1. Modules Description:
2. Top Module - Central Processing Unit (CPU):

The Top Module represents the core processing unit (CPU) and integrates various components responsible for instruction execution. It acts as the conductor, orchestrating the flow of data and control signals between these components.

Functionality:

Fetches instructions from memory.

Decodes instructions.

Executes arithmetic and logic operations using the ALU (Arithmetic Logic Unit).

Manages control signals based on the instruction type (opcode) through the control unit.

Inputs:

clk: Clock signal - synchronizes the operation of the CPU.

rst: Reset signal - initializes the CPU to a known state.

Modules Used:

N bitReg.v: N-bit register module - stores temporary data during instruction execution.

regFile.v: Register file module - stores general-purpose registers for data manipulation.

immGen.v: Immediate generator module - generates immediate values for certain instructions.

NbitMUX.v: N-bit multiplexer module - allows selection between different data sources.

NbitALU.v: N-bit ALU module - performs arithmetic and logic operations on data.

ctrl unit.v: Control unit module - generates control signals based on the instruction type.

nBitShifter.v: N-bit shifter module - shifts data by a specified number of bits.

alu ctrl.v: ALU control module - generates specific control signals for the ALU based on the instruction.

Branch Unit.v: Branch unit module - handles conditional branching instructions.

Memory.v: Memory module - stores both program instructions and data.

Hazard\_detection\_unit.v: Hazard detection module - identifies potential hazards during instruction execution.

Forwarding\_unit.v: Forwarding unit module - optimizes instruction execution by forwarding results from previous stages.

DecompUnit.v: Decompressing unit module – decompresses instructions if in compressed form.

1. Memory Module:

The memory module, a vital component of the CPU responsible for storing both program instructions and data.

Functionality:

Stores program instructions fetched from external memory.

Stores data operands used by the ALU (Arithmetic Logic Unit) during instruction execution.

Provides data to the CPU based on read requests and writes data to memory based on write requests.

Inputs:

sclk: Clock signal - synchronizes the operation of the memory module.

MemRead: Control signal indicating a read operation from memory.

MemWrite: Control signal indicating a write operation to memory.

signed\_inst: Signal indicating if the instruction being loaded is signed or unsigned (relevant for load instructions).

AU\_inst\_sel: Control signal from the ALU control unit specifying the size and type of data to be loaded (LW - Load Word, LH - Load Halfword, LHU - Load Halfword Unsigned, LB - Load Byte, LBU - Load Byte Unsigned).

addr: Address of the memory location to be read from or written to.

data\_in: Data to be written to memory during a write operation.

Outputs:

data\_out: Data read from memory during a read operation.

Memory Capacity and Organization:

The memory module has a capacity of 4 KB (4096 bytes).

Each memory location stores a single byte (8 bits).

A 32-bit word is retrieved by accessing four consecutive memory locations.

Loading Logic:

During a read operation (MemRead is high), the data\_out is determined based on the AU\_inst\_sel signal and the signed\_inst signal.

For LW (Load Word), all four bytes of the memory location specified by addr are concatenated to form the 32-bit word.

For LH (Load Halfword) and LHU (Load Halfword Unsigned), only the lower two bytes (16 bits) are retrieved. For LH, the sign is extended from the 4th bit of the second byte, while for LHU, the value is zero-extended (no sign extension).

For LB (Load Byte) and LBU (Load Byte Unsigned), only the byte at the specified address is retrieved. For LB, the sign is extended from the 7th bit of the byte, while for LBU, the value is zero-extended.

Storing Logic:

During a write operation (MemWrite is high on the falling edge of the clock (~sclk)), the data to be written (data\_in) is written to the memory location specified by addr.

For SW (Store Word), all four bytes of data\_in are written to four consecutive memory locations.

For SH (Store Halfword), only the lower 16 bits of data\_in are written to the two lower bytes of the memory location.

For SB (Store Byte), only the lower 8 bits of data\_in are written to the specified memory location.

