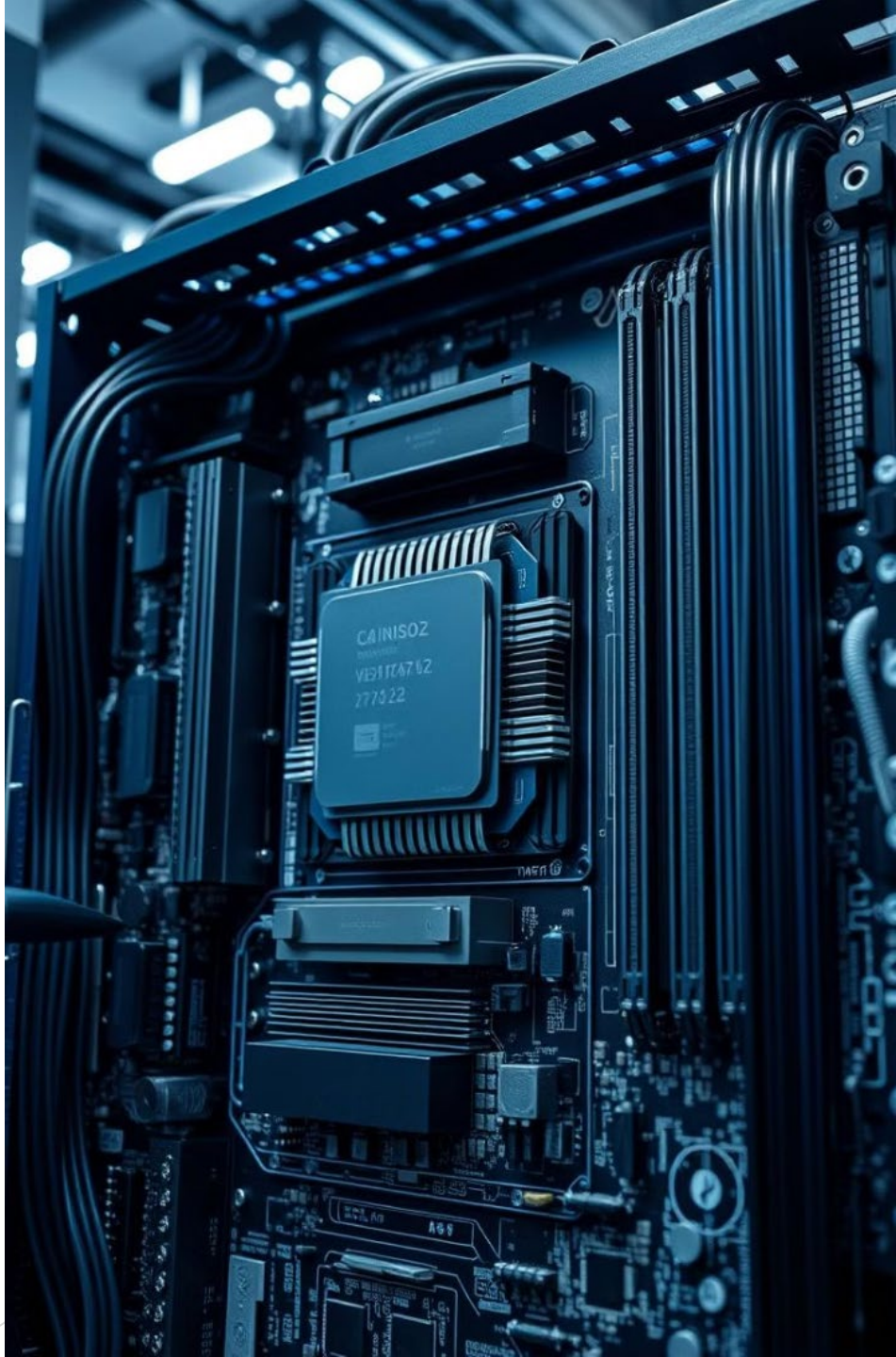
Implementing a 5-Stage Pipeline

By Register Jammers

Avik Mandal, Ankit Kumar, Kavy Vaghela, Navodit Verma, Utkarsh Kumar

github.com/utkarshk-iitr/MIPS_5-Stage_Pipeline

Contents



Acknowledgment

We would like to express our sincere gratitude to Prof. Peddoju Sateesh Kumar for providing us with the opportunity to undertake this project on implementing a 5-stage pipeline processor in the MIPS architecture. The project has enabled us to deepen our understanding of computer architecture and pipeline processing, offering valuable insights into practical applications of theoretical concepts. We also extend our thanks to our peers who provided their assistance and encouragement throughout this project. This experience has greatly contributed to our knowledge and skills in the field of computer engineering.

Overview of Project

Implementing a 5-Stage Pipelined Architecture in Verilog

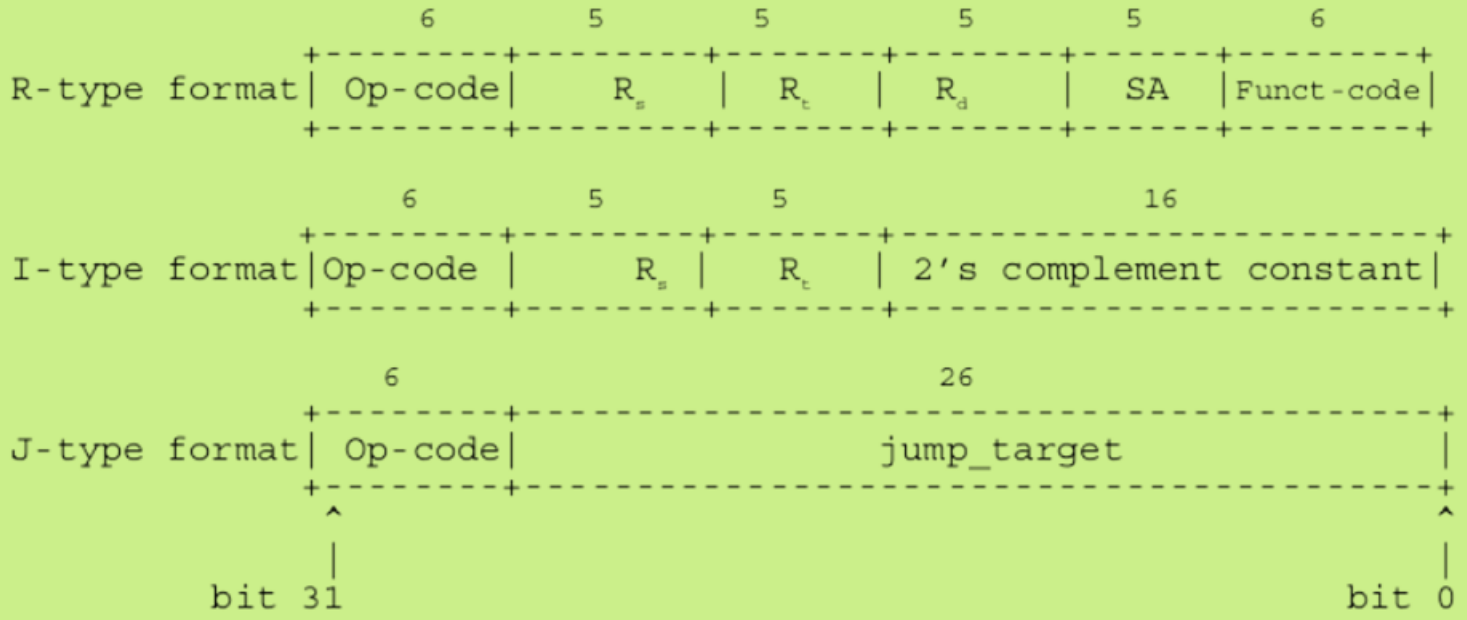
- Description: Design a 5-stage pipeline (Fetch, Decode, Execute, Memory, Writeback) for a MIPS processor. Implement the pipeline with hazards, forwarding, and stalling mechanisms.

- Deliverables:

1. Verilog source code for each stage of the pipeline.
2. Testbenches to validate individual stages and the complete pipeline.
3. Simulation results showing the execution of a simple instruction sequence.
4. Documentation explaining the pipeline stages, hazard handling, and the results

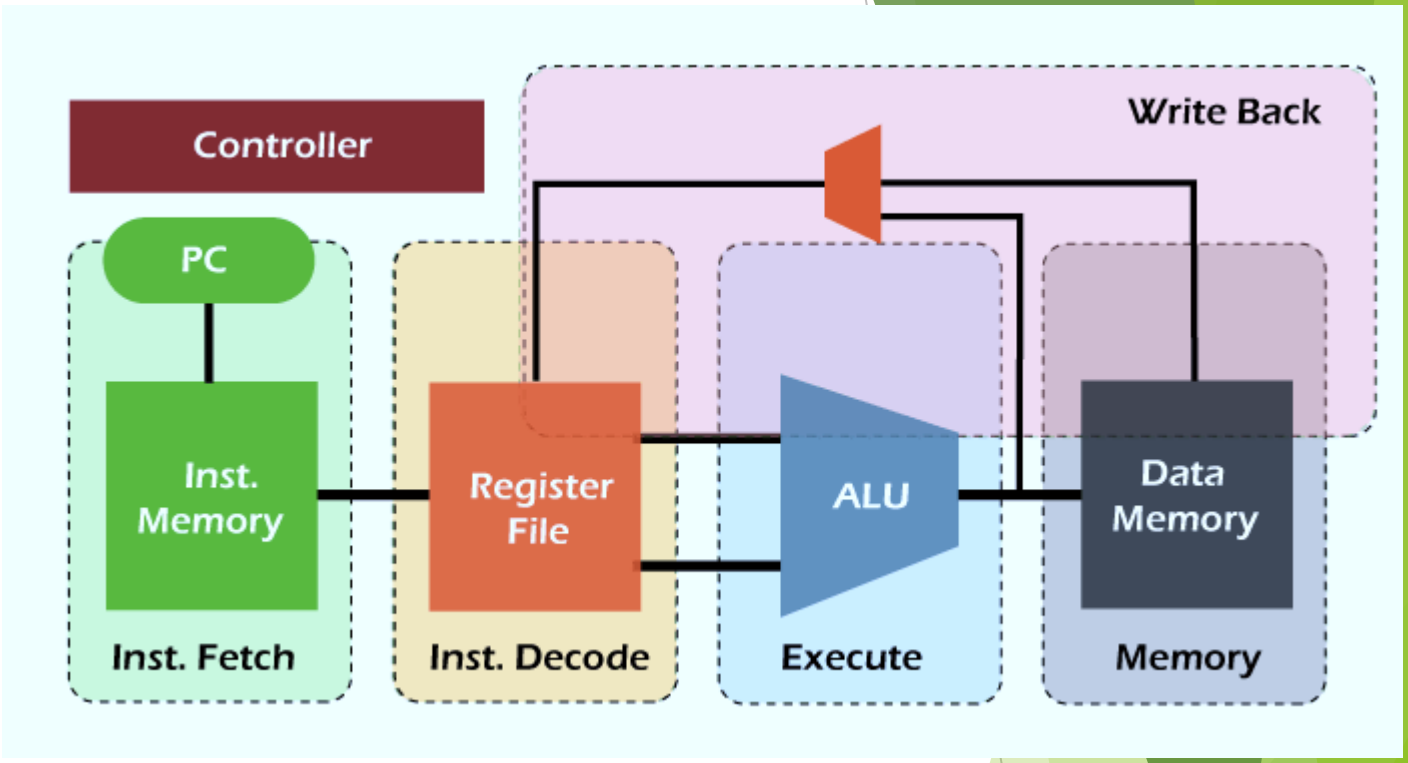
MIPS ISA Structure

Instruction Formats:
Instruction formats: all 32 bits wide (one word):



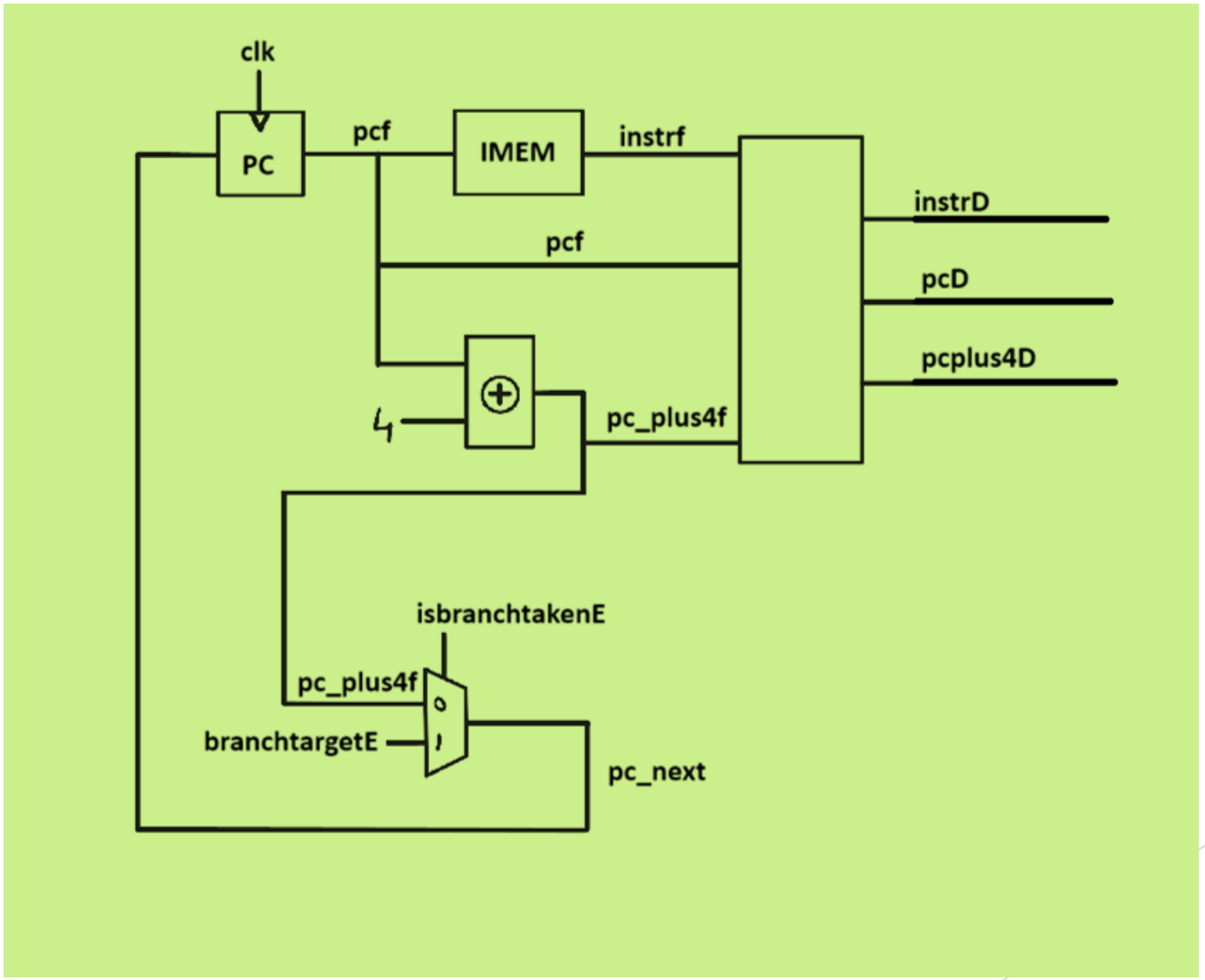
Pipelines

We design a pipelined processor by subdividing the single-cycle processor into five pipeline stages.



Thus, five instructions can execute simultaneously, one in each stage. Because each stage has only one-fifth of the entire logic, the clock frequency is approximately five times faster.

Diagram of Fetch Cycle



Implementation of Decode Cycle

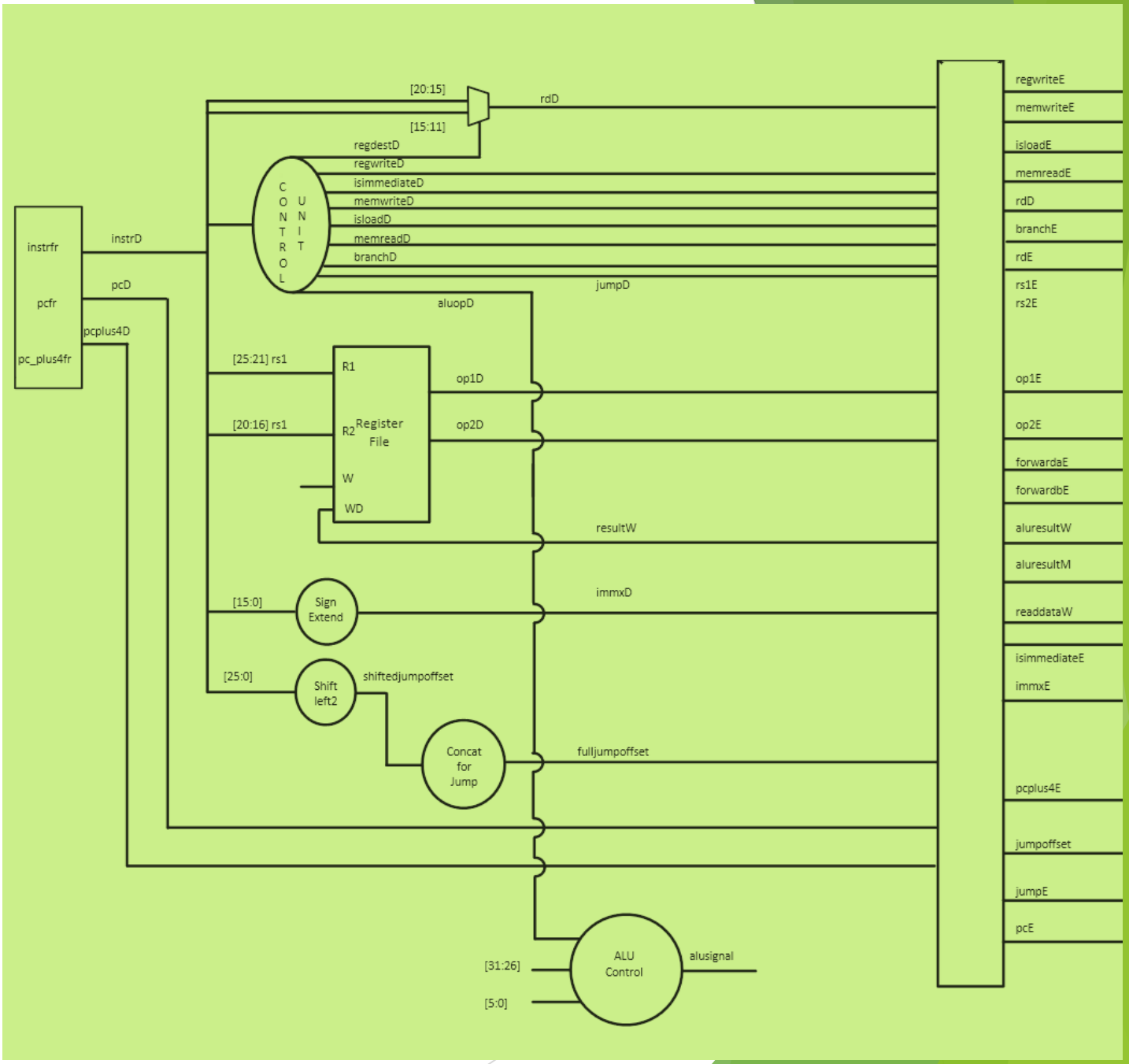
The Decode Cycle Module decodes the fetched instruction, generates control signals, and reads values from registers for the MIPS pipeline's next stage.

Component Breakdown:

- Control Unit: Generates control signals based on the opcode and instruction fields. Signals include RegWrite, ALUSrc, MemRead, MemWrite, Branch, and Jump.
- Register File: Reads values from two registers (RS1, RS2) based on instruction fields. Enables write-back with control signal RegWrite and provides the write-back register and data.
- ALU Control: Determines the ALU operation based on instruction fields (e.g., funct) and ALUop signal from the control unit.
- Pipeline Registers: Stores control signals (e.g., RegWrite, ALUSrc) and decoded values (op1, op2, immx) for the execute stage.

- Output Signals: Control signals (regwriteE, isimmediateE, etc.), register values (op1E, op2E), and addresses (rdE, rs1E, rs2E). Jump offset (jumpoffset) and updated PC values (pcE, pcplus4E).

Diagram of Decode Cycle



Implementation of Execute Cycle

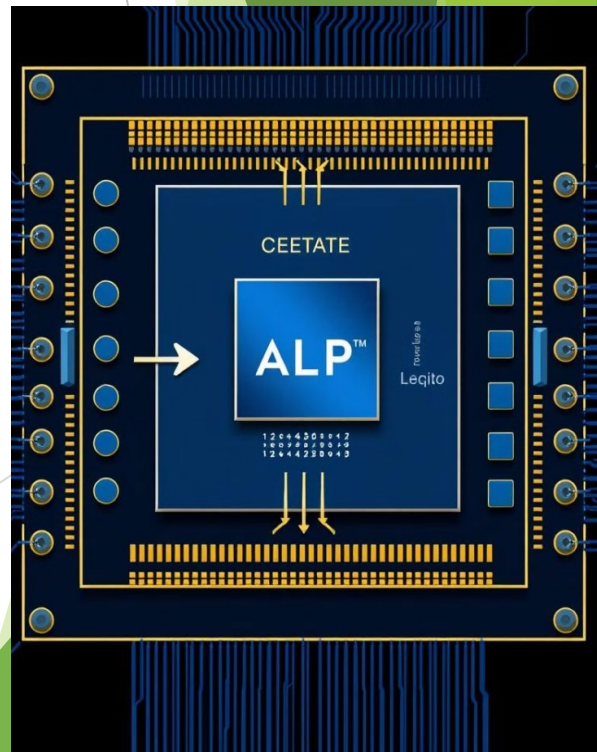
The Execute Cycle performs arithmetic/logic operations, calculates branch targets, and manages control signals for the memory stage in the MIPS pipeline. We also implemented hazard control later.

Component Breakdown:

- ALU Operations: The ALU performs operations based on alusignalE, using operands op1E and either op2E or immxE (selected by ALUSrc signal). Result is stored in aluresultM for the next stage.
- Branch Target Calculation: Shifts the immediate value and adds it to PC + 4 to compute the branch target. A multiplexer selects between the branch target and jump address, controlled by jumpE.
- Branch Decision Logic: Sets isbranchtakenE if a jump or branch condition is met.
- Pipeline Registers: Stores control signals and results for the memory stage.

Output Signals:

- isbranchtakenE: Indicates if a branch or jump is taken.
- branchtargetE: Target address for branch.
- regwriteM, memwriteM: Control signals for memory access.
- rdM: Destination register for the result.



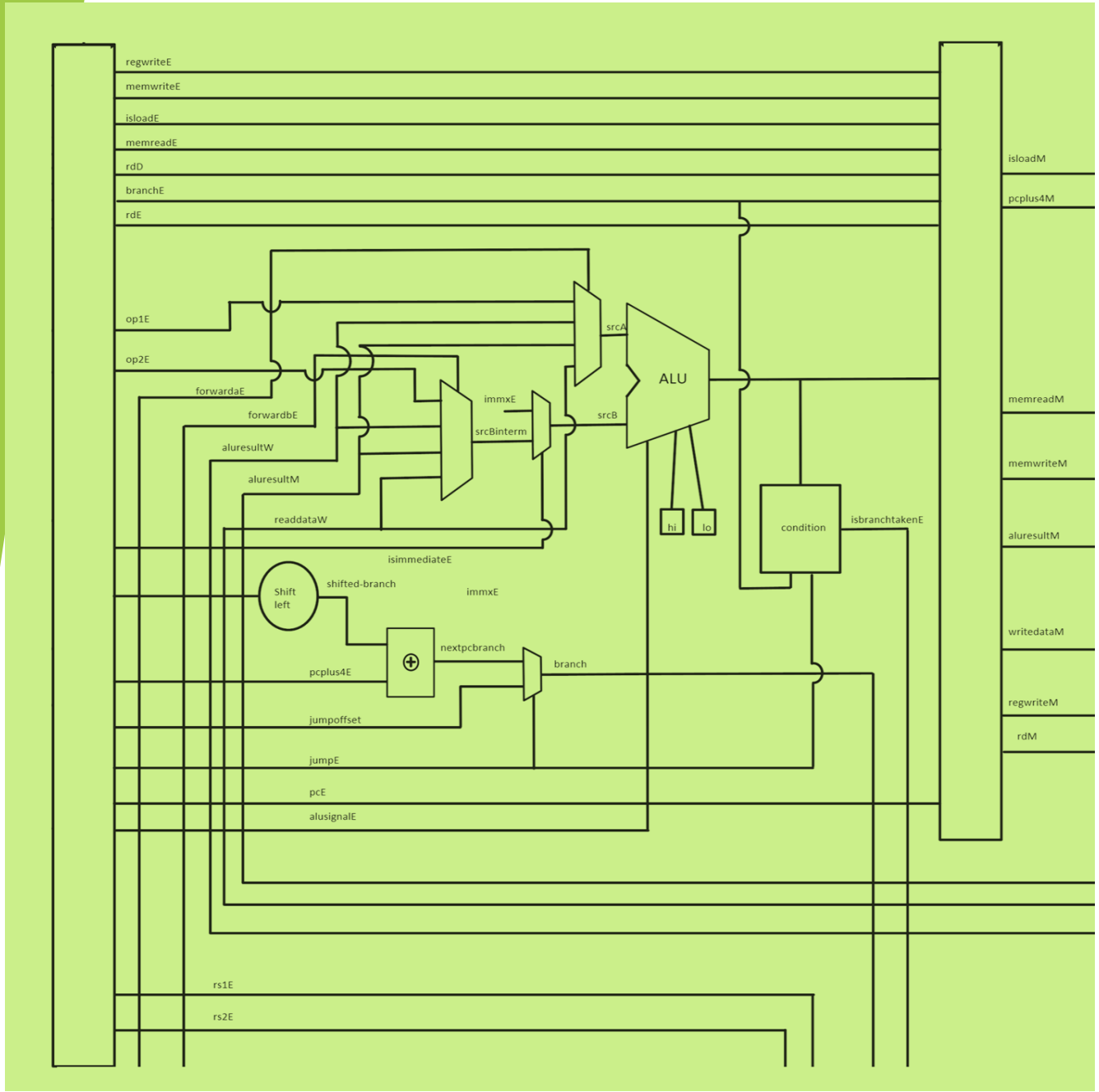


Diagram of Execute Cycle

Implementation of Memory Cycle

The Memory Cycle handles data memory access for load and store instructions and prepares control signals and data for the writeback stage.

Component Breakdown:

- Data Memory Access: DMemBank module performs read/write operations based on memreadM and memwriteM signals. aluresultM provides the address for memory access, and writedataM supplies data for store instructions. Read data from memory is stored in readdataW if memreadM is active.
- Pipeline Registers: Stores the computed ALU result, read data, program counter, and destination register for the next stage (write-back).Control signals (regwriteM, isloadM) are saved in regwrite and isloadW to indicate if data should be written to registers.

Output Signals:

- regwrite: Write-enable signal for register file in write-back.
- isloadW: Indicates if memory load result should be written back.
- pcplus4W: Incremented PC for the writeback stage.
- aluresultW, readdataW: ALU result and memory data.
- rd: Destination register for the writeback result.

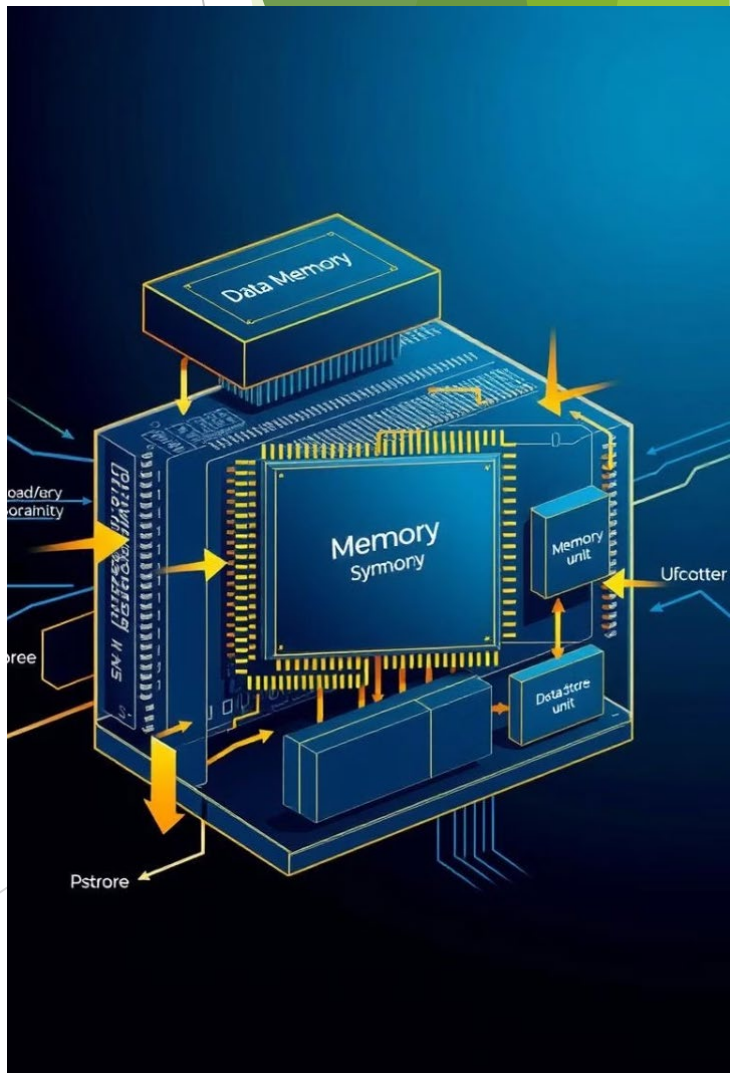
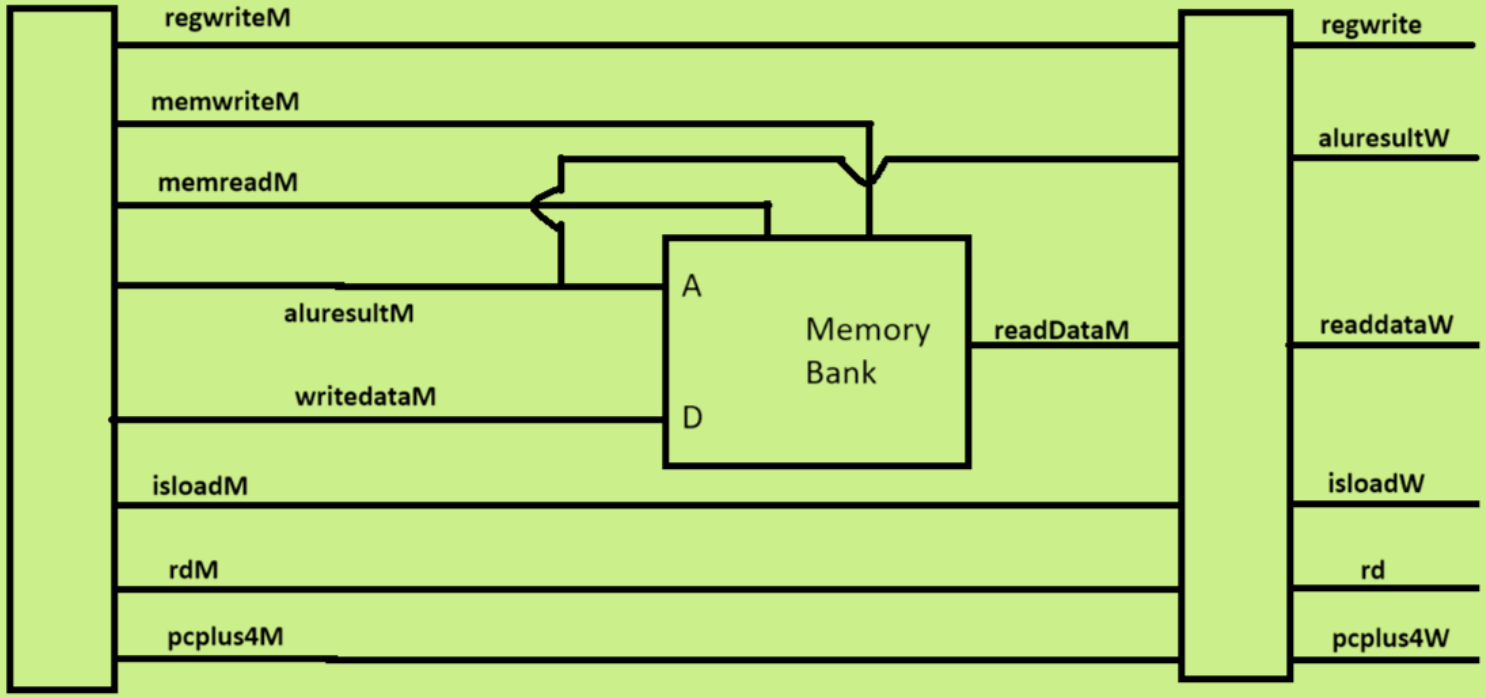


Diagram of Memory Cycle



Implementation of WriteBack Cycle

The Writeback Cycle module is the final stage in the MIPS pipeline, where computed data is written back to the register file. This stage ensures that either the ALU result, or memory data is available for further instructions.

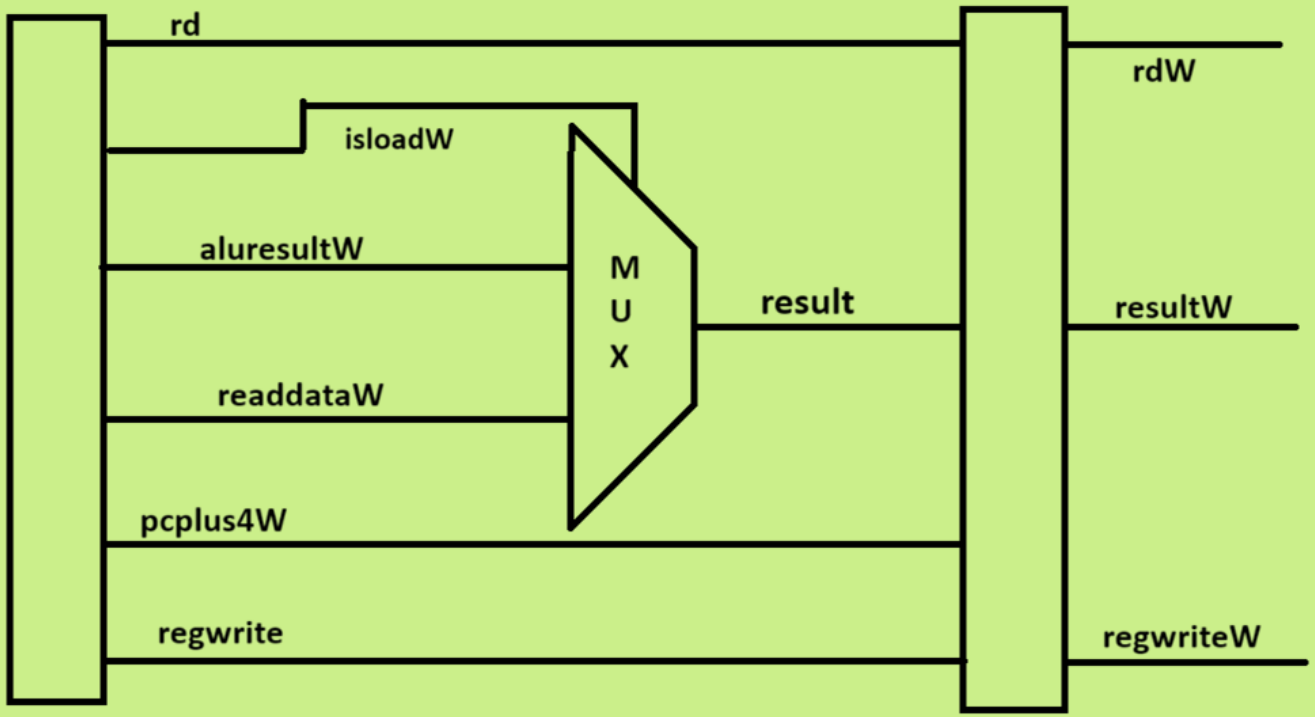
Component Breakdown:

- Data Selection (Mux): The writeback_mux selects between: aluresultW (output from ALU operation). readdataW (data loaded from memory). isloadW controls the selection, with memory data selected if isloadW is asserted.
- Pipeline Registers: resultW holds the final writeback result to be stored in the register file. regwriteW is a write-enable signal for the register file, controlled by the regwrite input. rdW specifies the destination register address where resultW will be written.

Output Signals:

- resultW: Final data for the register file.
- regwriteW: Enables writing to the register file.
- rdW: Specifies the destination register for write-back.

Diagram of WriteBack Cycle



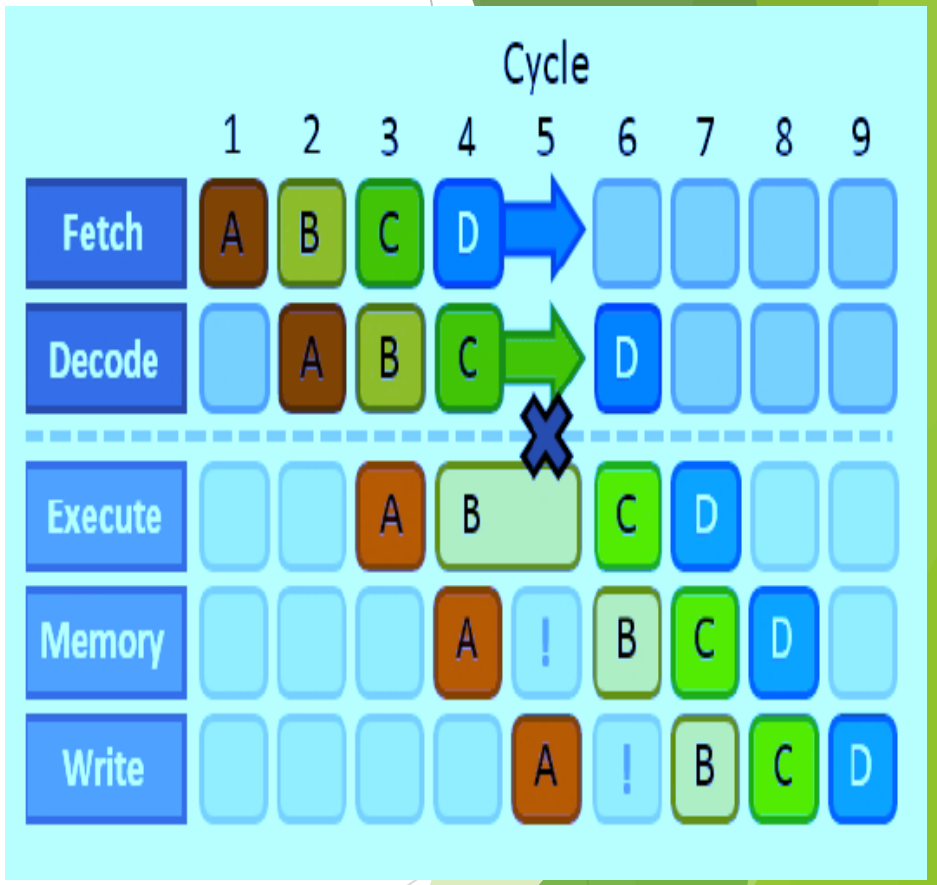
Pipeline Hazards

Pipeline hazards are issues that prevent the next instruction in the pipeline from executing in the following clock cycle. They reduce pipeline efficiency by causing stalls, which can limit performance gains from pipelining.

Data Hazards:

Definition: Occur when instructions depend on the results of previous instructions that are still in the pipeline.
Example: An instruction needs a value from a register that hasn't been written yet.

Solution: Forwarding (or bypassing) is commonly used to resolve data hazards by passing the required data from later stages directly to the execute stage.



Implementation of Hazard Unit

The Forwarding Module is essential for hazard resolution in the MIPS pipeline. It helps mitigate data hazards by forwarding values from later stages of the pipeline to earlier stages, allowing for the necessary data to be used immediately in dependent instructions.

Inputs:

- rst: Reset signal, used to reset forwarding.
- regwriteM and regwrite: Control signals indicating whether registers are written in the Memory and Writeback stages.
- isloadW: Indicates if the Writeback stage data is from a load instruction.
- rdM and rd: Destination registers in the Memory and Writeback stages.
- rs1E and rs2E: Source registers in the Execute stage.

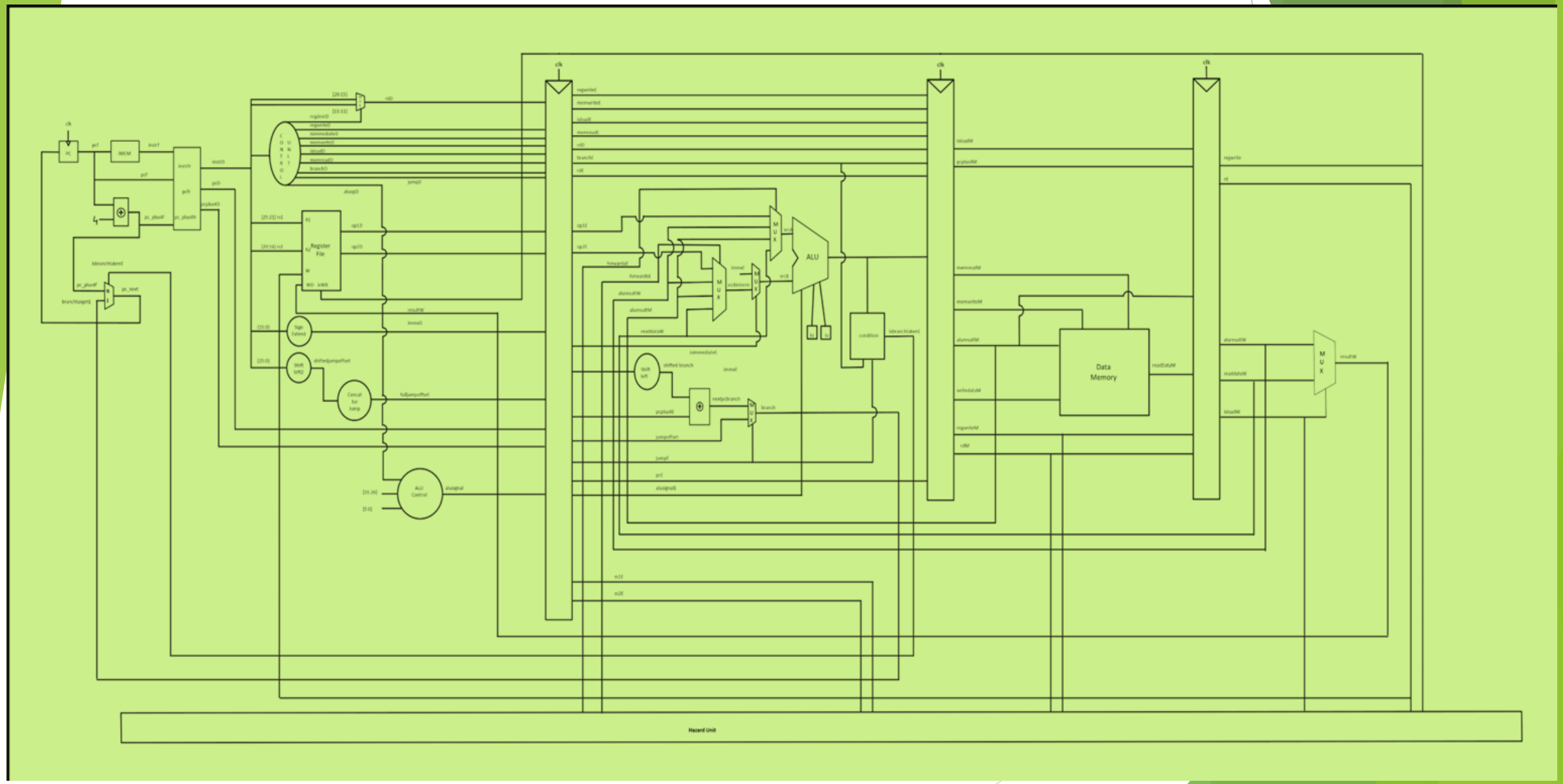
Outputs:

forwardaE and forwardbE: 2-bit control signals for forwarding

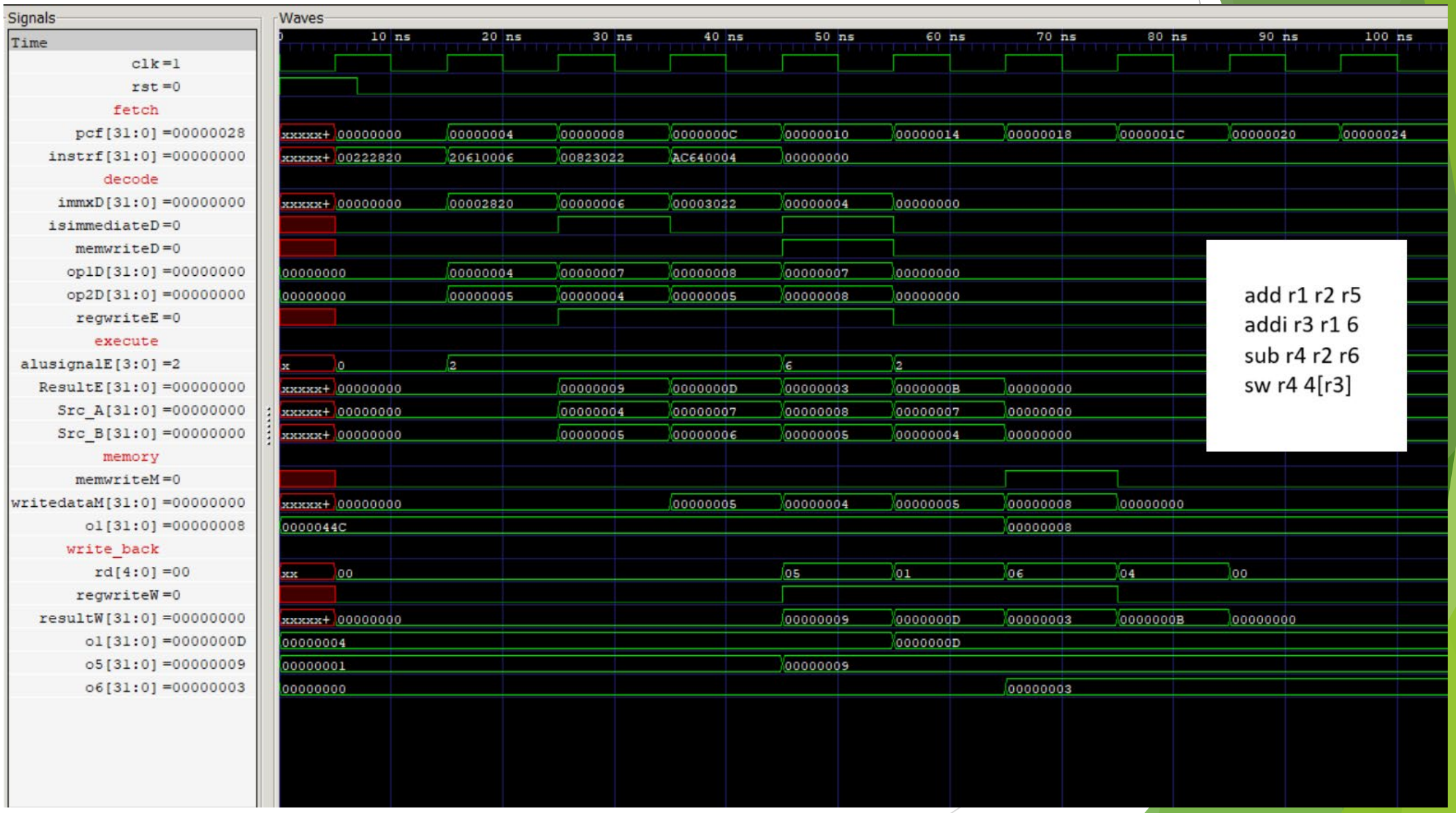
00: No forwarding
01: Forward from Writeback stage
10: Forward from Memory stage
11: Special case for load instructions

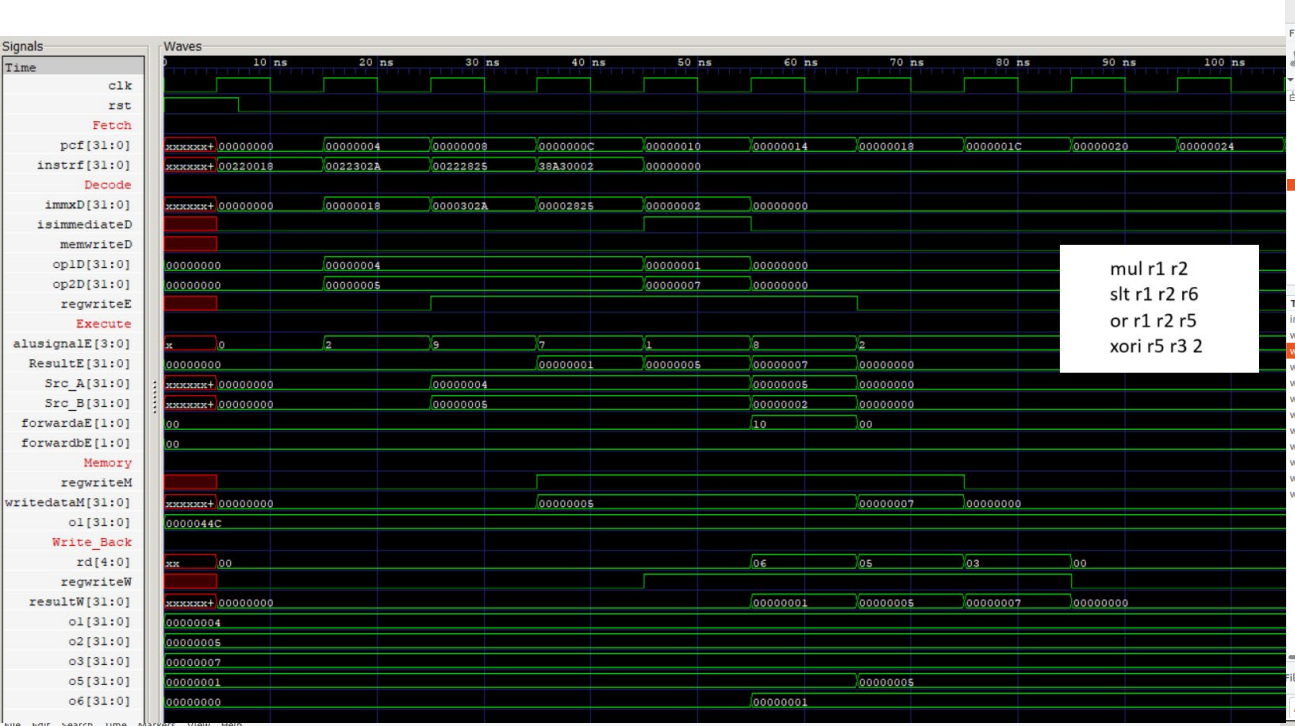
Forward to rs1E (forwardaE): Forward from Writeback if rd matches rs1E and isloadW is not asserted. Forward from Memory if rdM matches rs1E. For load instructions in Writeback, select 11 if rd matches rs1E.
Forward to rs2E (forwardbE): Logic is like forwardaE but checks rs2E.

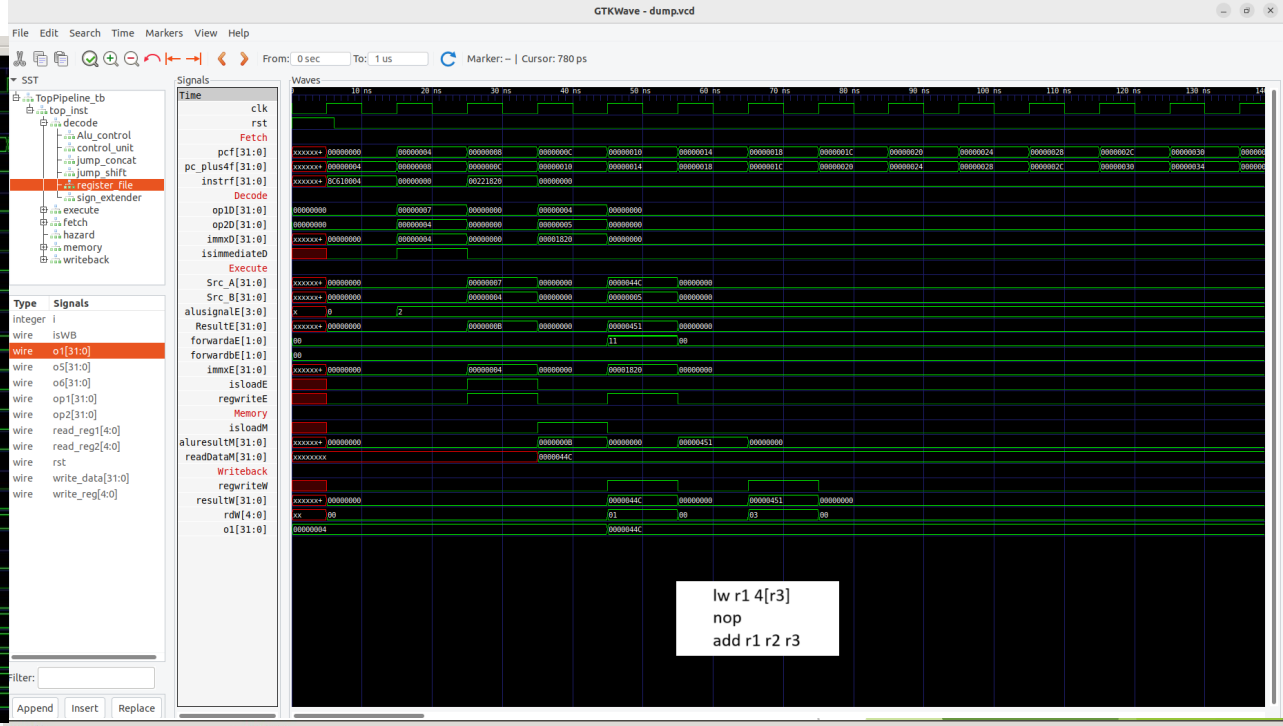
Final Circuit Diagram

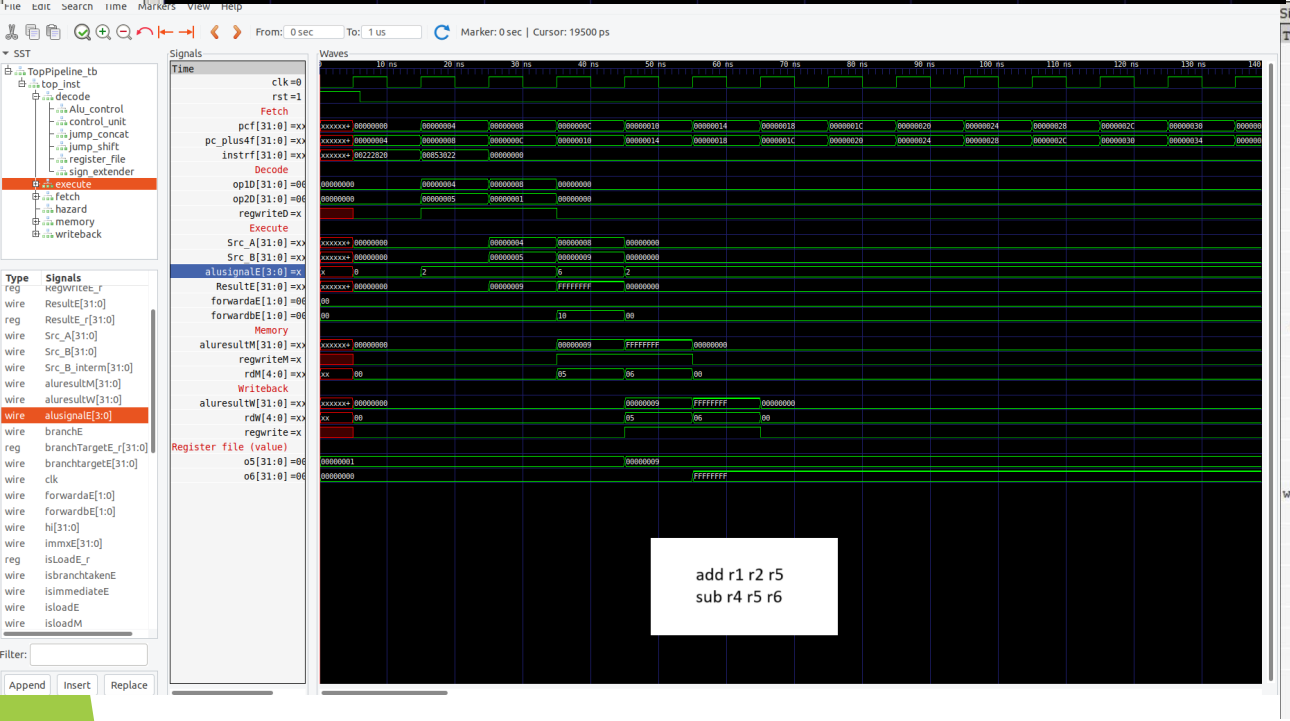


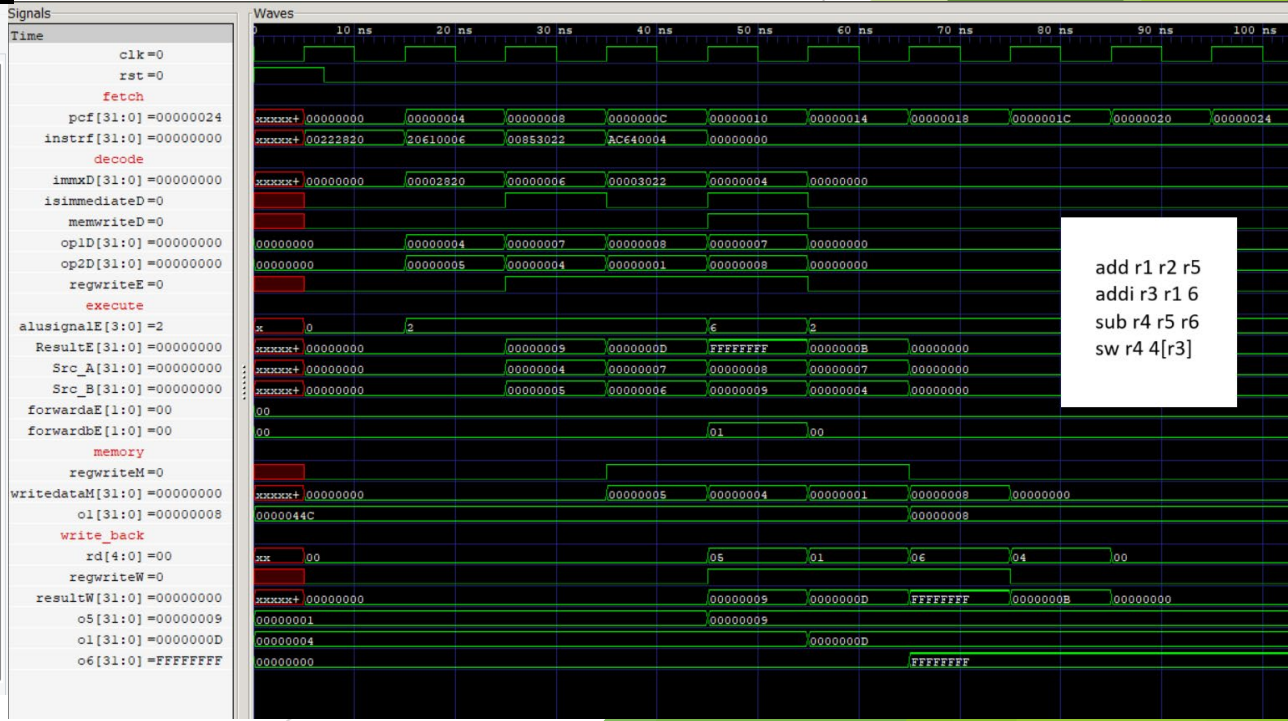
Example Output




Signals
Waves
mul r1 r2
slt r1 r2 r6
or r1 r2 r5
xori r5 r3 2


GTKWave - dump.vcd
lw r1 4[r3]
nop
add r1 r2 r3


add r1 r2 r5
sub r4 r5 r6


add r1 r2 r5
addi r3 r1 6
sub r4 r5 r6
sw r4 4[r3]

Conclusion

The MIPS pipeline divides instruction execution into stages to improve throughput and efficiency. Key stages—Fetch, Decode, Execute, Memory Access, and Writeback—each perform a specific function, allowing multiple instructions to overlap in execution.

Key Takeaways:

- Pipeline Hazards (Data, Control, Structural) are a challenge in pipelined processors but can be managed with techniques like forwarding, branch prediction, and resource separation.
- Forwarding Module: Essential for resolving data hazards, allowing dependent instructions to access required data without stalling.
- We also learnt a new hardware language Verilog on the fly.

Future Steps:

Explore more advanced pipeline techniques such as dynamic scheduling and out-of-order execution. Experiment with pipeline depth and additional hazard management techniques for optimized processor performance.

References

- https://en.m.wikipedia.org/wiki/MIPS_architecture
- https://web.archive.org/web/20220407105219/https://web.cse.ohio-state.edu/~crawfis.3/cse675-02/Slides/MIPS%20Instruction%20Set.pdf
- Computer Organization and Architecture by William Stallings
- https://www.youtube.com/playlist?list=PL3Soy1ohxlP1TLpcbYXYcVWItRy_XrUk8
- https://youtube.com/playlist?list=PL1C2GgOjAF-KFxGFauGAF3wOjYbmezNG0&si=UgJdHYntwCvH7McY

THANK YOU