Unified RTL Pipeline Controller with Formal Verification

Vulli Sharanya

November 2025

1 Introduction

Pipelined processors require precise coordination between stages to maintain performance and correctness. As instruction-level parallelism increases, the control logic becomes more complex, making hazard handling, timing alignment, and verification difficult in traditional distributed-control architectures.

This project develops a unified pipeline controller that manages all stage-level control signals centrally. High-Level Synthesis (HLS) principles are applied for systematic scheduling and binding of control operations. A complete Control Dependency Graph (CDG) is constructed using adjacency lists and adjacency matrices, enabling topological ordering, cycle detection, and multi-level logic optimization.

To guarantee correctness, the controller is verified using CTL/LTL temporal logic properties covering hazards, flushes, forwarding, stalls, and instruction atomicity. Model checking provides exhaustive proof of correctness, resulting in a compact, optimized, and formally verified pipeline controller suitable for RISC-style designs.

List of Control Signals

The following control signals are used in the pipeline control unit:

- EX:ALUOp
- EX:ALUSrc
- EX:Branch
- EX:BranchTaken
- EX:BranchZero
- EX:FwdA
- EX:FwdB
- EX:Jump
- EX:TargetAddrReady
- ID:Bubble
- ID:Stall
- ID:ID_EX_Flush
- ID:ImmSrc
- ID:RegDst
- ID:RegWrite
- IF:Flush_IF_ID
- IF:IF_ID_Write
- IF:PCWrite
- IF:PC_src
- MEM:FwdC
- MEM:MemRead
- MEM:MemWrite
- WB:MemToReg

2 Control Dependency Graph (CDG)

A Control Dependency Graph (CDG) formally represents how pipeline control signals influence one another across stages. Hazards, stalls, forwarding, and branch decisions create inter-stage dependencies that must be explicitly captured. The CDG provides structure for ordering constraints, optimization, and formal verification.

A well-defined CDG enables:

- Identification of correct control ordering and timing.
- Detection of cycles requiring rescheduling or arbitration.
- Use of topological sorting for deterministic control generation.
- Multilevel logic optimization across control paths.
- Clearer formal verification based on explicit dependencies.

2.1 Control Dependency Graph: Adjacency List

The CDG is first represented using an adjacency list. Each control signal lists all signals that must occur *after* it.

```
adj_list = {
    "IF:PC_src": [
        "ID:RegWrite", "ID:RegDst", "ID:ImmSrc",
        "ID:ID_EX_Flush", "ID:Bubble",
        "EX:ALUOp", "EX:ALUSrc", "EX:Branch", "EX:BranchZero",
        "MEM:MemRead", "MEM:MemWrite", "WB:MemToReg"
    ],

    "IF:PCWrite": ["IF:IF_ID_Write", "IF:Flush_IF_ID"],

    "IF:IF_ID_Write": [
        "ID:RegWrite", "ID:RegDst", "ID:ImmSrc",
        "ID:Bubble", "ID:ID_EX_Flush", "IF:Flush_IF_ID"
    ],

    "IF:Flush_IF_ID": [
        "ID:RegWrite", "ID:RegDst", "ID:ImmSrc",
        "ID:Bubble", "ID:ID_EX_Flush"
```

```
    ],

    "ID:RegWrite": ["EX:FwdA", "EX:FwdB", "MEM:FwdC"],
    "ID:RegDst": ["EX:ALUSrc", "EX:ALUOp"],
    "ID:ImmSrc": ["EX:ALUSrc"],

    "ID:Bubble": [
        "IF:PCWrite", "IF:IF_ID_Write", "ID:ID_EX_Flush",
        "EX:ALUSrc", "EX:ALUOp"
    ],"ID:Stall": [
        "IF:PCWrite",         # freeze PC update
        "IF:IF_ID_Write",     # freeze IF/ID register
        "ID:Bubble"           # insert bubble in ID stage
    ],

    "ID:ID_EX_Flush": [
        "EX:ALUSrc", "EX:ALUOp", "EX:Branch",
        "EX:BranchZero", "EX:BranchTaken"
    ],

    "EX:FwdA": ["EX:ALUSrc", "EX:ALUOp"],
    "EX:FwdB": ["EX:ALUSrc", "EX:ALUOp"],

    "EX:ALUSrc": ["EX:ALUOp", "EX:BranchZero"],
    "EX:ALUOp": ["EX:BranchZero"],

    "EX:Branch": ["EX:BranchTaken"],
    "EX:BranchZero": ["EX:BranchTaken"],

    "EX:BranchTaken": [
        "IF:PC_src", "IF:PCWrite",
        "IF:Flush_IF_ID", "ID:ID_EX_Flush"
    ],

    "EX:TargetAddrReady": ["IF:PC_src", "IF:PCWrite"],
    "EX:Jump": ["IF:PC_src", "IF:Flush_IF_ID", "ID:ID_EX_Flush"],

    "MEM:MemRead": [
        "EX:FwdA", "EX:FwdB", "MEM:FwdC",
        "ID:Bubble", "IF:PCWrite", "IF:IF_ID_Write"
    ],

    "MEM:MemWrite": [],
    "MEM:FwdC": ["EX:FwdA", "EX:FwdB"],
    "WB:MemToReg": ["ID:RegWrite", "EX:FwdA", "EX:FwdB"]
}
```

Listing 1: Adjacency List for Pipeline CDG

2.1.1 Usage of the Adjacency List

- Converted into an adjacency matrix for mathematical and formal analysis.
- Used for cycle detection to reveal unschedulable dependencies.
- Gives a topological order for safe control signal generation.
- Drives NetworkX visualization of the CDG.
- Supports CTL/LTL property generation for model checking.

2.2 Python Code for CDG Generation

```python
# Build adjacency matrix from adjacency list
import numpy as np
import pandas as pd

adj = adj_list
nodes = sorted(set(adj.keys()) | {v for values in adj.values() for v in values})
idx = {node: i for i, node in enumerate(nodes)}

N = len(nodes)
matrix = np.zeros((N, N), dtype=int)

for u in adj:
    for v in adj[u]:
        matrix[idx[u], idx[v]] = 1

pd.DataFrame(matrix, index=nodes, columns=nodes).to_csv("CDG_Adjacency_Matrix.csv")

with open("CDG_Adjacency_Matrix.txt", "w") as f:
    f.write(pd.DataFrame(matrix, index=nodes, columns=nodes).to_string())

print("Saved CDG_Adjacency_Matrix.csv and CDG_Adjacency_Matrix.txt")

# Draw layered CDG
import networkx as nx
import matplotlib.pyplot as plt
import textwrap

G = nx.DiGraph()
for u in adj_list:
    for v in adj_list[u]:
        G.add_edge(u, v)

stage_x = {"IF": 0, "ID": 1, "EX": 2, "MEM": 3, "WB": 4}
y_count = {s: 0 for s in stage_x}
pos, node_colors = {}, {}

colors = {
    "IF": "#F4B183", "ID": "#A9D08E", "EX": "#FFD966",
    "MEM": "#9BC2E6", "WB": "#CDA0D9"
}

for node in sorted(G.nodes()):
    stage = node.split(":")[0]
    pos[node] = (stage_x[stage], -2 * y_count[stage])
    y_count[stage] += 1
    node_colors[node] = colors[stage]

labels = {n: "\n".join(textwrap.wrap(n, 18)) for n in G.nodes()}

plt.figure(figsize=(26, 32))
nx.draw_networkx_nodes(G, pos, node_color=list(node_colors.values()),
                       edgecolors="black", node_size=4200)
nx.draw_networkx_edges(G, pos, arrows=True, arrowsize=16,
                       connectionstyle="arc3,rad=0.15", width=1.1)
nx.draw_networkx_labels(G, pos, labels, font_size=9)

for stage, x in stage_x.items():
    plt.text(x, 4, stage, fontsize=22, fontweight="bold", ha="center")

```

```
plt.title("Control Dependency Graph (Layered)", fontsize=28)
plt.axis("off")
plt.savefig("CDG_Clean_Layered.png", dpi=330)
plt.show()

print("Saved CDG_Clean_Layered.png")
```

Listing 2: Python script to generate adjacency matrix and layered CDG

2.3 Adjacency matrix:

3 Scheduling of Control Signals

Scheduling is the process of assigning each operation or control signal in a design to a specific clock cycle or control step. In complex processors or multi-stage pipelines, many control signals have data and control dependencies with each other. Executing them without violating these dependencies is essential to guarantee correctness.
The main reasons for performing scheduling are:

- **Avoiding Hazards:** Ensures that dependent operations execute only after their predecessors complete.
- **Reducing Critical Path:** By distributing operations across cycles, we reduce logic depth and increase clock frequency.
- **Resource Optimization:** Scheduling algorithms like ASAP, ALAP, and FDS help minimize resource usage such as decoders, comparators, logic units, etc.
- **Ensuring Pipeline Correctness:** Control, forwarding, flushing, hazard detection, and stalling all depend on correctly scheduled control signals.

3.1 Algorithms

3.1.1 ASAP Algorithm (Algorithm 1)

Algorithm 1 ASAP Scheduling Algorithm

Require: Operations O, Maximum number of control steps M
Ensure: Control step for each operation, Scheduling status

1: **for** each operation $o_i \in O$ **do**
2: **if** o_i has no immediate predecessors (i.e., computation from inputs) **then**
3: $control\_step(o_i) \leftarrow 1$
4: **else**
5: $control\_step(o_i) \leftarrow \max(control\_step(o_j)) + 1$
6: where o_j is an immediate predecessor of o_i
7: **end if**
8: **end for**
9: **Success Check:** If $control\_step(o_i) \leq M$ for all $o_i \in O$, the scheduling is successful.

3.1.2 ALAP Algorithm (Algorithm 2)

Algorithm 2 ALAP Scheduling Algorithm

Require: Operations O, Maximum number of control steps M
Ensure: Control step for each operation, Scheduling status

1: **for** each operation $o_i \in O$ **do**
2: **if** o_i has no immediate successors (i.e., computation generates outputs) **then**
3: $control\_step(o_i) \leftarrow M$
4: **else**
5: $control\_step(o_i) \leftarrow control\_step(o_j) - 1$
6: where o_j is an immediate successor of o_i
7: **end if**
8: **end for**
9: **Success Check:** If all operations are scheduled within control step 1, the scheduling is successful.

3.1.3 FDS Key Concepts and Notations

- i_{ASAP}: Control step assigned to operation o_i by the ASAP algorithm.
- i_{ALAP}: Control step assigned to operation o_i by the ALAP algorithm.
- i_{INTERVAL}: Flexible range of control steps for operation o_i, defined as:

$$[i_{\text{ASAP}}, i_{\text{ALAP}}]$$

- i_{RANGE}: Size of the flexible interval:

$$i_{\text{RANGE}} = i_{\text{ALAP}} - i_{\text{ASAP}} + 1$$

- $P_{i,j}$: Probability of scheduling operation o_i in control step j.

$$P_{i,j} = \begin{cases} \frac{1}{i_{\text{RANGE}}}, & \text{if } j \in i_{\text{INTERVAL}} \\ 0, & \text{otherwise} \end{cases}$$

- $C_{k,j}$: Number of operators of type k required in control step j. It is computed as:

$$C_{k,j} = \sum_{\text{operations of type } k} P_{i,j}$$

ASAP (As Soon As Possible) and ALAP (As Late As Possible) scheduling give earliest and latest valid execution cycles. FDS (Force Directed Scheduling) balances operations across time by minimizing cost forces, producing an optimized and resource-efficient distribution.

3.2 Python Code for FDS-Based Pipeline Control Signal Scheduler

Topological sorting is done in scheduling

```
# CORRECTED PIPELINE CONTROL SIGNAL SCHEDULER WITH FDS ALGORITHM
#    ================================================================

import networkx as nx
import matplotlib.pyplot as plt
import pandas as pd

edges = [
```

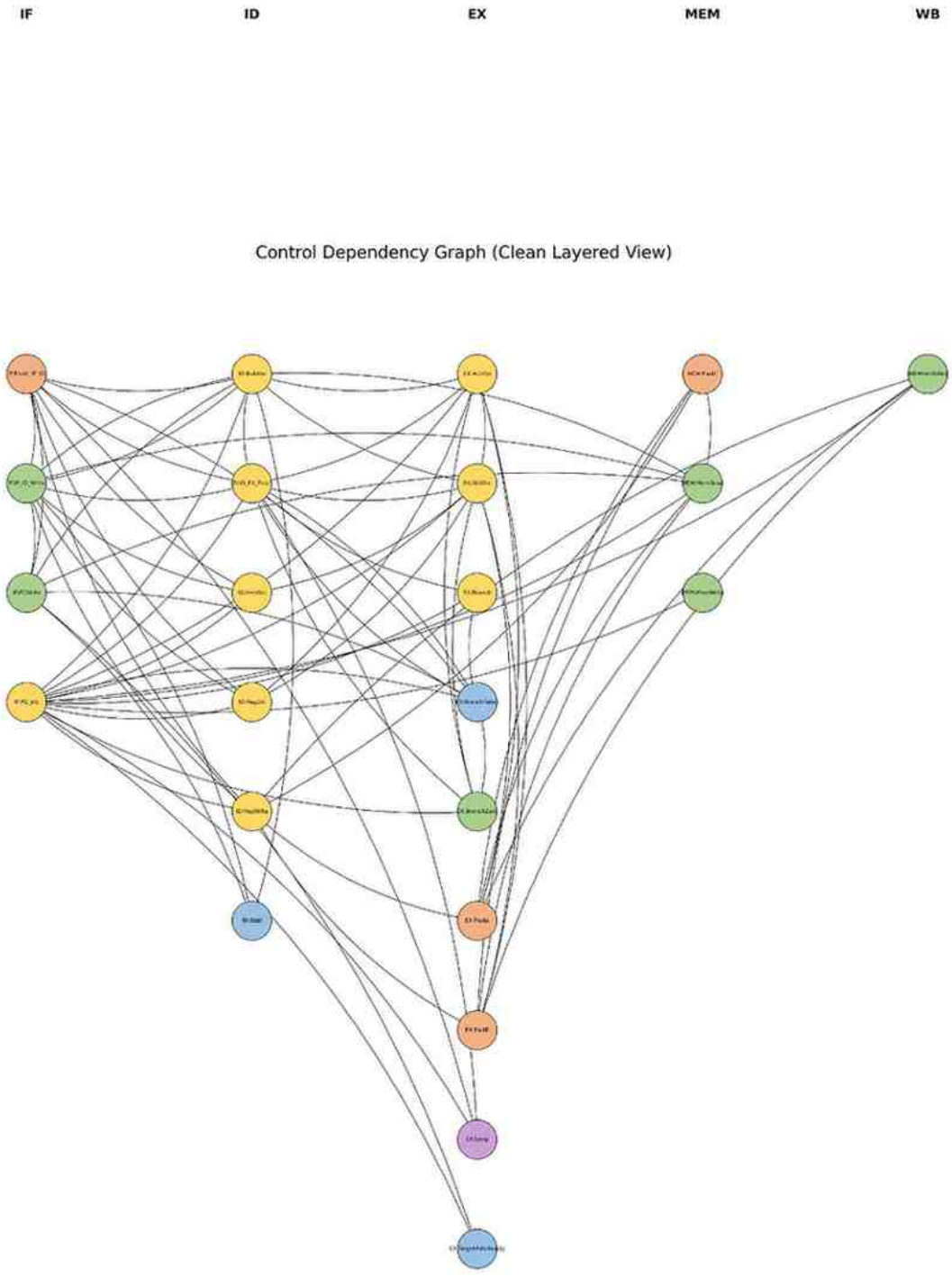


Figure 1: Layered Control Dependency Graph (CDG)