1. DecompUnit Module

The DecompUnit module, a component responsible for decompressing instructions in a compressed instruction set architecture (ISA).

Functionality:

Takes a 32-bit instruction as input (inst).

Analyzes the instruction format to identify potential compression opportunities.

If the instruction can be decompressed, generates a decompressed version (instOut).

Sets a compression flag (compress) to indicate if decompression occurred.

Passes through the original instruction unchanged if decompression is not applicable.

Inputs:

reset: Reset signal - initializes the module.

clk: Clock signal - synchronizes the operation of the module.

inst: 32-bit instruction to be decompressed.

Outputs:

instOut: 32-bit decompressed instruction (if applicable).

compress: Compression flag indicating if decompression was performed (1'b1 for compressed, 1'b0 for uncompressed).

Operation:

Reset: When reset is high, the instOut is set to zero, and the compress flag is cleared.

Falling Clock Edge: On the falling edge of the clock (~clk), the module bypasses any decompression and assigns the original instruction directly to instOut. This behavior might be related to a pipeline stall or hazard detection.

Instruction Decompression: If the reset is not active and the clock edge is positive, the module analyzes the instruction format based on specific bit fields:

Opcode (bits 1:0): Different opcode values trigger distinct decompression logic.

Other Control Bits (variable positions): Additional control bits within the instruction further refine the decompression process for specific instruction types.

Decompression Logic: Based on the identified opcode and control bits, the module applies specific bit manipulation operations to the original instruction to generate the decompressed version. These operations involve shifting, concatenation, and bit field replication to reconstruct the full instruction from its compressed format.

Compression Flag: If decompression is performed, the compress flag is set to 1'b1 to indicate a compressed instruction. Otherwise, the flag remains cleared (1'b0).

Unrecognized Instructions: If the opcode and control bits do not match any known decompression pattern, the instOut might be set to zero or retain its original value, depending on the specific implementation.

1. Hazard Detection Unit Module

The Hazard\_Detection\_Unit module, a critical component within a CPU pipeline responsible for identifying potential hazards during instruction execution.

Functionality:

Analyzes data dependencies between instructions in a pipelined processor.

Detects data hazards that could lead to incorrect results.

Signals the pipeline to stall the fetch (IF) stage if a hazard is detected.

Inputs:

IF\_ID\_RegisterRs1: Register number of the first operand for the instruction currently being fetched (IF stage).

IF\_ID\_RegisterRs2: Register number of the second operand for the instruction currently being fetched (IF stage).

ID\_EX\_RegisterRd: Register number where the result of the instruction currently being decoded (ID stage) will be written.

ID\_EX\_MemRead: Control signal indicating if the instruction currently being decoded (ID stage) is a load instruction that reads from memory.

Output:

stall: Control signal indicating a stall condition for the fetch (IF) stage (1'b1 for stall, 1'b0 for no stall).

Operation:

The module continuously monitors the register numbers used by instructions in the pipeline.

It compares the register numbers of the operands being fetched (in the IF stage) with the register number where the result of the instruction being decoded (in the ID stage) will be written (ID\_EX\_RegisterRd).

It also checks if the instruction being decoded is a load instruction (ID\_EX\_MemRead) that reads from memory.

Hazard Detection Logic:

A stall condition (stall is set to 1'b1) is triggered if:

Read After Write (RAW) Hazard:

Either IF\_ID\_RegisterRs1 or IF\_ID\_RegisterRs2 (operands being fetched) is equal to ID\_EX\_RegisterRd (destination register of the instruction being decoded).

This hazard occurs specifically when:

The instruction being decoded is a load instruction (ID\_EX\_MemRead is high).

The destination register of the load instruction (ID\_EX\_RegisterRd) is the same as the source register of an instruction being fetched.

Stalling the fetch stage ensures the load instruction finishes reading from memory before the fetched instruction attempts to use its value.

If no hazard is detected, the stall signal remains cleared (1'b0).

Benefits of Hazard Detection:

Prevents data corruption by ensuring instructions use valid data.

