

Design, Optimization, and Formal Verification of Stall Generation Logic in a 5-Stage Pipelined RISC-V Processor Using Karnaugh Maps, ESPRESSO Algorithm, and Model Checking

Siddhant Shah

Roll No.: B23334

Department of Microelectronics & VLSI
IIT MANDI

Abstract

This report presents the systematic design, minimization, hardware implementation, and formal verification of stall generation logic for data, control, and structural hazards in a five-stage pipelined RISC-V processor. Minimization techniques including Karnaugh Maps and the ESPRESSO logic optimization algorithm are applied and compared. Two-level and multilevel circuit realizations are synthesized in Vivado and evaluated based on area, delay, and power metrics. Additionally, temporal logic specifications are developed and verified using the NuXmv model checker to ensure that the implemented hazard detection and resolution mechanisms correctly prevent instruction loss and maintain pipeline correctness under all possible hazard scenarios.

1 Introduction

Pipelining improves instruction throughput but introduces hazards that must be detected and resolved correctly to maintain program correctness. This work focuses on designing complete stall-generation logic for a standard 5-stage pipeline (IF, ID, EX, MEM, WB) and formally verifying its correctness.

The processor considered for logic minimization assumes:

- No data forwarding
- Single-cycle memory model
- Branch resolution in EX stage
- Unified memory for instruction and data (structural hazard possible)

For formal verification, a realistic pipelined RISC-V processor with data forwarding and separate instruction/data memories is analyzed to validate hazard handling mechanisms.

The remainder of this report describes:

1. Derivation of hazard-related truth tables
2. Minimization using K-Maps
3. PLA-based representation and ESPRESSO minimization
4. Implementation in two-level and multilevel logic
5. Synthesis and comparison of implementations
6. Formal verification using temporal logic and model checking

2 Data Hazard Stall Logic

Since forwarding is not used, RAW dependencies between ID-stage source registers and EX/MEM-stage destination registers must cause stalls.

2.1 Variables Used

- EX_RW: EX stage writes a register
- EX_MR: EX stage performs memory read (load)
- MEM_RW: MEM stage writes register
- D_EX, T_EX: ID.rs1/rs2 depends on EX.rd
- D_MEM, T_MEM: ID.rs1/rs2 depends on MEM.rd

2.2 Truth Table

Table 1: Data Hazard Stall Conditions

EX_RW	EX_MR	MEM_RW	D_EX	T_EX	D_MEM	T_MEM	STALL
1	0	0	1	0	0	0	1
1	0	0	0	1	0	0	1
0	1	0	1	0	0	0	1
0	1	0	0	1	0	0	1
0	0	1	0	0	1	0	1
0	0	1	0	0	0	1	1

2.3 K-map Minimization

The minimized Boolean equation obtained from K-map is:

$$STALL_{data} = (EX_{RW} + EX_{MR})(D_{EX} + T_{EX}) + MEM_{RW}(D_{MEM} + T_{MEM}) \quad (1)$$

2.4 ESPRESSO Minimization

The PLA file was processed using the ESPRESSO tool [1]. The output cubes generated were:

```
0010010 1
0010001 1
1-01000 1
-101000 1
1-00100 1
-100100 1
```

These cubes represent:

- 0010010: $\overline{EX_{RW}} \cdot \overline{EX_{MR}} \cdot MEM_{RW} \cdot D_{MEM} \cdot \overline{T_{MEM}}$
- 0010001: $\overline{EX_{RW}} \cdot \overline{EX_{MR}} \cdot MEM_{RW} \cdot \overline{D_{MEM}} \cdot T_{MEM}$
- 1-01000: $EX_{RW} \cdot D_{EX}$
- -101000: $EX_{MR} \cdot D_{EX}$
- 1-00100: $EX_{RW} \cdot T_{EX}$
- -100100: $EX_{MR} \cdot T_{EX}$

Factoring these cubes yields the same expression as K-map, confirming correctness.

2.5 Two-Level Implementation

Two-Level SOP Form:

$$\begin{aligned} STALL_{data} = & (EX_{RW} \cdot D_{EX}) + (EX_{MR} \cdot D_{EX}) \\ & + (EX_{RW} \cdot T_{EX}) + (EX_{MR} \cdot T_{EX}) \\ & + (MEM_{RW} \cdot D_{MEM}) + (MEM_{RW} \cdot T_{MEM}) \end{aligned} \quad (2)$$

2.6 Multilevel Implementation

Multilevel Factored Form:

$$\begin{aligned} A &= EX_{RW} + EX_{MR} \\ B &= D_{EX} + T_{EX} \\ C &= D_{MEM} + T_{MEM} \\ D &= A \cdot B \\ E &= MEM_{RW} \cdot C \\ STALL_{data} &= D + E \end{aligned} \quad (3)$$

2.7 Comparison

Table 2: Data Hazard Implementation Comparison

Metric	Two-Level	Multilevel
LUT Count	2	6
Delay (ns)	5.924	6.497
Power (W)	0.415	0.436
Logic Depth	2	3
No. of literals	12	7

3 Control Hazard Logic

Control hazards arise from branches and jumps whose outcomes are resolved only in the EX stage.

3.1 Variables Used

- ID_isBranch: ID stage instruction is branch
- ID_isJump: ID stage instruction is jump
- ID_hasDep: ID instruction has dependency on EX/MEM
- EX_isBranch: EX stage instruction is branch
- BranchTaken: Branch decision outcome
- EX_isJump: EX stage instruction is jump

3.2 Truth Table

Table 3: Control Hazard Stall and Flush Conditions

ID_Br	ID_Jp	Dep	EX_Br	BTaken	EX_Jp	STALL	FLUSH
1	0	1	0	X	0	1	0
0	1	1	0	X	0	1	0
1	0	0	0	X	0	0	0
0	1	0	0	X	0	0	0
0	0	X	1	1	0	0	1
0	0	X	0	X	1	0	1

3.3 K-map Minimization

The minimized expressions are:

$$STALL_{ctrl} = ID_{hasDep}(ID_{isBranch} + ID_{isJump}) \quad (4)$$

$$FLUSH = (EX_{isBranch} \cdot BranchTaken) + EX_{isJump} \quad (5)$$

3.4 ESPRESSO Minimization

ESPRESSO Output for STALL_ctrl:

```
1010-0 10
0110-0 10
```

ESPRESSO Output for FLUSH:

```
00-110 01
00-0-1 01
```

These cubes confirm the K-map minimization results.

3.5 Two-Level Implementation

$$STALL_{ctrl} = (ID_{isBranch} \cdot ID_{hasDep}) + (ID_{isJump} \cdot ID_{hasDep}) \quad (6)$$

$$FLUSH = (EX_{isBranch} \cdot BranchTaken) + EX_{isJump} \quad (7)$$

3.6 Multilevel Implementation

$$A = ID_{isBranch} + ID_{isJump}$$

$$STALL_{ctrl} = A \cdot ID_{hasDep} \quad (8)$$

$$FLUSH = (EX_{isBranch} \cdot BranchTaken) + EX_{isJump} \quad (9)$$

3.7 Comparison

Table 4: Control Hazard Implementation Comparison

Metric	Two-Level	Multilevel
LUT Count	2	3
Delay (ns)	5.351	5.924
Power (W)	0.68	0.681
Logic Depth	2	2
No. of literals	4	3

4 Structural Hazard Logic

Shared resources lead to structural hazards in three cases:

- **Memory:** Unified memory accessed by both IF and MEM stages
- **ALU:** Multi-cycle operations may block EX stage
- **Register File:** Simultaneous read/write conflicts

4.1 Variables Used

- IF_MemReq: IF stage requests memory
- MEM_MemReq: MEM stage requests memory
- ALU_Busy: ALU currently in use
- EX_UsesALU: EX stage wants to use ALU
- RF_WriteBusy: Register file write in progress
- ID_UsesRF: ID stage wants to read register file

4.2 Truth Table

Table 5: Structural Stall Conditions

IF_MR	MEM_MR	ALU_B	EX_ALU	RF_WB	ID_RF	STALL
1	1	0	0	0	0	1
0	0	1	1	0	0	1
0	0	0	0	1	1	1

4.3 K-map Minimization

$$\begin{aligned} STALL_{struct} = & (IF_{MemReq} \cdot MEM_{MemReq}) \\ & + (ALU_{Busy} \cdot EX_{UsesALU}) \\ & + (RF_{WriteBusy} \cdot ID_{UsesRF}) \end{aligned} \quad (10)$$

4.4 ESPRESSO Minimization

ESPRESSO Output:

```
110000 1
001100 1
000011 1
```

These three cubes directly represent the three conflict conditions, matching the K-map result exactly.

4.5 Two-Level Implementation

$$\begin{aligned} STALL_{struct} = & (IF_{MemReq} \cdot MEM_{MemReq}) \\ & + (ALU_{Busy} \cdot EX_{UsesALU}) \\ & + (RF_{WriteBusy} \cdot ID_{UsesRF}) \end{aligned} \quad (11)$$

4.6 Multilevel Implementation

$$\begin{aligned}
mem_conflict &= IF_{MemReq} \cdot MEM_{MemReq} \\
alu_conflict &= ALU_{Busy} \cdot EX_{UsesALU} \\
A &= mem_conflict \cdot alu_conflict \\
rf_conflict &= RF_{WriteBusy} \cdot ID_{UsesRF} \\
STALL_{struct} &= A + rf_conflict
\end{aligned} \tag{12}$$

4.7 Comparison

Table 6: Structural Hazard Implementation Comparison

Metric	Two-Level	Multilevel
LUT Count	2	5
Delay (ns)	5.351	6.497
Power (W)	0.473	0.405
Logic Depth	2	3
No. of literals	6	6

5 Formal Verification of Pipeline Hazard Logic

The previous sections focused on deriving and minimizing stall-generation logic using Karnaugh Maps and the ESPRESSO algorithm under an abstract hazard model. To validate that the implemented control logic behaves correctly when embedded in a realistic processor, we also performed formal verification on a concrete 5-stage pipelined RISC-V design using temporal logic specifications and model checking.

In this implementation, we consider a standard five-stage pipeline (IF, ID, EX, MEM, WB) with the following assumptions:

- Data hazards are primarily resolved using **data forwarding** in the Execute stage, with additional load-use stall logic when forwarding is insufficient.
- Instruction and data memories are implemented as **separate** blocks, so memory access does not introduce structural hazards.
- Branch and jump outcomes are resolved in the EX stage, and younger instructions are flushed when a control transfer is taken.

5.1 Pipelined Processor Architecture for Verification

The RTL used for verification instantiates a datapath, control unit, and hazard unit that together implement a 32-bit RISC-V pipeline with forwarding and hazard resolution. The hazard unit generates stall and flush signals (`stall_f`, `stall_d`, `flush_d`, `flush_e`) and forwarding controls (`forward_ae`, `forward_be`) based on register dependencies and branch decisions.

Figure 1 conceptually illustrates the 5-stage pipelined RISC-V processor used for verification, including data forwarding paths and hazard unit connections.

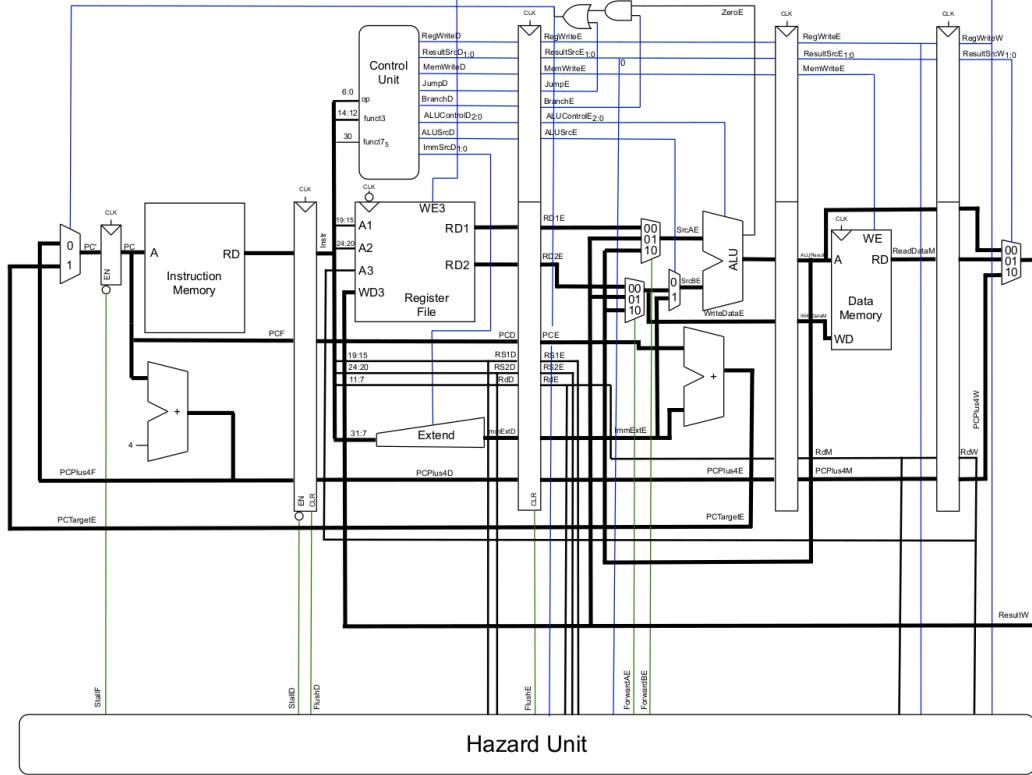


Figure 1: Five-stage pipelined RISC-V processor with data forwarding and separate instruction/data memories used for formal verification.

5.2 Simulation-Based Validation in Vivado

Before applying formal methods, the complete RTL (datapath, control unit, hazard unit, instruction and data memories) was synthesized and simulated in Vivado using representative test programs. Waveform inspection confirmed that:

- Load-use hazards between the EX-stage load destination and ID-stage source registers correctly trigger a one-cycle stall and bubble insertion in the EX stage.
- Branch and jump instructions resolved in the EX stage assert `pcsrc_e`, causing the Decode and Execute stages to be flushed while the Fetch stage redirects to the branch target.
- Forwarding paths prevent unnecessary stalls when ALU results can be bypassed directly from later pipeline stages.

These simulations provide evidence that the pipeline operates correctly for the tested programs, but they cannot exhaustively cover all hazard scenarios.

5.3 Temporal Logic Specifications for Hazard Behavior

To obtain stronger guarantees, we expressed key hazard requirements as temporal logic properties. Informally, the specifications require that:

- Stalls propagate correctly backward through the pipeline: whenever the Fetch stage is stalled, the Decode stage is also stalled, and both stall conditions are driven by the same load-use hazard.

- Instructions do not bypass stalled stages: if Decode is stalled, the instruction and program counter in the FD pipeline register remain stable; similarly, the Fetch program counter does not advance during a Fetch stall.
- Pipeline flushes work correctly: when a branch or jump is taken in the EX stage, both the Decode and Execute stages are flushed so that no incorrectly speculated instructions commit.
- Load-use hazards are always handled: when EX contains a load whose destination register matches a source register in Decode (and the destination is not $x0$), the processor inserts a bubble and asserts stalls as required.
- Stall scenarios do not cause instruction loss or incorrect execution: neither stalled nor flushed stages allow an instruction to “slip through” and execute with incorrect operands.

These informal requirements were first encoded as SystemVerilog assertions on the top-level RTL and then translated into temporal logic formulas suitable for model checking.

5.4 NuXmv Hazard Model and CTL Specifications

To make exhaustive verification tractable, we constructed an abstract model of the hazard, stall, and flush logic in the NuXmv input language. The model captures only the control aspects relevant to hazards, while abstracting away full 32-bit datapath values:

- Register indices (`rs1_d`, `rs2_d`, `rd_e`, `rd_m`, `rd_w`) are drawn from a small finite domain (e.g., 0–3), where 0 represents $x0$ and nonzero values represent arbitrary registers.
- The Fetch and Decode program counters (`pc_f`, `pc_d`) and the abstract Decode instruction (`instr_d`) are modeled as bounded integers sufficient to distinguish stability and change.
- The hazard unit outputs (`stall_f`, `stall_d`, `flush_d`, `flush_e`) are defined using the same Boolean equations as in the RTL hazard unit: a load-use hazard raises `stall_f` and `stall_d`, and branches/jumps raise `flush_d` and `flush_e`.
- The FD pipeline register is modeled to hold its contents when stalled, clear to a NOP when flushed, and otherwise accept new values from Fetch.

On this model, we formulated Computation Tree Logic (CTL) properties that directly correspond to the informal requirements:

- **Stall relationship:** $\text{AG}(\text{stall}_f \leftrightarrow \text{stall}_d)$ ensures that Fetch and Decode stalls always agree and are driven by the same load-use condition.
- **PC freeze under stall:** $\text{AG}(\text{stall}_f \rightarrow \text{AX } \text{pc}_f = \text{pc}_f_{\text{prev}})$ guarantees that the Fetch program counter does not advance while stalled.
- **FD register stability:** $\text{AG}(\text{stall}_d \rightarrow \text{AX}(\text{instr}_d = \text{instr}_d_{\text{prev}} \& \text{pc}_d = \text{pc}_d_{\text{prev}}))$ ensures that the instruction and PC in Decode do not change during a Decode stall.

- **Flush correctness:** $\text{AG}(\text{pcsrc_e} \rightarrow (\text{flush_d} \& \text{flush_e}))$ states that when a branch or jump is taken, both Decode and Execute stages are flushed.
- **Load-use hazard handling:** whenever EX contains a load whose destination matches a Decode source and is not $x0$, a CTL property requires that Fetch and Decode are stalled and the Execute stage is flushed.
- **Stall/flush interaction:** an additional property ensures that Decode is not simultaneously stalled and flushed unless a branch is taken, preventing contradictory control signals.

Invariants were also included to assert that the abstract hazard equations in the model exactly match the intended RTL equations for stall and flush signals.

5.5 Model Checking Results and Interpretation

The NuXmv model checker was run on the abstract hazard model with all CTL specifications and invariants. The tool proved all properties to be true, showing that under all possible combinations of register dependencies and branch outcomes:

- Stalls propagate correctly backward through the pipeline, and the Fetch and Decode stages never disagree on whether a load-use hazard is present.
- The Fetch program counter and FD pipeline register contents remain stable during stalls, so instructions do not bypass stalled stages.
- Pipeline flushes triggered by branches and jumps correctly clear younger instructions in Decode and Execute, preventing incorrect execution along mispredicted paths.
- Load-use hazards are always resolved by stalling and bubble insertion, so no instruction reads an operand before it has been written.
- Stall and flush signals are mutually consistent, avoiding control patterns that could cause instruction loss or duplication.

```

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\liddman> cd "C:\Users\liddman\Desktop\Downloads\nuXmv-2.1.0-win64\nuXmv-2.1.0-win64"
PS C:\Users\liddman\Desktop\Downloads\nuXmv-2.1.0-win64> nuXmv -go check_ctlspec
*** This is nuXmv 2.1.0 (compiled on Thu Nov 28 16:15:03 2019)
*** Copyright (c) 2018-2024, Fondazione Bruno Kessler
*** Please report bugs to <nuXmv-users@fbk.eu>
*** or email to <nuXmv@fbk.eu>
*** Please report bugs at Please report bugs to <nuXmv-users@fbk.eu>
*** or email to <nuXmv@fbk.eu>
*** Alternatively write to <nuXmv@fbk.eu>
*** Please report bugs to <nuXmv@fbk.eu>
*** or email to <nuXmv-users@fbk.eu>
*** Alternatively write to <nuXmv@fbk.eu>

*** This version of nuXmv is linked to the NuSMV 2.7.0.
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@fbk.eu>
*** Copyright (c) 2018-2024, Fondazione Bruno Kessler
*** This version of nuXmv is linked to the CUDD Library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado
*** This version of nuXmv is linked to the MiniSat SAT solver.
*** See <http://minisat.se>
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson
*** This version of nuXmv is linked to the MathSAT
*** Copyright (c) 2009-2019 by Fondazione Bruno Kessler
*** See <http://mathsat.fbk.eu>
*** See http://mathsat.fbk.eu

nuXmv> -go
nuXmv> check_ctlspec
-- specification AG ((stall_f == stall_d) & true) is true
-- specification AG ((stall_f == true) & (flush_e == true)) is true
-- specification AG (((resultrc_e0 & (rs1_d == rd_e) | rs2_d == rd_e) & rd_e != 0) & (stall_f & stall_d) & flush_e)) is true
-- specification AG ((stall_d == 0 & pc_src_e == 0) & (flush_e == 1)) is true
-- specification AG ((stall_d == 1 & pc_src_e == 1) & (flush_e == 0)) is true
-- specification AG (stall_d == AX (instr_d == instr_d_prev & pc_d == pc_d_prev)) is true
nuXmv> check_invar
-- invariant (stall_d <-> pcsrc_e) is true
-- invariant (stall_f <-> (resultrc_e0 & (rs1_d == rd_e | rs2_d == rd_e))) is true
-- invariant (stall_d <-> (resultrc_e0 & (rs1_d == rd_e | rs2_d == rd_e))) is true
-- invariant (flush_e <-> ((resultrc_e0 & (rs1_d == rd_e | rs2_d == rd_e)) | pcsrc_e)) is true
nuXmv>

```

Figure 2: NuXmv model checking output showing CTL specifications and invariants evaluated on the hazard model.

Together with the Vivado simulations, these results provide strong evidence that the implemented hazard unit and pipeline control logic correctly handle all hazard scenarios without instruction loss or incorrect execution.

6 Conclusion

This comprehensive study demonstrates several key findings across design, optimization, and verification:

1. **Logic Minimization:** K-map and ESPRESSO produce identical minimized equations for all hazard stall conditions, validating both manual and algorithmic approaches to Boolean minimization.
2. **Implementation Trade-offs:** Two-level logic consistently yields faster hardware due to reduced logic depth (2 levels vs 3-4 levels), while multilevel logic can reduce literal count and occasionally power consumption for complex expressions. The choice depends on whether delay or area optimization is the primary design constraint.
3. **Synthesis Results:** Vivado synthesis of both implementations confirms that ESPRESSO-derived expressions lead to efficient hardware realizations with measurable differences in LUT count, delay, and power consumption across different hazard types.
4. **Formal Verification:** Temporal logic specifications encoded in CTL and verified using NuXmv model checking prove that the hazard detection and resolution logic correctly handles all possible hazard scenarios. The verification confirms that stall and flush signals propagate correctly, pipeline registers maintain stability during stalls, and no instruction loss or incorrect execution occurs under any combination of data and control hazards.
5. **Complete Design Methodology:** The combination of theory (K-maps), algorithmic minimization (ESPRESSO), practical synthesis (Vivado), and formal verification (NuXmv with CTL) provides a complete end-to-end framework for designing, optimizing, and validating stall-generation hardware in pipelined processors.

This work demonstrates that rigorous application of both classical digital design techniques and modern formal methods is essential for building correct and efficient pipelined processor control logic. The formal verification step, in particular, provides mathematical guarantees beyond what simulation alone can achieve, ensuring that the hazard logic behaves correctly under all possible execution scenarios.

References

- [1] ESPRESSO Logic Minimizer (WebAssembly Implementation), <https://nudelerde.github.io/Espresso-Wasm-Web/index.html>
- [2] David M. Harris and Sarah L. Harris, *Digital Design and Computer Architecture, RISC-V Edition*. Morgan Kaufmann, 2021.
- [3] NuXmv Model Checker, <https://nuxmv.fbk.eu/download.html>