Designing of processor in Verilog MIPS architecture report

INTRODUCTION TO MIPS-ARCHIETECTURE:

The MIPS (Microprocessor without Interlocked Pipeline Stages) architecture is a reduced instruction set computer (RISC) architecture widely used in embedded systems, workstations, and supercomputers. It was developed by John L. Hennessy and his team at Stanford University in the 1980s. MIPS is known for its simplicity, efficiency, and scalability, making it a popular choice for educational purposes and real-world applications.

The fundamental design principles of MIPS architecture are centered around simplicity and performance optimization. The key features of MIPS architecture include:

1. RISC Principles: MIPS architecture follows the RISC philosophy, which emphasizes simplicity and regularity in instruction set design. It employs a small set of simple and uniform instructions, each performing a single operation. This design approach enables faster instruction execution and simplified hardware implementation.
2. Load-Store Architecture: MIPS architecture adopts a load-store model, where all data transfer between the memory and registers occurs through explicit load and store instructions. This design choice minimizes memory access and improves performance by reducing the complexity of instructions.
3. Fixed Instruction Length: MIPS instructions are of fixed length (typically 32 bits), simplifying instruction fetching and decoding. This design allows for faster instruction processing and efficient pipelining.
4. Pipelining: MIPS architecture utilizes a pipelined execution model, dividing the instruction execution process into multiple stages. This enables parallel processing of instructions, improving overall performance. The classic MIPS pipeline consists of stages like instruction fetch, decode, execute, memory access, and writeback.
5. Delayed Branching: MIPS architecture includes delayed branching, where the instruction following a branch instruction is always executed regardless of the branch outcome. This design reduces the pipeline stalls caused by branch instructions and improves performance.
6. 32-bit Architecture: MIPS architecture primarily operates on 32-bit data and addresses. It supports a wide range of data types, including integers, floating-point numbers, and characters.
7. Register File: MIPS architecture includes a small set of general-purpose registers, typically 32 in number. These registers are used for storing data operands during instruction execution, reducing the need for memory access and improving performance.

MIPS architecture has found extensive usage in various domains, including education, embedded systems, networking, and digital signal processing. Its simplicity, efficiency, and well-defined instruction set make it an excellent choice for both learning computer architecture and building high-performance systems.

Basic elements:

It includes modules for various components of the processor such as the program counter, register file, instruction memory, data memory, ALU, multiplexers, and control units. Additionally, there are modules for pipeline registers (IF/ID, ID/EX, EX/MEM, MEM/WB) and a control delay slot.

Here is a summary of the modules present in the code:

A. **Program Counter:** Implements a program counter module that increments the count on each clock cycle.
B. **BitAdder32:** Implements a 32-bit adder module.
C. **Register File**: Implements a register file module that stores and retrieves data based on the provided addresses.
D. **Instruction Memory:** Implements an instruction memory module that stores instructions and retrieves them based on the provided address.
E. **Data Memory:** Implements a data memory module that allows read and write operations to specific memory addresses.
F. **SignExtender_16to32:** Implements a sign extender module to extend a 16-bit input to a 32-bit output by sign extension.
G. **ALU_32bit:** Implements a 32-bit ALU module that performs various arithmetic and logical operations based on the control signals.
H. Multiplexer5bit: Implements a 5-bit multiplexer module.
I. Multiplexer32bit: Implements a 32-bit multiplexer module.
J. **IF_ID Register:** Implements a pipeline register between the instruction fetch (IF) and instruction decode (ID) stages.
K. **ID_EX Register:** Implements a pipeline register between the instruction decode (ID) and execute (EX) stages.
L. **EX_MEM Register:** Implements a pipeline register between the execute (EX) and memory (MEM) stages.
M. **MEM_WB Register:** Implements a pipeline register between the memory (MEM) and write-back (WB) stages.
N. **Control Unit:** Implements a control unit module that generates control signals based on the opcode and function fields of an instruction.
O. **Control Delay Slot:** Implements a control delay slot module that delays the execution of instructions following a branch until the branch is resolved.