Maintains the correctness of the pipeline execution.

1. Forwarding Unit Module

The Forwarding\_Unit module, a vital component within a pipelined processor responsible for data forwarding to resolve data hazards.

Functionality:

Analyzes data dependencies between instructions in a pipelined CPU.

Identifies potential RAW (Read After Write) hazards.

Selects the appropriate data source for the ALU operands based on the hazard detection.

Provides forwarding control signals to the ALU for efficient data access.

Inputs:

ID\_EX\_RegisterRs1: Register number of the first operand for the instruction currently being decoded (ID stage).

ID\_EX\_RegisterRs2: Register number of the second operand for the instruction currently being decoded (ID stage).

MEM\_WB\_RegWrite: Control signal indicating if the instruction currently in the Write Back (WB) stage is writing a value to a register.

EX\_MEM\_RegWrite: Control signal indicating if the instruction currently in the Memory (MEM) stage is writing a value to a register.

MEM\_WB\_RegisterRd: Register number of the destination register for the instruction currently in the WB stage.

EX\_MEM\_RegisterRd: Register number of the destination register for the instruction currently in the MEM stage.

Outputs:

forwardA\_ALU: Control signal for forwarding data to the first ALU operand (2'b10 for forwarding from EX stage, 2'b01 for forwarding from WB stage, 2'b00 for no forwarding).

forwardB\_ALU: Control signal for forwarding data to the second ALU operand (same format as forwardA\_ALU).

Operation:

The module continuously monitors the register numbers used by instructions throughout the pipeline stages.

It compares the register numbers of the operands being decoded (in the ID stage) with the destination register of instructions currently in the EX (Execute) and WB (Write Back) stages.

It also checks the RegWrite control signals for both stages to identify instructions that are writing to registers.

Forwarding Logic:

The module identifies two potential RAW hazard scenarios and determines the appropriate forwarding path for each ALU operand (forwardA\_ALU and forwardB\_ALU).

Forwarding from EX stage (2'b10):

If an instruction in the EX stage writes to a register (EX\_MEM\_RegWrite is high) and its destination register (EX\_MEM\_RegisterRd) matches the first operand register of the instruction being decoded (ID\_EX\_RegisterRs1), a RAW hazard is detected.

In this case, forwardA\_ALU is set to 2'b10, indicating that the data should be forwarded directly from the output of the EX stage ALU to the first ALU operand of the decoded instruction.

Forwarding from WB stage (2'b01):

If an instruction in the WB stage writes to a register (MEM\_WB\_RegWrite is high) and its destination register (MEM\_WB\_RegisterRd) matches either operand register of the instruction being decoded (ID\_EX\_RegisterRs1 or ID\_EX\_RegisterRs2), a potential RAW hazard exists.

However, forwarding from the WB stage only occurs if there's no earlier hazard in the EX stage (checked by the additional condition in the else-if block).

If forwarding from WB is necessary, the corresponding control signal (forwardA\_ALU or forwardB\_ALU) is set to 2'b01 to indicate data forwarding from the WB stage register.

No forwarding (2'b00):

If no RAW hazard is detected (i.e., none of the conditions above are met), both forwarding control signals (forwardA\_ALU and forwardB\_ALU) are set to 2'b00, indicating that the operands should be fetched from the register file as usual.

Benefits of Forwarding:

Eliminates pipeline stalls caused by RAW hazards.

Improves overall pipeline performance by avoiding unnecessary stalls and waiting for data from the register file.

1. Branch Unit Module

The Branch\_Unit module, a component responsible for controlling branching behavior within a processor.

Functionality:

Analyzes the instruction type and branch condition flags.

Determines if a branch instruction should be taken.

Generates control signals to indicate the type of branch (conditional or unconditional) and the target address calculation method.

Inputs:

inst[2:0]: Lower three bits of the instruction (likely representing the branch opcode).

Branch: Control signal indicating if the current instruction is a branch instruction.

jump: Control signal indicating if the current instruction is an unconditional jump instruction.

cf: Carry Flag from the ALU.

zf: Zero Flag from the ALU.

vf: Overflow Flag from the ALU.

sf: Sign Flag from the ALU.

