

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.

```
1 adj_list = {  
2   "IF:PC_src": [  
3     "ID:RegWrite", "ID:RegDst", "ID:  
4     ImmSrc",  
5     "ID:ID_EX_Flush", "ID:Bubble",  
6     "EX:ALUOp", "EX:ALUSrc", "EX:Branch"  
7     , "EX:BranchZero",  
8     "MEM:MemRead", "MEM:MemWrite", "WB:  
9     MemToReg"  
10    ],  
11  
12    "IF:PCWrite": ["IF:IF_ID_Write", "IF:  
13    Flush_IF_ID"],  
14  
15    "IF:IF_ID_Write": [  
16      "ID:RegWrite", "ID:RegDst", "ID:  
17      ImmSrc",  
18      "ID:Bubble", "ID:ID_EX_Flush", "IF:  
19      Flush_IF_ID"  
20    ],  
21  
22    "IF:Flush_IF_ID": [  
23      "ID:RegWrite", "ID:RegDst", "ID:  
24      ImmSrc",  
25      "ID:Bubble", "ID:ID_EX_Flush"  
26    ]  
27  }
```

```

19 ],
20
21 "ID:RegWrite": ["EX:FwdA", "EX:FwdB", "
MEM:FwdC"],
22 "ID:RegDst": ["EX:ALUSrc", "EX:ALUOp"],
23 "ID:ImmSrc": ["EX:ALUSrc"],
24
25 "ID:Bubble": [
26     "IF:PCWrite", "IF:IF_ID_Write", "ID:
ID_EX_Flush",
27     "EX:ALUSrc", "EX:ALUOp"
28 ], "ID:Stall": [
29     "IF:PCWrite",           # freeze PC
30     update
31     "IF:IF_ID_Write",       # freeze IF/ID
32     register
33     "ID:Bubble"             # insert bubble
34     in ID stage
35 ],
36
37 "ID:ID_EX_Flush": [
38     "EX:ALUSrc", "EX:ALUOp", "EX:Branch"
39 ],
40     "EX:BranchZero", "EX:BranchTaken"
41 ],
42
43 "EX:FwdA": ["EX:ALUSrc", "EX:ALUOp"],
44 "EX:FwdB": ["EX:ALUSrc", "EX:ALUOp"],
45
46 "EX:ALUSrc": ["EX:ALUOp", "EX:BranchZero
"],
47 "EX:ALUOp": ["EX:BranchZero"],
48
49 "EX:Branch": ["EX:BranchTaken"],
50 "EX:BranchZero": ["EX:BranchTaken"],
51
52 "EX:BranchTaken": [
53     "IF:PC_src", "IF:PCWrite",
54     "IF:Flush_IF_ID", "ID:ID_EX_Flush"
55 ],
56
57 "EX:TargetAddrReady": ["IF:PC_src", "IF:
PCWrite"],
58 "EX:Jump": ["IF:PC_src", "IF:Flush_IF_ID
", "ID:ID_EX_Flush"],
59
60 "MEM:MemRead": [
61     "EX:FwdA", "EX:FwdB", "MEM:FwdC",
62     "ID:Bubble", "IF:PCWrite", "IF:
IF_ID_Write"
63 ],
64
65 "MEM:MemWrite": [],
66 "MEM:FwdC": ["EX:FwdA", "EX:FwdB"],
67 "WB:MemToReg": ["ID:RegWrite", "EX:FwdA
", "EX:FwdB"]
68 }