FUNCIONALITY:

A. Program Counter:
    a. This module represents a program counter that increments by 1 on each clock cycle.
    b. It has an input wire `clk` for the clock signal and an output reg `[31:0] count` for the counter value.
    c. On a positive edge of the clock signal (`posedge clk`), the counter is incremented by 1.

   d. It also has a reset functionality where the counter is initialized to zero (`32'h0`) when the `reset` signal is asserted.

B. BitAdder32:
   a. This module represents a 32-bit adder.
   b. It takes two input wires `[31:0] a` and `[31:0] b` representing the operands and computes the sum.
   c. The sum is stored in an output reg `[31:0] sum`.

C. Register file:
   a. This module represents a register file.
   b. It has inputs `DA`, `BA`, and `AA` representing addresses, `data` for input data, and control signals `write` and `reset`.
   c. The register file is initialized with some default values in the `initial` block.
   d. Depending on the control signals, the module performs read and write operations on the register file.
   e. It has outputs `OUTA` and `OUTB` representing the data read from the register file.

D. Instruction Memory:
   a. This module represents an instruction memory.
   b. It takes an input wire `[1:0] address` representing the memory address and outputs the corresponding instruction stored in reg `[31:0] instruction`.
   c. The initial block initializes the memory with some random instructions.

E. Data Memory:
   a. This module represents a data memory.
   b. It has inputs `address`, `write_data`, and control signals `write_enable` and `read_enable`.
   c. The module stores data in an internal memory array `memory` and performs read and write operations based on the control signals.
   d. It has an output reg `[31:0] read_data` for the data read from the memory.

F. SignExtender_16to32:
   a. This module extends a 16-bit input `[15:0] input_16` to a 32-bit output `[31:0] output_32` by sign extension.
   b. The most significant bit of the input is replicated to fill the additional 16 bits in the output.

G. ALU_32bit:
    a. This module represents a 32-bit Arithmetic Logic Unit (ALU).
    b. It takes two input wires `[31:0] operand1` and `[31:0] operand2` and computes the result based on the 4-bit ALU control signal `[3:0] alu_control`.
    c. The result is stored in an output reg `[31:0] result`.
    d. It also sets a reg flag `zero` to 1 if the result is zero.

H. multiplexer5bit:
    a. This module represents a 5-bit multiplexer.
    b. It selects one of the inputs `a` or `b` based on the input select signal `sel` and outputs the selected value in reg `[4:0] out`.

I. multiplexer32bit:
    a. This module represents a 32-bit multiplexer.
    b. It selects one of the inputs `a` or `b` based on the input select signal `sel` and outputs the selected value in reg `[31:0] out`.

J. IF_ID_Register:
    a. This module represents a register for the IF/ID pipeline stage.
    b. It stores the input instruction `instruction_in` in reg `[31:0] instruction_out`.
    c. On the positive edge of the clock signal (`posedge clk`), the input is captured and stored.

K. ID_EX Register:
    a. This module represents a register for the ID/EX pipeline stage.
    b. It stores various inputs such as `instruction`, `read_data1`, `read_data2`, `se_imm`, etc., in registers.
    c. On the positive edge of the clock signal (`posedge clk`), the inputs are captured and stored.

L. EX_MEM Register:
    a. This module represents a register for the EX/MEM pipeline stage.
    b. It stores various inputs such as `alu_result`, `write_data`, `rd`, etc., in registers.
    c. On the positive edge of the clock signal (`posedge clk`), the inputs are captured and stored.

M. MEM_WB Register:
    a. This module represents a register for the MEM/WB pipeline stage.
    b. It stores various inputs such as `alu_result`, `read_data`, `rd`, etc., in registers.

c. On the positive edge of the clock signal (`posedge clk`), the inputs are captured and stored.