Output:

flag: Two-bit control signal:

Bit 0: Branch Taken Indication (1'b1 for branch taken, 1'b0 for branch not taken).

Bit 1: Branch/Jump Type (1'b0 for conditional branch, 1'b1 for jump).

Operation:

The module checks the Branch signal to determine if the current instruction is a branch instruction.

If it's a branch instruction (Branch is high):

It analyzes the lower three bits of the instruction (inst[2:0]) to identify the specific branch condition.

Based on the branch condition and the ALU flags (cf, zf, vf, and sf), it determines if the branch should be taken.

It utilizes a case statement to handle different branch conditions:

BR\_BEQ: Branch on Equal (branch if zf is set and Branch is high).

BR\_BNE: Branch on Not Equal (branch if zf is cleared and Branch is high).

BR\_BLT: Branch on Less Than (branch if sf is not equal to vf and Branch is high).

BR\_BGE: Branch on Greater Than or Equal (branch if sf is equal to vf and Branch is high).

BR\_BLTU: Branch on Less Than Unsigned (branch if cf is cleared and Branch is high).

BR\_BGEU: Branch on Greater Than or Equal Unsigned (branch if cf is set and Branch is high).

For each condition, the appropriate bit in the flag output is set to indicate a branch taken (1'b1) or not taken (1'b0) based on the condition evaluation.

If the branch condition is not recognized (default case), the flag is set to neutral (2'b0).

If the instruction is not a branch (Branch is low):

The flag output is set based on the jump signal:

If jump is high, it's an unconditional jump (branch taken), so flag is set to 2'b10 (branch taken, jump).

If jump is low, it's neither a branch nor a jump, so flag is set to 2'b00 (branch not taken, no jump).

Benefits:

Controls the flow of execution within a program based on branch instructions and jump instructions.

Enables conditional execution based on specific ALU results (e.g., branching based on zero or negative values).

1. Control Unit Module

This module It takes a 32-bit instruction as input (inst) and generates various control signals for the datapath. Here's a breakdown of the code:

Inputs:

rst: Reset signal (active low)

inst: The 32-bit instruction to be decoded

Outputs:

Branch: Control signal for branch instructions (active high)

jump: Control signal for jump instructions (active high)

MemRead: Control signal for reading from memory (active high)

MemtoReg: Control signal to determine whether data from memory or ALU output goes to the register file (active high for memory data)

MemWrite: Control signal for writing to memory (active high)

ALUSrc: Control signal to determine the second operand for the ALU (active high for immediate value from instruction)

RegWrite: Control signal for writing to the register file (active high)

signed\_inst: Control signal to indicate if the instruction is signed (active high)

PC\_en: Control signal to enable updating the program counter (PC)

RF\_MUX\_sel: Selects which data to write back to the register file (0 for ALU output, 1 for data loaded from memory with sign extension, 2 for data loaded from memory without sign extension)

AU\_inst\_sel: Selects the type of load/store operation (0 for word, 1 for halfword, 2 for byte)

Functionality:

The code first checks for a reset (rst). If asserted, it sets all control signals to zero.

Otherwise, it checks if the instruction is zero (no operation). If so, it sets all control signals to zero.

The main logic is inside a case statement that checks the opcode of the instruction (bits 6:2 of inst). Different opcodes have different control signal settings based on the function code (bits 14:12 of inst).

OPCODE\_Arith\_R\_M: Handles arithmetic instructions for register and immediate values (e.g., ADD, SUB).

Based on the function code, it sets control signals for ALU operation, register write, signed operation, etc.

OPCODE\_Load: Handles load instructions from memory (e.g., LW, LH, LB).

Based on the function code, it sets control signals for memory read, register write, data size (word, halfword, byte), and sign extension.

OPCODE\_Store: Handles store instructions to memory (e.g., SW, SH, SB).

Based on the function code, it sets control signals for memory write, data size, and sign extension.

