Organization of Digital Computer Lab

EECS 112L

Lab 4: Pipeline Processor

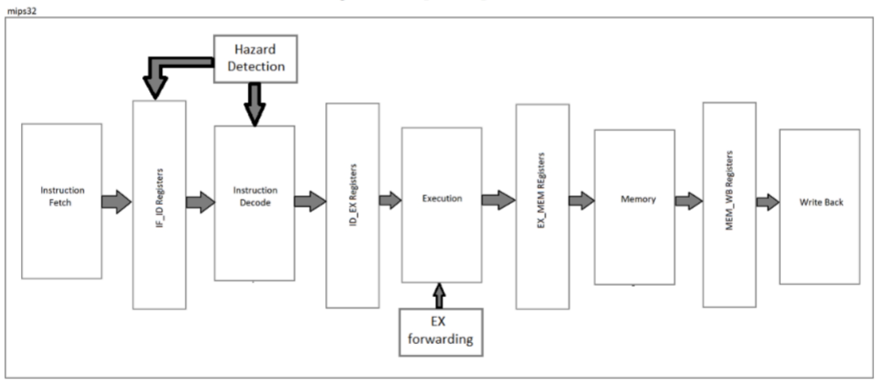
Name: Esther Anaya

Student ID: 82547392

Due: November 25, 2024

1 Objective

Figure 1: Pipeline processor



The objective of this lab is to design and implement a pipelined processor based on the MIPS architecture, as illustrated in **Figure 1: Pipeline Processor**. This design involves dividing the datapath into five distinct stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EXE), Memory Access (MEM), and Write Back (WB), with pipeline registers separating each stage. The pipeline processor improves instruction throughput by allowing multiple instructions to overlap in execution, ensuring that all stages of the processor remain busy during every clock cycle.

The processor must handle the three types of hazards inherent to pipelining:

- **Structural Hazards**: When hardware resources are insufficient to execute all instructions in the pipeline simultaneously.
- **Data Hazards**: When an instruction depends on the results of a previous instruction that has not yet completed.
- **Control Hazards**: When the flow of instruction addresses changes due to branch or jump instructions.

To resolve these hazards:

1. **Forwarding and Hazard Detection**: Implement hardware mechanisms to detect data dependencies and enable data forwarding to minimize stalls.
2. **Pipeline Stalling and Flushing**: Introduce stalls and flushes where necessary, particularly in cases where hazards cannot be resolved by forwarding alone.
3. **Instruction Flow Management**: Utilize control logic to ensure accurate instruction flow, especially during branch and jump operations.

Figure 1 outlines the structure of the pipeline processor, highlighting the flow of instructions through the pipeline stages, the role of pipeline registers in transferring control and data signals, and the mechanisms to address hazards.

The primary goal of this lab is to develop each stage of the processor, integrate the pipeline registers, and incorporate hazard resolution techniques. By testing the processor with a series of instructions and observing waveform simulations, the lab will verify:

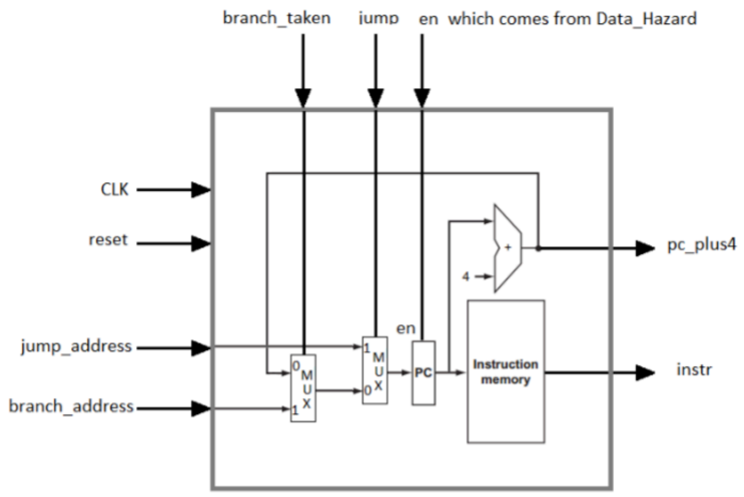
- Accurate instruction flow through the pipeline.
- Proper execution of arithmetic, logical, load, store, branch, and jump instructions.
- Effective handling of hazards to ensure the correctness and efficiency of the pipeline.

Through this lab, the student will gain practical experience in implementing a pipelined processor, understanding the challenges and solutions involved in pipelining, and demonstrating the advantages of pipelined architectures in modern computing systems.

2 Procedure

1. Instruction Fetch Stage (IF_pipe_stage)

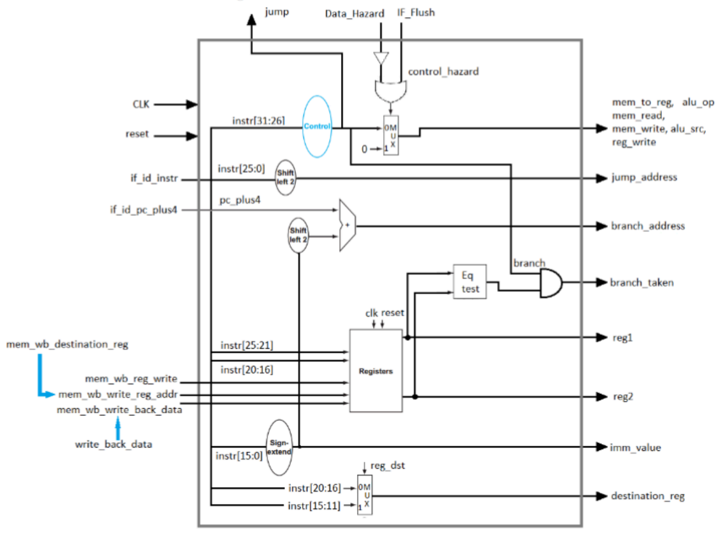
Figure 2: Instruction Fetch.