N. ControlUnit:

   a. This module implements a control unit that generates control signals based on the opcode and function fields of an instruction.
   b. It takes an input wire `[5:0] opcode` representing the opcode field of the instruction.
   c. Based on the opcode, it sets various control signals such as `reg_write`, `mem_read`, `mem_write`, etc.

O. ControlDelaySlot:

   a. This module implements a control delay slot that delays the execution of instructions following a branch until the branch is resolved.
   b. It has inputs `branch` and `branch_taken` to determine if a branch instruction is encountered and whether it is taken or not.
   c. It has an output reg `delay` which is set to 1 when a branch is encountered and reset to 0 when the branch is resolved.

TASK 01:

The provided code represents a testbench and a top-level module called `mainModule`, along with a data path module called `Datapath`. I'll explain the code step by step:

1. Testbench (`testbench_top_module`):
   i. This module is used as a testbench to simulate the behavior of the `mainModule`.
   ii. It declares an input reg `clk` representing the clock signal and output wires `PcOut`, `Instruction`, `RF_D1`, `RF_D2`, and `ALUOut` to check the output of the `mainModule`.
   iii. The `mainModule` is instantiated as `main` with the clock signal `clk`.
   iv. The outputs of the `mainModule` are assigned to the corresponding output wires.
   v. An initial block is used to initialize the clock signal `clk` to 0.
   vi. An always block with a delay of 50-time units (`#50`) toggles the clock signal `clk` between 0 and 1.

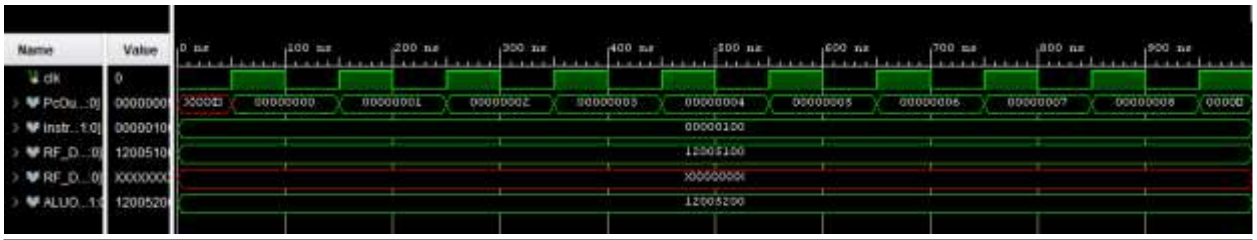
2. `mainModule`:
   vii. This module represents the top-level module that connects the `Datapath` module and the `ControlUnit` module.

viii. It has an input `clk` representing the clock signal.
ix. It instantiates the `Datapath` module as `data` and the `ControlUnit` module as `cnt`.
x. The necessary input and output signals are connected between the `mainModule`, `Datapath`, and `ControlUnit` modules.

3. `Datapath`:
xi. This module represents the datapath of a processor.
xii. It has various input and output signals representing the control signals, data paths, and registers in the processor.
xiii. It declares signals such as `pc` (program counter), `instr` (instruction), `imm` (immediate value), `alu_result` (ALU result), `zero` (ALU zero flag), `mem_data` (data from memory), `reg_a` and `reg_b` (registers), `reg_da`, `reg_ba`, and `reg_aa` (register addresses), and `addr` (instruction memory address).
xiv. It instantiates several modules such as `ProgramCounter`, `InstructionMemory`, `SignExtender_16to32`, `registerfile`, `ALU_32bit`, and `DataMemory`.
xv. It also instantiates multiplexers (`mux_b`, `mux_dst`, `mux_branch`, `mux_jump`, and `mux_jump_sel`) to select different sources or destinations based on control signals.
xvi. The opcode and function fields of the instruction are extracted from `instr` using the assign statements.