```

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

```

1 # Build adjacency matrix from adjacency list
2 import numpy as np
3 import pandas as pd
4
5 adj = adj_list
6 nodes = sorted(set(adj.keys()) | {v for
values in adj.values() for v in values})
7 idx = {node: i for i, node in enumerate(
nodes)}
8
9 N = len(nodes)
10 matrix = np.zeros((N, N), dtype=int)
11
12 for u in adj:
13     for v in adj[u]:
14         matrix[idx[u], idx[v]] = 1
15
16 pd.DataFrame(matrix, index=nodes, columns=
nodes).to_csv("CDG_Adjacency_Matrix.csv")
17
18 with open("CDG_Adjacency_Matrix.txt", "w")
as f:
19     f.write(pd.DataFrame(matrix, index=nodes
, columns=nodes).to_string())
20
21 print("Saved CDG_Adjacency_Matrix.csv and
CDG_Adjacency_Matrix.txt")
22
23 # Draw layered CDG
24 import networkx as nx
25 import matplotlib.pyplot as plt
26 import textwrap
27
28 G = nx.DiGraph()
29 for u in adj_list:
30     for v in adj_list[u]:
31         G.add_edge(u, v)
32
33 stage_x = {"IF": 0, "ID": 1, "EX": 2, "MEM":
3, "WB": 4}
34 y_count = {s: 0 for s in stage_x}
35 pos, node_colors = {}, {}
36
37 colors = {
38     "IF": "#F4B183", "ID": "#A9D08E", "EX":
"#FFD966",
39     "MEM": "#9BC2E6", "WB": "#CDA0D9"
40 }
41
42 for node in sorted(G.nodes()):
43     stage = node.split(":")[0]
44     pos[node] = (stage_x[stage], -2 *
y_count[stage])
45     y_count[stage] += 1
46     node_colors[node] = colors[stage]
47
48 labels = {n: "\n".join(textwrap.wrap(n, 18))
for n in G.nodes()}
49
50 plt.figure(figsize=(26, 32))
51 nx.draw_networkx_nodes(G, pos, node_color=
list(node_colors.values()),
52                        edgecolors="black",
53                        node_size=4200)
54 nx.draw_networkx_edges(G, pos, arrows=True,
arrowsize=16,
55                        connectionstyle="
arc3,rad=0.15", width=1.1)
56 nx.draw_networkx_labels(G, pos, labels,
font_size=9)
57
58 for stage, x in stage_x.items():
59     plt.text(x, 4, stage, fontsize=22,
fontweight="bold", ha="center")

```

```

60 plt.title("Control Dependency Graph (Layered
    )", fontsize=28)
61 plt.axis("off")
62 plt.savefig("CDG_Clean_Layered.png", dpi
    =330)
63 plt.show()
64
65 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:

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

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

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

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

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

- $C_{k,j}$: 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

```

1 # CORRECTED PIPELINE CONTROL SIGNAL
  SCHEDULER WITH FDS ALGORITHM
2 #
  =====
3 import networkx as nx
4 import matplotlib.pyplot as plt
5 import pandas as pd
6
7 edges = [

```

IF

EX

MEM

WB

Control Dependency Graph (Clean Layered View)

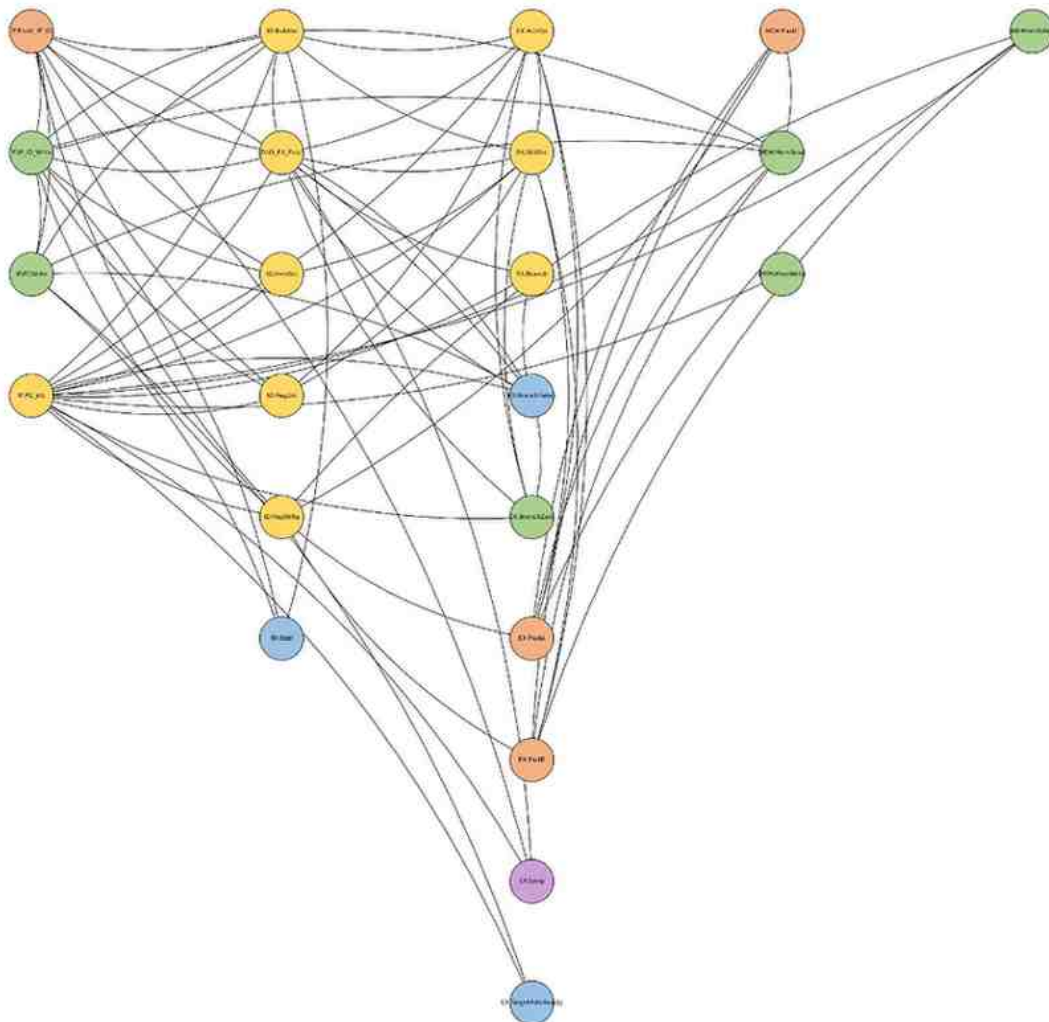


Figure 1: Layered Control Dependency Graph (CDG)

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