- The first stage fetches instructions from memory using the address in the Program Counter (PC).
- As shown in Figure 2, the PC is incremented by 4 (PC_PLUS4) for sequential instructions. For branch and jump instructions, the branch target (BRANCH_ADDRESS) or jump address (JUMP_ADDRESS) is selected using multiplexers, controlled by BRANCH_TAKEN and JUMP.
- The fetched instruction and PC_PLUS4 are stored in the IF/ID pipeline register (Figure 3) for use in the next stage.

2. Instruction Decode Stage (ID_pipe_stage)

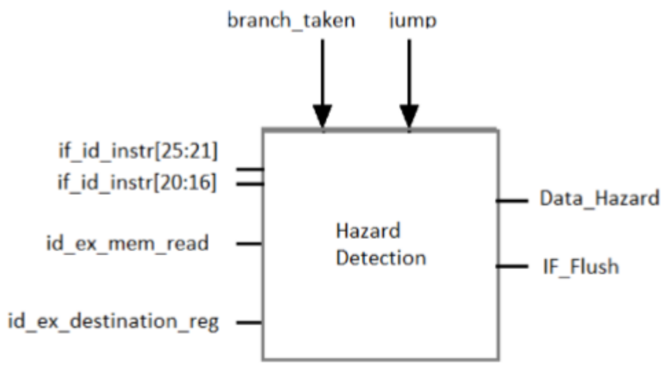
Figure 4: Instruction Decode.



- In this stage, the instruction fetched in the previous stage is decoded, and the required registers are read from the register file.
- Figure 4 shows the instruction decode stage, where:
    1. The control unit generates signals (e.g., MEM_READ, ALU_OP, REG_WRITE) based on the instruction opcode.
    2. The immediate value (IMM_VALUE) is sign-extended for later use.
    3. Branch addresses are calculated (BRANCH_ADDRESS), and a comparison determines whether the branch is taken (BRANCH_TAKEN).
- These outputs, along with control signals, are stored in the ID/EX pipeline register (Figure 5) for use in the execution stage.

3. Hazard Detection

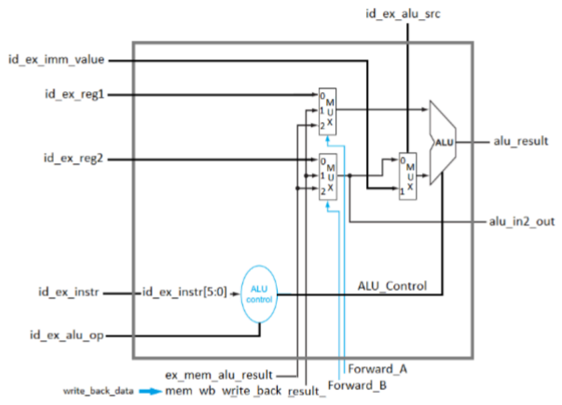
Figure 6: Hazard Detection.



- Hazards are handled using the Hazard Detection Unit (Figure 6).
- Data Hazards: If an instruction depends on the result of a previous instruction that hasn't completed, the pipeline stalls. Data_Hazard indicates a stall, and a NOP (no-operation) is inserted.
- Control Hazards: For branch and jump instructions, the pipeline flushes the instructions fetched after the branch or jump by setting IF/ID pipeline registers to zero (IF_Flush).

4. Execution Stage (EX_pipe_stage)

Figure 7: Execution.

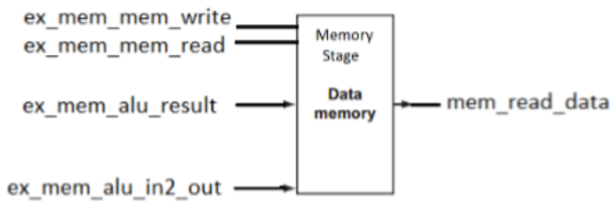


- In this stage, arithmetic and logical operations, as well as address calculations, are performed by the ALU, as shown in Figure 7.

- Forwarding logic (Figure 8) resolves data dependencies by selecting inputs for the ALU from:
    1. The current pipeline registers.
    2. Results from later stages (e.g., EX/MEM, MEM/WB).
- The ALU Control Unit (Figure 7) generates a 4-bit control signal (ALU_Control) based on the instruction's Function field and the 2-bit ALU_OP signal from the control unit.
- Results from this stage (ALU_RESULT) and other control signals are passed to the EX/MEM pipeline register (Figure 8) for use in the memory stage.