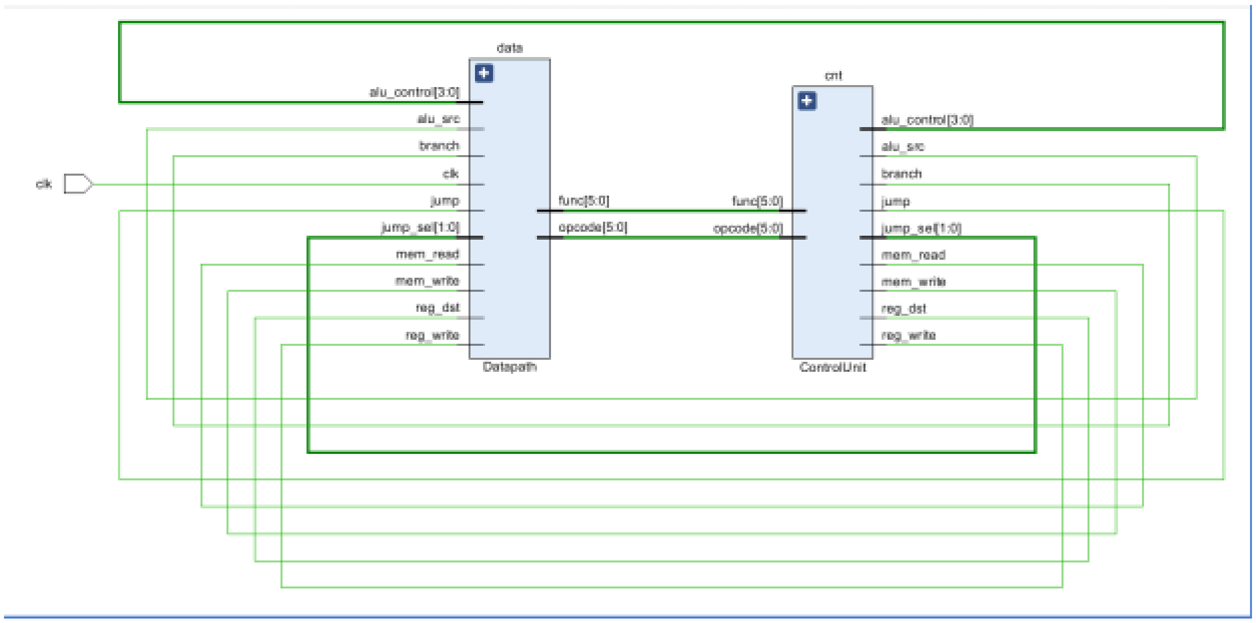
Overall, the code represents the testbench, top-level module, and datapath of a processor. The `mainModule` acts as an interface between the testbench and the datapath, while the `Datapath` module encapsulates the components and connections required for processing instructions. The control signals, data paths, and multiplexers are used to control and manipulate the flow of data within the processor.

