IPA REPORT

Adithi Samudrala – 2022102065
Zainab Raza – 2022102013

SEQUENTIAL

FETCH:

Inputs and Outputs: The module has inputs such as `PC` (Program Counter) and `clk` (clock), and outputs like `icode`, `ifun`, `rA`, `rB`, `val_P`, `val_C`, `halt`, `instr_valid`, `imem_error`, and `IDR`.

- **Instruction Memory Initialization**: The initial block loads instruction memory contents from an external file "1.txt" into the `imem` array.
- **Instruction Fetch and Decoding**:
    - The module continuously reads instructions from memory based on the current value of the program counter (`PC`).
    - It decodes the fetched instruction into opcode (`icode`) and function code (`ifun`).
- **Handling of Different Instruction Types**:
    - The module handles various instruction types such as halt, no operation, data movement, arithmetic/logical operations, control flow, and function calls based on the opcode (`icode`).
    - For each instruction type, it updates the program counter (`val_P`) to point to the next instruction.
- **Error Detection**:
    - It detects errors such as invalid memory access (`imem_error`) or illegal register access (`IDR`).
    - In case of an error, it sets flags accordingly and potentially halts the processor.
- **Instruction Validity**:
    - If the fetched instruction does not match any defined opcode, it sets an instruction valid flag (`instr_valid`).

Overall, the "fetch" module serves as the entry point for fetching and decoding instructions in a processor, ensuring proper instruction execution and error handling.

\DECODE:

The declaration `reg [63:0] register[0:14];` creates an array of registers named `register`, where each register is 64 bits wide. The array contains 15 elements, indexed from 0 to 14. This construct essentially creates 15 separate 64-bit registers, allowing for the storage of 64-bit values for each register.

Similarly, `reg [63:0] d_rvalB, d_rvalA;` declares two 64-bit registers named `d_rvalB` and `d_rvalA`.

These registers are likely used to store data within a Verilog module, possibly for maintaining state, buffering input or output data, or storing temporary values during computation. In the context of a processor design, such registers could be used to store the values of data or addresses being processed at various stages of the pipeline. They may also hold intermediate results during arithmetic or logical operations.

EXECUTE:

`full_adder`:

- Implements a full adder circuit with inputs A, B, and Cin, and outputs Sum and Carry.

2. **`adder`**:
   - Uses instances of the `full_adder` module to perform addition on two 64-bit inputs A and B.
   - Generates a carry chain for the addition operation.
   - Outputs the sum and a carry flag (`add_cc`).
3. **`subtractor`**:
   - Similar to the `adder` module but performs subtraction by inverting one of the inputs (A) and adding it to the other input (B).
   - Outputs the difference and a carry flag (`sub_cc`).
4. **`xorgate`**, **`andgate`**:
   - Implement XOR and AND gates for each bit of the input.
   - Output the result and condition code flags.
5. **`ALU_A`**, **`ALU_B`**:
   - Modules to select ALU inputs based on the instruction code (`icode`) and input values (`valA`, `valB`).
   - ALU_A selects the appropriate value for operand A.
   - ALU_B selects the appropriate value for operand B.
6. **`ALU`**:
   - Combines ALU_A and ALU_B to perform arithmetic and logic operations based on `icode` and `ifun`.
   - Uses `add_out`, `sub_out`, `and_out`, `xor_out` from adder, subtractor, and logic gate modules to produce `valE`, the ALU output value.
   - Sets condition code flags (`CC`) based on the operation performed.
7. **`cond`**:
   - Implements conditional logic for conditional move and jump instructions based on `icode` and `ifun`.
   - Sets the `cnd` flag based on condition code flags (`CC`), sign flag (`sf`), zero flag (`zf`), and overflow flag (`of`).

Overall Functionality:

- The `full_adder`, `adder`, and `subtractor` modules implement basic arithmetic operations for addition and subtraction.
- The `xorgate` and `andgate` modules implement logic operations.
- `ALU_A` and `ALU_B` select appropriate operands based on instruction code (`icode`).
- `ALU` module performs arithmetic and logic operations based on `icode` and `ifun`, producing `valE` as output.
- `cond` module determines whether to execute a conditional move or jump instruction based on condition code flags (`CC`), sign flag (`sf`), zero flag (`zf`), and overflow flag (`of`).

MEMORY:

`clk`: Clock signal.

- `val_A`, `val_E`, `val_P`: 64-bit signed values representing different memory addresses or data values.
- `icode`: 4-bit instruction code.

Outputs:

- `val_M`: 64-bit signed value representing the data read from memory.
- `dmem_er`: Data memory error flag.

Internal Variables:

- `mem`: Array of 64-bit registers representing the memory.

Read Operation:

- If the instruction code is `0101` (indicating a memory read operation), the module checks if the memory address (`val_E`) is within the valid memory range (0 to 1023). If it's within the range, the corresponding data is read from memory and stored in `val_M`. Otherwise, `dmem_er` is set to `1` to indicate a data memory error.

Write Operations:

- On the positive edge of the clock (`posedge clk`), the module performs write operations for different instruction codes:
    - If the instruction code is `0100` (indicating a memory write operation), the module checks if the memory address (`val_E`) is within the valid memory range. If so, it writes the value of `val_A` to the memory address.
    - If the instruction code is `1000` (indicating a call operation), the module writes the value of `val_P` to the memory address `val_E` if it's within the valid memory range.
    - If the instruction code is `1010` (indicating a push operation), the module writes the value of `val_A` to the memory address `val_E` if it's within the valid memory range.