5. Memory Stage

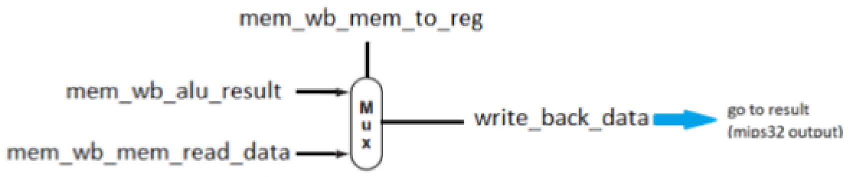
Figure 9: Memory.



- The data memory module (Figure 9) is accessed in this stage for:
    1. Load Instructions (lw): The memory address is calculated using the ALU, and data is read from the memory.
    2. Store Instructions (sw): Data is written to memory at the calculated address.
- Results and control signals are passed to the MEM/WB pipeline register (Figure 10) for the final write-back stage.

6. Write-Back Stage

Figure 11: Write Back.



- The final stage writes results back to the register file (Figure 11).

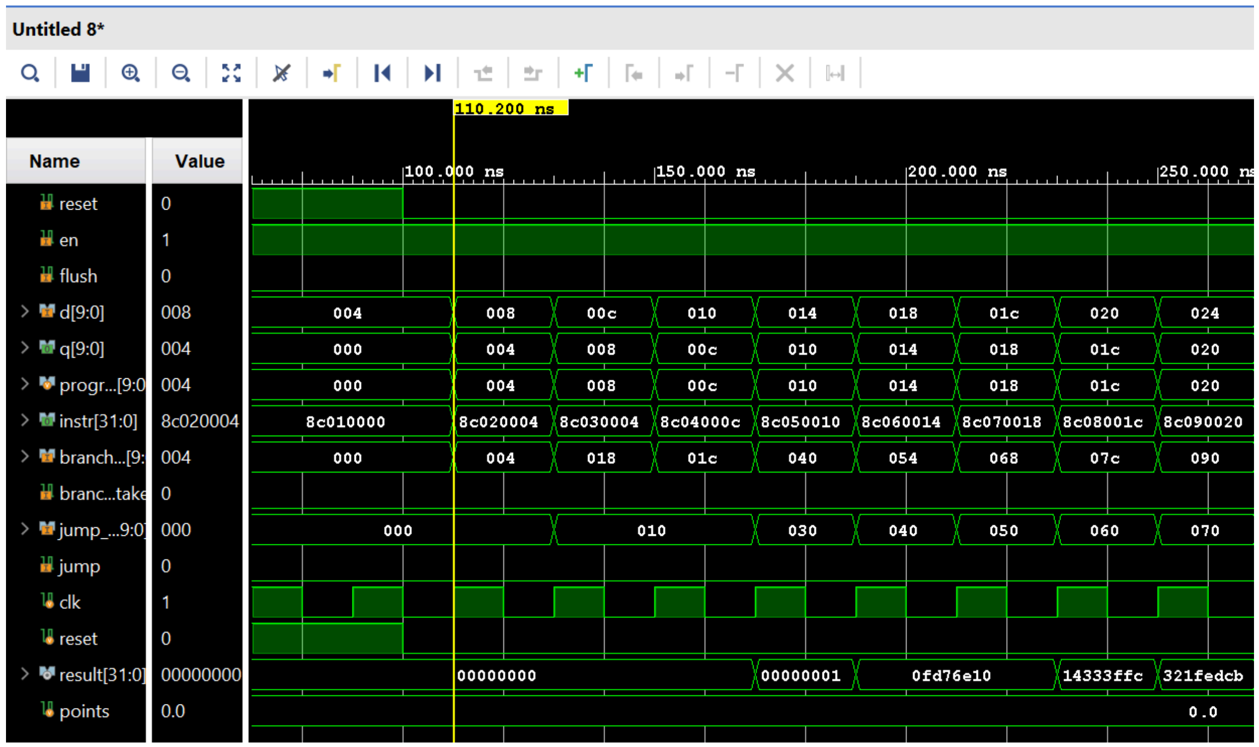
- A multiplexer selects the data to write:
    1. ALU results (for arithmetic and logical operations).
    2. Data from memory (for load instructions).
- The selected data is written to the destination register specified in the instruction.

7. Integration (mips_32 Module)

- All stages are integrated using pipeline registers (IF/ID, ID/EX, EX/MEM, MEM/WB),
- Hazard detection (Hazard_detection) and forwarding (EX_Forwarding_unit) ensure proper instruction flow and handle dependencies efficiently.