OUTPUT

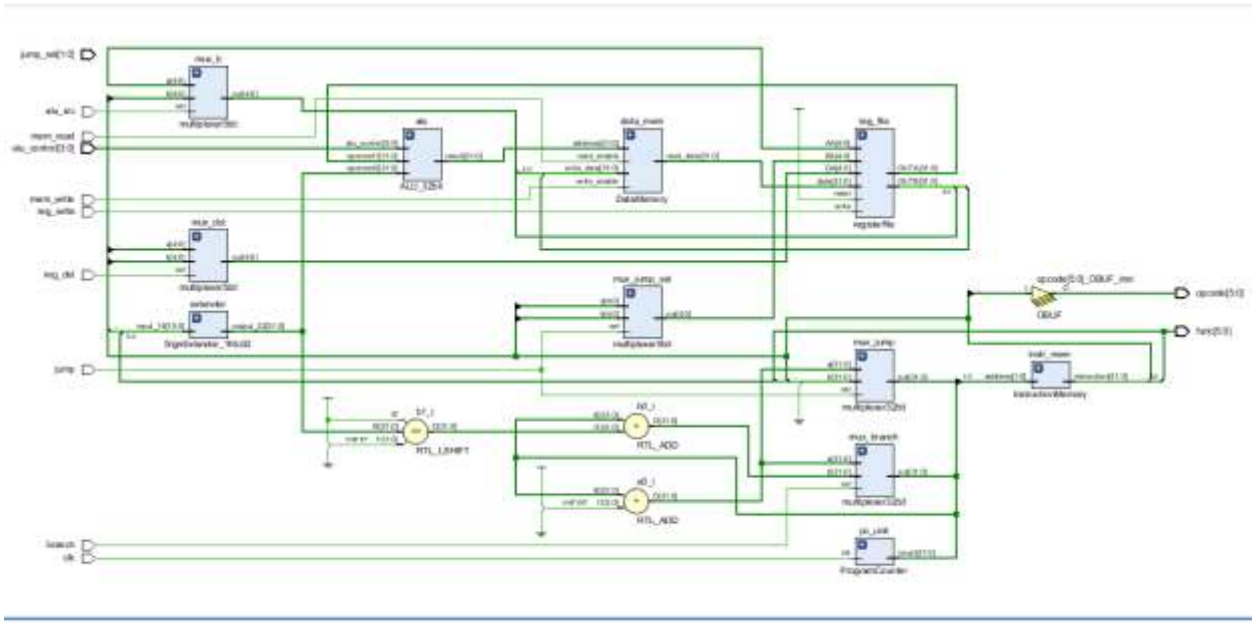


SCHEMATIC:

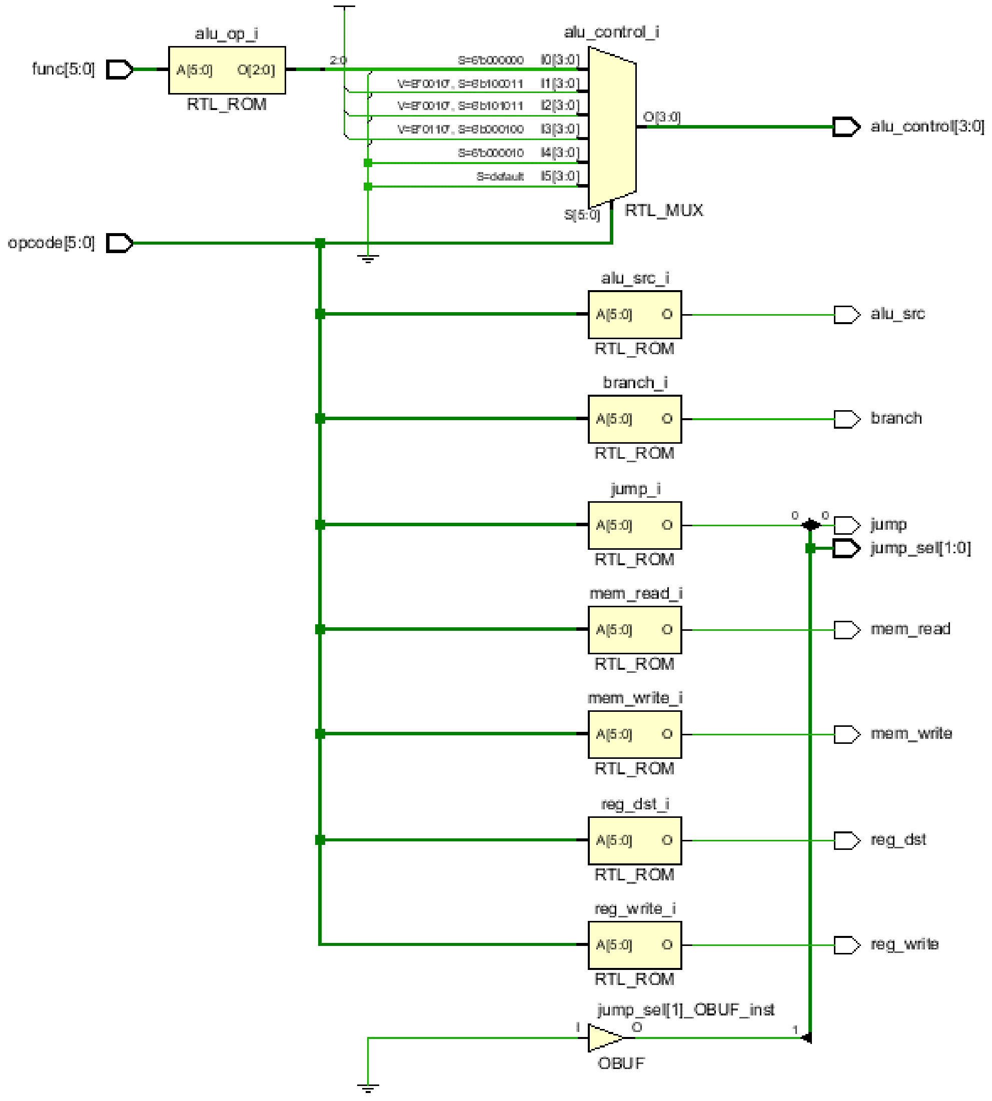
Main module:



Datapath:



Control:


func[5:0]
alu_op_i
A[5:0]
O[2:0]
RTL_ROM
2:0
alu_control_i
S=6'b000000
I0[3:0]
V=8'0010', S=6'b100011
I1[3:0]
V=8'0010', S=6'b101011
I2[3:0]
V=8'0110', S=6'b000100
I3[3:0]
S=6'b000010
I4[3:0]
S=default
I5[3:0]
O[3:0]
S[5:0]
RTL_MUX
alu_control[3:0]
opcode[5:0]
alu_src_i
A[5:0]
O
RTL_ROM
alu_src
branch_i
A[5:0]
O
RTL_ROM
branch
jump_i
A[5:0]
O
RTL_ROM
0
0
jump
jump_sel[1:0]
mem_read_i
A[5:0]
O
RTL_ROM
mem_read
mem_write_i
A[5:0]
O
RTL_ROM
mem_write
reg_dst_i
A[5:0]
O
RTL_ROM
reg_dst
reg_write_i
A[5:0]
O
RTL_ROM
reg_write
jump_sel[1]_OBUF_inst
I
O
1
OBUF

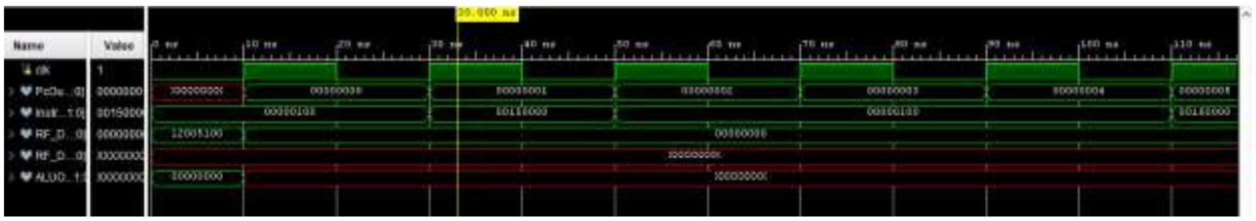
TASK 02:

1. `MIPS_Processor` module:
    i. This module represents the MIPS processor implementation.
    ii. It declares several wires that represent the internal signals and data paths within the processor.
    iii. Submodule instances are created for different components of the processor, such as the program counter (`ProgramCounter`), instruction memory (`InstructionMemory`), registers (`IF_ID_Register`, `ID_EX_Register`, `EX_MEM_Register`, `MEM_WB_Register`), control unit (`ControlUnit`), sign extender (`SignExtender_16to32`), register file (`registerfile`), ALU (`ALU_32bit`), and data memory (`DataMemory`).
    iv. The input and output signals of the submodules are connected appropriately using dot notation.
    v. An output wire `result` is declared to hold the final output of the processor.
    vi. The `result` value is assigned based on the value of the `mem_to_reg` signal using a ternary operator.