Remarks:

- The module uses sequential logic to handle write operations on the positive edge of the clock.
- It uses combinatorial logic to handle read operations.
- The module ensures that memory accesses are within the valid memory range (0 to 1023) to prevent memory access violations.
- The `dmem_er` flag is set to `1` to indicate a data memory error if the memory address is out of range during a read or write operation.

Overall, the `memory` module simulates the behavior of a simple data memory unit in a processor, supporting read and write operations with error detection for out-of-range memory accesses.

PC UPDATE:

Inputs:

- `clk`: Clock signal.
- `icode`: 4-bit instruction code.
- `cnd`: Condition flag indicating whether a conditional jump should be taken.
- `valC`, `valM`, `valP`: 64-bit signed values representing different program counter update values.

Outputs:

- `pc_update`: 64-bit value representing the updated program counter.

Operation:

- The module uses combinational logic to determine the appropriate value for updating the program counter based on the current instruction (`icode`), condition flag (`cnd`), and other input values.
- It evaluates the `icode` to determine the type of instruction being executed.
- If the instruction is a `call` (icode = 1000), the module sets the `pc_update` to the value of `valC`, which typically represents the target address of the call instruction.
- If the instruction is a `ret` (icode = 1001), the module sets the `pc_update` to the value of `valM`, which typically represents the return address stored in memory.
- If the instruction is a conditional jump (icode = 0111), the module checks the condition flag (`cnd`). If `cnd` is true (`1`), indicating that the condition for the jump is met, it sets `pc_update` to `valC`, representing the target address of the jump. Otherwise, it sets `pc_update` to `valP`, representing the next sequential instruction address.
- For all other instructions, the module sets `pc_update` to `valP`, indicating that the program counter should be updated to the next sequential instruction address.

Remarks:

- The module operates entirely using combinational logic, meaning it updates the `pc_update` output as soon as the inputs change.
- It allows for conditional branching based on the condition flag (`cnd`), allowing for control flow changes in the program execution.
- The output `pc_update` reflects the updated program counter value based on the executed instruction and the current state of the processor.

The Verilog module `sequential` integrates various modules responsible for executing instructions sequentially in a processor. Here's an explanation of its functionality:

FINAL COMBINED CODE:

Inputs and Outputs:

- **Inputs:**
  - `clk`: Clock signal.
  - `icode`, `ifun`, `rA`, `rB`: Instruction code, function, and register IDs.
  - `val_P`, `val_C`, `val_A`, `val_B`, `val_E`, `val_M`: Values corresponding to the program counter (`val_P`), constant (`val_C`), register A (`val_A`), register B (`val_B`), effective address (`val_E`), and memory (`val_M`).
  - `halt`: Signal indicating whether the processor should halt.
- **Outputs:**
  - `instr_valid`, `imem_error`, `dmem_er`: Signals indicating whether the instruction is valid, there's an instruction memory error, or a data memory error.
  - `Cnd`: Condition flag indicating whether a conditional jump should be taken.
  - `rax`, `rcx`, `rdx`, `rbx`, `rsp`, `rbp`, `rsi`, `rdi`, `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14`: Register values.
  - `alu_A`, `alu_B`: Values for the ALU operations.
  - `CC`: Condition code.
  - `idr_temp`: Signal indicating an invalid register.
  - `AOK`, `INS`, `HLT`, `IDR`: Signals indicating the processor's state.
  - `pc_update`: Updated program counter value.

Operation:

- The `sequential` module initializes various signals and variables.
- It instantiates the following modules:
  - `fetch`: Fetches instructions from memory.
  - `decode`: Decodes instructions and retrieves register values.
  - `ALU_A`, `ALU_B`: Compute values for ALU operations based on instructions.
  - `ALU`: Performs arithmetic and logical operations.
  - `cond`: Determines whether a conditional jump should be taken based on condition flags.
  - `memory`: Handles memory operations.

    - `pc_update`: Updates the program counter based on instruction execution.
- It monitors the state of the processor and updates flags accordingly (e.g., halt, invalid instruction, etc.).
- It finishes the simulation if the processor reaches certain states (e.g., halt, invalid instruction, etc.).
- It updates the clock signal for simulation purposes.
- It updates the program counter based on the `pc_update` signal.
- The `initial` block initializes simulation parameters and dumps variables for waveform analysis.
- The `always @(posedge clk)` block updates the program counter on each clock cycle.

