UNIVERSITY OF NEVADA LAS VEGAS. DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING LABORATORIES.

Class:	**CPE300L 1001**			Semester:	**Fall 2024**
Points		Document author:	**Alireza Bolourian**		
		Author's email:	**bolouria@unlv.nevada.edu**		
		Document topic:	**Final Project Report**		
Instructor's comments:					

Goal: The goal of this project is to demonstrate an understanding of pipelining and address the challenges associated with it. While pipelining can increase latency, it compensates by enhancing throughput, making it advantageous in practical applications where higher computational output improves overall performance.

Abstract: In this project a five-stage pipeline for a subset of the ARMv4 instruction set architecture is designed starting from a single-cycle implementation provided in the *Digital Design and Computer Architecture ARM Edition by Sarah & David Harris*. Pipelining involves understanding and addressing challenges related, such as data hazards, control hazards, and forwarding, while maintaining functional correctness. This improves performance by overlapping instruction execution stages (fetch, decode, execute, memory, and writeback), showcasing how modern processors optimize instruction throughput. This pipeline model demonstrates key concepts in computer architecture, including hazard detection, branch prediction, and efficient resource utilization.

Pipeline Diagram

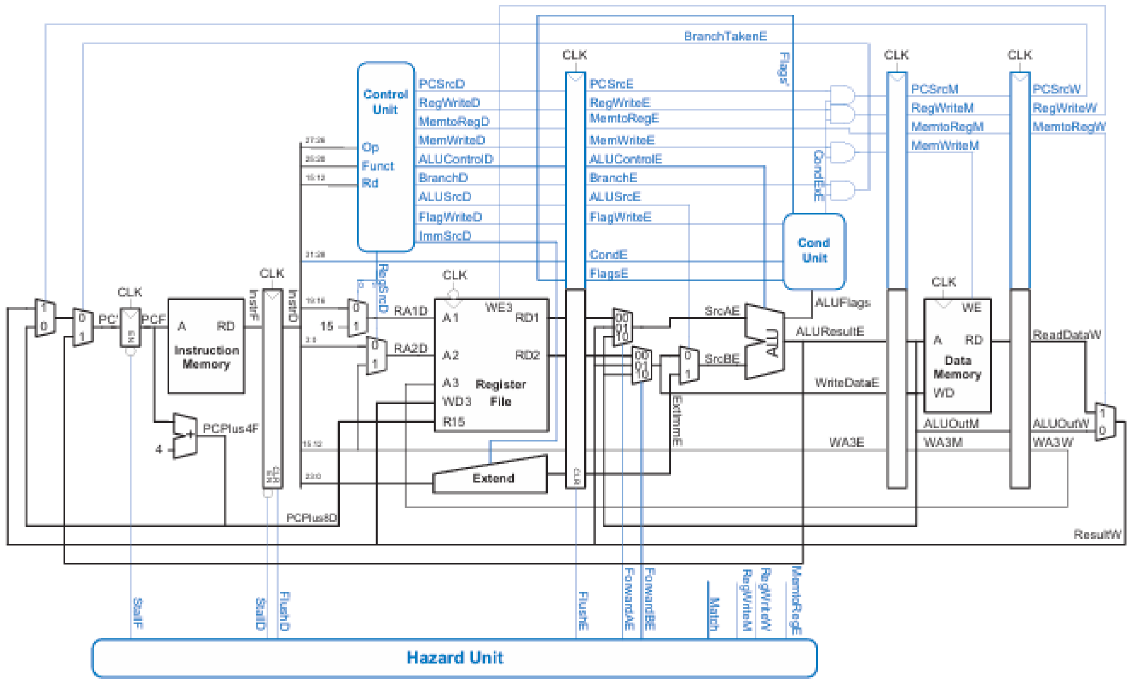


Figure 7.58 **Pipelined processor with full hazard handling**

This diagram shows the components and signals necessary for the pipeline implementation. Four registers divide the processor into 5 stages. These stages are Fetch, Decode, Execute, Memory and Writeback. These registers save the signals that have been processed in the previous steps. Two multiplexers have been added to allow for forwarding from memory or writeback into the execution stage. The hazard unit is responsible for detecting hazards and determines whether stalls and flushes. One interesting detail is that in the writeback stage, writing into the register occurs at the negative edge of the clock, so that in the second half of the cycle data can be read from the register in the decode stage without the need for forwarding.

Modules Overview:

Testbench

- **Purpose**: Simulates the ARM pipeline implementation.
- **Main Features**:
  - Generates a clock signal to synchronize tests.
  - Initializes the system by asserting and deasserting the reset signal.
  - Verifies correct memory write operations during simulation, displaying success or failure messages.

Top Module

- **Purpose**: Integrates the processor, instruction memory (imem), and data memory (dmem).
- **Main Features**:
  - Connects the processor (ARM pipeline) to instruction and data memories.
  - Interfaces the testbench signals with the ARM processor.

Instruction Memory (imem)

- **Purpose**: Stores and provides instructions to the processor.
- **Main Features**:
  - A memory initialized with a program (memfile.dat).
  - Outputs word-aligned instruction data based on the address.

Data Memory (dmem)

- **Purpose**: Stores data for processor operations.
- **Main Features**:
  - Word-aligned memory for read and write operations.
  - Allows data writes on the positive clock edge when enabled (we).

ARM Processor (arm)

- **Purpose**: Implements the 5-stage ARM pipeline processor.
- **Main Features**:
  - Handles instruction fetch, decode, execute, memory, and write-back stages.
  - Interfaces with the controller, datapath, and hazard modules for control signals, data processing, and hazard management.

Controller

- **Purpose**: Generates control signals for the pipeline based on the current instruction and ALU flags.
- **Main Features**:
    - Decodes instructions to determine operation type (e.g., data processing, memory, branch).
    - Computes control signals for pipeline stages and condition checks for branching.
    - Controls pipeline flush, stall, and execution based on hazards and branch conditions.

Condition Checker (condcheck)

- **Purpose**: Evaluates condition codes and updates flags for conditional execution.
- **Main Features**:
    - Determines if an instruction should be executed based on condition flags (e.g., zero, negative).
    - Updates processor flags conditionally based on ALU operations.

Datapath

- **Purpose**: Implements the data flow and operations of the ARM processor.
- **Main Features**:
    - Contains pipeline registers for all stages (fetch, decode, execute, memory, write-back).
    - Includes key components like the ALU, multiplexers, and registers.
    - Manages branching, forwarding, and ALU operations.
    - Interfaces with hazard management for stall and flush signals.

Hazard Unit

- **Purpose**: Manages hazards (data, structural, control) in the pipeline.
- **Main Features**:
    - Implements data forwarding for dependencies between instructions.
    - Detects and handles load-use hazards with pipeline stalls.
    - Controls flushing and stalling of pipeline stages to ensure correct operation.

Register File (regfile)

- **Purpose**: Implements a multi-ported register file for ARM general-purpose registers.
- **Main Features**:
    - Provides two combinational read ports and one synchronous write port.
    - Special handling for register 15 to output the PC+8 value.
    - Allows write operations on the falling clock edge.

ALU

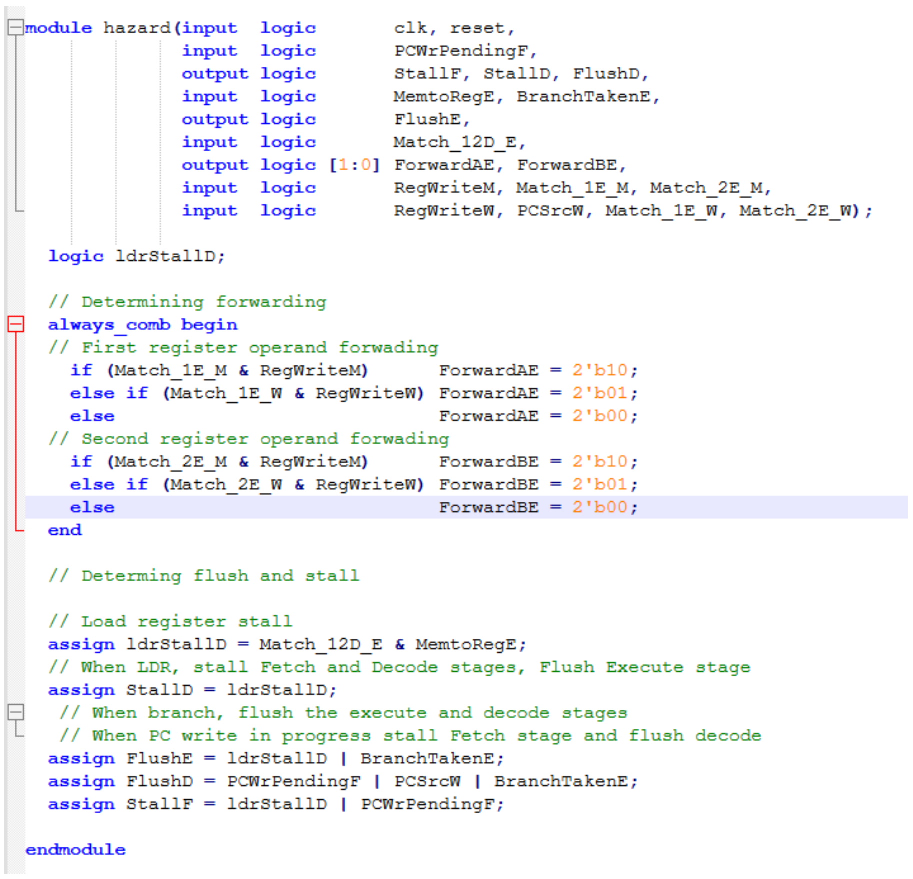
- **Purpose**: Performs arithmetic and logical operations.
- **Main Features**:
  - Executes operations such as addition, subtraction, AND, and OR based on control inputs.
  - Updates condition flags based on operation results.

Supporting Modules

- **Multiplexers (mux2, mux3)**:
  - Dynamically select input data for operations based on control signals.
- **Equality Comparator (eqcmp)**:
  - Checks equality between addresses/registers to detect hazards.
- **Adder**:
  - Calculates PC increments and branch targets.

Implementation and Demonstration:

The pipeline was implemented using systemVerilog with comments provided in the code. The name of signals was changed to correspond to the stage of the pipeline, and new signals and modules were integrated. The new hazard module is shown here.



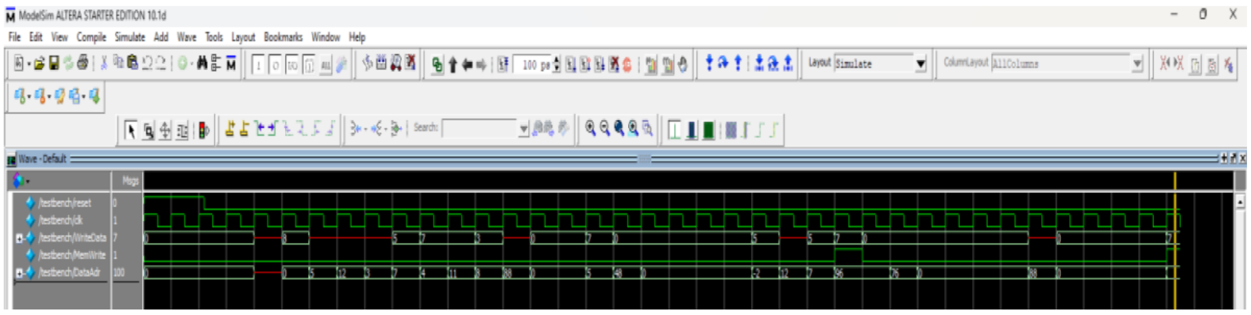
```
module hazard(input  logic       clk, reset,
              input  logic       PCWrPendingF,
              output logic       StallF, StallD, FlushD,
              input  logic       MemtoRegE, BranchTakenE,
              output logic       FlushE,
              input  logic       Match_12D_E,
              output logic [1:0] ForwardAE, ForwardBE,
              input  logic       RegWriteM, Match_1E_M, Match_2E_M,
              input  logic       RegWriteW, PCSrcW, Match_1E_W, Match_2E_W);

  logic ldrStallD;

  // Determining forwarding
  always_comb begin
  // First register operand forwading
    if (Match_1E_M & RegWriteM)      ForwardAE = 2'b10;
    else if (Match_1E_W & RegWriteW) ForwardAE = 2'b01;
    else                             ForwardAE = 2'b00;
  // Second register operand forwading
    if (Match_2E_M & RegWriteM)      ForwardBE = 2'b10;
    else if (Match_2E_W & RegWriteW) ForwardBE = 2'b01;
    else                             ForwardBE = 2'b00;
  end

  // Determing flush and stall

  // Load register stall
  assign ldrStallD = Match_12D_E & MemtoRegE;
  // When LDR, stall Fetch and Decode stages, Flush Execute stage
  assign StallD = ldrStallD;
   // When branch, flush the execute and decode stages
   // When PC write in progress stall Fetch stage and flush decode
  assign FlushE = ldrStallD | BranchTakenE;
  assign FlushD = PCWrPendingF | PCSrcW | BranchTakenE;
  assign StallF = ldrStallD | PCWrPendingF;

endmodule
```

The design was tested using the instruction provided in memfile.dat. This is the same datafile that was used in the single cycle ARM architecture in the lab and has all the standard instructions including branches:

ADDR	PROGRAM	; COMMENTS	BINARY MACHINE CODE	HEX CODE
00 MAIN	SUB R0, R15, R15	; R0 = 0	1110 000 0010 0 1111 0000 0000 0000 1111	E04F000F
04	ADD R2, R0, #5	; R2 = 5	1110 001 0100 0 0000 0010 0000 0000 0101	E2802005
08	ADD R3, R0, #12	; R3 = 12	1110 001 0100 0 0000 0011 0000 0000 1100	E280300C
0C	SUB R7, R3, #9	; R7 = 3	1110 001 0010 0 0011 0111 0000 0000 1001	E2437009
10	ORR R4, R7, R2	; R4 = 3 OR 5 = 7	1110 000 1100 0 0111 0100 0000 0000 0010	E1874002
14	AND R5, R3, R4	; R5 = 12 AND 7 = 4	1110 000 0000 0 0011 0101 0000 0000 0100	E0035004
18	ADD R5, R5, R4	; R5 = 4 + 7 = 11	1110 000 0100 0 0101 0101 0000 0000 0100	E0855004
1C	SUBS R8, R5, R7	; R8 = 11 - 3 = 8, set Flags	1110 000 0010 1 0101 1000 0000 0000 0111	E0558007
20	BEQ END	; shouldn't be taken	0000 1010 0000 0000 0000 0000 0000 1100	0A00000C
24	SUBS R8, R3, R4	; R8 = 12 - 7 = 5	1110 000 0010 1 0011 1000 0000 0000 0100	E0538004
28	BGE AROUND	; should be taken	1010 1010 0000 0000 0000 0000 0000 0000	AA000000
2C	ADD R5, R0, #0	; should be ski pped	1110 001 0100 0 0000 0101 0000 0000 0000	E2805000
30 AROUND	SUBS R8, R7, R2	; R8 = 3 - 5 = -2, set Flags	1110 000 0010 1 0111 1000 0000 0000 0010	E0578002
34	ADDLT R7, R5, #1	; R7 = 11 + 1 = 12	1011 001 0100 0 0101 0111 0000 0000 0001	B2857001
38	SUB R7, R7, R2	; R7 = 12 - 5 = 7	1110 000 0010 0 0111 0111 0000 0000 0010	E0477002
3C	STR R7, [R3, #84]	; mem[12+84] = 7	1110 010 1100 0 0011 0111 0000 0101 0100	E5837054
40	LDR R2, [R0, #96]	; R2 = mem[96] = 7	1110 010 1100 1 0000 0010 0000 0110 0000	E5902060
44	ADD R15, R15, R0	; PC = PC+8 (skips next)	1110 000 0100 0 1111 1111 0000 0000 0000	E08FF000
48	ADD R2, R0, #14	; shouldn't happen	1110 001 0100 0 0000 0010 0000 0000 0001	E280200E
4C	B END	; always taken	1110 1010 0000 0000 0000 0000 0000 0001	EA000001
50	ADD R2, R0, #13	; shouldn't happen	1110 001 0100 0 0000 0010 0000 0000 0001	E280200D
54	ADD R2, R0, #10	; shouldn't happen	1110 001 0100 0 0000 0010 0000 0000 0001	E280200A
58 END	STR R2, [R0, #100]	; mem[100] = 7	1110 010 1100 0 0000 0010 0000 0101 0100	E5802064

Figure 7.60 **Assembly and machine code for test program**

Now the same instructions are processed inside an ARM pipeline architecture. The testbench waveform correctly shows that the value of 100 is at address 7 which confirms the design. Additionally, the waveform shows that address 96 has the value of 7 as expected from the instructions.



Lessons Learned:

During this project, I learned about the challenges of adding new functionalities to an existing system while maintaining stability. It became clear how critical attention to detail is, especially when ensuring that signal names are consistently and correctly used across the code. Even a small mismatch can cause significant debugging challenges.

One of the most difficult aspects of this assignment was carefully integrating changes without introducing unintended side effects, particularly in a pipelined system. I gained a deeper appreciation for how pipelining improves throughput and performance but also introduces overhead and complexity in managing data hazards, timing, and dependencies between stages.

This experience reinforced the importance of modular design, proper documentation, and systematic testing to ensure scalability and reliability in complex systems. It also highlighted how foundational principles, like clear naming conventions and well-structured code, play a crucial role in building robust systems.

Future Steps:

One key step would be to implement more advanced optimization techniques to reduce the overhead associated with pipelining, such as better hazard detection and resolution mechanisms. For example, for branch instructions there is one less cycle penalty if the branch is asserted in the decode stage rather than execute stage. Exploring alternative architectural designs or adding support for parallelism could further enhance performance and scalability. Finally, it would be interesting to use an FPGA to demonstrate the design and use time analysis to assess the performance benefits.

Conclusions:

This project provided valuable insights into the complexities of system design and implementation, particularly when incorporating advanced concepts like pipelining and simulating the hardware using systemVerilog. Through the challenges of ensuring signal consistency, debugging, and balancing performance trade-offs, I gained a deeper understanding of how small changes can impact the broader system. Despite the difficulties, the experience reinforced the importance of careful planning, attention to detail, and iterative testing in achieving a functional and efficient design.

References

Harris, S., & Harris, D. (2015). *Digital Design and Computer Architecture*. Morgan Kaufmann