OPCODE\_Arith\_I: Handles arithmetic instructions with immediate values (e.g., ADDI, SLTI).

Similar to OPCODE\_Arith\_R\_M but uses the immediate value as the second operand.

OPCODE\_Branch: Handles branch instructions (e.g., BEQ, BNE).

Based on the function code, it sets control signals for branch operation and signed comparison.

Default: If none of the above opcodes match, it sets all control signals to zero.

1. alu\_ctrl Module

The alu\_ctrl module, determines the appropriate operation for the Arithmetic Logic Unit (ALU) based on the instruction type and function code.

Inputs:

ALUOp: A 3-bit input representing the ALU operation code.

inst: A 32-bit input representing the entire instruction being decoded.

Output:

ALU\_selection: A 5-bit output register that holds the encoded ALU operation.

Functionality:

The alu\_ctrl module uses a case statement based on the ALUOp to determine the ALU operation for different instruction categories:

I and S format instructions (ALUOp = 000):

Decodes the instruction based on bits 14:12 (function code).

Sets ALU\_selection based on the function code:

000: ALU\_ADD (addition)

001: ALU\_SLL (shift left logical) if bits 5:4 are 01, otherwise ALU\_ADD.

010: ALU\_SLT (set on less than) or ALU\_ADD based on bits 5:4.

Similar logic for other function codes (ALU\_SLTU, ALU\_XOR, ALU\_OR, ALU\_AND).

Default case sets ALU\_selection to zero (invalid operation).

Branch and jal statements (ALUOp = 001):

Sets ALU\_selection to ALU\_SUB (subtraction) for branch instructions.

R-format functions (ALUOp = 010):

Similar to I/S format, decodes based on function code (bits 14:12).

Sets ALU\_selection for various operations like addition, subtraction, shift, compare, etc.

Default case sets ALU\_selection to zero.

LUI and AUIPC instructions (ALUOp = 011):

Sets ALU\_selection to ALU\_PASS for load upper immediate and add upper immediate with PC instructions, indicating no ALU operation needed.

M-type instructions (Division and Multiplication) (ALUOp = 100):

Decodes based on function code (bits 14:12).

Sets ALU\_selection for specific multiply, divide, or remainder operations based on dedicated ALU functions (ALU\_MUL, ALU\_DIV, etc.).

Default case sets ALU\_selection to zero.

Default:

If ALUOp doesn't match any category, sets ALU\_selection to zero (invalid operation).

1. regFile Module

The regFile module, a register file, which stores general-purpose temporary data during program execution.

Parameters:

N: Defines the number of registers in the file (default 5, can be customized).

Inputs:

rst: Reset signal (active high).

clk: Clock signal for synchronous operation.

RegWrite: Control signal to enable writing to the register file.

data: 32-bit data to be written to the register file.

read1: Address (register number) of the first register to read (N-1 bits).

read2: Address (register number) of the second register to read (N-1 bits).

write: Address (register number) of the register to write (N-1 bits).

Outputs:

r1: 32-bit data read from the first register (register read1).

r2: 32-bit data read from the second register (register read2).

Functionality:

The regFile module uses two main mechanisms:

Register Array:

Declares an internal register array x with 32 elements (enough to hold 32-bit data for all possible registers).

Assigns the output ports r1 and r2 to directly connect to the corresponding elements in the array based on the read1 and read2 addresses. This allows simultaneous read access to two registers.

Write Logic:

Uses a positive edge triggered clock (posedge clk) and a negative edge reset (negedge clk) to handle writes and initialization.

If rst is high (active reset), the always block initializes all registers in the x array to zero.

During normal operation (clock pulses):

If RegWrite is high (write enabled), the module writes data to the register specified by the write address.

There's a special case where writing to register zero (write == 5'd0) always writes zero, likely to avoid unintended behavior with the zero register.

Otherwise, no write operation occurs.

1. Remarks:

We have faced challenges when it comes to the forwarding unit as it is not fully functioning correctly.