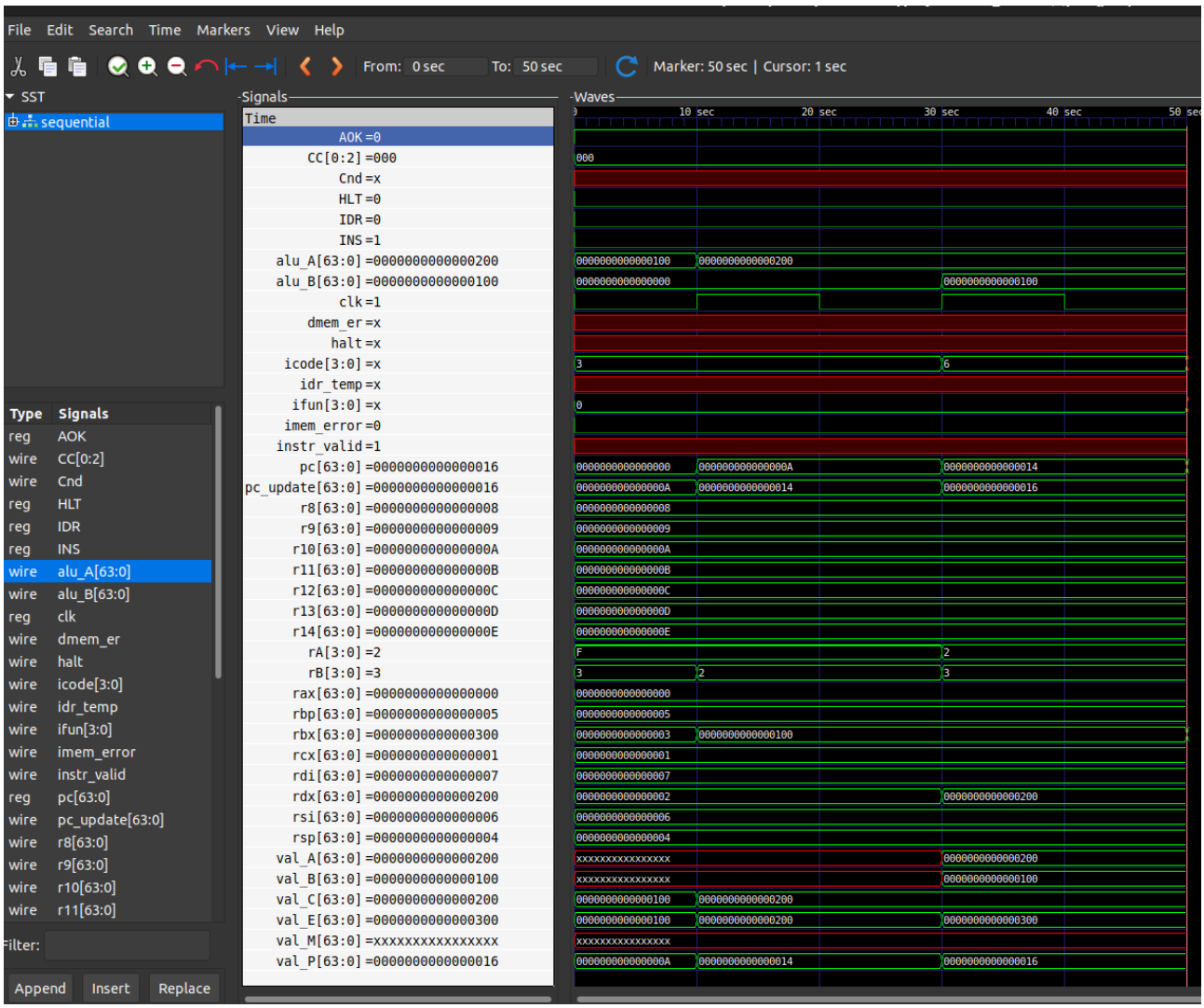
Remarks:

- This module integrates various components of a processor to execute instructions sequentially.
- It handles instruction fetching, decoding, execution, memory operations, and program counter updates.
- It monitors the processor's state and reacts accordingly, including halting the simulation if necessary.
- The use of `always @(posedge clk)` ensures that the program counter updates on each clock cycle, facilitating sequential instruction execution.

GTK WAVE PLOT:

Testcase:

Sample testcase given



PIPELINE:

`select_PC` Module:

- **Inputs and Outputs**: This module takes several inputs including `clk`, instruction-related signals (`F_predPC`, `W_valM`, `M_valA`, `M_icode`, `W_icode`), and `M_Cnd`. It produces an output `pc_new` representing the selected program counter value.
- **Functionality**:
    - The module selects a new program counter value (`pc_new`) based on certain conditions:
        - If the writeback stage instruction opcode (`W_icode`) is equal to 9 (indicating a return instruction), then `pc_new` is set to the value of memory writeback stage's memory value (`W_valM`).
        - If the memory stage instruction opcode (`M_icode`) is equal to 7 (indicating a conditional jump instruction), it checks the condition `M_Cnd`. If `M_Cnd` is false, `pc_new` is set to the memory stage's memory value (`M_valA`); otherwise, it's set to the predicted program counter value (`F_predPC`).

- For all other cases, `pc_new` is set to the predicted program counter value (`F_predPC`).

`fetch` Module:

- **Outputs**: This module produces various outputs including instruction details (`f_icode`, `f_ifun`, `f_rA`, `f_rB`, `f_val_P`, `f_val_C`, `f_predPC`), control signals (`halt`, `instr_valid`, `imem_error`, `IDR`).
- **Functionality**:
  - The module reads instructions from memory (`imem`) based on the current program counter (`f_PC`).
  - It decodes fetched instructions and determines their opcode (`f_icode`), function code (`f_ifun`), register operands (`f_rA`, `f_rB`), immediate values (`f_val_C`), and predicted program counter (`f_predPC`).
  - It handles various instruction types such as halt, no operation, conditional move, immediate move, register-memory move, memory-register move, arithmetic operation, jump, call, return, push, and pop.
  - The `imem_error` flag is set if the program counter exceeds the memory size.
  - The `halt` flag is set if a halt instruction is encountered.
  - The `instr_valid` flag indicates the validity of the fetched instruction.
  - The `IDR` flag is set if an invalid register is encountered.
- **Initialization**:
  - The `imem` is initialized with instructions from a memory file (`7.txt`).
- **Combination Logic**:
  - The module selects the predicted program counter value based on the instruction opcode (`f_icode`). If the opcode corresponds to a conditional jump or a call instruction, it selects the immediate value (`f_val_C`) as the predicted program counter; otherwise, it selects the incremented program counter (`f_val_P`).