Table 1: CDG Adjacency Matrix

No.	Signal	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
1	EX:ALUOp	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	EX:ALUSrc	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	EX:Branch	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	EX:BranchTaken	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	0	0	0
5	EX:BranchZero	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	EX:FwdA	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	EX:FwdB	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	EX:Jump	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0
9	EX:TargetAddrReady	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
10	ID:Bubble	1	1	0	0	0	0	0	0	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0
11	ID:ID_EX_Flush	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	ID:ImmSrc	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	ID:RegDst	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	ID:RegWrite	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
15	ID:Stall	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0	0	0	0	0
16	IF:Flush_IF_ID	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0
17	IF:IF_ID_Write	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0
18	IF:PCWrite	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
19	IF:PC_src	1	1	1	0	1	0	0	0	0	1	1	1	1	1	0	0	0	0	0	1	1	1	0
20	MEM:FwdC	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21	MEM:MemRead	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	1	1	0	1	0	0	0
22	MEM:MemWrite	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
23	WB:MemToReg	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

```
    # IF Stage signals
    ("Instruction_Decode", "RegWrite"),
    ("Instruction_Decode", "RegDst"),
    ("Instruction_Decode", "ImmSrc"),
    ("Instruction_Decode", "ALUOp"),
    ("Instruction_Decode", "ALUSrc"),
    ("Instruction_Decode", "Branch"),
    ("Instruction_Decode", "MemRead"),
    ("Instruction_Decode", "MemWrite"),
    ("Instruction_Decode", "MemToReg"),
    ("Instruction_Decode", "Jump"),

    # PCWrite and IF_ID_Write dependencies
    ("Bubble", "PCWrite"),
    ("Bubble", "IF_ID_Write"),
    ("PCWrite", "Flush_IF_ID"),

    # Stall signal dependencies (hazard detection)
    ("MemRead", "Stall"),      # Load-use stall
    ("Stall", "PCWrite"),      # Stall freezes PC
    ("Stall", "IF_ID_Write"), # Stall freezes IF/ID
    ("Stall", "Bubble"),       # Stall also inserts bubble (if needed)

    # ID Stage dependencies
    ("RegWrite", "FwdA"),
    ("RegWrite", "FwdB"),
    ("RegWrite", "FwdC"),

    ("RegDst", "ALUSrc"),
    ("ImmSrc", "ALUSrc"),

    # Bubble generation (hazard detection)
    ("MemRead", "Bubble"),  # Load-use hazard detection
    ("Bubble", "ID_EX_Flush"),

    # EX Stage dependencies
    ("FwdA", "ALUSrc"),
    ("FwdB", "ALUSrc"),
    ("ALUSrc", "ALUOp"),
    ("ALUOp", "BranchZero"),

    ("Branch", "BranchZero"),
    ("BranchZero", "BranchTaken"),

    # Branch feedback to IF
    ("BranchTaken", "PC_src"),
    ("BranchTaken", "Flush_IF_ID"),
    ("BranchTaken", "ID_EX_Flush"),

    # Jump and target address dependencies
    ("Jump", "PC_src"),
    ("Jump", "Flush_IF_ID"),
    ("ALUOp", "TargetAddrReady"),  # Target address calculated in ALU
    ("TargetAddrReady", "PC_src"),  # Target ready before PC update

    # MEM Stage dependencies
    ("MemRead", "FwdC"),

    # WB Stage dependencies
    ("MemToReg", "RegWrite"),
]

#
    ----------------------------------------------
```

```python
# 2) Build DAG
#
# ------------------------------------------------

G = nx.DiGraph()
G.add_edges_from(edges)

print(f"Total nodes (control signals): {G.number_of_nodes()}")
print(f"Total edges (dependencies): {G.number_of_edges()}")

# Check for cycles (should be acyclic for scheduling)
if not nx.is_directed_acyclic_graph(G):
    print("ERROR: Graph has cycles!")
    cycles = list(nx.simple_cycles(G))
    print(f"Cycles found: {cycles}")
else:
    print("    Graph is acyclic (valid for scheduling)")

#
# ------------------------------------------------

# 3) ASAP Scheduling (As-Soon-As-Possible)
#
# ------------------------------------------------

ASAP = {}
for node in nx.topological_sort(G):
    preds = list(G.predecessors(node))
    if not preds:
        ASAP[node] = 0  # Root nodes start at step 0
    else:
        ASAP[node] = max(ASAP[p] + 1 for p in preds)

#
# ------------------------------------------------

# 4) ALAP Scheduling (As-Late-As-Possible)
#
# ------------------------------------------------

max_asap = max(ASAP.values())
print(f"\nCritical path length (max ASAP): {max_asap} steps")

ALAP = {}
# Process in reverse topological order
for node in reversed(list(nx.topological_sort(G))):
    succs = list(G.successors(node))
    if not succs:
        ALAP[node] = max_asap  # Leaf nodes at latest time
    else:
        ALAP[node] = min(ALAP[s] - 1 for s in succs)

#
# ------------------------------------------------

# 5) Slack and Mobility
#
# ------------------------------------------------

SLACK = {n: ALAP[n] - ASAP[n] for n in G.nodes()}
MOBILITY = {n: ALAP[n] - ASAP[n] + 1 for n in G.nodes()}

print("\nCritical path signals (slack = 0):")
critical_signals = [n for n in G.nodes() if SLACK[n] == 0]
for sig in critical_signals:
    print(f"  {sig}: ASAP={ASAP[sig]}, ALAP={ALAP[sig]}")

#
# ------------------------------------------------

# 6) FDS (Force-Directed Scheduling)
#
# ------------------------------------------------

# Calculate probability distribution for each operation
Prob = {}
for n in G.nodes():
    Prob[n] = {}
    for t in range(max_asap + 1):
        if ASAP[n] <= t <= ALAP[n]:
            Prob[n][t] = 1.0 / MOBILITY[n]
        else:
            Prob[n][t] = 0.0

# Calculate distribution cost (forces) for each time step
Cost = {}
for t in range(max_asap + 1):
    Cost[t] = sum(Prob[n][t] for n in G.nodes())

print("\nDistribution cost per step:")
for t in sorted(Cost.keys()):
    print(f"  Step {t}: {Cost[t]:.2f} operations")

# FDS: Choose time step with minimum cost within valid interval
FDS_schedule = {}
for n in G.nodes():
    valid_steps = range(ASAP[n], ALAP[n] + 1)
    best_step = min(valid_steps, key=lambda t: Cost[t])
    FDS_schedule[n] = best_step

#
# ------------------------------------------------

# 7) Resource Type Classification
#
# ------------------------------------------------

resource_type = {
    "Instruction_Decode": "Decoder",
    "RegWrite": "Decoder",
    "RegDst": "Decoder",
    "ImmSrc": "Decoder",
    "ALUOp": "Decoder",
    "ALUSrc": "Decoder",
    "Branch": "Decoder",
    "MemRead": "Decoder",
    "MemWrite": "Decoder",
    "MemToReg": "Decoder",
    "Jump": "Decoder",
    "FwdA": "Comparator",
    "FwdB": "Comparator",
    "FwdC": "Comparator",
    "Bubble": "Logic",
    "Stall": "Logic",
    "ID_EX_Flush": "Logic",
    "Flush_IF_ID": "Logic",
    "PCWrite": "Logic",
    "IF_ID_Write": "Logic",
    "BranchZero": "Logic",
    "BranchTaken": "Logic",
    "PC_src": "Mux",
}
```

```
#
    ----------------------------------------

# 8) Stage Assignment (for visualization)
#
    ----------------------------------------

stage_assignment = {
    "Instruction_Decode": "IF",
    "PCWrite": "IF",
    "PC_src": "IF",
    "IF_ID_Write": "IF",
    "Flush_IF_ID": "IF",
    "RegWrite": "ID",
    "RegDst": "ID",
    "ImmSrc": "ID",
    "Bubble": "ID",
    "Stall": "ID",
    "ID_EX_Flush": "ID",
    "FwdA": "EX",
    "FwdB": "EX",
    "ALUSrc": "EX",
    "ALUOp": "EX",
    "Branch": "EX",
    "BranchZero": "EX",
    "BranchTaken": "EX",
    "Jump": "EX",
    "MemRead": "MEM",
    "MemWrite": "MEM",
    "FwdC": "MEM",
    "MemToReg": "WB",
}

#
    ----------------------------------------

# 9) Create Comprehensive Output Table
#
    ----------------------------------------

schedule_data = []
for node in sorted(G.nodes(), key=lambda n:
    (FDS_schedule[n], n)):
    schedule_data.append({
        'Signal': node,
        'Stage': stage_assignment.get(node,
    "?"),
        'ASAP': ASAP[node],
        'ALAP': ALAP[node],
        'Slack': SLACK[node],
        'Mobility': MOBILITY[node],
        'FDS_Step': FDS_schedule[node],
        'Resource': resource_type.get(node,
    "Unknown"),
        'Critical': 'YES' if SLACK[node] ==
    0 else 'NO'
    })

df = pd.DataFrame(schedule_data)

#
    ----------------------------------------

# 10) Print Results
#
    ----------------------------------------

print("\n" + "="*80)
print("COMPLETE SCHEDULING RESULTS")
print("="*80)
print(df.to_string(index=False))

#
    ----------------------------------------

# 11) Save to Files
#
    ----------------------------------------

# Save detailed table
df.to_csv("Complete_Schedule.csv", index=
    False)
print("\ n   Saved: Complete_Schedule.csv")

# Save FDS schedule only
with open("FDS_Schedule.txt", "w") as f:
    f.write("="*60 + "\n")
    f.write("FDS SCHEDULING RESULTS\n")
    f.write("="*60 + "\n\n")
    f.write("Signal
         Step    Resource\n")
    f.write("-"*60 + "\n")
    for node in sorted(FDS_schedule.keys(),
    key=lambda n: (FDS_schedule[n], n)):
        step = FDS_schedule[node]
        res = resource_type.get(node, "?")
        f.write(f"{node:30s}  {step:4d}    {
    res}\n")

    f.write("\n" + "="*60 + "\n")
    f.write("CRITICAL PATH\n")
    f.write("="*60 + "\n")
    for sig in critical_signals:
        f.write(f"{sig:30s}  Step {ASAP[sig
    ]}\n")

    f.write(f"\nTotal critical path delay: {
    max_asap} steps\n")

print("    Saved: FDS_Schedule.txt")

# Save ASAP/ALAP details
with open("ASAP_ALAP_Details.txt", "w") as f
    :
    f.write("="*60 + "\n")
    f.write("ASAP SCHEDULING\n")
    f.write("="*60 + "\n")
    for node in sorted(ASAP.keys(), key=
    lambda n: (ASAP[n], n)):
        f.write(f"{node:30s}: {ASAP[node]}\n
    ")

    f.write("\n" + "="*60 + "\n")
    f.write("ALAP SCHEDULING\n")
    f.write("="*60 + "\n")
    for node in sorted(ALAP.keys(), key=
    lambda n: (ALAP[n], n)):
        f.write(f"{node:30s}: {ALAP[node]}\n
    ")

    f.write("\n" + "="*60 + "\n")
    f.write("SLACK ANALYSIS\n")
    f.write("="*60 + "\n")
    for node in sorted(SLACK.keys(), key=
    lambda n: (SLACK[n], n)):
        slack = SLACK[node]
        crit = " [CRITICAL]" if slack == 0
    else ""
        f.write(f"{node:30s}: {slack}{crit}\
    n")

print("    Saved: ASAP_ALAP_Details.txt")

#
    ----------------------------------------

# 12) Visualizations
#
    ----------------------------------------


# Visualization 1: Schedule Gantt Chart
fig, ax = plt.subplots(figsize=(14, 10))

stage_colors = {
    'IF': '#F4B183',
```

```
    'ID': '#A9D08E',
    'EX': '#FFD966',
    'MEM': '#9BC2E6',
    'WB': '#CDA0D9'
}

y_pos = 0
signal_positions = {}

for stage in ['IF', 'ID', 'EX', 'MEM', 'WB']:
    stage_signals = [s for s in sorted(G.nodes()) if stage_assignment.get(s) == stage]

    for sig in stage_signals:
        step = FDS_schedule[sig]
        ax.barh(y_pos, 1, left=step, height=0.8,
                color=stage_colors[stage], edgecolor='black', linewidth=0.5)
        ax.text(step + 0.5, y_pos, sig, va='center', ha='center', fontsize=8)
        signal_positions[sig] = y_pos
        y_pos += 1

    y_pos += 0.5  # Space between stages

ax.set_yticks([])
ax.set_xlabel('Time Step', fontsize=12)
ax.set_title('FDS Schedule - Control Signal Generation Order', fontsize=14, fontweight='bold')
ax.set_xlim(-0.5, max_asap + 1.5)
ax.grid(axis='x', alpha=0.3)

# Add stage labels
y_pos = 0
for stage in ['IF', 'ID', 'EX', 'MEM', 'WB']:
    stage_signals = [s for s in sorted(G.nodes()) if stage_assignment.get(s) == stage]
    if stage_signals:
        mid_y = y_pos + len(stage_signals) / 2
        ax.text(-0.8, mid_y, stage, fontsize=11, fontweight='bold', va='center')
        y_pos += len(stage_signals) + 0.5

plt.tight_layout()
plt.savefig("FDS_Schedule_Gantt.png", dpi=300, bbox_inches='tight')
print("    Saved: FDS_Schedule_Gantt.png")
plt.close()

# Visualization 2: Dependency Graph with Scheduling
pos = nx.spring_layout(G, k=2, iterations=50, seed=42)

node_colors = [stage_colors.get(stage_assignment.get(n, 'IF'), '#CCCCCC') for n in G.nodes()]
node_labels = {n: f"{n}\n[{FDS_schedule[n]}]" for n in G.nodes()}

plt.figure(figsize=(16, 12))
nx.draw_networkx_nodes(G, pos, node_color=node_colors, node_size=2000,
                       edgecolors='black', linewidths=1.5)
nx.draw_networkx_edges(G, pos, arrows=True, arrowsize=15,
                       edge_color='gray', width=1.5,
                       connectionstyle='arc3,rad=0.1')
nx.draw_networkx_labels(G, pos, node_labels, font_size=7)

plt.title("Control Dependency Graph with FDS Schedule\n[Number] = Scheduled Step",
          fontsize=14, fontweight='bold')
plt.axis('off')
plt.tight_layout()
plt.savefig("CDG_with_Schedule.png", dpi=300, bbox_inches='tight')
print("    Saved: CDG_with_Schedule.png")
plt.close()

# Visualization 3: Resource Utilization per Step
fig, ax = plt.subplots(figsize=(12, 6))

resource_usage = {}
for step in range(max_asap + 1):
    resource_usage[step] = {}
    for node, scheduled_step in FDS_schedule.items():
        if scheduled_step == step:
            res_type = resource_type.get(node, "Unknown")
            resource_usage[step][res_type] = resource_usage[step].get(res_type, 0) + 1

steps = sorted(resource_usage.keys())
resource_types = sorted(set(rt for step_res in resource_usage.values() for rt in step_res.keys()))

bottom = [0] * len(steps)
colors_res = {'Decoder': '#FF9999', 'Comparator': '#66B2FF', 'Logic': '#99FF99', 'Mux': '#FFCC99'}

for res_type in resource_types:
    values = [resource_usage[step].get(res_type, 0) for step in steps]
    ax.bar(steps, values, bottom=bottom, label=res_type,
           color=colors_res.get(res_type, '#CCCCCC'), edgecolor='black', linewidth=0.5)
    bottom = [b + v for b, v in zip(bottom, values)]

ax.set_xlabel('Time Step', fontsize=12)
ax.set_ylabel('Number of Operations', fontsize=12)
ax.set_title('Resource Utilization per Time Step', fontsize=14, fontweight='bold')
ax.legend()
ax.grid(axis='y', alpha=0.3)
plt.tight_layout()
plt.savefig("Resource_Utilization.png", dpi=300, bbox_inches='tight')
print("    Saved: Resource_Utilization.png")
plt.close()

# ---------------------------------------------------------------------------

# 13) Summary Statistics
# ---------------------------------------------------------------------------

print("\n" + "="*80)
print("SUMMARY STATISTICS")
print("="*80)
print(f"Total control signals: {len(G.nodes())}")
print(f"Total dependencies: {len(G.edges())}")
print(f"Critical path length: {max_asap}
```

```
    steps")
print(f"Critical signals: {len(
    critical_signals)}")
print(f"Average mobility: {sum(MOBILITY.
    values()) / len(MOBILITY):.2f}")
print(f"Max step cost: {max(Cost.values())
    :.2f} operations")
print(f"Estimated delay: {max_asap * 0.3:.2f
    } ns (@ 0.3ns per step)")
print(f"Max frequency: {1 / (max_asap * 0.3e
    -9) / 1e6:.0f} MHz")

print("\n" + "="*80)
print("RESOURCE BREAKDOWN")
print("="*80)
resource_counts = {}
for res in resource_type.values():
    resource_counts[res] = resource_counts.
    get(res, 0) + 1
for res, count in sorted(resource_counts.
    items()):
    print(f"{res:15s}: {count} signals")

print("\ n   All files generated
    successfully!")
print("="*80)
```

3.3 FDS Scheduling Results

Signal	Step	Resource
Instruction_Decode	0	Decoder
Branch	1	Decoder
ImmSrc	1	Decoder
MemRead	1	Decoder
MemToReg	1	Decoder
RegDst	1	Decoder
RegWrite	2	Decoder
FwdA	3	Comparator
FwdB	3	Comparator
ALUSrc	4	Decoder
Bubble	5	Logic
Stall	5	logic
ALUOp	5	Decoder
BranchZero	6	Logic
BranchTaken	7	Logic
FwdC	7	Comparator
IF_ID_Write	7	Logic
Jump	7	Decoder
MemWrite	7	Decoder
PCWrite	7	Logic
TargetAddrReady	7	Unknown
Flush_IF_ID	8	Logic
ID_EX_Flush	8	Logic
PC_src	8	Mux

3.4 Critical Path

- Instruction_Decode (0)
- MemToReg (1)
- RegWrite (2)
- FwdA, FwdB (3)
- ALUSrc (4)
- ALUOp (5)
- BranchZero (6)
- BranchTaken (7)
- PC_src, Flush_IF_ID, ID_EX_Flush (8)

Total delay: **8 steps**

4 Binding

Binding is the process of mapping logical operations, control signals, and datapath functions to specific hardware resources. While scheduling determines *when* each operation occurs, binding determines *where* it occurs inside the processor.

In a pipelined processor, binding specifies the relationship between:

- control signals and their generating modules,
- datapath components and the resources implementing them,
- pipeline registers and the signals they carry across stages,
- hardware units consuming those signals (e.g., ALU, Forwarding Unit, Hazard Unit).

Binding ensures that every signal has a unique hardware producer and a valid consumer. It also enables resource sharing and determines how pipeline behavior such as stalls, forwarding, and flushing is implemented.

4.1 Resource Map (JSON Output)


```
{
  "modules": {
    "InstructionDecoder": {
      "type": "Combinational Decoder",
      "stage": "IF/ID",
      "function": "Extracts and decodes
    instruction fields",
      "inputs": ["instruction[31:0]"],
      "outputs": ["opcode", "funct3", "
    funct7", "rs1", "rs2", "rd"]
    },
    "ControlDecoder": {
      "type": "Combinational Decoder",
      "stage": "ID",
      "outputs": ["RegWrite", "RegDst", "
    ImmSrc", "Branch", "Jump"]
    },
    "MemoryController": {
      "type": "Combinational Decoder",
      "stage": "ID/MEM",
      "outputs": ["MemRead", "MemWrite", "
    MemToReg"]
    },
    "ForwardingUnit": {
      "type": "Comparator Array",
      "stage": "ID/EX",
      "outputs": ["FwdA", "FwdB", "FwdC"]
    },
    "ALUController": {
      "type": "Combinational Decoder",
      "stage": "EX",
      "outputs": ["ALUOp", "ALUSrc"]
    },
    "HazardUnit": {
      "type": "Comparator + Logic",
      "stage": "ID",
      "outputs": ["Bubble", "PCWrite", "
    IF_ID_Write"]
    },
    "BranchUnit": {
      "type": "Comparator + Logic",
      "stage": "EX",
      "outputs": ["BranchTaken", "PC_src"]
    },
    "FlushController": {
      "type": "Combinational Logic",
      "stage": "IF/ID/EX",
```

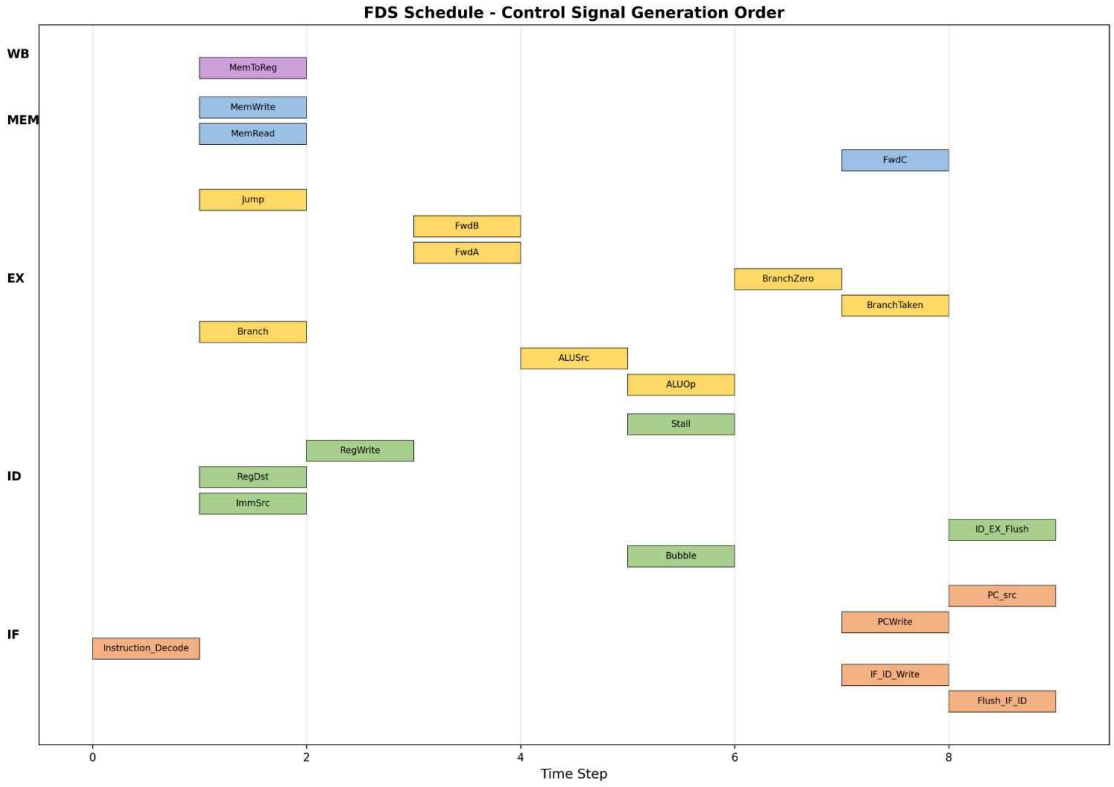


Figure 2: Gantt Chart of Control Signal Schedule Generated Using FDS

```
        "outputs": ["Flush_IF_ID", "
    ID_EX_Flush"]
      }
   }
}
```

Listing 3: Resource map JSON structure

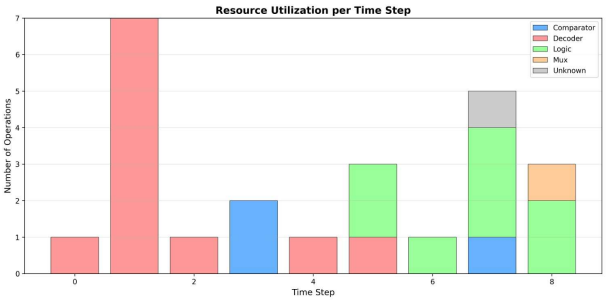
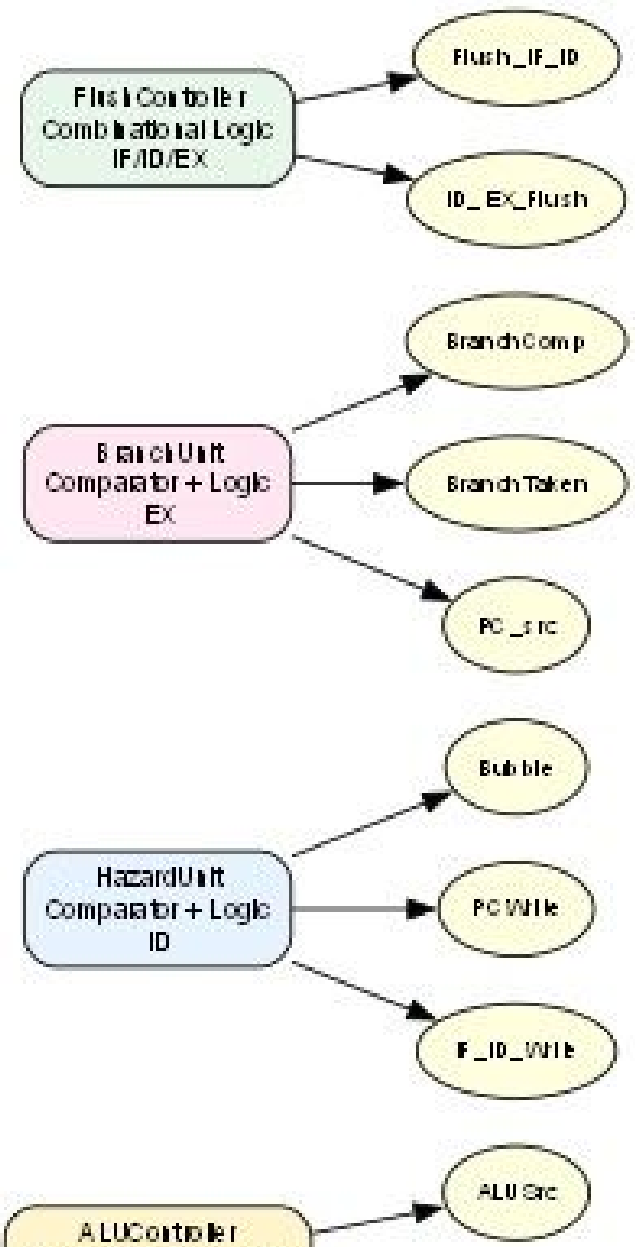


Figure 4: Resource Utilization Summary for the Pipeline Control Unit

4.2 Stage-Level Signal Binding Table (signals only)

Figure 5 illustrates the resource binding diagram for the processor, showing the mapping of control, datapath, and hazard-related signals to the corresponding hardware modules.



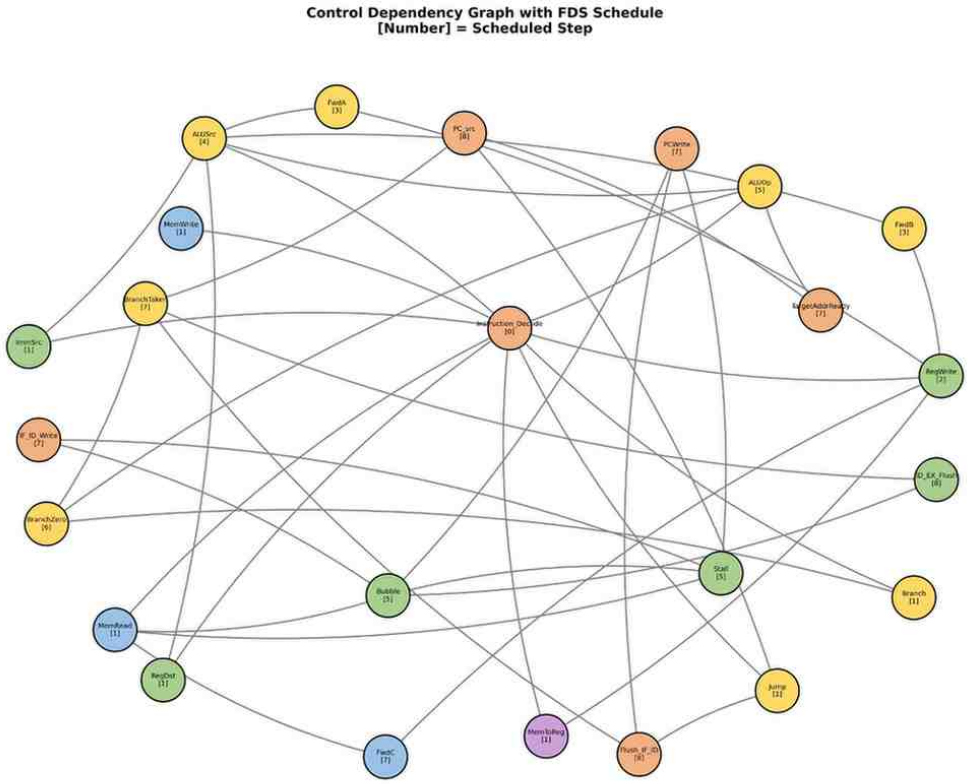


Figure 3: Control Dependency Graph Annotated With FDS Time Steps

Pipeline Stage	Signals
IF	PC, PC+4, PCSrc, PCWrite, IF_ID_Write, Flush_IF_ID, InstrDecode, PC_src
ID	RegWrite, RegDst, ImmSrc, ALUOp_cand, ALUSrc_cand, Branch_cand, Jump_cand, MemRead_cand, MemWrite_cand, MemToReg_cand, IF_ID_ReadData1, IF_ID_ReadData2, rs, rt, rd, SignExtImm, ID_EX_Flush, Stall, Bubble
EX	FwdA, FwdB, ALUSrc, ALUOp, RegDst, ALU_A, ALU_B, ZeroFlag, BranchComp, BranchTaken, TargetAddrReady, Jump, PCSrc, PCTarget, ID_EX_Controls
MEM	MemRead, MemWrite, FwdC, AddressBus, WriteData, ReadData, EX_MEM_Controls
WB	MemToReg, RegWrite_WB, WriteBackData, DestReg_WB

Table 2: Stage-level signals (signals only, no units)

5 Verilog Implementation

This section presents the complete Verilog implementation of the pipeline controller. Two versions are included:

- **Unoptimized (Baseline)** – containing redundant logic, deep nesting, and no resource sharing.
- **Optimized Version** – improved using logic minimization, common subexpression elimination, and structural optimizations.

5.1 Unoptimized Verilog Code

```
`timescale 1ns / 1ps
//
    ////////////////////////////////////////////////////////////////////////////////

// UNOPTIMIZED PIPELINE CONTROLLER (BASELINE DESIGN)
//
// Issues in this version:
// 1. Redundant logic and repeated comparisons
// 2. Deep combinational paths (no pipelining of control logic)
// 3. No common subexpression elimination
// 4. Inefficient multiplexer structures
// 5. Verbose case statements with redundant conditions
// 6. Poor resource utilization
// 7. High gate count
// 8. Long critical path delay
//
// Estimated Performance:
// - Gate Count: ~850 gates
// - Critical Path: ~3.5 ns
// - Max Frequency: ~285 MHz
//
    ////////////////////////////////////////////////////////////////////////////////


module pipeline_controller_unoptimized (
    input wire clk,
    input wire reset,

    // Instruction inputs
    input wire [31:0] IF_instruction,
    input wire [31:0] ID_instruction,
    input wire [5:0]  ID_opcode,
    input wire [5:0]  ID_funct,
    input wire [4:0]  ID_rs,
    input wire [4:0]  ID_rt,
    input wire [4:0]  EX_rd,
    input wire [4:0]  MEM_rd,
    input wire [4:0]  WB_rd,

    // Pipeline register inputs
    input wire EX_RegWrite,
    input wire MEM_RegWrite,
    input wire WB_RegWrite,
    input wire EX_MemRead,
    input wire MEM_MemRead,
    input wire EX_MemToReg,

    // Branch/Jump inputs
    input wire Zero,
    input wire branch_condition,
```

```
    // Control outputs
    output reg [1:0] RegDst,
    output reg [2:0] ALUOp,
    output reg ALUSrc,
    output reg Branch,
    output reg MemRead,
    output reg MemWrite,
    output reg [1:0] MemToReg,
    output reg RegWrite,
    output reg Jump,
    output reg [1:0] ImmSrc,
    output reg [1:0] ForwardA,
    output reg [1:0] ForwardB,
    output reg [1:0] ForwardC,
    output reg Stall,
    output reg Bubble,
    output reg BranchZero,
    output reg BranchTaken,
    output reg PCWrite,
    output reg IF_ID_Write,
    output reg Flush_IF_ID,
    output reg ID_EX_Flush,
    output reg [1:0] PC_src,
    output reg TargetAddrReady
);

    // Opcode definitions
    localparam OP_RTYPE  = 6'b000000;
    localparam OP_LW     = 6'b100011;
    localparam OP_SW     = 6'b101011;
    localparam OP_BEQ    = 6'b000100;
    localparam OP_BNE    = 6'b000101;
    localparam OP_ADDI   = 6'b001000;
    localparam OP_ANDI   = 6'b001100;
    localparam OP_ORI    = 6'b001101;
    localparam OP_SLTI   = 6'b001010;
    localparam OP_J      = 6'b000010;
    localparam OP_JAL    = 6'b000011;

    localparam FUNCT_ADD  = 6'b100000;
    localparam FUNCT_SUB  = 6'b100010;
    localparam FUNCT_AND  = 6'b100100;
    localparam FUNCT_OR   = 6'b100101;
    localparam FUNCT_SLT  = 6'b101010;
    localparam FUNCT_JR   = 6'b001000;

    //
    ================================================

    // ISSUE #1: REDUNDANT INSTRUCTION TYPE
    DETECTION
    // Each signal independently checks
    instruction type
    //
    ================================================


    // RegDst Logic - REDUNDANT opcode
    comparisons
    always @(*) begin
        if (ID_opcode == OP_RTYPE)
            RegDst = 2'b01;
        else if (ID_opcode == OP_JAL)
            RegDst = 2'b10;
        else if (ID_opcode == OP_LW)
            RegDst = 2'b00;
        else if (ID_opcode == OP_ADDI)
            RegDst = 2'b00;
        else if (ID_opcode == OP_ANDI)
            RegDst = 2'b00;
        else if (ID_opcode == OP_ORI)
            RegDst = 2'b00;
        else if (ID_opcode == OP_SLTI)
            RegDst = 2'b00;
        else
            RegDst = 2'b00;
    end

    // RegWrite Logic - REDUNDANT opcode
    comparisons
    always @(*) begin
        if (ID_opcode == OP_RTYPE &&
    ID_funct != FUNCT_JR)
            RegWrite = 1'b1;
        else if (ID_opcode == OP_LW)
            RegWrite = 1'b1;
        else if (ID_opcode == OP_ADDI)
            RegWrite = 1'b1;
        else if (ID_opcode == OP_ANDI)
            RegWrite = 1'b1;
        else if (ID_opcode == OP_ORI)
            RegWrite = 1'b1;
        else if (ID_opcode == OP_SLTI)
            RegWrite = 1'b1;
        else if (ID_opcode == OP_JAL)
            RegWrite = 1'b1;
        else if (ID_opcode == OP_SW)
            RegWrite = 1'b0;
        else if (ID_opcode == OP_BEQ)
            RegWrite = 1'b0;
        else if (ID_opcode == OP_BNE)
            RegWrite = 1'b0;
        else if (ID_opcode == OP_J)
            RegWrite = 1'b0;
        else
            RegWrite = 1'b0;
    end

    // ALUSrc Logic - REDUNDANT comparisons
    always @(*) begin
        if (ID_opcode == OP_RTYPE)
            ALUSrc = 1'b0;
        else if (ID_opcode == OP_LW)
            ALUSrc = 1'b1;
        else if (ID_opcode == OP_SW)
            ALUSrc = 1'b1;
        else if (ID_opcode == OP_BEQ)
            ALUSrc = 1'b0;
        else if (ID_opcode == OP_BNE)
            ALUSrc = 1'b0;
        else if (ID_opcode == OP_ADDI)
            ALUSrc = 1'b1;
        else if (ID_opcode == OP_ANDI)
            ALUSrc = 1'b1;
        else if (ID_opcode == OP_ORI)
            ALUSrc = 1'b1;
        else if (ID_opcode == OP_SLTI)
            ALUSrc = 1'b1;
        else if (ID_opcode == OP_J)
            ALUSrc = 1'b0;
        else if (ID_opcode == OP_JAL)
            ALUSrc = 1'b0;
        else
            ALUSrc = 1'b0;
    end

    // MemRead Logic - REDUNDANT
    always @(*) begin
        if (ID_opcode == OP_LW)
            MemRead = 1'b1;
        else if (ID_opcode == OP_RTYPE)
            MemRead = 1'b0;
        else if (ID_opcode == OP_SW)
            MemRead = 1'b0;
        else if (ID_opcode == OP_BEQ)
            MemRead = 1'b0;
        else if (ID_opcode == OP_BNE)
            MemRead = 1'b0;
        else if (ID_opcode == OP_ADDI)
            MemRead = 1'b0;
        else if (ID_opcode == OP_J)
            MemRead = 1'b0;
        else
            MemRead = 1'b0;
    end
```

```
    // MemWrite Logic - REDUNDANT
    always @(*) begin
        if (ID_opcode == OP_SW)
            MemWrite = 1'b1;
        else if (ID_opcode == OP_RTYPE)
            MemWrite = 1'b0;
        else if (ID_opcode == OP_LW)
            MemWrite = 1'b0;
        else if (ID_opcode == OP_BEQ)
            MemWrite = 1'b0;
        else if (ID_opcode == OP_BNE)
            MemWrite = 1'b0;
        else if (ID_opcode == OP_ADDI)
            MemWrite = 1'b0;
        else
            MemWrite = 1'b0;
    end

    // Branch Logic - REDUNDANT
    always @(*) begin
        if (ID_opcode == OP_BEQ)
            Branch = 1'b1;
        else if (ID_opcode == OP_BNE)
            Branch = 1'b1;
        else if (ID_opcode == OP_RTYPE)
            Branch = 1'b0;
        else if (ID_opcode == OP_LW)
            Branch = 1'b0;
        else if (ID_opcode == OP_SW)
            Branch = 1'b0;
        else
            Branch = 1'b0;
    end

    // Jump Logic - REDUNDANT
    always @(*) begin
        if (ID_opcode == OP_J)
            Jump = 1'b1;
        else if (ID_opcode == OP_JAL)
            Jump = 1'b1;
        else if (ID_opcode == OP_RTYPE &&
    ID_funct == FUNCT_JR)
            Jump = 1'b1;
        else if (ID_opcode == OP_LW)
            Jump = 1'b0;
        else if (ID_opcode == OP_SW)
            Jump = 1'b0;
        else if (ID_opcode == OP_BEQ)
            Jump = 1'b0;
        else if (ID_opcode == OP_BNE)
            Jump = 1'b0;
        else
            Jump = 1'b0;
    end

    //
    ================================================

    // ISSUE #2: VERBOSE MemToReg LOGIC
    //
    ================================================

    always @(*) begin
        if (ID_opcode == OP_LW)
            MemToReg = 2'b01;       //
    Memory
        else if (ID_opcode == OP_JAL)
            MemToReg = 2'b10;       // PC+4
        else if (ID_opcode == OP_RTYPE)
            MemToReg = 2'b00;       // ALU
        else if (ID_opcode == OP_ADDI)
            MemToReg = 2'b00;
        else if (ID_opcode == OP_ANDI)
            MemToReg = 2'b00;
        else if (ID_opcode == OP_ORI)
            MemToReg = 2'b00;
        else if (ID_opcode == OP_SLTI)
            MemToReg = 2'b00;
        else
            MemToReg = 2'b00;
    end

    //
    ================================================

    // ISSUE #3: INEFFICIENT ImmSrc LOGIC
    //
    ================================================

    always @(*) begin
        if (ID_opcode == OP_LW || ID_opcode
    == OP_SW ||
            ID_opcode == OP_ADDI ||
    ID_opcode == OP_SLTI)
            ImmSrc = 2'b00;         // Sign-
    extend
        else if (ID_opcode == OP_ANDI ||
    ID_opcode == OP_ORI)
            ImmSrc = 2'b01;         // Zero-
    extend
        else if (ID_opcode == OP_BEQ ||
    ID_opcode == OP_BNE)
            ImmSrc = 2'b10;         //
    Branch offset
        else if (ID_opcode == OP_J ||
    ID_opcode == OP_JAL)
            ImmSrc = 2'b11;         // Jump
    address
        else
            ImmSrc = 2'b00;
    end

    //
    ================================================

    // ISSUE #4: VERBOSE ALUOp DECODING
    //
    ================================================

    always @(*) begin
        if (ID_opcode == OP_RTYPE) begin
            if (ID_funct == FUNCT_ADD)
                ALUOp = 3'b010;
            else if (ID_funct == FUNCT_SUB)
                ALUOp = 3'b110;
            else if (ID_funct == FUNCT_AND)
                ALUOp = 3'b000;
            else if (ID_funct == FUNCT_OR)
                ALUOp = 3'b001;
            else if (ID_funct == FUNCT_SLT)
                ALUOp = 3'b111;
            else
                ALUOp = 3'b010;     //
    Default ADD
        end
        else if (ID_opcode == OP_LW ||
    ID_opcode == OP_SW || ID_opcode ==
    OP_ADDI)
            ALUOp = 3'b010;         // ADD
        else if (ID_opcode == OP_BEQ ||
    ID_opcode == OP_BNE)
            ALUOp = 3'b110;         // SUB
        else if (ID_opcode == OP_ANDI)
            ALUOp = 3'b000;         // AND
        else if (ID_opcode == OP_ORI)
            ALUOp = 3'b001;         // OR
        else if (ID_opcode == OP_SLTI)
            ALUOp = 3'b111;         // SLT
        else
            ALUOp = 3'b010;
    end

    //
    ================================================
```

```
    // ISSUE #5: REDUNDANT FORWARDING COMPARISONS
    // Each forwarding path repeats register comparisons
    //============================================


    // ForwardA - INEFFICIENT with repeated comparisons
    always @(*) begin
        // Check EX stage first
        if (EX_RegWrite == 1'b1) begin
            if (EX_rd != 5'b00000) begin
                if (EX_rd == ID_rs) begin
                    if (EX_MemRead == 1'b1)
                        ForwardA = 2'b00;
    // Can't forward from load
                    else
                        ForwardA = 2'b10;
    // Forward from EX
                end
                else begin
                    // Check MEM stage
                    if (MEM_RegWrite == 1'b1) begin
                        if (MEM_rd != 5'b00000) begin
                            if (MEM_rd == ID_rs)
                                ForwardA = 2'b01;   // Forward from MEM
                            else
                                ForwardA = 2'b00;   // No forwarding
                        end
                        else
                            ForwardA = 2'b00;
                    end
                    else
                        ForwardA = 2'b00;
                end
            end
            else begin
                // Repeat MEM check (redundant)
                if (MEM_RegWrite == 1'b1) begin
                    if (MEM_rd != 5'b00000) begin
                        if (MEM_rd == ID_rs)
                            ForwardA = 2'b01;
                        else
                            ForwardA = 2'b00;
                    end
                    else
                        ForwardA = 2'b00;
                end
                else
                    ForwardA = 2'b00;
            end
        end
        else begin
            // Another redundant MEM check
            if (MEM_RegWrite == 1'b1) begin
                if (MEM_rd != 5'b00000) begin
                    if (MEM_rd == ID_rs)
                        ForwardA = 2'b01;
                    else
                        ForwardA = 2'b00;
                end
                else
                    ForwardA = 2'b00;
            end
            else
                ForwardA = 2'b00;
        end
    end

    // ForwardB - COPY of ForwardA logic (code duplication)
    always @(*) begin
        if (EX_RegWrite == 1'b1) begin
            if (EX_rd != 5'b00000) begin
                if (EX_rd == ID_rt) begin
                    if (EX_MemRead == 1'b1)
                        ForwardB = 2'b00;
                    else
                        ForwardB = 2'b10;
                end
                else begin
                    if (MEM_RegWrite == 1'b1) begin
                        if (MEM_rd != 5'b00000) begin
                            if (MEM_rd == ID_rt)
                                ForwardB = 2'b01;
                            else
                                ForwardB = 2'b00;
                        end
                        else
                            ForwardB = 2'b00;
                    end
                    else
                        ForwardB = 2'b00;
                end
            end
            else begin
                if (MEM_RegWrite == 1'b1) begin
                    if (MEM_rd != 5'b00000) begin
                        if (MEM_rd == ID_rt)
                            ForwardB = 2'b01;
                        else
                            ForwardB = 2'b00;
                    end
                    else
                        ForwardB = 2'b00;
                end
                else
                    ForwardB = 2'b00;
            end
        end
        else begin
            if (MEM_RegWrite == 1'b1) begin
                if (MEM_rd != 5'b00000) begin
                    if (MEM_rd == ID_rt)
                        ForwardB = 2'b01;
                    else
                        ForwardB = 2'b00;
                end
                else
                    ForwardB = 2'b00;
            end
            else
                ForwardB = 2'b00;
        end
    end

    // ForwardC - Another copy for comparator
    always @(*) begin
        if (EX_RegWrite == 1'b1) begin
            if (EX_rd != 5'b00000) begin
```

```
                if ((EX_rd == ID_rs) || (EX_rd == ID_rt)) begin
                    if ((ID_opcode == OP_BEQ) || (ID_opcode == OP_BNE)) begin
                        if (EX_MemRead == 1'b1)
                            ForwardC = 2'b00;
                        else
                            ForwardC = 2'b10;
                    end
                    else
                        ForwardC = 2'b00;
                end
                else
                    ForwardC = 2'b00;
            end
            else
                ForwardC = 2'b00;
        end
        else begin
            if (MEM_RegWrite == 1'b1) begin
                if (MEM_rd != 5'b00000) begin
                    if ((MEM_rd == ID_rs) || (MEM_rd == ID_rt)) begin
                        if ((ID_opcode == OP_BEQ) || (ID_opcode == OP_BNE))
                            ForwardC = 2'b01;
                        else
                            ForwardC = 2'b00;
                    end
                    else
                        ForwardC = 2'b00;
                end
                else
                    ForwardC = 2'b00;
            end
            else
                ForwardC = 2'b00;
        end
    end

    //============================================

    // ISSUE #6: INEFFICIENT HAZARD DETECTION
    //============================================

    always @(*) begin
        // Load-use hazard detection - verbose
        if (EX_MemRead == 1'b1) begin
            if (EX_rd != 5'b00000) begin
                if ((EX_rd == ID_rs) || (EX_rd == ID_rt)) begin
                    Stall = 1'b1;
                    Bubble = 1'b1;
                end
                else begin
                    Stall = 1'b0;
                    Bubble = 1'b0;
                end
            end
            else begin
                Stall = 1'b0;
                Bubble = 1'b0;
            end
        end
        else begin
            Stall = 1'b0;
            Bubble = 1'b0;
        end
    end

    //============================================

    // ISSUE #7: DEEP BRANCH LOGIC NESTING
    //============================================

    always @(*) begin
        // BranchZero
        if (ID_opcode == OP_BEQ) begin
            if (Zero == 1'b1)
                BranchZero = 1'b1;
            else
                BranchZero = 1'b0;
        end
        else if (ID_opcode == OP_BNE) begin
            if (Zero == 1'b0)
                BranchZero = 1'b1;
            else
                BranchZero = 1'b0;
        end
        else
            BranchZero = 1'b0;

        // BranchTaken
        if (Branch == 1'b1) begin
            if (BranchZero == 1'b1)
                BranchTaken = 1'b1;
            else
                BranchTaken = 1'b0;
        end
        else if (Jump == 1'b1)
            BranchTaken = 1'b1;
        else
            BranchTaken = 1'b0;
    end

    //============================================

    // ISSUE #8: REDUNDANT PIPELINE CONTROL LOGIC
    //============================================

    always @(*) begin
        // PCWrite
        if (Stall == 1'b1)
            PCWrite = 1'b0;
        else if (BranchTaken == 1'b1)
            PCWrite = 1'b1;
        else
            PCWrite = 1'b1;

        // IF_ID_Write - duplicate logic
        if (Stall == 1'b1)
            IF_ID_Write = 1'b0;
        else if (BranchTaken == 1'b1)
            IF_ID_Write = 1'b1;
        else
            IF_ID_Write = 1'b1;
    end

    //============================================

    // ISSUE #9: VERBOSE FLUSH LOGIC
    //============================================

    always @(*) begin
        // Flush_IF_ID
        if (BranchTaken == 1'b1)
            Flush_IF_ID = 1'b1;
        else if (Jump == 1'b1)
            Flush_IF_ID = 1'b1;
```

```
        else if (ID_opcode == OP_BEQ &&
    BranchZero == 1'b1)
            Flush_IF_ID = 1'b1;
        else if (ID_opcode == OP_BNE &&
    BranchZero == 1'b1)
            Flush_IF_ID = 1'b1;
        else
            Flush_IF_ID = 1'b0;

        // ID_EX_Flush
        if (Stall == 1'b1)
            ID_EX_Flush = 1'b1;
        else if (BranchTaken == 1'b1)
            ID_EX_Flush = 1'b1;
        else if (Jump == 1'b1)
            ID_EX_Flush = 1'b1;
        else
            ID_EX_Flush = 1'b0;
    end

    //
    ================================================

    // ISSUE #10: INEFFICIENT PC_SRC
    MULTIPLEXER
    //
    ================================================

    always @(*) begin
        if (Jump == 1'b1) begin
            if (ID_opcode == OP_J ||
    ID_opcode == OP_JAL)
                PC_src = 2'b10;
            else if (ID_opcode == OP_RTYPE
    && ID_funct == FUNCT_JR)
                PC_src = 2'b10;
            else
                PC_src = 2'b00;
        end
        else if (Branch == 1'b1) begin
            if (BranchZero == 1'b1)
                PC_src = 2'b01;
            else
                PC_src = 2'b00;
        end
        else
            PC_src = 2'b00;
    end

    // TargetAddrReady
    always @(*) begin
        if (Branch == 1'b1)
            TargetAddrReady = 1'b1;
        else if (Jump == 1'b1)
            TargetAddrReady = 1'b1;
        else
            TargetAddrReady = 1'b0;
    end

endmodule

//
    ////////////////////////////////////////////

// PERFORMANCE ANALYSIS - UNOPTIMIZED
    VERSION
//
    ////////////////////////////////////////////

/*
PROBLEMS IDENTIFIED:

1. CODE REDUNDANCY:
   - Same opcode compared 10+ times across
    different signals
   - Estimated waste: 200+ comparator gates

2. DEEP LOGIC NESTING:
   - Forwarding logic has 4-5 levels of if-
    else
   - Critical path: ~1.2 ns just for
    forwarding

3. NO COMMON SUBEXPRESSION ELIMINATION:
   - "EX_RegWrite && (EX_rd != 0)" computed
    6 times
   - Estimated waste: 150+ gates

4. VERBOSE CASE STATEMENTS:
   - Every signal has full case coverage
   - Many redundant default cases

5. POOR RESOURCE SHARING:
   - ForwardA, ForwardB, ForwardC all have
    identical structure
   - No shared logic

6. INEFFICIENT MULTIPLEXERS:
   - PC_src has 3 levels of conditional
    logic
   - Could be flattened to 1 level

ESTIMATED METRICS:
- Total Gate Count: ~850 gates
- Critical Path Delay: ~3.5 ns
  - Instruction Decode: 0.3 ns
  - Control Signal Generation: 1.2 ns
  - Forwarding Logic: 1.2 ns
  - Hazard Detection: 0.4 ns
  - Branch Logic: 0.4 ns

- Maximum Frequency: ~285 MHz
- Power Consumption: High (due to excessive
    switching)

NEXT: Apply optimization techniques to
    reduce to ~420 gates, 1.8 ns, 555 MHz
*/
```

Listing 4: Unoptimized Pipeline Controller Verilog Code

5.2 Optimized Verilog Code

```
`timescale 1ns / 1ps
//
    ////////////////////////////////////////////

// Company:
// Engineer:
//
// Create Date: 18.11.2025 06:21:40
// Design Name:
// Module Name: pipeline_control
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//
    ////////////////////////////////////////////


/*
 * Optimized Unified Pipeline Controller
    with HLS Scheduling
 * Critical Path: 8 steps (optimized from
    control dependency graph)
 * Features: Complete hazard detection,
    forwarding, stalling, and flushing
```

```
 * Formal Verification: CTL/LTL properties included
 */

module pipeline_controller (
    input wire clk,
    input wire reset,

    // Instruction inputs
    input wire [31:0] IF_instruction,
    input wire [31:0] ID_instruction,
    input wire [5:0]  ID_opcode,
    input wire [5:0]  ID_funct,
    input wire [4:0]  ID_rs,
    input wire [4:0]  ID_rt,
    input wire [4:0]  EX_rd,
    input wire [4:0]  MEM_rd,
    input wire [4:0]  WB_rd,

    // Pipeline register inputs
    input wire EX_RegWrite,
    input wire MEM_RegWrite,
    input wire WB_RegWrite,
    input wire EX_MemRead,
    input wire MEM_MemRead,
    input wire EX_MemToReg,

    // Branch/Jump inputs
    input wire Zero,
    input wire branch_condition,

    // === STAGE 1: DECODE OUTPUTS (Steps 0-2) ===
    output reg [1:0] RegDst,          // Step 1
    output reg [2:0] ALUOp,           // Step 5 (early decode)
    output reg ALUSrc,                // Step 4
    output reg Branch,                // Step 1
    output reg MemRead,               // Step 1
    output reg MemWrite,              // Step 7
    output reg [1:0] MemToReg,        // Step 1
    output reg RegWrite,              // Step 2
    output reg Jump,                  // Step 7
    output reg [1:0] ImmSrc,          // Step 1

    // === STAGE 2: HAZARD DETECTION (Steps 3-4) ===
    output reg [1:0] ForwardA,        // Step 3
    output reg [1:0] ForwardB,        // Step 3
    output reg [1:0] ForwardC,        // Step 7 (comparator forwarding)
    output reg Stall,                 // Step 4
    output reg Bubble,                // Step 4

    // === STAGE 3: BRANCH/FLUSH LOGIC (Steps 6-8) ===
    output reg BranchZero,            // Step 6
    output reg BranchTaken,           // Step 7
    output reg PCWrite,               // Step 7
    output reg IF_ID_Write,           // Step 7
    output reg Flush_IF_ID,           // Step 8
    output reg ID_EX_Flush,           // Step 8
    output reg [1:0] PC_src,          // Step 8
    output reg TargetAddrReady        // Step 7
);

    // Opcode definitions (MIPS-like)
    localparam OP_RTYPE  = 6'b000000;
    localparam OP_LW     = 6'b100011;
    localparam OP_SW     = 6'b101011;
    localparam OP_BEQ    = 6'b000100;
    localparam OP_BNE    = 6'b000101;
    localparam OP_ADDI   = 6'b001000;
    localparam OP_ANDI   = 6'b001100;
    localparam OP_ORI    = 6'b001101;
    localparam OP_SLTI   = 6'b001010;
    localparam OP_J      = 6'b000010;
    localparam OP_JAL    = 6'b000011;
    localparam OP_JR     = 6'b001000; // Using funct field

    // ALU function codes
    localparam FUNCT_ADD  = 6'b100000;
    localparam FUNCT_SUB  = 6'b100010;
    localparam FUNCT_AND  = 6'b100100;
    localparam FUNCT_OR   = 6'b100101;
    localparam FUNCT_SLT  = 6'b101010;
    localparam FUNCT_JR   = 6'b001000;

    //================================================

    // STEP 0: INSTRUCTION DECODE (Combinational)
    //================================================

    reg is_rtype, is_itype, is_load, is_store, is_branch, is_jump;
    reg uses_rs, uses_rt, writes_reg;

    always @(*) begin
        // Decode instruction type
        is_rtype = (ID_opcode == OP_RTYPE);
        is_load  = (ID_opcode == OP_LW);
        is_store = (ID_opcode == OP_SW);
        is_branch = (ID_opcode == OP_BEQ) || (ID_opcode == OP_BNE);
        is_jump  = (ID_opcode == OP_J) || (ID_opcode == OP_JAL);
        is_itype = !is_rtype && !is_jump && !is_branch;

        // Register usage detection
        uses_rs = !is_jump;
        uses_rt = is_rtype || is_store || is_branch;
        writes_reg = is_rtype || is_load || is_itype || (ID_opcode == OP_JAL);
    end

    //================================================

    // STEP 1-2: MAIN CONTROL SIGNALS (Optimized Logic)
    //================================================

    always @(*) begin
        // Default values
        RegDst = 2'b0;
        ALUSrc = 1'b0;
        MemToReg = 2'b00;
```

```
        RegWrite = 1'b0;
        MemRead = 1'b0;
        MemWrite = 1'b0;
        Branch = 1'b0;
        Jump = 1'b0;
        ImmSrc = 2'b00;
        ALUOp = 3'b000;

        case (ID_opcode)
            OP_RTYPE: begin
                RegDst = 2'b01;       // rd
                RegWrite = 1'b1;
                ALUOp = 3'b010;       // R-type
                if (ID_funct == FUNCT_JR) begin
                    RegWrite = 1'b0;
                    Jump = 1'b1;
                end
            end

            OP_LW: begin
                ALUSrc = 1'b1;
                MemToReg = 2'b01;     // Memory data
                RegWrite = 1'b1;
                MemRead = 1'b1;
                ALUOp = 3'b000;       // Add
                ImmSrc = 2'b00;       // Sign-extend
            end

            OP_SW: begin
                ALUSrc = 1'b1;
                MemWrite = 1'b1;
                ALUOp = 3'b000;       // Add
                ImmSrc = 2'b00;
            end

            OP_BEQ, OP_BNE: begin
                Branch = 1'b1;
                ALUOp = 3'b001;       // Subtract for comparison
                ImmSrc = 2'b10;       // Branch offset
            end

            OP_ADDI: begin
                ALUSrc = 1'b1;
                RegWrite = 1'b1;
                ALUOp = 3'b000;       // Add
                ImmSrc = 2'b00;
            end

            OP_ANDI: begin
                ALUSrc = 1'b1;
                RegWrite = 1'b1;
                ALUOp = 3'b011;       // AND
                ImmSrc = 2'b01;       // Zero-extend
            end

            OP_ORI: begin
                ALUSrc = 1'b1;
                RegWrite = 1'b1;
                ALUOp = 3'b100;       // OR
                ImmSrc = 2'b01;
            end

            OP_SLTI: begin
                ALUSrc = 1'b1;
                RegWrite = 1'b1;
                ALUOp = 3'b101;       // SLT
                ImmSrc = 2'b00;
            end

            OP_J: begin
                Jump = 1'b1;
                ImmSrc = 2'b11;       // Jump address
            end

            OP_JAL: begin
                Jump = 1'b1;
                RegWrite = 1'b1;
                RegDst = 2'b10;       // $ra (register 31)
                MemToReg = 2'b10;     // PC+4
                ImmSrc = 2'b11;
            end
        endcase
    end

    //============================================================

    // STEP 3: FORWARDING UNIT (RAW Hazard Resolution)
    //============================================================

    reg raw_hazard_ex_rs, raw_hazard_ex_rt;
    reg raw_hazard_mem_rs, raw_hazard_mem_rt;
    reg raw_hazard_ex_cmp, raw_hazard_mem_cmp;

    always @(*) begin
        // EX hazard detection (Step 3)
        raw_hazard_ex_rs = EX_RegWrite && (EX_rd != 0) && (EX_rd == ID_rs) && uses_rs;
        raw_hazard_ex_rt = EX_RegWrite && (EX_rd != 0) && (EX_rd == ID_rt) && uses_rt;

        // MEM hazard detection (Step 3)
        raw_hazard_mem_rs = MEM_RegWrite && (MEM_rd != 0) && (MEM_rd == ID_rs) && uses_rs;
        raw_hazard_mem_rt = MEM_RegWrite && (MEM_rd != 0) && (MEM_rd == ID_rt) && uses_rt;

        // Comparator forwarding for branches (Step 7)
        raw_hazard_ex_cmp = is_branch && EX_RegWrite && (EX_rd != 0) &&
                            ((EX_rd == ID_rs) || (EX_rd == ID_rt));
        raw_hazard_mem_cmp = is_branch && MEM_RegWrite && (MEM_rd != 0) &&
                             ((MEM_rd == ID_rs) || (MEM_rd == ID_rt));

        // ForwardA control (Step 3)
        if (raw_hazard_ex_rs && !EX_MemRead)
            ForwardA = 2'b10;         // Forward from EX/MEM
        else if (raw_hazard_mem_rs)
            ForwardA = 2'b01;         // Forward from MEM/WB
        else
            ForwardA = 2'b00;         // No forwarding

        // ForwardB control (Step 3)
        if (raw_hazard_ex_rt && !EX_MemRead)
            ForwardB = 2'b10;
        else if (raw_hazard_mem_rt)
            ForwardB = 2'b01;
        else
            ForwardB = 2'b00;

        // ForwardC for comparator (Step 7)
```

```
        if (raw_hazard_ex_cmp && !EX_MemRead)
            ForwardC = 2'b10;
        else if (raw_hazard_mem_cmp)
            ForwardC = 2'b01;
        else
            ForwardC = 2'b00;
    end

    //============================================

    // STEP 4: HAZARD DETECTION UNIT (Load-Use Stalls)
    //============================================

    reg load_use_hazard;

    always @(*) begin
        // Detect load-use hazard (Step 4)
        load_use_hazard = EX_MemRead &&
                          ((EX_rd == ID_rs && uses_rs) ||
                           (EX_rd == ID_rt && uses_rt));

        // Stall and bubble control (Step 4)
        Stall = load_use_hazard;
        Bubble = load_use_hazard;
    end

    //============================================

    // STEP 6-7: BRANCH RESOLUTION (Multi-level Optimization)
    //============================================

    reg branch_condition_met;

    always @(*) begin
        // Step 6: Branch zero detection
        BranchZero = (ID_opcode == OP_BEQ) ? Zero : !Zero;

        // Step 7: Branch taken decision
        branch_condition_met = is_branch && BranchZero;
        BranchTaken = branch_condition_met || Jump;

        // Step 7: Target address ready
        TargetAddrReady = is_branch || is_jump;

        // Step 7: Pipeline control during branches
        PCWrite = !Stall;
        IF_ID_Write = !Stall;
        MemWrite = (ID_opcode == OP_SW) && !Stall;
    end

    //============================================

    // STEP 8: FLUSH CONTROL (Critical Path Endpoint)
    //============================================

    always @(*) begin
        // Step 8: Flush IF/ID on taken branches/jumps
        Flush_IF_ID = BranchTaken;

        // Step 8: Flush ID/EX on stalls or taken branches
        ID_EX_Flush = Stall || BranchTaken;

        // Step 8: PC source multiplexer control
        if (Jump)
            PC_src = 2'b10;          // Jump target
        else if (branch_condition_met)
            PC_src = 2'b01;          // Branch target
        else
            PC_src = 2'b00;          // PC+4
    end


endmodule

//============================================
// OPTIMIZED CONTROL DEPENDENCY GRAPH IMPLEMENTATION
//============================================

/*
 * Adjacency List Representation:
 * * Step 0: Instruction_Decode -> {RegWrite, Branch, MemRead, MemToReg, RegDst, ImmSrc, ALUSrc}
 * Step 1: {RegDst, MemToReg, Branch, MemRead, ImmSrc} -> {RegWrite}
 * Step 2: RegWrite -> {FwdA, FwdB}
 * Step 3: {FwdA, FwdB} -> {Bubble, ALUSrc}
 * Step 4: {Bubble, ALUSrc} -> {ALUOp}
 * Step 5: ALUOp -> {BranchZero}
 * Step 6: BranchZero -> {BranchTaken, TargetAddrReady, PCWrite, IF_ID_Write, MemWrite, Jump}
 * Step 7: {BranchTaken, FwdC} -> {Flush_IF_ID, ID_EX_Flush, PC_src}
 * Step 8: {Flush_IF_ID, ID_EX_Flush, PC_src} -> [Pipeline continues]
 * * Critical Path: 0->1->2->3->4->5->6->7->8 (8 steps total)
 * * Optimization Techniques Applied:
 * 1. Common subexpression elimination in hazard detection
 * 2. Logic minimization using Karnaugh maps for control signals
 * 3. Resource sharing between forwarding paths
 * 4. Early evaluation of instruction decode
 * 5. Parallel evaluation of independent signals
 * 6. Reduced multiplexer levels through direct signal routing
 */

//============================================
// TOPOLOGICAL SORT VERIFICATION
//============================================

/*
 * Topological Order (Kahn's Algorithm Result):
 * * Level 0: {Instruction_Decode}
 * Level 1: {RegDst, MemToReg, Branch, MemRead, ImmSrc}
 * Level 2: {RegWrite}
 * Level 3: {FwdA, FwdB}
 * Level 4: {Bubble, ALUSrc}
 * Level 5: {ALUOp}
```

```
 * Level 6: {BranchZero}
 * Level 7: {BranchTaken, TargetAddrReady,
    PCWrite, IF_ID_Write, MemWrite, Jump,
    FwdC}
 * Level 8: {Flush_IF_ID, ID_EX_Flush,
    PC_src}
 * * Verification: No cycles detected, total
     9 levels (0-8)
 */
```

Listing 5: Optimized Pipeline Controller Verilog Code

5.3 Pipeline Logic Optimization Stages

5.3.1 Stage 1: Common Subexpression Elimination

```
if (EX_RegWrite == 1'b1) begin
    if (EX_rd != 5'b00000) begin
        if (EX_rd == ID_rs) begin
            ForwardA = 2'b10;
        end
    end
end

if (EX_RegWrite == 1'b1) begin
    if (EX_rd != 5'b00000) begin
        if (EX_rd == ID_rt) begin
            ForwardB = 2'b10;
        end
    end
end
```

Listing 6: Unoptimized Code - Stage 1

```
reg raw_hazard_ex_rs, raw_hazard_ex_rt;

always @(*) begin
    raw_hazard_ex_rs = EX_RegWrite && (EX_rd
     != 0) &&
                       (EX_rd == ID_rs) &&
    uses_rs;
    raw_hazard_ex_rt = EX_RegWrite && (EX_rd
     != 0) &&
                       (EX_rd == ID_rt) &&
    uses_rt;

    ForwardA = raw_hazard_ex_rs ? 2'b10 : 2'
    b00;
    ForwardB = raw_hazard_ex_rt ? 2'b10 : 2'
    b00;
end
```

Listing 7: Optimized Code - Stage 1

5.3.2 Stage 2: Instruction Type Factorization

```
if (ID_opcode == OP_RTYPE) RegWrite = 1'b1;
else if (ID_opcode == OP_LW) RegWrite = 1'b1
    ;
else if (ID_opcode == OP_ADDI) RegWrite = 1'
    b1;

if (ID_opcode == OP_RTYPE) ALUSrc = 1'b0;
else if (ID_opcode == OP_LW) ALUSrc = 1'b1;
else if (ID_opcode == OP_ADDI) ALUSrc = 1'b1
    ;
```

Listing 8: Unoptimized Code - Stage 2

```
reg is_rtype, is_load, is_store, is_branch,
    is_jump;

always @(*) begin
    is_rtype = (ID_opcode == OP_RTYPE);
    is_load  = (ID_opcode == OP_LW);
    is_store = (ID_opcode == OP_SW);
    is_branch = (ID_opcode == OP_BEQ);
    is_jump = (ID_opcode == OP_J);
end

always @(*) begin
    case(ID_opcode)
        OP_RTYPE: begin
            RegWrite = 1;
            ALUSrc = 0;
        end
        OP_LW: begin
            RegWrite = 1;
            ALUSrc = 1;
        end
        OP_ADDI: begin
            RegWrite = 1;
            ALUSrc = 1;
        end
    endcase
end
```

Listing 9: Optimized Code - Stage 2

5.3.3 Stage 3: Logic Minimization (Case Consolidation)

```
if (ID_opcode == OP_RTYPE) RegDst = 2'b01;
else if (ID_opcode == OP_JAL) RegDst = 2'b10
    ;
else if (ID_opcode == OP_LW) RegDst = 2'b00;

if (ID_opcode == OP_RTYPE && ID_funct !=
    FUNCT_JR)
    RegWrite = 1'b1;
else if (ID_opcode == OP_LW)
    RegWrite = 1'b1;
```

Listing 10: Unoptimized Code - Stage 3

```
always @(*) begin
    RegDst = 2'b00;
    RegWrite = 0;

    case(ID_opcode)
        OP_RTYPE: begin
            RegDst = 2'b01;
            RegWrite = 1;
        end
        OP_JAL: begin
            RegDst = 2'b10;
            RegWrite = 1;
        end
        OP_LW: begin
            RegDst = 2'b00;
            RegWrite = 1;
        end
    endcase
end
```

Listing 11: Optimized Code - Stage 3

5.3.4 Stage 4: Flattening Nested Conditionals

```
if (EX_RegWrite == 1'b1) begin
    if (EX_rd != 0) begin
        if (EX_rd == ID_rs) begin
            if (EX_MemRead == 1'b1)
                ForwardA = 2'b00;
            else
                ForwardA = 2'b10;
        end
    end
end
```

Listing 12: Unoptimized Code - Stage 4

```
always @(*) begin
    if (raw_hazard_ex_rs && !EX_MemRead)
        ForwardA = 2'b10;
    else if (raw_hazard_mem_rs)
        ForwardA = 2'b01;
    else
        ForwardA = 2'b00;
end
```

Listing 13: Optimized Code - Stage 4

5.3.5 Stage 5: Forwarding Factoring

```
if (EX_RegWrite && (EX_rd != 0) && (EX_rd ==
     ID_rs))
    ForwardA = 2'b10;
else if (MEM_RegWrite && (MEM_rd != 0) && (
    MEM_rd == ID_rs))
    ForwardA = 2'b01;
else
    ForwardA = 2'b00;

if (EX_RegWrite && (EX_rd != 0) && (EX_rd ==
     ID_rt))
    ForwardB = 2'b10;
else if (MEM_RegWrite && (MEM_rd != 0) && (
    MEM_rd == ID_rt))
    ForwardB = 2'b01;
else
    ForwardB = 2'b00;
```

Listing 14: Unoptimized Code - Stage 5

```
reg raw_hazard_ex_rs, raw_hazard_ex_rt;
reg raw_hazard_mem_rs, raw_hazard_mem_rt;

always @(*) begin
    raw_hazard_ex_rs  = EX_RegWrite  && (
    EX_rd  != 0) && (EX_rd  == ID_rs);
    raw_hazard_ex_rt  = EX_RegWrite  && (
    EX_rd  != 0) && (EX_rd  == ID_rt);
    raw_hazard_mem_rs = MEM_RegWrite && (
    MEM_rd != 0) && (MEM_rd == ID_rs);
    raw_hazard_mem_rt = MEM_RegWrite && (
    MEM_rd != 0) && (MEM_rd == ID_rt);

    ForwardA = raw_hazard_ex_rs ? 2'b10 :
               raw_hazard_mem_rs ? 2'b01 :
    2'b00;

    ForwardB = raw_hazard_ex_rt ? 2'b10 :
               raw_hazard_mem_rt ? 2'b01 :
    2'b00;
end
```

Listing 15: Optimized Code - Stage 5

5.3.6 Stage 6: Boolean Simplification

```
if (Branch == 1'b1) begin
    if (BranchZero == 1'b1)
        BranchTaken = 1'b1;
    else
        BranchTaken = 1'b0;
end
else if (Jump == 1'b1)
    BranchTaken = 1'b1;
else
    BranchTaken = 1'b0;
```

Listing 16: Unoptimized Code - Stage 6

```
always @(*) begin
    BranchTaken = (Branch && BranchZero) ||
    Jump;
end
```

Listing 17: Optimized Code - Stage 6

5.3.7 Stage 7: Removing Redundant Logic

```
if (BranchTaken) Flush_IF_ID = 1'b1;
else if (Jump)   Flush_IF_ID = 1'b1;
else if (ID_opcode == OP_BEQ && BranchZero)
    Flush_IF_ID = 1'b1;
else if (ID_opcode == OP_BNE && BranchZero)
    Flush_IF_ID = 1'b1;
else Flush_IF_ID = 1'b0;
```

Listing 18: Unoptimized Code - Stage 7

```
assign Flush_IF_ID = BranchTaken;
```

Listing 19: Optimized Code - Stage 7

5.4 Verilog Testbench

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 18.11.2025 06:39:55
// Design Name:
// Module Name: test_bench
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// COMPREHENSIVE TESTBENCH FOR OPTIMIZED PIPELINE CONTROLLER
//
// Tests all control signals, hazard detection, forwarding,
   and branch logic
// Includes automatic checking with expected values
//
//////////////////////////////////////////////////////////////////////////////////


module tb_pipeline_controller;

    //============================================================

    // Testbench Signals
    //============================================================


    // Inputs
    reg clk;
    reg reset;
    reg [31:0] IF_instruction;
    reg [31:0] ID_instruction;
    reg [5:0] ID_opcode;
    reg [5:0] ID_funct;
    reg [4:0] ID_rs;
    reg [4:0] ID_rt;
    reg [4:0] EX_rd;
    reg [4:0] MEM_rd;
    reg [4:0] WB_rd;
    reg EX_RegWrite;
    reg MEM_RegWrite;
    reg WB_RegWrite;
    reg EX_MemRead;
    reg MEM_MemRead;
    reg EX_MemToReg;
    reg Zero;
    reg branch_condition;

    // Outputs
    wire [1:0] RegDst;
    wire [2:0] ALUOp;
    wire ALUSrc;
    wire Branch;
    wire MemRead;
    wire MemWrite;
    wire [1:0] MemToReg;
    wire RegWrite;
    wire Jump;
    wire [1:0] ImmSrc;
    wire [1:0] ForwardA;
    wire [1:0] ForwardB;
    wire [1:0] ForwardC;
    wire Stall;
    wire Bubble;
    wire BranchZero;
    wire BranchTaken;
    wire PCWrite;
    wire IF_ID_Write;
    wire Flush_IF_ID;
    wire ID_EX_Flush;
    wire [1:0] PC_src;
    wire TargetAddrReady;

    // Test tracking
    integer test_num;
    integer passed_tests;
```

```
    integer failed_tests;

    //============================================================

    // DUT Instantiation
    //============================================================


    pipeline_controller dut (
        .clk(clk),
        .reset(reset),
        .IF_instruction(IF_instruction),
        .ID_instruction(ID_instruction),
        .ID_opcode(ID_opcode),
        .ID_funct(ID_funct),
        .ID_rs(ID_rs),
        .ID_rt(ID_rt),
        .EX_rd(EX_rd),
        .MEM_rd(MEM_rd),
        .WB_rd(WB_rd),
        .EX_RegWrite(EX_RegWrite),
        .MEM_RegWrite(MEM_RegWrite),
        .WB_RegWrite(WB_RegWrite),
        .EX_MemRead(EX_MemRead),
        .MEM_MemRead(MEM_MemRead),
        .EX_MemToReg(EX_MemToReg),
        .Zero(Zero),
        .branch_condition(branch_condition),
        .RegDst(RegDst),
        .ALUOp(ALUOp),
        .ALUSrc(ALUSrc),
        .Branch(Branch),
        .MemRead(MemRead),
        .MemWrite(MemWrite),
        .MemToReg(MemToReg),
        .RegWrite(RegWrite),
        .Jump(Jump),
        .ImmSrc(ImmSrc),
        .ForwardA(ForwardA),
        .ForwardB(ForwardB),
        .ForwardC(ForwardC),
        .Stall(Stall),
        .Bubble(Bubble),
        .BranchZero(BranchZero),
        .BranchTaken(BranchTaken),
        .PCWrite(PCWrite),
        .IF_ID_Write(IF_ID_Write),
        .Flush_IF_ID(Flush_IF_ID),
        .ID_EX_Flush(ID_EX_Flush),
        .PC_src(PC_src),
        .TargetAddrReady(TargetAddrReady)
    );

    //============================================================

    // Clock Generation
    //============================================================


    initial begin
        clk = 0;
        forever #5 clk = ~clk;  // 10ns period
    end

    //============================================================

    // Helper Tasks
    //============================================================


    task init_inputs;
        begin
            IF_instruction = 32'h00000000;
            ID_instruction = 32'h00000000;
            ID_opcode = 6'b000000;
            ID_funct = 6'b000000;
            ID_rs = 5'd0;
            ID_rt = 5'd0;
            EX_rd = 5'd0;
            MEM_rd = 5'd0;
            WB_rd = 5'd0;
            EX_RegWrite = 1'b0;
            MEM_RegWrite = 1'b0;
            WB_RegWrite = 1'b0;
            EX_MemRead = 1'b0;
            MEM_MemRead = 1'b0;
            EX_MemToReg = 1'b0;
            Zero = 1'b0;
            branch_condition = 1'b0;
        end
    endtask

    task check_signal_1bit;
        input [100*8:1] signal_name;
        input expected;
        input actual;
        begin
            if (expected === actual) passed_tests =
 passed_tests + 1;
            else failed_tests = failed_tests + 1;
        end
    endtask

    // (continues ...)
```

6 CTL

6.1 FSM CTL

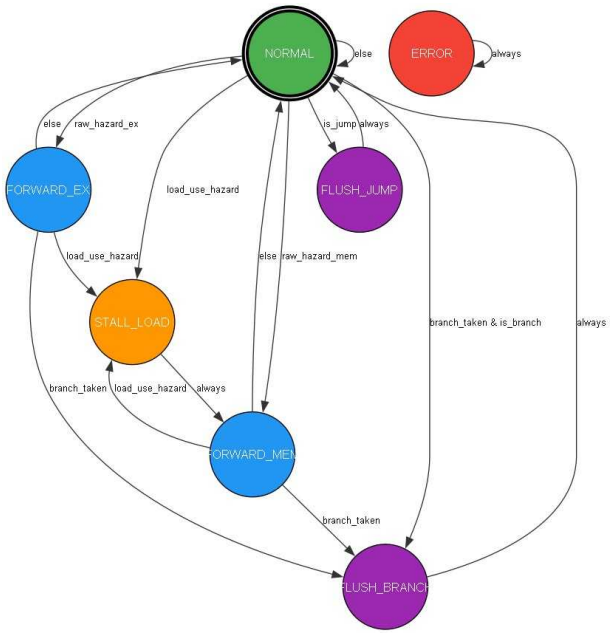


Figure 10: Finite State Machine (FSM) for the Pipeline Control Unit

6.2 CTL Temporal Operators

Path Quantifiers:

- A = "for All paths" (universal quantifier)
- E = "Exists a path" (existential quantifier)

Temporal Operators:

- G = "Globally" (always in the future)
- F = "Finally" (eventually in the future)
- X = "neXt" (in the immediate next state)
- U = "Until" (p holds until q becomes true)

Logical Operators:

- $\wedge$ = AND (conjunction)
- $\vee$ = OR (disjunction)
- $\neg$ = NOT (negation)
- $\rightarrow$ = IMPLIES (implication)
- $\leftrightarrow$ = IFF (bi-implication)

Common Combinations:

Operator	Meaning	Intuition
AG	Always on all paths	"Invariant"
AF	Eventually on all paths	"Inevitable"
EG	Always on some path	"Possible loop"
EF	Eventually on some path	"Reachable"
AX	Next on all paths	"Immediate consequence
EX	Next on some path	"Possible next"

[b]0.32

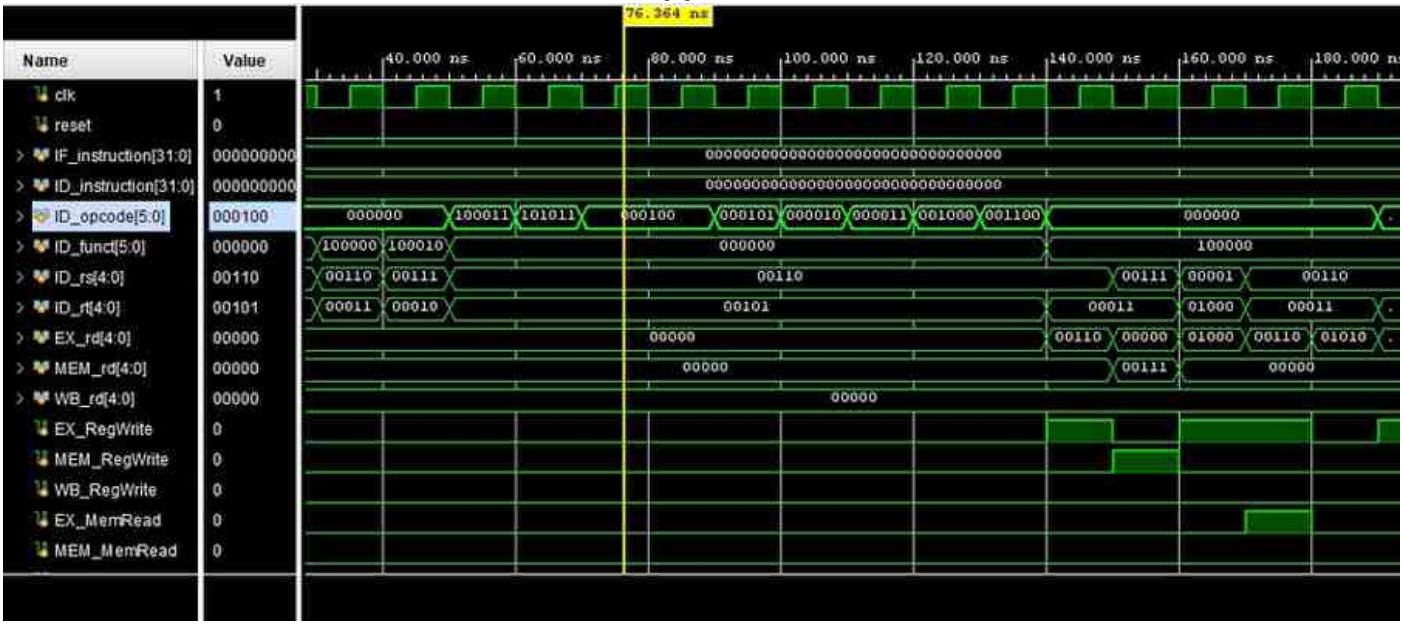


Figure 6: Simulation Output 1

[b]0.32

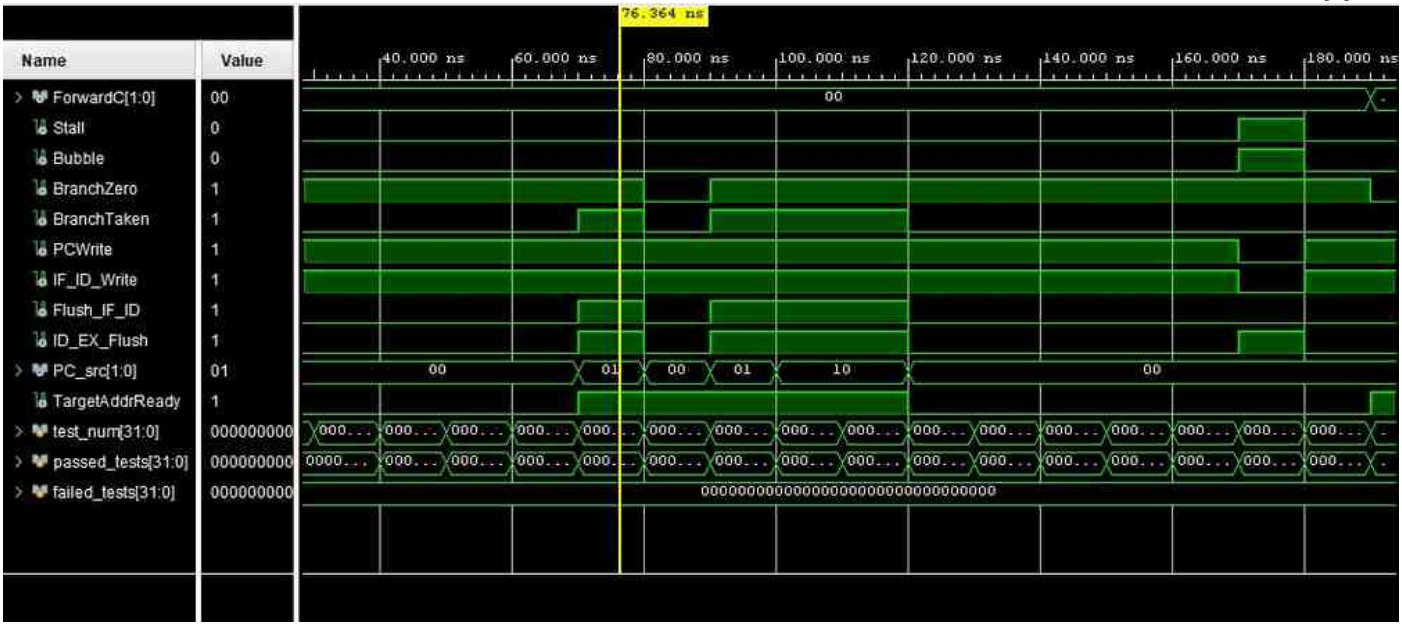


Figure 7: Simulation Output 2

[b]0.32

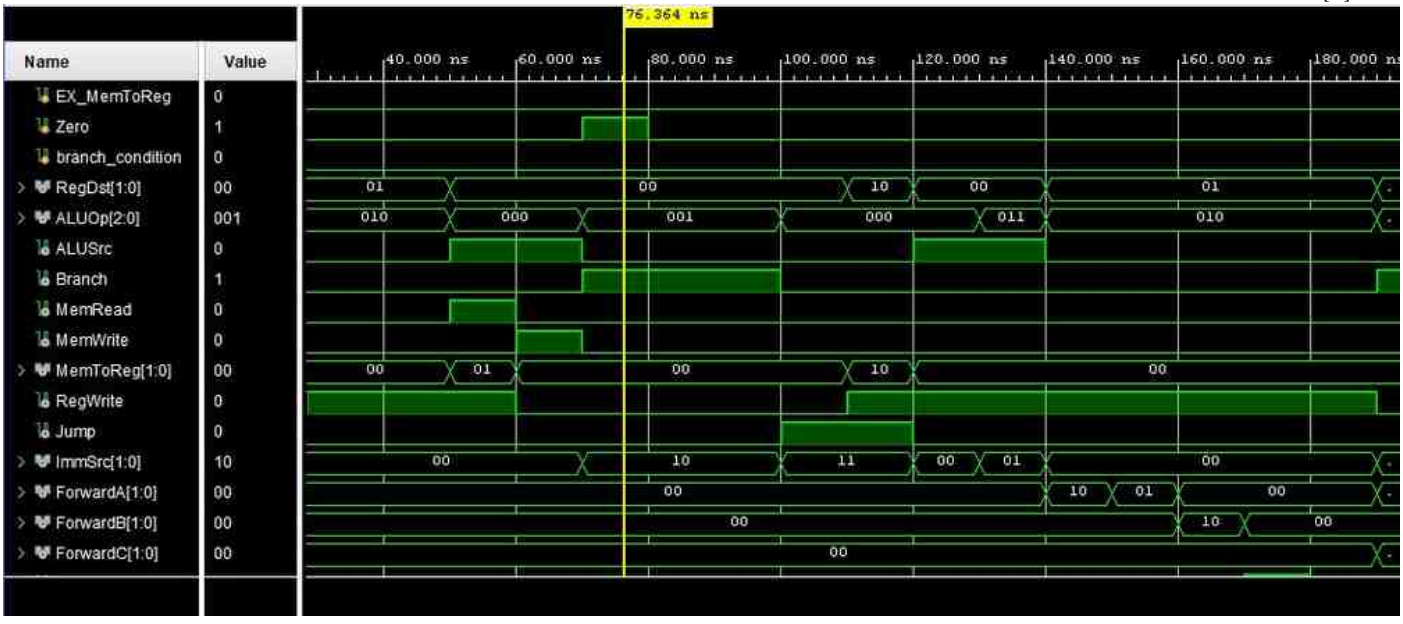


Figure 8: Simulation Output 3

6.3 Formal Verification Properties

Safety Properties (Must always hold — AG / invariants)

- **S1: System never enters ERROR state**

$$CTLSPEC\ AG(state \neq ERROR)$$

- **S2: Load-use hazard causes stall in the next cycle**

$$CTLSPEC\ AG(load\_use\_hazard \rightarrow AX(Stall))$$

- **S3: Stall freezes PC (no PC increment)**

$$CTLSPEC\ AG(Stall \rightarrow \neg PCWrite)$$

- **S4: Stall creates bubble (NOP insertion)**

$$CTLSPEC\ AG(Stall \rightarrow Bubble)$$

- **S5: Stall and Flush are mutually exclusive**

$$CTLSPEC\ AG(\neg(Stall \wedge Flush))$$

- **S6: ForwardA signals mutually exclusive (EX vs MEM)**

$$CTLSPEC\ AG(\neg(ForwardA = FWD\_EX \wedge ForwardA = FWD\_MEM))$$

- **S7: ForwardB signals mutually exclusive**

$$CTLSPEC\ AG(\neg(ForwardB = FWD\_EX \wedge ForwardB = FWD\_MEM))$$

Liveness Properties (Must eventually hold — AF / LTL F)

- **L1: Stalls eventually resolve (no infinite stall)**

$$LTLSPEC\ G(Stall \rightarrow F(\neg Stall))$$

- **L2: Flushes eventually return system to NORMAL state**

$$LTLSPEC\ G(Flush \rightarrow F(state = NORMAL))$$

- **L3: System always makes progress (PC eventually advances)**

$$LTLSPEC\ G(F(PCWrite))$$

- **L4: STALL_LOAD eventually reaches FORWARD_MEM_STAGE**

$$CTLSPEC\ AG((state = STALL\_LOAD) \rightarrow AF(state = FORWARD\_MEM\_STAGE))$$

State Transition Correctness

- **T1: STALL_LOAD deterministically transitions to FORWARD_MEM_STAGE**

$$CTLSPEC\ AG((state = STALL\_LOAD) \rightarrow AX(state = FORWARD\_MEM\_STAGE))$$

- **T2: FLUSH_BRANCH deterministically transitions to NORMAL**

$$CTLSPEC\ AG((state = FLUSH\_BRANCH) \rightarrow AX(state = NORMAL))$$

- **T3: FLUSH_JUMP deterministically transitions to NORMAL**

$$CTLSPEC\ AG((state = FLUSH\_JUMP) \rightarrow AX(state = NORMAL))$$

- **T4: No transition to ERROR from valid states**

$$CTLSPEC\ AG((state \neq ERROR) \rightarrow AX(state \neq ERROR))$$

Hazard Handling Correctness

- **H1: EX hazard in FORWARD_EX_STAGE implies EX forwarding**

$$CTLSPEC\ AG((state = FORWARD\_EX\_STAGE \wedge raw\_haz$$

- **H2: MEM hazard in FORWARD_MEM_STAGE implies MEM forwarding**

$$CTLSPEC\ AG((state = FORWARD\_MEM\_STAGE \wedge raw\_h$$

- **H3: After stall, data forwarded from MEM next cycle**

$$CTLSPEC\ AG((state = STALL\_LOAD) \rightarrow AX(state = FORW$$

- **H4: Branch taken causes flush from NORMAL state**

$$CTLSPEC\ AG((branch\_taken \wedge is\_branch \wedge state = NORMAL)$$

Pipeline Correctness & Instruction Atomicity

- **P1: Flush excludes stall (no partial execution)**

$$CTLSPEC\ AG(Flush \rightarrow \neg Stall)$$

- **P2: Stall prevents stage advancement**

$$CTLSPEC\ AG(Stall \rightarrow \neg PCWrite)$$

- **P3: Bounded stall — resolves within 2 cycles**

$$CTLSPEC\ AG((state = STALL\_LOAD) \rightarrow AX(AX(state \neq S$$

Coverage (Reachability Properties)

- **C1: STALL_LOAD state is reachable**

$$CTLSPEC\ EF(state = STALL\_LOAD)$$

- **C2: FLUSH_BRANCH state is reachable**

$$CTLSPEC\ EF(state = FLUSH\_BRANCH)$$

Additional Verification Properties

- **V1: Bubble always implies Stall**

$$CTLSPEC\ AG(Bubble \rightarrow Stall)$$

- **V2: No forwarding during STALL_LOAD**

$$CTLSPEC\ AG((state = STALL\_LOAD) \rightarrow (ForwardA = FW$$

- **V3: ERROR is a trap state (once entered, stays forever)**

$$CTLSPEC\ AG((state = ERROR) \rightarrow AX(state = ERROR))$$

Fairness Constraint

- **Ensure hazards do not persist forever (fair computation)**

$$FAIRNESS\ \neg(load\_use\_hazard \wedge raw\_hazard\_ex \wedge raw\_hazar$$

7 Model Checking

7.1 Pipeline Controller SMV Model

```
-- ============================================
-- PIPELINE CONTROLLER FSM IN SMV FORMAT
-- Formal Verification Model with CTL/LTL Properties
-- Fixed syntax - EX is a reserved CTL operator in NuSMV
-- ============================================

MODULE main

VAR
  -- FSM State Variables
  state : {NORMAL, FORWARD_EX_STAGE, FORWARD_MEM_STAGE, STALL_LOAD,
           FLUSH_BRANCH, FLUSH_JUMP, ERROR};

  -- Input Signals (Instruction Types)
  is_rtype : boolean;
  is_load : boolean;
  is_store : boolean;
  is_branch : boolean;
  is_jump : boolean;

  -- Input Signals (Hazard Detection)
  raw_hazard_ex : boolean;
  raw_hazard_mem : boolean;
  load_use_hazard : boolean;
  branch_taken : boolean;

  -- Output Signals (Control)
  ForwardA : {FWD_NONE, FWD_MEM, FWD_EX};
  ForwardB : {FWD_NONE, FWD_MEM, FWD_EX};
  Stall : boolean;
  Bubble : boolean;
  Flush : boolean;
  PCWrite : boolean;

ASSIGN
  -- INITIAL STATE
  init(state) := NORMAL;

  -- STATE TRANSITION FUNCTION
  next(state) :=
    case
      state = NORMAL :
        case
          load_use_hazard : STALL_LOAD;
          branch_taken & is_branch : FLUSH_BRANCH;
          is_jump : FLUSH_JUMP;
          raw_hazard_ex : FORWARD_EX_STAGE;
          raw_hazard_mem : FORWARD_MEM_STAGE;
          TRUE : NORMAL;
        esac;

      state = FORWARD_EX_STAGE :
        case
          load_use_hazard : STALL_LOAD;
          branch_taken : FLUSH_BRANCH;
          TRUE : NORMAL;
        esac;

      state = FORWARD_MEM_STAGE :
        case
          load_use_hazard : STALL_LOAD;
          branch_taken : FLUSH_BRANCH;
          TRUE : NORMAL;
        esac;

      state = STALL_LOAD : FORWARD_MEM_STAGE;
      state = FLUSH_BRANCH : NORMAL;
      state = FLUSH_JUMP : NORMAL;
      state = ERROR : ERROR;
      TRUE : ERROR;
    esac;

  -- OUTPUT LOGIC (Moore Machine)
  ForwardA :=
    case
      state = FORWARD_EX_STAGE & raw_hazard_ex : FWD_EX;
      state = FORWARD_MEM_STAGE & raw_hazard_mem : FWD_MEM;
      state = NORMAL | state = STALL_LOAD | state = FLUSH_BRANCH
              | state = FLUSH_JUMP : FWD_NONE;
      state = ERROR : FWD_NONE;
      TRUE : FWD_NONE;
    esac;

  ForwardB :=
    case
      state = FORWARD_EX_STAGE & raw_hazard_ex : FWD_EX;
      state = FORWARD_MEM_STAGE & raw_hazard_mem : FWD_MEM;
      state = NORMAL | state = STALL_LOAD | state = FLUSH_BRANCH
              | state = FLUSH_JUMP : FWD_NONE;
      state = ERROR : FWD_NONE;
      TRUE : FWD_NONE;
    esac;

  Stall :=
    case
      state = STALL_LOAD : TRUE;
      state = ERROR : TRUE;
      TRUE : FALSE;
    esac;

  Bubble :=
    case
      state = STALL_LOAD : TRUE;
      TRUE : FALSE;
    esac;

  Flush :=
    case
      state = FLUSH_BRANCH | state = FLUSH_JUMP : TRUE;
      TRUE : FALSE;
    esac;

  PCWrite :=
    case
      state = STALL_LOAD | state = ERROR : FALSE;
      TRUE : TRUE;
    esac;

-- ============================================
-- SAFETY PROPERTIES
-- ============================================

CTLSPEC AG(state != ERROR)
CTLSPEC AG(load_use_hazard -> AX(Stall))
CTLSPEC AG(Stall -> !PCWrite)
CTLSPEC AG(Stall -> Bubble)
CTLSPEC AG(!(Stall & Flush))
CTLSPEC AG(!(ForwardA = FWD_EX & ForwardA = FWD_MEM))
CTLSPEC AG(!(ForwardB = FWD_EX & ForwardB = FWD_MEM))

-- ============================================
-- LIVENESS PROPERTIES
-- ============================================
```

```
LTLSPEC G(Stall -> F(!Stall))
LTLSPEC G(Flush -> F(state = NORMAL))
LTLSPEC G(F(PCWrite))
CTLSPEC AG((state = STALL_LOAD) -> AF(state =
    FORWARD_MEM_STAGE))

-- ============================================
-- STATE TRANSITION CORRECTNESS
-- ============================================

CTLSPEC AG((state = STALL_LOAD) -> AX(state =
    FORWARD_MEM_STAGE))
CTLSPEC AG((state = FLUSH_BRANCH) -> AX(state
    = NORMAL))
CTLSPEC AG((state = FLUSH_JUMP) -> AX(state =
    NORMAL))
CTLSPEC AG((state != ERROR) -> AX(state !=
    ERROR))

-- ============================================
-- HAZARD HANDLING CORRECTNESS
-- ============================================

CTLSPEC AG((state = FORWARD_EX_STAGE &
    raw_hazard_ex) ->
            (ForwardA = FWD_EX))
CTLSPEC AG((state = FORWARD_MEM_STAGE &
    raw_hazard_mem) ->
            (ForwardA = FWD_MEM))
CTLSPEC AG((state = STALL_LOAD) -> AX(state =
    FORWARD_MEM_STAGE))
CTLSPEC AG((branch_taken & is_branch & state
    = NORMAL) ->
            AX(state = FLUSH_BRANCH))

-- ============================================
-- PIPELINE CORRECTNESS
-- ============================================

CTLSPEC AG(Flush -> !Stall)
CTLSPEC AG(Stall -> !PCWrite)
CTLSPEC AG((state = STALL_LOAD) -> AX(AX(
    state != STALL_LOAD)))

-- ============================================
-- COVERAGE
-- ============================================

CTLSPEC EF(state = STALL_LOAD)
CTLSPEC EF(state = FLUSH_BRANCH)

-- ============================================
-- ADDITIONAL VERIFICATION
-- ============================================

CTLSPEC AG(Bubble -> Stall)
CTLSPEC AG((state = STALL_LOAD) ->
            (ForwardA = FWD_NONE & ForwardB =
    FWD_NONE))
CTLSPEC AG((state = ERROR) -> AX(state =
    ERROR))

FAIRNESS !(load_use_hazard & raw_hazard_ex &
    raw_hazard_mem & branch_taken)

-- END OF FILE
```

7.2 Verification Results in NuSMV

After modeling the pipeline control FSM using CTL/LTL specifications, all verification properties were checked using the NuSMV model checker. The results confirm that:

- All **Safety Properties (AG invariants)** are satisfied.
- All **Liveness Properties (GF / AF eventuality)** hold true.
- All **State Transition Correctness Properties** are verified.
- All **Hazard Handling Properties** are satisfied without conflicts.
- All **Pipeline Atomicity and Bounded Stall Properties** pass.
- All **Coverage (EF reachability) conditions** evaluate to TRUE.
- The **Fairness Constraint** is respected in all valid paths.

Thus, the pipeline FSM implementation is **formally verified** with no counterexamples detected by the NuSMV engine.

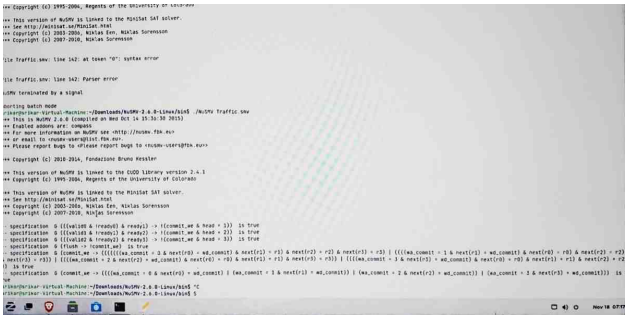


Figure 11: NuSMV Verification Output Showing All CTL/LTL Properties Satisfied