3 Simulation Results

1. Basic Instruction Flow (No Hazards)



This waveform represents the ideal pipelined processor behavior:

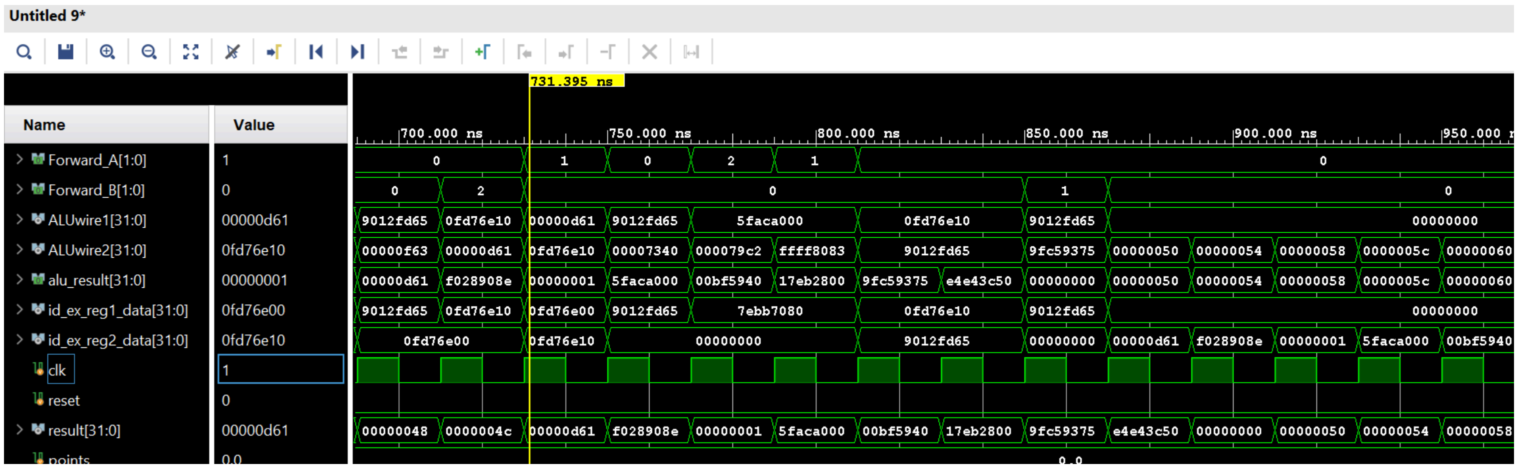
- All instructions are executed in a sequential manner without any stalls, hazards, or interruptions.
- The absence of branch or jump instructions ensures there are no control hazards.
- No data dependencies exist, so no forwarding or stalling mechanisms are required.

- Each stage of the pipeline is fully utilized, demonstrating optimal performance.

Signals:

- Instruction (instr[31:0]) Progression:
  - The instr values update sequentially at every clock cycle (e.g., 8c020004, 8c010000, 8c030004, etc.), indicating that new instructions are fetched and executed smoothly without any pipeline interruptions.
- Program Counter (program_counter[9:0]):
  - The program_counter increments in steps of 4 at each clock cycle (e.g., 004, 008, 00C, 010), confirming the sequential execution of instructions with no branching or jumps.
- Branch and Jump Signals:
  - branch_taken and jump are both 0, showing no branch or jump instructions are executed, ensuring a straightforward flow of instructions through the pipeline.
  - The branch_address and jump_address are unused, as there are no control hazards present in this scenario.
- Pipeline Continuity:
  - The flush signal remains 0, indicating that no instructions are being cleared or stalled in the pipeline.
  - The en signal is consistently 1, allowing the pipeline to remain active and progress instructions efficiently.
- Output Result (result[31:0]):
  - The result signal updates regularly, reflecting the output of executed instructions. This indicates that the instructions are processed without delay, and the pipeline is working as intended.
- Clock Signal (clk):
  - The clock operates consistently, triggering updates at predictable intervals, ensuring smooth instruction flow through the pipeline stages.

2. Data Hazard with Forwarding



```
Name                  Value      Waveform
Forward_A[1:0]        1          0 | 1 | 0 | 2 | 1 | 0
Forward_B[1:0]        0          0 | 2 | 0 | 1 | 0
ALUwire1[31:0]        00000d61   9012fd65 | 0fd76e10 | 00000d61 | 9012fd65 | 5faca000 | 0fd76e10 | 9012fd65 | 00000000
ALUwire2[31:0]        0fd76e10   00000f63 | 00000d61 | 0fd76e10 | 00007340 | 000079c2 | ffff8083 | 9012fd65 | 9fc59375 | 00000050 | 00000054 | 00000058 | 0000005c | 00000060
alu_result[31:0]      00000001   00000d61 | f028908e | 00000001 | 5faca000 | 00bf5940 | 17eb2800 | 9fc59375 | e4e43c50 | 00000000 | 00000050 | 00000054 | 00000058 | 0000005c | 00000060
id_ex_reg1_data[31:0] 0fd76e00   9012fd65 | 0fd76e10 | 0fd76e00 | 9012fd65 | 7ebb7080 | 0fd76e10 | 9012fd65 | 00000000
id_ex_reg2_data[31:0] 0fd76e10   0fd76e00 | 0fd76e10 | 00000000 | 9012fd65 | 00000000 | 00000d61 | f028908e | 00000001 | 5faca000 | 00bf5940
clk                   1
reset                 0
result[31:0]          00000d61   00000048 | 0000004c | 00000d61 | f028908e | 00000001 | 5faca000 | 00bf5940 | 17eb2800 | 9fc59375 | e4e43c50 | 00000000 | 00000050 | 00000054 | 00000058
points                0.0        0.0
```