FETCH REG:

Inputs:

- `clk`: Clock signal for synchronous operation.
- `F_stall`: Signal indicating whether the fetch stage is stalled.
- `f_predPC`: Predicted program counter value.
- `f_PC`: Current program counter value.
- `F_stat`: Status code indicating the state of the fetch stage.

- **Outputs**:
  - `F_predPC`: Output signal representing the predicted program counter.
  - `f_stat`: Output signal representing the status code of the fetch stage.
- **Functionality**:
  - The module operates on the rising edge of the clock (`posedge clk`).

- It updates the `f_stat` output based on the `F_stat` input.
- If the fetch stage is not stalled (`!F_stall`), the predicted program counter (`F_predPC`) is updated with the value of `f_predPC`.
- If the fetch stage is stalled, the predicted program counter is held constant, maintaining the current program counter value (`f_PC`).

Summary:

The `fetch_reg` module synchronously updates the predicted program counter based on the stall condition and the current clock cycle. It ensures that the fetch stage operates correctly within the processor pipeline, allowing for the sequential fetching of instructions while handling stall conditions effectively.

DECODE:

Inputs and Outputs: The module takes various inputs such as `clk` (clock signal), `D_icode` (instruction opcode for the current cycle), `D_ifun` (instruction function code), `D_stat` (instruction status), `W_icode` (instruction opcode from the write-back stage), register specifiers (`D_rA`, `D_rB`), immediate values (`D_valC`, `D_valP`), along with values and destinations from different pipeline stages. It produces outputs representing register values (`rax`, `rcx`, etc.), decoded instruction details (`d_icode`, `d_ifun`, `d_stat`), and operand values (`d_valA`, `d_valB`, `d_valC`) and destinations (`d_dstE`, `d_dstM`, `d_srcA`, `d_srcB`).

- **Register Initialization**: The initial block initializes an array of registers (`register`) with specific values.
- **Instruction Decoding and Operand Handling**:
    - The module decodes the instruction based on its opcode and function code and determines the appropriate operand sources (`d_srcA`, `d_srcB`) and destinations (`d_dstE`, `d_dstM`).
    - It computes the values of operand A and operand B (`d_valA`, `d_valB`) based on the operand sources and pipeline stage values.
    - Additionally, it handles immediate values (`D_valC`, `D_valP`) for specific instruction types.
- **Register Updates**:
    - The module updates the register file synchronously on the positive edge of the clock based on the executed instruction from the write-back stage (`W_icode`) and its associated values (`W_valE`, `W_valM`).
- **Synchronous Register Updates**:
    - The module updates register values synchronously on the positive edge of the clock based on the executed instruction.

Overall, the "decode" module serves as a vital component in the processor's pipeline, responsible for decoding instructions, computing operand values, determining operand sources and destinations, and updating register values accordingly.

DECODE REG:

Inputs and Outputs: The module takes various inputs such as `clk` (clock signal), `D_bubble` (bubble signal), `D_stall` (stall signal), along with various signals (`f_rA`, `f_rB`, `f_valC`, `f_valP`, `f_stat`, `f_icode`, `f_ifun`) representing details from the fetch stage. It also takes inputs representing system status (`instr_valid`, `imem_error`, `halt`). The module produces outputs (`D_rA`, `D_rB`, `D_valC`, `D_valP`, `D_stat`, `D_icode`, `D_ifun`) representing decoded instruction details for the decode stage.

- **Instruction Decoding Logic**:
    - The module decodes the instruction based on the signals from the fetch stage when neither a stall nor a bubble condition is present (`D_stall` and `D_bubble` are both low).
    - If there's a stall condition (`D_stall` is high), the module maintains the current instruction details without updating them.
    - If there's a bubble condition (`D_bubble` is high), the module inserts a bubble in the pipeline, resetting the instruction details accordingly.
    - The `stat` variable is determined based on the system status signals and the fetched instruction (`f_icode`). It's used to determine the status of the current instruction.
- **Synchronous Updates**:
    - The updates to the instruction details (`D_rA`, `D_rB`, `D_valC`, `D_valP`, `D_stat`, `D_icode`, `D_ifun`) occur synchronously on the positive edge of the clock.
- **State Assignment**:
    - The `stat` variable is assigned based on various conditions such as instruction validity (`instr_valid`), instruction memory error (`imem_error`), and the opcode of the fetched instruction (`f_icode`). It represents the state of the current instruction.

Overall, the "decode_reg" module serves as a crucial component in the processor pipeline, responsible for decoding fetched instructions and determining the appropriate instruction details for the decode stage, while also handling stall and bubble conditions effectively.

MEMORY:

Module Explanation: `Addr`

This module is responsible for determining the memory address (`addr`) based on the current microinstruction code (`M_icode`) and associated values (`M_valE`, `M_valA`).