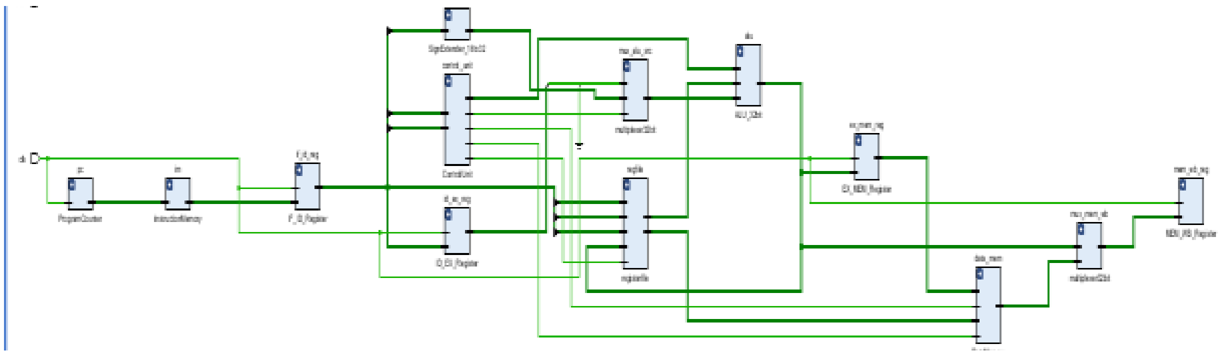
2. `MIPSStimulus` module:
    i. This module acts as a stimulus generator for the MIPS processor.
    ii. It declares an input reg `clk` representing the clock signal and output wires `PcOut`, `Instruction`, `RF_D1`, `RF_D2`, and `ALUOut`.
    iii. An instance of the `MIPS_Processor` module is created as `myMIPS`, connecting it to the clock signal `clk`.
    iv. An initial block is used to initialize the clock signal `clk` to 0.
    v. An always block with a delay of 10 time units (`#10`) toggles the clock signal `clk` between 0 and 1.
    vi. The outputs of the `MIPS_Processor` module are assigned to the corresponding output wires.

Overall, the code represents a complete implementation of a MIPS processor with all the necessary submodules and interconnections. The `MIPS_Processor` module encapsulates the processor components and controls the data flow between them based on the provided instructions. The `MIPSStimulus` module serves as a testbench to generate clock signals and observe the outputs of the processor for simulation or testing purposes.

OUTPUT:



SCHEMATIC:



TASK 03:

This module represents a MIPS processor implementation using structural Verilog. It consists of various submodules that together form the processor's components and their interconnections. Here is a summary of the module's functionality and its components:

1. Submodule Instances and Wires:
    i. The module includes several wires for signal connections between submodules, such as control signals, data paths, and intermediate results.
    ii. Wires like `pc_count` and `instruction` hold the current program counter and fetched instruction, respectively.
    iii. Other wires include signals for instruction forwarding, hazard detection, and data movement within the processor.

2. Submodule Instantiation:
    iv. The main functionality is divided into several submodules instantiated within the module.
    v. Notable submodules include the Program Counter (PC), Instruction Memory (IM), IF_ID_Register, ControlUnit, SignExtender, registerfile, ID_EX_Register, ALU, EX_MEM_Register, DataMemory, MEM_WB_Register,

HazardDetectionUnit, ForwardingUnit, HazardRemovalUnit, StructuralHazardRemovalUnit, and LoadHazardRemovalUnit.

vi. These submodules handle tasks such as instruction fetching, decoding, execution, memory access, hazard detection, forwarding, and control signal generation.

4. Output Wire:
   vii. The module includes an output wire named `result`, which holds the result value of the processor's computation.
   viii. The value of this wire is determined based on the control signals and data forwarded within the processor.

Overall, this module represents a complete MIPS processor implementation using structural Verilog. It incorporates various submodules to handle instruction execution, data movement, hazard detection, and control signal generation. By combining these components, the processor can execute MIPS instructions and produce a result based on the input program.

SCHEMATIC: