemMux2Sel),
547     .dataOut(MemMux2Out)
548 );
549 
550 mux #(
551     .BIT_WIDTH(8),
552     .DEPTH(3)
553 ) U_MEMMUX3(
554     .dataIn({EX_MEM_ReadData2[31:24], EX_MEM_ReadData2[15:8],
            EX_MEM_ReadData2[7:0]}),
555     .sel(MemMux3Sel),
556     .dataOut(MemMux3Out)
557 );
558 
559 // ---- Data Memory ---- //
560 ram U_ram0(
561     .q(DataMemOut[7:0]),
562     .d(EX_MEM_ReadData2[7:0]),
```

```
563     .a(ALUaddress),
564     .rst(rst),
565     .we(MemWrite[0]),
566     .re(MemRead[0]),
567     .clk(clk)
568 );
569 
570 ram U_ram1(
571     .q(DataMemOut[15:8]),
572     .d(MemMux1Out),
573     .a(ALUaddress),
574     .rst(rst),
575     .we(MemWrite[1]),
576     .re(MemRead[1]),
577     .clk(clk)
578 );
579 
580 ram U_ram2(
581     .q(DataMemOut[23:16]),
582     .d(MemMux2Out),
583     .a(ALUaddress),
584     .rst(rst),
585     .we(MemWrite[2]),
586     .re(MemRead[2]),
587     .clk(clk)
588 );
589 
590 ram U_ram3(
591     .q(DataMemOut[31:24]),
592     .d(MemMux3Out),
593     .a(ALUaddress),
594     .rst(rst),
595     .we(MemWrite[3]),
596     .re(MemRead[3]),
597     .clk(clk)
598 );
599 
600  memout_r0 # (
601     .DATA_WIDTH(8)
602 ) U_MEMOUT (
603     .Mem0Out(DataMemOut[7:0]),
604     .Mem1Out(DataMemOut[15:8]),
605     .Mem2Out(DataMemOut[23:16]),
606     .Mem3Out(DataMemOut[31:24]),
607     .MemSel(MemSelect),
608     .Opcode(EX_MEM_Opcode),
609     .MemOut(MemOut)
610 
611 );
612 
613 memcontroller_r0 U_MEMCONTROLLER(
614     .opcode(EX_MEM_Opcode),
615     .MemSelect(MemSelect),
616     .MemWrite(MemWrite),
617     .MemRead(MemRead),
618     .MemMux1Sel(MemMux1Sel),
```

```
619     .MemMux2Sel(MemMux2Sel),
620     .MemMux3Sel(MemMux3Sel)
621  );
622
623  // ----- WWRITE BACK (WB) ----- //
624   mux #(
625     .BIT_WIDTH(DATA_WIDTH),
626     .DEPTH(2)
627  )U_MEMTOREGMUX(
628     .dataIn({MEM_WB_MemOut, MEM_WB_ALUOut}),
629     .sel(MEM_WB_MemtoReg),
630     .dataOut(MemtoRegOut)
631  );
632
633   mux #(
634     .BIT_WIDTH(DATA_WIDTH),
635     .DEPTH(3)
636  )U_REGWRITESRCMUX(
637     .dataIn({MEM_WB_PCPlus4, {MEM_WB_SignExtOut[15:0], 16'h0000},
           MemtoRegOut}),
638     .sel(MEM_WB_RegWriteSrc),
639     .dataOut(WriteData)
640  );
641
642  assign PCPlus4 = PC + 4;
643  assign address = PC >> 2;  // Shift PC by two since ROM is byte-addressable
644
645  // ------------------ Memory Signals ---------------------- //
646  assign ALUaddress = EX_MEM_ALUOut[7:2];    // Calculated Address bits
647  assign MemSelect = EX_MEM_ALUOut[1:0]; // Select bits from ALU output
648
649  always @(posedge clk) begin
650     if(rst == 1'b1) begin
651         PC <= {(DATA_WIDTH){1'b0}};
652     end else begin
653         if(PCWrite) begin
654             PC <= JumpRegOut;
655         end
656     end
657  end
658 endmodule
```

Datapath implementation of the processor includes an instance of each module and takes care of the PC.

0.5.8 delay r0

```
1  module delay_r0 #(
2      parameter BIT_WIDTH = 4,
3      parameter DEPTH = 2,
4      parameter DELAY = 4
5  )(
6      input clk,
7      input rst,
8      input en_n,
9      input [BIT_WIDTH*DEPTH - 1:0] dataIn,
10     output [BIT_WIDTH*DEPTH - 1:0] dataOut
```

```
);
    `define PACK_ARRAY(PK_WIDTH,PK_DEPTH,PK_SRC,PK_DEST, BLOCK_ID, GEN_VAR)
          genvar GEN_VAR; generate for (GEN_VAR=0; GEN_VAR<(PK_DEPTH);
       GEN_VAR=GEN_VAR+1) begin: BLOCK_ID assign
       PK_DEST[((PK_WIDTH)*GEN_VAR+((PK_WIDTH)-1)):((PK_WIDTH)*GEN_VAR)] =
       PK_SRC[GEN_VAR][((PK_WIDTH)-1):0]; end endgenerate
    `define UNPACK_ARRAY(PK_WIDTH,PK_DEPTH,PK_DEST,PK_SRC, BLOCK_ID,
       GEN_VAR)  genvar GEN_VAR; generate for (GEN_VAR=0;
       GEN_VAR<(PK_DEPTH); GEN_VAR=GEN_VAR+1) begin: BLOCK_ID assign
       PK_DEST[GEN_VAR][((PK_WIDTH)-1):0] =
       PK_SRC[((PK_WIDTH)*GEN_VAR+(PK_WIDTH-1)):((PK_WIDTH)*GEN_VAR)]; end
       endgenerate

    integer i,j; //iterators
    wire [BIT_WIDTH - 1:0] tmp [DEPTH - 1:0]; //input as array
    wire [BIT_WIDTH*DEPTH - 1:0] tmpOut; //wire for output, more of this at
       end
    reg [BIT_WIDTH - 1:0] pipe [DELAY-1:0][DEPTH - 1:0]; //data pipeline
    `UNPACK_ARRAY(BIT_WIDTH,DEPTH,tmp,dataIn, U_BLK_0, idx_0)
    always @(posedge clk) begin
        if(rst == 1'b1) begin
            for(j=0; j<DELAY; j=j+1) begin //For all delay layers
                for(i=0; i<DEPTH; i=i+1) begin //For all depth of input array
                    pipe[j][i] <= {(BIT_WIDTH){1'b0}};
                end
            end
        end

        else begin

            // Pipeline delay
            if(en_n == 1'b0) begin
                for(i=0;i<DEPTH; i=i+1) begin //For all depth of input array
                    pipe[0][i] <= tmp[i];
                end
            end
            for(i=0; i<DELAY-1; i=i+1) begin
                for(j=0;j<DEPTH; j=j+1) begin //For all depth of input array
                    pipe[i+1][j] <= pipe[i][j]; //Pipe shifts makes delay
                end
            end
        end
    end

    `PACK_ARRAY(BIT_WIDTH,DEPTH,pipe[(DELAY-1)],tmpOut,U_BLK_1,idx_1)

    generate
    if(DELAY > 0)
        assign dataOut = tmpOut;

    else
        assign dataOut = dataIn;
    endgenerate

endmodule
```

This Verilog code implements a delay module that introduces a configurable number of clock cycles de-

lay to the input data. The delay is achieved by using a pipeline structure with multiple stages. The input data is loaded into the first stage of the pipeline, and in each clock cycle, the data is shifted through the pipeline stages. The delayed data is then available at the output. The module also includes logic for resetting the pipeline and handling the case where the delay is set to zero.

0.5.9 rom

```
// Instruction Memory
// 128 Words Long
// $readmemh will load the given program from the .hex file

module rom(q, a);
   output[31:0] q;
   input [6:0] a;

   reg [31:0] mem [127:0];

   initial $readmemh("data.hex", mem, 0, 127) ;
   assign q = mem[a];

endmodule
```

This ROM module acts as a storage unit for instructions, providing them to other components of the digital system when requested using an address.

0.5.10 signextender r0

```
module signextender_r0 #(
    parameter IN_WIDTH = 16,
    parameter OUT_WIDTH = 32,
    parameter DEPTH = 1,
    parameter DELAY = 0
)(
    input [DEPTH*IN_WIDTH - 1:0] dataIn,
    output [DEPTH*OUT_WIDTH - 1:0] dataOut,
    input isSigned,

    // Delay Inputs
    input clk,
    input rst,
    input en_n
);

`define PACK_ARRAY(PK_WIDTH,PK_DEPTH,PK_SRC,PK_DEST, BLOCK_ID, GEN_VAR)
    genvar GEN_VAR; generate for (GEN_VAR=0; GEN_VAR<(PK_DEPTH);
    GEN_VAR=GEN_VAR+1) begin: BLOCK_ID assign
    PK_DEST[((PK_WIDTH)*GEN_VAR+((PK_WIDTH)-1)):((PK_WIDTH)*GEN_VAR)] =
    PK_SRC[GEN_VAR][((PK_WIDTH)-1):0]; end endgenerate
`define UNPACK_ARRAY(PK_WIDTH,PK_DEPTH,PK_DEST,PK_SRC, BLOCK_ID, GEN_VAR)
    genvar GEN_VAR; generate for (GEN_VAR=0; GEN_VAR<(PK_DEPTH);
    GEN_VAR=GEN_VAR+1) begin: BLOCK_ID assign
    PK_DEST[GEN_VAR][((PK_WIDTH)-1):0] =
    PK_SRC[((PK_WIDTH)*GEN_VAR+(PK_WIDTH-1)):((PK_WIDTH)*GEN_VAR)]; end
    endgenerate

```

```
wire [IN_WIDTH - 1:0] tmpIn [DEPTH - 1:0];
reg [OUT_WIDTH - 1:0] extTmp [DEPTH - 1:0]; // temp to hold the vector being
    created
wire [DEPTH*OUT_WIDTH - 1:0] outTmp;

integer i;

  `UNPACK_ARRAY(IN_WIDTH, DEPTH, tmpIn, dataIn, U_BLK_0, idx_0)
 `PACK_ARRAY(OUT_WIDTH, DEPTH, extTmp, outTmp, U_BLK_1, idx_1)


 delay #(
    .BIT_WIDTH(OUT_WIDTH),
    .DEPTH(DEPTH),
    .DELAY(DELAY)
 ) U_IP(
    .clk(clk),
    .rst(rst),
    .en_n(en_n),
    .dataIn(outTmp),
    .dataOut(dataOut)
 );

 always @(tmpIn, isSigned) begin
    for(i=0; i<DEPTH; i=i+1) begin
        if(isSigned == 1'b1) begin
            extTmp[i][OUT_WIDTH - 1:IN_WIDTH] <= {(OUT_WIDTH -
                IN_WIDTH){tmpIn[i][IN_WIDTH - 1]}};
            extTmp[i][IN_WIDTH - 1:0] <= tmpIn[i];
        end else begin
            extTmp[i][OUT_WIDTH - 1:IN_WIDTH] <= {(OUT_WIDTH -
                IN_WIDTH){1'b0}};
            extTmp[i][IN_WIDTH - 1:0] <= tmpIn[i];
        end
    end
 end

 //assign dataOut = outTmp; // assign the output to the newly created vector
endmodule
```

This module expands the width of input data while maintaining its signedness, making it suitable for various data processing tasks in digital systems.

0.5.11 sub r0

```
module sub_r0 #(
    parameter DATA_WIDTH = 32
)(
    input [DATA_WIDTH - 1:0] input1,
    input [DATA_WIDTH - 1:0] input2,
    output[DATA_WIDTH - 1:0] dataOut,
    output C,
    output Z,
    output V,
    output S
);
```

```
reg [DATA_WIDTH:0] tmpSub;
reg Ctmp, Ztmp, Vtmp, Stmp;

 always @(input1, input2) begin
    Ctmp = 0;
    Ztmp = 0;
    Vtmp = 0;
    Stmp = 0;

    tmpSub = input1 - input2;

    Ctmp = tmpSub[DATA_WIDTH];

    if(tmpSub[DATA_WIDTH-1:0] == {(DATA_WIDTH){1'b0}}) begin
        Ztmp = 1;
    end

    if((input1[DATA_WIDTH - 1] != input2[DATA_WIDTH - 1]) &&
       (tmpSub[DATA_WIDTH - 1] == input2[DATA_WIDTH - 1])) begin
        Vtmp = 1;
    end

    Stmp = tmpSub[DATA_WIDTH - 1];
 end

 assign dataOut = tmpSub[DATA_WIDTH - 1:0];
 assign C = Ctmp;
 assign Z = Ztmp;
 assign V = Vtmp;
 assign S = Stmp;

endmodule
```

This module subtracts two input values and provides informative status flags, making it useful for arithmetic operations in digital systems.

0.6 Synthesis and FPGA Implementation

0.7 Conclusion

In conclusion, our MIPS Processor implementation represents a significant achievement in the realm of processor design and showcases the successful integration of key architectural features. By combining a streamlined pipeline architecture, efficient hazard handling techniques, and careful RTL design, we

have created a high-performance MIPS Processor capable of executing instructions with speed and accuracy.

Throughout the development process, we focused on optimizing performance, minimizing hazards, and ensuring the reliability of our design. The pipeline architecture allowed for parallel instruction execution, maximizing throughput and harnessing the full potential of the MIPS instruction set. Our hazard handling techniques, including forwarding and hazard detection, greatly reduced pipeline stalls and improved overall performance.

We thoroughly tested and verified our processor design using rigorous methodologies, including comprehensive testbenches and simulations. This validation process ensured that the processor operates correctly under various scenarios and adheres to the MIPS architecture specifications.

Performance evaluations demonstrated the superiority of our MIPS Processor design compared to reference implementations. The efficient handling of hazards, the streamlined pipeline architecture, and the careful RTL design collectively contributed to significant performance improvements, enabling faster and more efficient execution of instructions.

Looking ahead, there are several avenues for further enhancements and optimizations. Future iterations of our MIPS Processor could explore additional hazard handling techniques, such as branch prediction, to reduce the impact of control hazards further. Additionally, incorporating more advanced features, such as cache memory or out-of-order execution, could lead to even higher performance gains.

Overall, our MIPS Processor presents a robust and efficient solution for computing tasks that require the MIPS instruction set. Whether it is for educational purposes, research endeavors, or practical applications, our processor's design and performance make it a valuable asset in the field of computer architecture.

We are excited to share our MIPS Processor implementation and hope that it inspires further advancements in the domain of processor design and contributes to the broader computing community.

References

[1] [Online image] Figure 1: MIPS basic pipeline. Available at: `https://www.researchgate.net/figure/Figure-2-MIPS-basic-pipeline-25_fig2_342107319`.