- **Inputs**:
    - `M_icode`: Microinstruction code indicating the current operation.
    - `M_valE`: Value from the execution stage.
    - `M_valA`: Value from the address calculation stage.
- **Outputs**:

    - `addr`: Output memory address.
- **Functionality**:
    - The module operates combinational logic to determine the memory address based on the microinstruction code.
    - If the microinstruction code corresponds to a return or a pop operation (`M_icode` equals `4'b1001` or `4'b1011`), the address is set to `M_valA`.
    - For other instructions (`M_icode` equals `4'b0100`, `4'b0101`, `4'b1001`, or `4'b1000`), the address is set to `M_valE`.

Module Explanation: `mem_read`

This module determines whether a memory read operation should be performed based on the microinstruction code (`M_icode`).

- **Inputs**:
    - `M_icode`: Microinstruction code indicating the current operation.
- **Outputs**:
    - `read`: Output signal indicating whether a memory read should be performed.
- **Functionality**:
    - The module uses combinational logic to determine whether a read operation is required based on the microinstruction code.
    - If the microinstruction code corresponds to memory read operations (`M_icode` equals `4'b0101`, `4'b1011`, or `4'b1001`), `read` is set to `1`.
    - For all other microinstruction codes, `read` is set to `0`.

Module Explanation: `mem_write`

This module determines whether a memory write operation should be performed based on the microinstruction code (`M_icode`).

- **Inputs**:
    - `M_icode`: Microinstruction code indicating the current operation.
    - `clk`: Clock signal for synchronous operation.
- **Outputs**:
    - `write`: Output signal indicating whether a memory write should be performed.
- **Functionality**:
    - The module uses sequential logic triggered by the rising edge of the clock (`clk`) to determine whether a write operation is required based on the microinstruction code.
    - If the microinstruction code corresponds to memory write operations (`M_icode` equals `4'b0100`, `4'b1010`, or `4'b1000`), `write` is set to `1`.
    - For all other microinstruction codes, `write` is set to `0`.

Module Explanation: `data_memory`

This module simulates the data memory and handles memory read and write operations.

- **Inputs**:
  - `addr`: Memory address for read/write operations.
  - `data`: Data to be written to memory.
  - `read`: Signal indicating a memory read operation.
  - `write`: Signal indicating a memory write operation.
- **Outputs**:
  - `m_valM`: Output value read from memory.
  - `dmem_error`: Signal indicating a memory access error.
- **Internal Variables**:
  - `mem`: Array representing the memory.
- **Functionality**:
  - The module updates the `dmem_error` signal if the memory address is out of range (less than 0 or greater than 1023).
  - If the address is valid and a read operation is requested, the module reads the data from memory (`mem`) and outputs it as `m_valM`.
  - If the address is valid and a write operation is requested, the module writes the input data (`data`) to the specified memory address.

Module Explanation: `stat`

This module handles the status code (`m_stat`) based on the memory access error (`dmem_error`) and the microinstruction status code (`M_stat`).

- **Inputs**:
  - `dmem_error`: Signal indicating a memory access error.
  - `M_stat`: Microinstruction status code.
- **Outputs**:
  - `m_stat`: Output status code.
- **Functionality**:
  - If a memory access error is detected (`dmem_error`), the module sets the status code (`m_stat`) to `4'b0100`.
  - Otherwise, it propagates the microinstruction status code (`M_stat`) to `m_stat`.

These modules together manage memory read and write operations, handle memory addresses, and ensure proper error handling in the data memory stage of the processor pipeline.

MEMORY REG:

The `memory_reg` module handles the transition of values from the Execute stage (`E`) to the Memory stage (`M`) in the processor pipeline. It updates various control signals and data values based on the clock signal and bubble condition.

- **Inputs**:
    - `clk`: Clock signal for synchronous operation.
    - `M_bubble`: Signal indicating whether the Memory stage is in a bubble state.
    - `E_stat`: Status code indicating the state of the Execute stage.
    - `E_icode`: Instruction code from the Execute stage.
    - `e_Cnd`: Condition flag from the Execute stage.
    - `e_dstE`: Destination register for Execute stage results.
    - `E_dstM`: Destination register for Execute stage memory results.
    - `E_valA`: Value of register A from the Execute stage.
    - `e_valE`: Value computed by the Execute stage.
- **Outputs**:
    - `M_stat`: Status code indicating the state of the Memory stage.
    - `M_icode`: Instruction code passed from the Execute stage to the Memory stage.
    - `M_Cnd`: Condition flag passed from the Execute stage to the Memory stage.
    - `M_dstE`: Destination register for Execute stage results passed to the Memory stage.
    - `M_dstM`: Destination register for Execute stage memory results passed to the Memory stage.
    - `M_valE`: Value computed by the Execute stage passed to the Memory stage.
    - `M_valA`: Value of register A from the Execute stage passed to the Memory stage.
- **Functionality**:
    - The module operates on the rising edge of the clock (`posedge clk`).
    - It updates the `M_stat` output with the value of `E_stat`.
    - It updates the `M_Cnd` output with the value of `e_Cnd`.
    - If `M_bubble` is asserted (indicating a bubble state), `M_icode` is set to indicate a no-operation instruction (`4'b0001`).
    - If not in a bubble state, the module passes the values from the Execute stage to the Memory stage: `M_valE`, `M_dstM`, `M_icode`, `M_dstE`, and `M_valA` are updated with the corresponding values from the Execute stage.