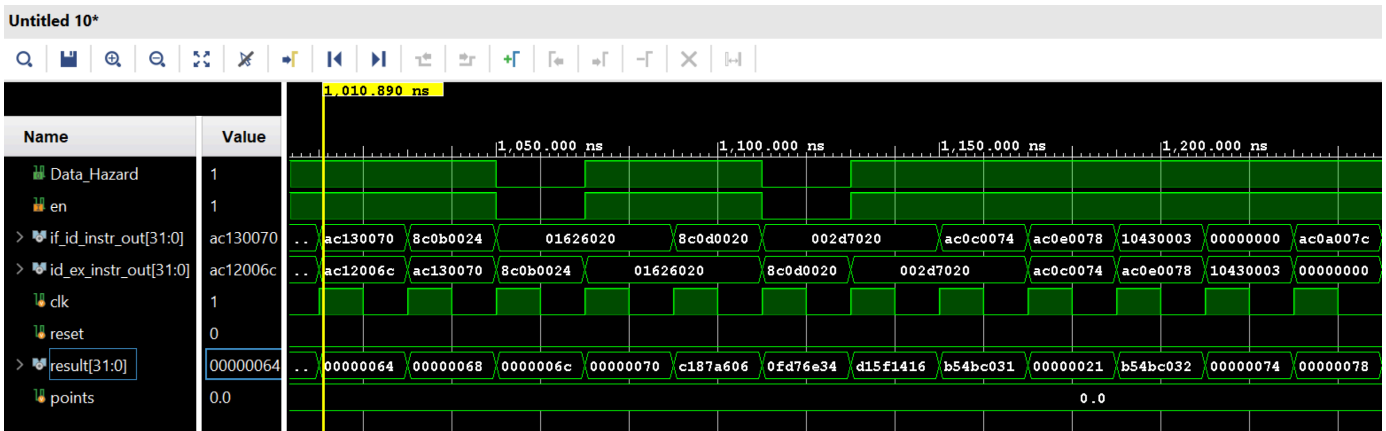
The waveform shows:

- This waveform illustrates a data hazard caused by dependent instructions. Instead of stalling the pipeline, the hazard is resolved through data forwarding (bypassing).
- The forwarding control logic identifies the dependency and routes the needed values directly to the ALU inputs.
- This mechanism ensures the pipeline operates efficiently, maintaining a high throughput without introducing stalls.

Signals:

- Forwarding Control Signals (Forward_A[1:0], Forward_B[1:0]):
    - At specific cycles (e.g., 750 ns), Forward_A is 1 and Forward_B is 0, indicating that data from a later pipeline stage is forwarded to resolve a dependency on source registers for the current instruction.
- ALU Inputs and Forwarded Data:
    - ALUwire1 and ALUwire2 receive values directly from the forwarding paths (5faca000 and 0fd76e10), demonstrating that the required operands are forwarded instead of fetched from the register file.
    - These forwarded values allow the ALU to compute the result without stalling.
- Instruction Dependency Resolution:
    - Dependencies are resolved dynamically using the forwarding mechanism. For instance:
        - The instruction in the EX stage requires data produced by a previous instruction in the MEM or WB stage.
        - Forwarding provides these values directly from the later stage outputs to the ALU inputs.
- Sequential Execution Continues:
    - Despite the data dependency, the clk signal triggers new operations at each cycle, and the result is updated regularly (e.g., 5faca000, 0bf5940, 17eb2800), confirming no pipeline stall.
- No Pipeline Stall (Forwarding Success):
    - The Forward_A and Forward_B signals eliminate the need for stalling by bypassing the register file read stage. The pipeline remains active and functional across all stages.

3. Data Hazard Without Forwarding (Pipeline Stall)



Name	Value
Data_Hazard	1
en	1
if_id_instr_out[31:0]	ac130070
id_ex_instr_out[31:0]	ac12006c
clk	1
reset	0
result[31:0]	00000064
points	0.0

```
if_id_instr_out  .. ac130070 8c0b0024 01626020 8c0d0020 002d7020 ac0c0074 ac0e0078 10430003 00000000 ac0a007c
id_ex_instr_out  .. ac12006c ac130070 8c0b0024 01626020 8c0d0020 002d7020 ac0c0074 ac0e0078 10430003 00000000
result           .. 00000064 00000068 0000006c 00000070 c187a606 0fd76e34 d15f1416 b54bc031 00000021 b54bc032 00000074 00000078
points           0.0
```

The waveform Shows:

- In this scenario, the hazard detection unit identifies a data dependency between instructions. Without forwarding, the pipeline must stall until the dependent data is written back to the register file. This ensures correct instruction execution but reduces throughput as the pipeline cannot execute instructions during the stall period.

Signals:

Data_Hazard Signal:

- The Data_Hazard signal is asserted (1), indicating a data dependency that requires stalling the pipeline to prevent incorrect instruction execution.

Instruction Stalling:

- The values of if_id_instr_out and id_ex_instr_out remain constant (ac130070 and ac12006c) for multiple cycles.
- This shows that the pipeline is frozen at the instruction fetch and decode stages, as new instructions cannot proceed until the data dependency is resolved.

Pipeline Stall:

- The en signal is active, enabling the stalling mechanism while the dependency persists.
- During this time, the pipeline does not fetch or decode new instructions, causing a visible delay in the flow of instructions through the pipeline.

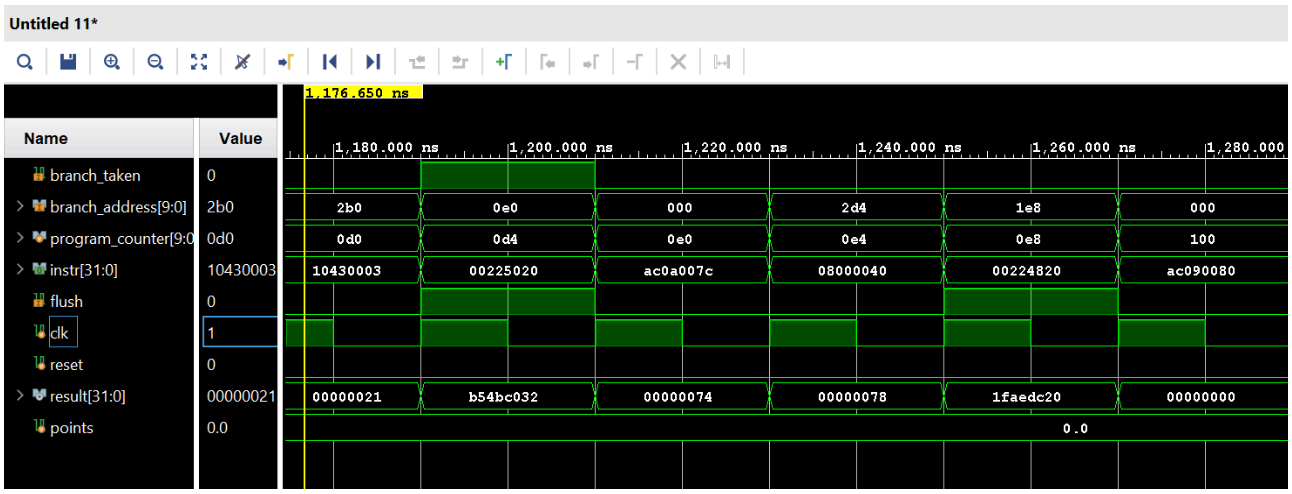
result Signal Updates:

- The result signal advances slower compared to hazard-free scenarios, e.g., progressing from 00000064 to 00000068 and 0000006c after the stall resolves.
- This confirms that the execution of dependent instructions is delayed due to the stall.

Clock Continuity:

- Despite the stall, the clock (clk) continues, confirming that the pipeline is stalled rather than halted.

4. Control Hazard (Branch Taken)



Name	Value
branch_taken	0
branch_address[9:0]	2b0
program_counter[9:0	0d0
instr[31:0]	10430003
flush	0
clk	1
reset	0
result[31:0]	00000021
points	0.0

The waveform shows:

a control hazard caused by a branch instruction. The pipeline resolves the hazard by:

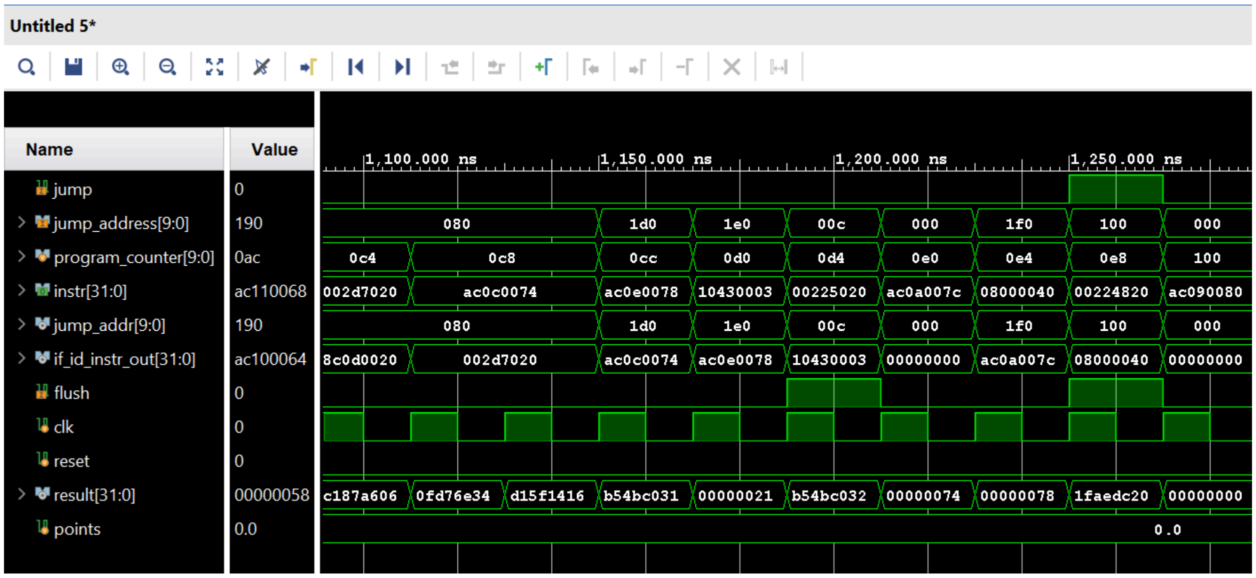
- Redirecting the program_counter to the branch target address (2b0).
- Flushing incorrect instructions.
- Resuming correct execution flow after the branch.

Signals:

- Branch Decision (branch_taken):
  - At 1,180 ns, branch_taken becomes 1, indicating that the branch condition is met, and the pipeline should redirect to the branch address (branch_address).
- Program Counter (program_counter) Update:
  - Before the branch decision, program_counter follows sequential instruction addresses (e.g., 0d0, 0e0).
  - When branch_taken is asserted, the program_counter redirects to the specified branch_address value (2b0), demonstrating the branch redirection.
- Instruction Fetch (instr):
  - Prior to the branch being taken, the instructions in the pipeline are fetched sequentially (e.g., 10430003).

    - Once the branch is taken, the instruction at the branch target address (ac0a007c) is fetched, indicating the pipeline correctly redirected to the branch target.
- Pipeline Flush (flush):
    - The flush signal is asserted momentarily to clear incorrect instructions in the pipeline that were fetched before the branch decision, ensuring no incorrect execution occurs.
- Result Signal (result):
    - Despite the branch hazard, the pipeline successfully executes instructions after the branch, and result progresses correctly (e.g., 00000021, 00000074, etc.).

5. Control Hazard (Jump Instruction)



The waveform shows:

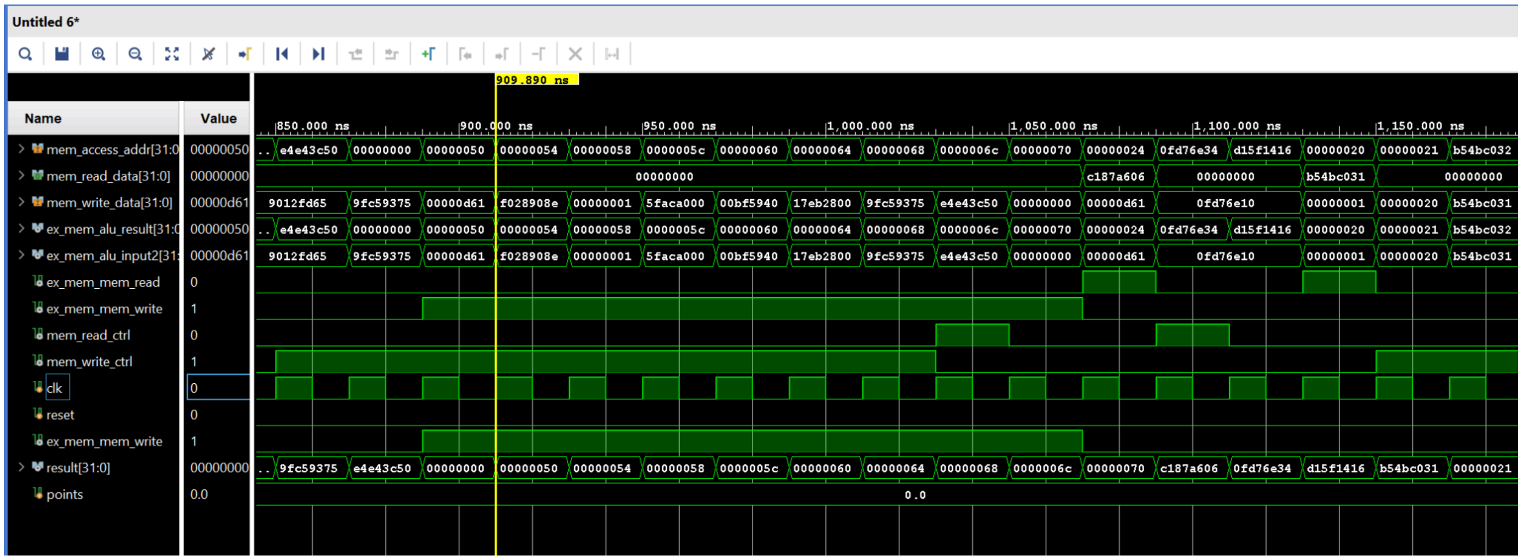
- This visualization captures the disruption and redirection caused by a jump instruction, along with the pipeline's recovery process.

Signals:

- Jump Signal Activation:
    - The jump signal is set high (becomes active) to indicate a jump instruction. This triggers a redirection of the program counter to the jump_address (190).
- Program Counter Update:
    - Before the jump, the program_counter follows the normal incrementing sequence (0ac, 0c4, 0c8). After the jump, the program_counter abruptly switches to the jump target address (190), skipping the intermediate instructions.
- Instruction Replacement:

    - The instr and if_id_instr_out signals change from their previous sequence (e.g., ac110068, ac0c0074) to instructions fetched from the jump target address (ac100064).
- Pipeline Flush:
    - The jump causes a flush in the pipeline, clearing the instructions from the incorrect path. This is evident as the pipeline stages are overwritten with instructions fetched from the jump target.
- Hazard Resolution:
    - The flush and redirection show the control hazard resolution caused by the jump instruction, ensuring that the pipeline fetches and executes the correct instructions following the jump.

6. Memory Operations (Load and Store)



The waveform shows:

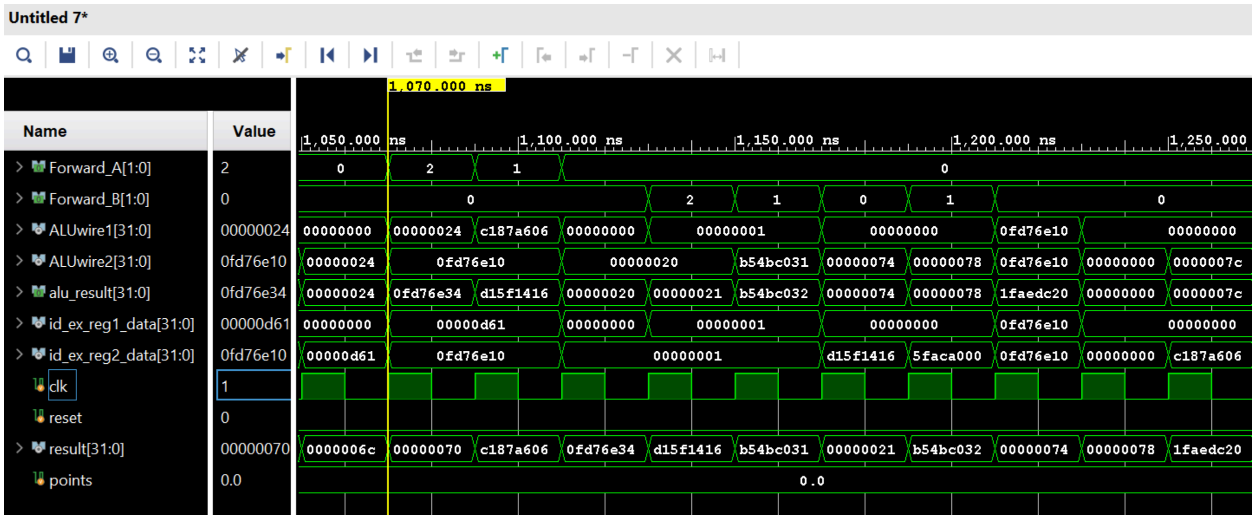
- This visualization captures the disruption and redirection caused by a jump instruction, along with the pipeline's recovery process.

Signals:

- Memory Read (Load) Operation:
    - Signals: The mem_read_ctrl signal is active during a load operation, indicating that data is being fetched from the memory.
    - Data Read: The mem_access_addr specifies the address being accessed, and mem_read_data displays the value read from memory. For example, when mem_access_addr points to a specific address, the corresponding value appears on mem_read_data.

- Memory Write (Store) Operation:
  - Signals: The mem_write_ctrl signal is active during a store operation, indicating that data is being written to memory.
  - Data Write: The mem_write_data shows the value being written to memory at the address specified by mem_access_addr.
- Address and Data Correlation:
  - For each clock cycle, the mem_access_addr correlates with either a load or store operation, depending on the state of mem_read_ctrl and mem_write_ctrl.
  - In the waveform, you can observe alternating read and write operations as indicated by changes in mem_read_ctrl and mem_write_ctrl.
- Pipeline Integration:
  - The ex_mem_alu_result and ex_mem_alu_input2 wires show how the ALU computes the effective memory address and passes it down the pipeline for memory operations.
  - The data being loaded or stored is properly integrated with the pipeline registers, ensuring that subsequent stages receive the correct values.
- Output Results:
  - The result register updates correctly as memory operations are performed. For instance, the result register reflects the value loaded from memory during a load operation or remains unaffected during a store operation.

7. Forwarding to Resolve Complex Dependencies



Name	Value
Forward_A[1:0]	2
Forward_B[1:0]	0
ALUwire1[31:0]	00000024
ALUwire2[31:0]	0fd76e10
alu_result[31:0]	0fd76e34
id_ex_reg1_data[31:0]	00000d61
id_ex_reg2_data[31:0]	0fd76e10
clk	1
reset	0
result[31:0]	00000070
points	0.0

Forwarding Signals:

- Forward_A and Forward_B signals indicate active forwarding paths. For example:
  - Forward_A = 2 indicates that the value is forwarded from the EX_MEM stage to the ALU input.

    - Forward_B = 0 means no forwarding is needed for that operand, as the value is directly available.

ALU Input Values:

- ALUwire1 and ALUwire2 show the operands being fed into the ALU. These values are directly forwarded from previous pipeline stages when necessary, resolving data hazards caused by dependencies.
- For instance:
    - At a particular cycle, ALUwire1 receives 0fd76e10, which was the ALU_result of a previous instruction in the EX_MEM stage.
    - Similarly, ALUwire2 gets the appropriate forwarded value to prevent a stall.

Dependency Resolution:

- The id_ex_reg1_data and id_ex_reg2_data wires reflect the original values for the current instruction's operands. Forwarding ensures that these values are updated or replaced when dependencies exist.
- For example:
    - The id_ex_reg1_data initially holds 0fd76e10, which is forwarded to the ALU as shown by Forward_A = 2.

Pipeline Continuity:

- Despite dependencies, the pipeline continues without stalls. The result register updates correctly, indicating that the forwarding mechanism is successfully handling dependencies between instructions.
- For instance, the result value progresses through 0000006c, c187a606, and 0fd76e10 over consecutive clock cycles, showing that computations proceed without interruption.

Control of Forwarding:

- The control logic ensures the correct selection of forwarding paths, as seen in the transitions of Forward_A and Forward_B. These signals adjust dynamically based on the dependencies detected between stages.