Summary:

The `memory_reg` module facilitates the transfer of control signals and data values from the Execute stage to the Memory stage in the processor pipeline. It ensures that the Memory stage receives the correct instruction code, data values, and control signals for proper operation, accounting for bubble conditions to maintain pipeline integrity.

W REG:

The `w_reg` module is responsible for capturing the results from the Memory stage (`M`) and storing them in registers for use in the Writeback stage (`W`) of the processor pipeline. It operates on the rising edge of the clock signal and takes into account the stall condition to control the register updates.

- **Inputs**:
    - `clk`: Clock signal for synchronous operation.
    - `W_stall`: Signal indicating whether the Writeback stage is stalled.
    - `m_stat`: Status code indicating the state of the Memory stage.
    - `m_icode`: Instruction code from the Memory stage.
    - `m_dstE`: Destination register for Memory stage results.
    - `m_dstM`: Destination register for Memory stage memory results.
    - `m_valE`: Value computed by the Memory stage.
    - `m_valM`: Value of memory data computed by the Memory stage.
- **Outputs**:
    - `m_Cnd`: Condition flag passed from the Memory stage to the Writeback stage.
    - `W_stat`: Status code indicating the state of the Writeback stage.
    - `W_icode`: Instruction code passed from the Memory stage to the Writeback stage.
    - `W_dstE`: Destination register for Memory stage results passed to the Writeback stage.
    - `W_dstM`: Destination register for Memory stage memory results passed to the Writeback stage.
    - `W_valE`: Value computed by the Memory stage passed to the Writeback stage.
    - `W_valM`: Value of memory data computed by the Memory stage passed to the Writeback stage.
- **Functionality**:
    - The module operates on the rising edge of the clock (`posedge clk`).
    - If the Writeback stage is not stalled (`W_stall != 1`), the module captures the results from the Memory stage and updates the corresponding registers in the Writeback stage.
    - The values captured include the status code (`W_stat`), instruction code (`W_icode`), computed values (`W_valE`, `W_valM`), and destination registers (`W_dstE`, `W_dstM`).
    - The `m_Cnd` output is not directly updated in this module, indicating that it might be computed or determined in a different stage of the pipeline.

Summary:

The `w_reg` module serves as a register bank for storing the results from the Memory stage to be used in the Writeback stage of the processor pipeline. It ensures proper synchronization of data transfer between pipeline stages and handles stall conditions to maintain pipeline integrity.

CONTROL LOGIC:

The `control_logic` module is responsible for managing the control signals in the pipeline stages of a processor. It controls the stall and bubble signals based on various conditions arising from the instructions being executed in different pipeline stages.

- **Inputs**:
  - `D_icode`: Instruction code from the Decode stage (`D`).
  - `M_icode`: Instruction code from the Memory stage (`M`).
  - `E_icode`: Instruction code from the Execute stage (`E`).
  - `m_stat`: Status code indicating the state of the Memory stage.
  - `W_stat`: Status code indicating the state of the Writeback stage.
  - `d_srcA`: Source register A for the Decode stage.
  - `d_srcB`: Source register B for the Decode stage.
  - `E_dstM`: Destination register for Memory stage results passed to the Execute stage.
  - `e_Cnd`: Condition flag passed from the Execute stage.
- **Outputs**:
  - `F_stall`: Stall signal for the Fetch stage (`F`).
  - `D_stall`: Stall signal for the Decode stage.
  - `W_stall`: Stall signal for the Writeback stage.
  - `D_bubble`: Bubble signal for the Decode stage.
  - `E_bubble`: Bubble signal for the Execute stage.
  - `M_bubble`: Bubble signal for the Memory stage.
- **Functionality**:
  - The module initializes all output signals to `0` during the `initial` block.
  - It evaluates the conditions based on the current instruction codes and status codes to determine whether to stall or introduce bubbles in the pipeline stages.
  - `F_stall` is set to `1` if certain conditions related to potential data hazards are met, which prevents the Fetch stage from fetching new instructions.
  - `D_stall` is set to `1` if the Decode stage is potentially stalled due to data hazards.
  - `W_stall` is set to `1` if the Writeback stage is potentially stalled due to certain status codes.
  - `D_bubble` is set to `1` if conditions for introducing a bubble in the Decode stage are met.
  - `E_bubble` is set to `1` if conditions for introducing a bubble in the Execute stage are met.
  - `M_bubble` is set to `1` if conditions for introducing a bubble in the Memory stage are met.

Summary:

The `control_logic` module plays a crucial role in managing the pipeline stages of the processor by controlling stall and bubble signals to handle data hazards and ensure proper

instruction execution flow. It evaluates various conditions based on instruction codes and status codes to make decisions regarding pipeline operation.

FINAL PROCESSOR:

The `processor` module integrates various components of a processor, including the fetch, decode, execute, memory, and writeback stages, along with control logic. It simulates the pipeline architecture of a processor, where each stage processes instructions in parallel.

- **Inputs**:
    - `clk`: Clock signal for synchronous operation.
    - `f_PC`: Program counter value from the Fetch stage.
    - `F_stat`: Status code indicating the state of the Fetch stage.
- **Outputs**:
    - Various wires connecting different stages of the pipeline, such as `D_bubble`, `D_stall`, `f_rA`, `f_rB`, etc.
- **Internal Components**:
    - **Fetch Stage**:
        - `fetch_reg`: Fetches instructions from memory and passes them to the Decode stage.
    - **Decode Stage**:
        - `decode_reg`: Decodes the fetched instruction and sends the decoded information to subsequent stages.
    - **Execute Stage**:
        - `execute_reg`: Executes instructions based on decoded information.
    - **Memory Stage**:
        - `memory_reg`: Performs memory-related operations like load/store instructions.
    - **Writeback Stage**:
        - `w_reg`: Writes back results to registers.
- **Control Logic**:
    - `control_logic`: Manages control signals to handle pipeline hazards, stalls, and bubbles.
- **Simulation Setup**:
    - Initializes the clock signal and other relevant variables.
    - Initializes the processor's components.
    - Implements clock toggling for synchronous operation.
    - Monitors the Writeback stage's status and finishes simulation upon specific conditions.

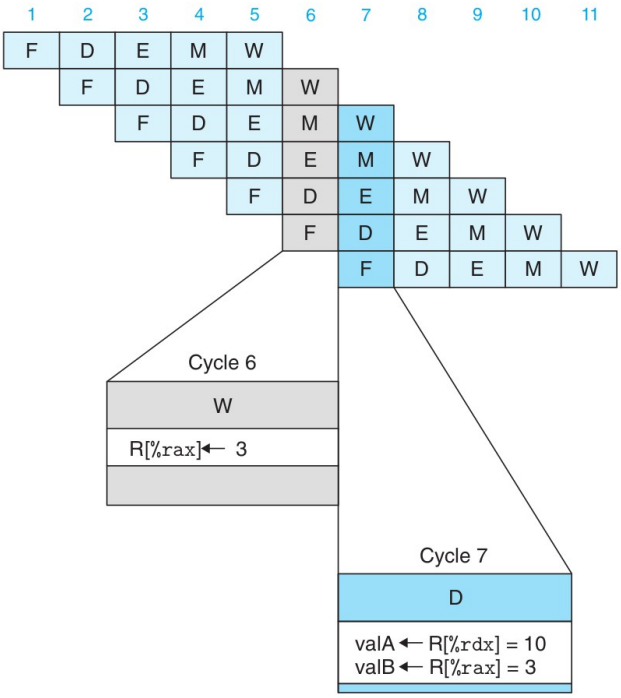
Summary:

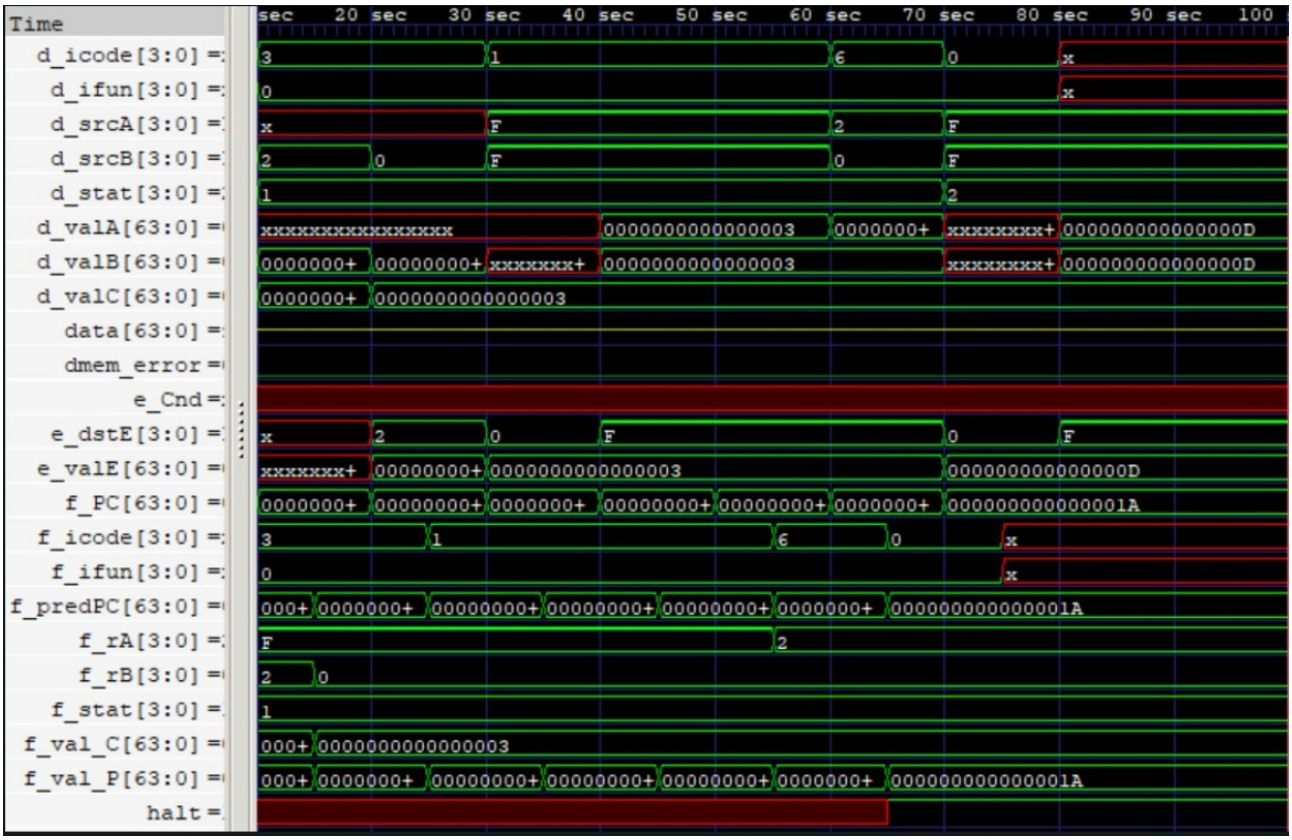
The `processor` module serves as the top-level entity that brings together the components of a processor, including its pipeline stages and control logic. It orchestrates the flow of instructions

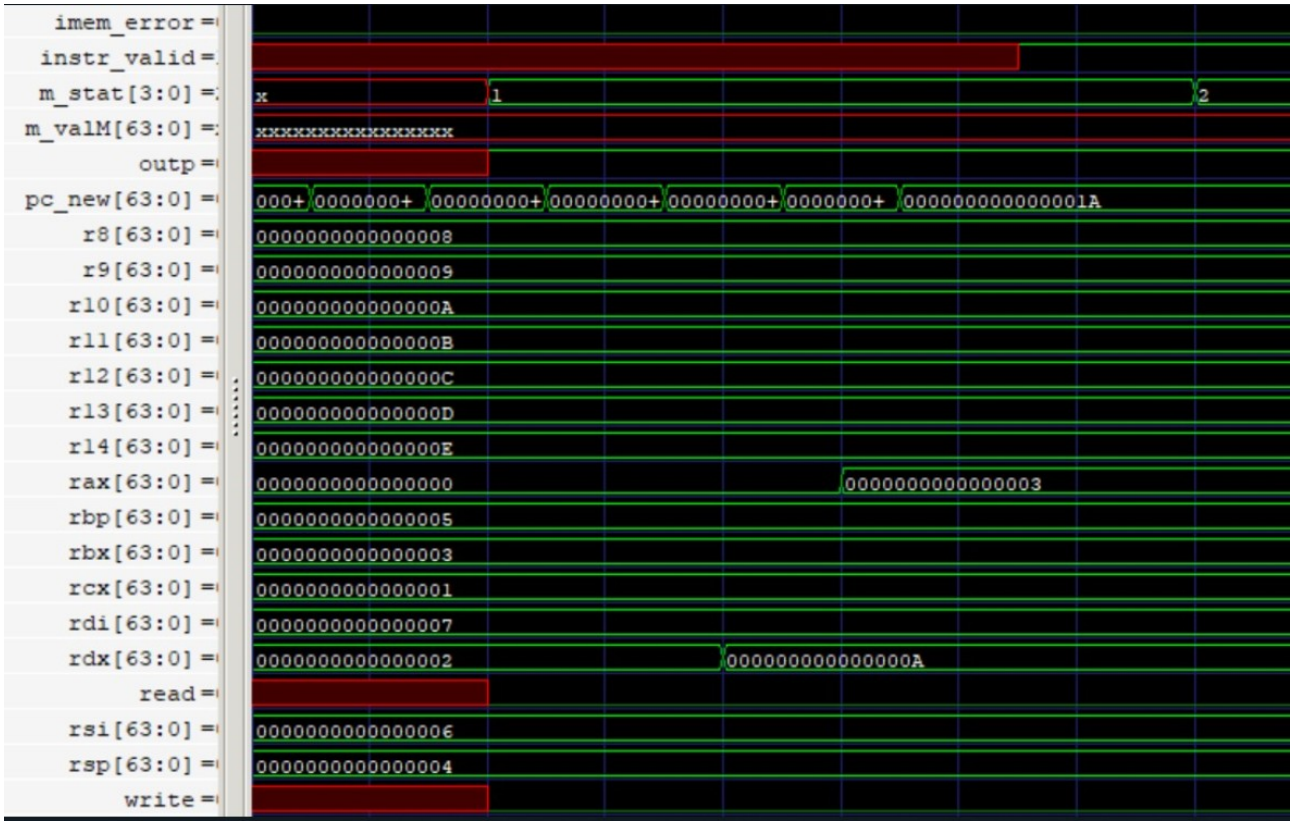
through the pipeline and manages control signals to ensure proper instruction execution. The simulation setup allows for testing and verification of the processor's functionality.

TESTCASE:

```
# progl
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: nop
0x017: addq %rdx,%rax
0x019: halt
```






imem_error
instr_valid
m_stat[3:0]
m_valM[63:0]
outp
pc_new[63:0]
r8[63:0]
r9[63:0]
r10[63:0]
r11[63:0]
r12[63:0]
r13[63:0]
r14[63:0]
rax[63:0]
rbp[63:0]
rbx[63:0]
rcx[63:0]
rdi[63:0]
rdx[63:0]
read
rsi[63:0]
rsp[63:0]
write
x
1
2
xxxxxxxxxxxxxxxx
000+
0000000+
00000000+
000000000+
00000000+
0000000+
000000000000001A
0000000000000008
0000000000000009
000000000000000A
000000000000000B
000000000000000C
000000000000000D
000000000000000E
0000000000000000
0000000000000003
0000000000000005
0000000000000003
0000000000000001
0000000000000007
0000000000000002
000000000000000A
0000000000000006
0000000000000004