```

8  # IF Stage signals
9  ("Instruction_Decompile", "RegWrite"),
10 ("Instruction_Decompile", "RegDst"),
11 ("Instruction_Decompile", "ImmSrc"),
12 ("Instruction_Decompile", "ALUOp"),
13 ("Instruction_Decompile", "ALUSrc"),
14 ("Instruction_Decompile", "Branch"),
15 ("Instruction_Decompile", "MemRead"),
16 ("Instruction_Decompile", "MemWrite"),
17 ("Instruction_Decompile", "MemToReg"),
18 ("Instruction_Decompile", "Jump"),
19
20 # PCWrite and IF_ID_Write dependencies
21 ("Bubble", "PCWrite"),
22 ("Bubble", "IF_ID_Write"),
23 ("PCWrite", "Flush_IF_ID"),
24
25 # Stall signal dependencies (hazard
26 # detection)
27 ("MemRead", "Stall"),          # Load-use
28                                # stall
29 ("Stall", "PCWrite"),          # Stall
30                                # freezes PC
31 ("Stall", "IF_ID_Write"),      # Stall
32                                # freezes IF/ID
33 ("Stall", "Bubble"),          # Stall also
34                                # inserts bubble (if needed)
35
36 # ID Stage dependencies
37 ("RegWrite", "FwdA"),
38 ("RegWrite", "FwdB"),
39 ("RegWrite", "FwdC"),
40
41 ("RegDst", "ALUSrc"),
42 ("ImmSrc", "ALUSrc"),
43
44 # Bubble generation (hazard detection)
45 ("MemRead", "Bubble"), # Load-use
46                                # hazard detection
47 ("Bubble", "ID_EX_Flush"),
48
49 # EX Stage dependencies
50 ("FwdA", "ALUSrc"),
51 ("FwdB", "ALUSrc"),
52 ("ALUSrc", "ALUOp"),
53 ("ALUOp", "BranchZero"),
54
55 ("Branch", "BranchZero"),
56 ("BranchZero", "BranchTaken"),
57
58 # Branch feedback to IF
59 ("BranchTaken", "PC_src"),
60 ("BranchTaken", "Flush_IF_ID"),
61 ("BranchTaken", "ID_EX_Flush"),
62
63 # Jump and target address dependencies
64 ("Jump", "PC_src"),
65 ("Jump", "Flush_IF_ID"),
66 ("ALUOp", "TargetAddrReady"), # Target
67                                # address calculated in ALU
68 ("TargetAddrReady", "PC_src"), # Target
69                                # ready before PC update
70
71 # MEM Stage dependencies
72 ("MemRead", "FwdC"),
73
74 # WB Stage dependencies
75 ("MemToReg", "RegWrite"),
76
77 ]
78 #
79 -----

```



```

71 # 2) Build DAG
72 #
-----
73 G = nx.DiGraph()
74 G.add_edges_from(edges)
75
76 print(f"Total nodes (control signals): {G.
77     number_of_nodes()}")
77 print(f"Total edges (dependencies): {G.
78     number_of_edges()}")
78
79 # Check for cycles (should be acyclic for
80     scheduling)
80 if not nx.is_directed_acyclic_graph(G):
81     print("ERROR: Graph has cycles!")
82     cycles = list(nx.simple_cycles(G))
83     print(f"Cycles found: {cycles}")
84 else:
85     print("    Graph is acyclic (valid for
86         scheduling)")
86
87 #
-----
88 # 3) ASAP Scheduling (As-Soon-As-Possible)
89 #
-----
90 ASAP = {}
91 for node in nx.topological_sort(G):
92     preds = list(G.predecessors(node))
93     if not preds:
94         ASAP[node] = 0 # Root nodes start
95         at step 0
96     else:
97         ASAP[node] = max(ASAP[p] + 1 for p
98             in preds)
98
99 #
-----
100 # 4) ALAP Scheduling (As-Late-As-Possible)
101 #
-----
102 max_asap = max(ASAP.values())
103 print(f"\nCritical path length (max ASAP): {
104     max_asap} steps")
104
105 ALAP = {}
106 # Process in reverse topological order
107 for node in reversed(list(nx.
108     topological_sort(G))):
109     succs = list(G.successors(node))
110     if not succs:
111         ALAP[node] = max_asap # Leaf nodes
112         at latest time
113     else:
114         ALAP[node] = min(ALAP[s] - 1 for s
115             in succs)
115
116 #
-----
117 # 5) Slack and Mobility
118 #
-----
119 SLACK = {n: ALAP[n] - ASAP[n] for n in G.
120     nodes()}
121 MOBILITY = {n: ALAP[n] - ASAP[n] + 1 for n
122     in G.nodes()}
123
124 print("\nCritical path signals (slack = 0):"
125     )
126 critical_signals = [n for n in G.nodes() if
127     SLACK[n] == 0]
128
129 for sig in critical_signals:
130     print(f"    {sig}: ASAP={ASAP[sig]}, ALAP
131         ={ALAP[sig]}")
131
132 #
-----
133 # 6) FDS (Force-Directed Scheduling)
134 #
-----
135 # Calculate probability distribution for
136     each operation
137 Prob = {}
138 for n in G.nodes():
139     Prob[n] = {}
140     for t in range(max_asap + 1):
141         if ASAP[n] <= t <= ALAP[n]:
142             Prob[n][t] = 1.0 / MOBILITY[n]
143         else:
144             Prob[n][t] = 0.0
145
146 # Calculate distribution cost (forces) for
147     each time step
148 Cost = {}
149 for t in range(max_asap + 1):
150     Cost[t] = sum(Prob[n][t] for n in G.
151         nodes())
152
153 print("\nDistribution cost per step:")
154 for t in sorted(Cost.keys()):
155     print(f"    Step {t}: {Cost[t]:.2f}
156         operations")
156
157 # FDS: Choose time step with minimum cost
158     within valid interval
159 FDS_schedule = {}
160 for n in G.nodes():
161     valid_steps = range(ASAP[n], ALAP[n] +
162         1)
163     best_step = min(valid_steps, key=lambda
164         t: Cost[t])
165     FDS_schedule[n] = best_step
165
166 #
-----
167 # 7) Resource Type Classification
168 #
-----
169 resource_type = {
170     "InstructionDecode": "Decoder",
171     "RegWrite": "Decoder",
172     "RegDst": "Decoder",
173     "ImmSrc": "Decoder",
174     "ALUOp": "Decoder",
175     "ALUSrc": "Decoder",
176     "Branch": "Decoder",
177     "MemRead": "Decoder",
178     "MemWrite": "Decoder",
179     "MemToReg": "Decoder",
180     "Jump": "Decoder",
181     "FwdA": "Comparator",
182     "FwdB": "Comparator",
183     "FwdC": "Comparator",
184     "Bubble": "Logic",
185     "Stall": "Logic",
186     "ID_EX_Flush": "Logic",
187     "Flush_IF_ID": "Logic",
188     "PCWrite": "Logic",
189     "IF_ID_Write": "Logic",
190     "BranchZero": "Logic",
191     "BranchTaken": "Logic",
192     "PC_src": "Mux",
193 }

```

```

182 # -----
183 # 8) Stage Assignment (for visualization)
184 # -----
185 stage_assignment = {
186     "InstructionDecode": "IF",
187     "PCWrite": "IF",
188     "PC_src": "IF",
189     "IF_ID_Write": "IF",
190     "Flush_IF_ID": "IF",
191     "RegWrite": "ID",
192     "RegDst": "ID",
193     "ImmSrc": "ID",
194     "Bubble": "ID",
195     "Stall": "ID",
196     "ID_EX_Flush": "ID",
197     "FwdA": "EX",
198     "FwdB": "EX",
199     "ALUSrc": "EX",
200     "ALUOp": "EX",
201     "Branch": "EX",
202     "BranchZero": "EX",
203     "BranchTaken": "EX",
204     "Jump": "EX",
205     "MemRead": "MEM",
206     "MemWrite": "MEM",
207     "FwdC": "MEM",
208     "MemToReg": "WB",
209 }
210 # -----
211 # 9) Create Comprehensive Output Table
212 # -----
213 # -----
214 schedule_data = []
215 for node in sorted(G.nodes(), key=lambda n:
216     (FDS_schedule[n], n)):
217     schedule_data.append({
218         'Signal': node,
219         'Stage': stage_assignment.get(node,
220             "?"),
221         'ASAP': ASAP[node],
222         'ALAP': ALAP[node],
223         'Slack': SLACK[node],
224         'Mobility': MOBILITY[node],
225         'FDS_Step': FDS_schedule[node],
226         'Resource': resource_type.get(node,
227             "Unknown"),
228         'Critical': 'YES' if SLACK[node] ==
229             0 else 'NO'
230     })
231 df = pd.DataFrame(schedule_data)
232 # -----
233 # 10) Print Results
234 # -----
235 print("\n" + "="*80)
236 print("COMPLETE SCHEDULING RESULTS")
237 print("="*80)
238 print(df.to_string(index=False))
239 # -----
240 # 11) Save to Files
241 # -----
242 # Save detailed table
243 df.to_csv("Complete_Schedule.csv", index=
244     False)
245 print("\n    Saved: Complete_Schedule.csv")
246 # Save FDS schedule only
247 with open("FDS_Schedule.txt", "w") as f:
248     f.write("="*60 + "\n")
249     f.write("FDS SCHEDULING RESULTS\n")
250     f.write("="*60 + "\n\n")
251     f.write("Signal
252         Step    Resource\n")
253     for node in sorted(FDS_schedule.keys(),
254         key=lambda n: (FDS_schedule[n], n)):
255         step = FDS_schedule[node]
256         res = resource_type.get(node, "?")
257         f.write(f"{node:30s} {step:4d} {
258             res}\n")
259     f.write("\n" + "="*60 + "\n")
260     f.write("CRITICAL PATH\n")
261     f.write("="*60 + "\n")
262     for sig in critical_signals:
263         f.write(f"{sig:30s} Step {ASAP[sig
264             ]}\n")
265     f.write(f"\nTotal critical path delay: {
266         max_asap} steps\n")
267 print("    Saved: FDS_Schedule.txt")
268 # Save ASAP/ALAP details
269 with open("ASAP_ALAP_Details.txt", "w") as f
270 :
271     f.write("="*60 + "\n")
272     f.write("ASAP SCHEDULING\n")
273     f.write("="*60 + "\n")
274     for node in sorted(ASAP.keys(), key=
275         lambda n: (ASAP[n], n)):
276         f.write(f"{node:30s}: {ASAP[node]}\n
277             ")
278     f.write("\n" + "="*60 + "\n")
279     f.write("ALAP SCHEDULING\n")
280     f.write("="*60 + "\n")
281     for node in sorted(ALAP.keys(), key=
282         lambda n: (ALAP[n], n)):
283         f.write(f"{node:30s}: {ALAP[node]}\n
284             ")
285     f.write("\n" + "="*60 + "\n")
286     f.write("SLACK ANALYSIS\n")
287     f.write("="*60 + "\n")
288     for node in sorted(SLACK.keys(), key=
289         lambda n: (SLACK[n], n)):
290         slack = SLACK[node]
291         crit = " [CRITICAL]" if slack == 0
292         else ""
293         f.write(f"{node:30s}: {slack}{crit}\n
294             ")
295 print("    Saved: ASAP_ALAP_Details.txt")
296 # -----
297 # 12) Visualizations
298 # -----
299 # Visualization 1: Schedule Gantt Chart
300 fig, ax = plt.subplots(figsize=(14, 10))
301 stage_colors = {
302     'IF': '#F4B183',

```

```

300     'ID': '#A9D08E',
301     'EX': '#FFD966',
302     'MEM': '#9BC2E6',
303     'WB': '#CDA0D9'
304 }
305
306 y_pos = 0
307 signal_positions = {}
308
309 for stage in ['IF', 'ID', 'EX', 'MEM', 'WB']:
310     stage_signals = [s for s in sorted(G.
311 nodes()) if stage_assignment.get(s) ==
312 stage]
313
314     for sig in stage_signals:
315         step = FDS_schedule[sig]
316         ax.barh(y_pos, 1, left=step, height
317 =0.8,
318                 color=stage_colors[stage],
319                 edgecolor='black', linewidth=0.5)
320         ax.text(step + 0.5, y_pos, sig, va='
321 center', ha='center', fontsize=8)
322         signal_positions[sig] = y_pos
323         y_pos += 1
324
325     y_pos += 0.5 # Space between stages
326
327 ax.set_yticks([])
328 ax.set_xlabel('Time Step', fontsize=12)
329 ax.set_title('FDS Schedule - Control Signal
330 Generation Order', fontsize=14,
331 fontweight='bold')
332
333 ax.set_xlim(-0.5, max_asap + 1.5)
334 ax.grid(axis='x', alpha=0.3)
335
336 # Add stage labels
337 y_pos = 0
338 for stage in ['IF', 'ID', 'EX', 'MEM', 'WB']:
339     stage_signals = [s for s in sorted(G.
340 nodes()) if stage_assignment.get(s) ==
341 stage]
342     if stage_signals:
343         mid_y = y_pos + len(stage_signals) /
344 2
345         ax.text(-0.8, mid_y, stage, fontsize
346 =11, fontweight='bold', va='center')
347         y_pos += len(stage_signals) + 0.5
348
349 plt.tight_layout()
350 plt.savefig("FDS_Schedule_Gantt.png", dpi
351 =300, bbox_inches='tight')
352 print(" Saved: FDS_Schedule_Gantt.png")
353 plt.close()
354
355 # Visualization 2: Dependency Graph with
356 Scheduling
357 pos = nx.spring_layout(G, k=2, iterations
358 =50, seed=42)
359
360 node_colors = [stage_colors.get(
361 stage_assignment.get(n, 'IF'), '#CCCCCC')
362 for n in G.nodes()]
363 node_labels = {n: f"{n}\n[{FDS_schedule[n]}]
364 " for n in G.nodes()}
365
366 plt.figure(figsize=(16, 12))
367 nx.draw_networkx_nodes(G, pos, node_color=
368 node_colors, node_size=2000,
369 edgecolors='black',
370 linewidths=1.5)
371 nx.draw_networkx_edges(G, pos, arrows=True,
372 arrowsize=15,
373 edge_color='gray',
374 width=1.5,
375 connectionstyle='
376 arc3,rad=0.1')
377
378 nx.draw_networkx_labels(G, pos, node_labels,
379 font_size=7)
380
381 plt.title("Control Dependency Graph with FDS
382 Schedule\n[Number] = Scheduled Step",
383 font_size=14, fontweight='bold')
384 plt.axis('off')
385 plt.tight_layout()
386 plt.savefig("CDG_with_Schedule.png", dpi
387 =300, bbox_inches='tight')
388 print(" Saved: CDG_with_Schedule.png")
389 plt.close()
390
391 # Visualization 3: Resource Utilization per
392 Step
393 fig, ax = plt.subplots(figsize=(12, 6))
394
395 resource_usage = {}
396 for step in range(max_asap + 1):
397     resource_usage[step] = {}
398     for node, scheduled_step in FDS_schedule
399 .items():
400         if scheduled_step == step:
401             res_type = resource_type.get(
402 node, "Unknown")
403             resource_usage[step][res_type] =
404 resource_usage[step].get(res_type, 0) +
405 1
406
407 steps = sorted(resource_usage.keys())
408 resource_types = sorted(set(rt for step_res
409 in resource_usage.values() for rt in
410 step_res.keys()))
411
412 bottom = [0] * len(steps)
413 colors_res = {'Decoder': '#FF9999', '
414 Comparator': '#66B2FF', 'Logic': '#99FF99',
415 'Mux': '#FFCC99'}
416
417 for res_type in resource_types:
418     values = [resource_usage[step].get(
419 res_type, 0) for step in steps]
420     ax.bar(steps, values, bottom=bottom,
421 label=res_type,
422 color=colors_res.get(res_type, '
423 #CCCCCC'), edgecolor='black', linewidth
424 =0.5)
425     bottom = [b + v for b, v in zip(bottom,
426 values)]
427
428 ax.set_xlabel('Time Step', fontsize=12)
429 ax.set_ylabel('Number of Operations',
430 fontsize=12)
431 ax.set_title('Resource Utilization per Time
432 Step', fontsize=14, fontweight='bold')
433 ax.legend()
434 ax.grid(axis='y', alpha=0.3)
435 plt.tight_layout()
436 plt.savefig("Resource_Utilization.png", dpi
437 =300, bbox_inches='tight')
438 print(" Saved: Resource_Utilization.png")
439 plt.close()
440
441 #
442 -----
443
444 # 13) Summary Statistics
445 #
446 -----
447
448 print("\n" + "="*80)
449 print("SUMMARY STATISTICS")
450 print("="*80)
451 print(f"Total control signals: {len(G.nodes
452 ())}")
453 print(f"Total dependencies: {len(G.edges())}
454 ")
455 print(f"Critical path length: {max_asap}")

```



```

steps")
406 print(f"Critical signals: {len(
    critical_signals)}")
407 print(f"Average mobility: {sum(MOBILITY.
    values()) / len(MOBILITY):.2f}")
408 print(f"Max step cost: {max(Cost.values())
    :.2f} operations")
409 print(f"Estimated delay: {max_asap * 0.3:.2f
    } ns (@ 0.3ns per step)")
410 print(f"Max frequency: {1 / (max_asap * 0.3e
    -9) / 1e6:.0f} MHz")
411
412 print("\n" + "="*80)
413 print("RESOURCE BREAKDOWN")
414 print("="*80)
415 resource_counts = {}
416 for res in resource_type.values():
417     resource_counts[res] = resource_counts.
    get(res, 0) + 1
418 for res, count in sorted(resource_counts.
    items()):
419     print(f"{res:15s}: {count} signals")
420
421 print("\n All files generated
    successfully!")
422 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)

```

1 {
2   "modules": {
3     "InstructionDecoder": {
4       "type": "Combinational Decoder",
5       "stage": "IF/ID",
6       "function": "Extracts and decodes
    instruction fields",
7       "inputs": ["instruction[31:0]"],
8       "outputs": ["opcode", "funct3", "
    funct7", "rs1", "rs2", "rd"]
9     },
10    "ControlDecoder": {
11      "type": "Combinational Decoder",
12      "stage": "ID",
13      "outputs": ["RegWrite", "RegDst", "
    ImmSrc", "Branch", "Jump"]
14    },
15    "MemoryController": {
16      "type": "Combinational Decoder",
17      "stage": "ID/MEM",
18      "outputs": ["MemRead", "MemWrite", "
    MemToReg"]
19    },
20    "ForwardingUnit": {
21      "type": "Comparator Array",
22      "stage": "ID/EX",
23      "outputs": ["FwdA", "FwdB", "FwdC"]
24    },
25    "ALUController": {
26      "type": "Combinational Decoder",
27      "stage": "EX",
28      "outputs": ["ALUOp", "ALUSrc"]
29    },
30    "HazardUnit": {
31      "type": "Comparator + Logic",
32      "stage": "ID",
33      "outputs": ["Bubble", "PCWrite", "
    IF_ID_Write"]
34    },
35    "BranchUnit": {
36      "type": "Comparator + Logic",
37      "stage": "EX",
38      "outputs": ["BranchTaken", "PC_src"]
39    },
40    "FlushController": {
41      "type": "Combinational Logic",
42      "stage": "IF/ID/EX",

```

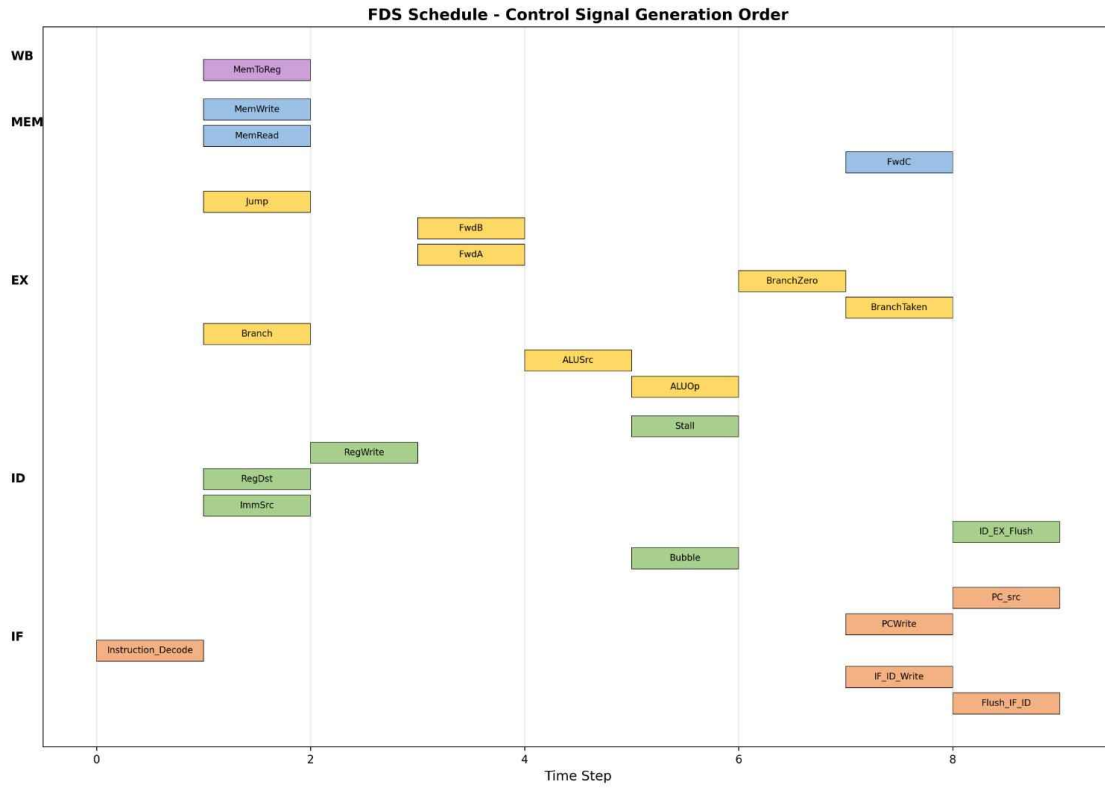


Figure 2: Gantt Chart of Control Signal Schedule Generated Using FDS

```

43   "outputs": ["Flush_IF_ID", "
44   ID_EX_Flush"]
45   }
46 }

```

Listing 3: Resource map JSON structure

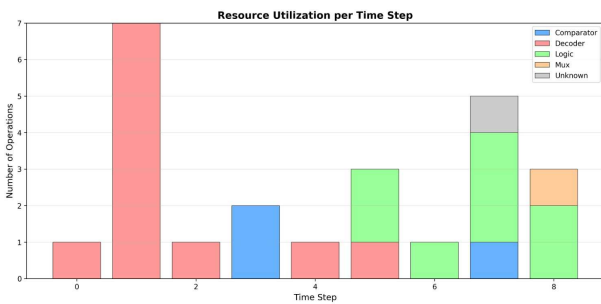
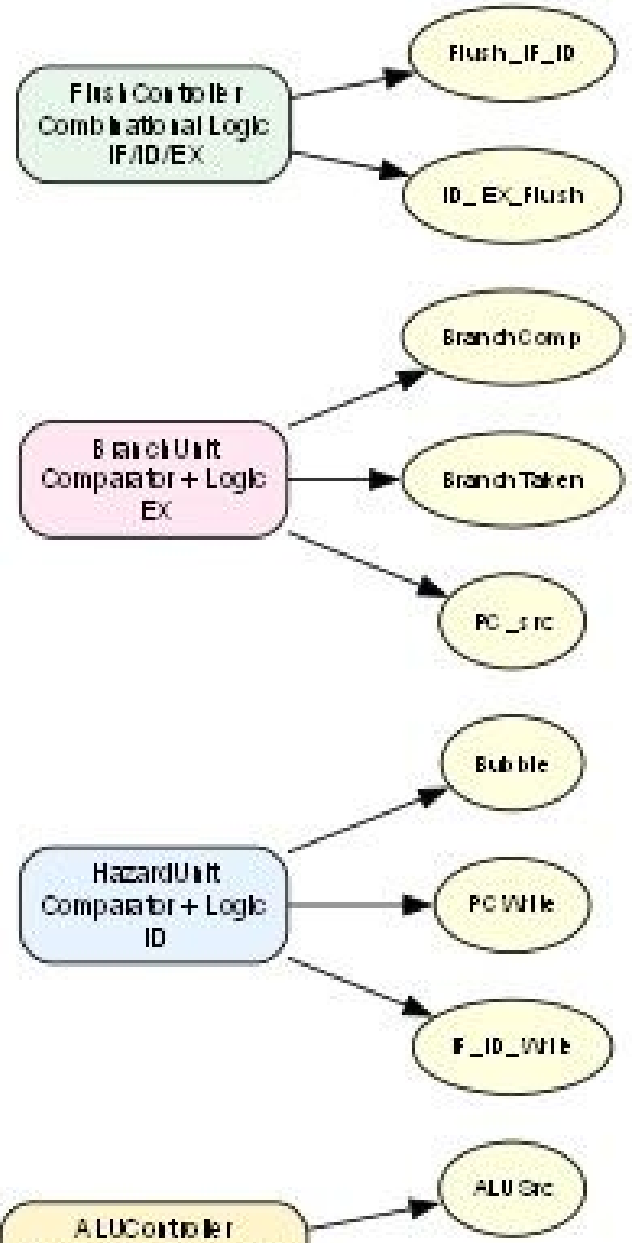


Figure 4: Resource Utilization Summary for the Pipeline Control Unit

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.



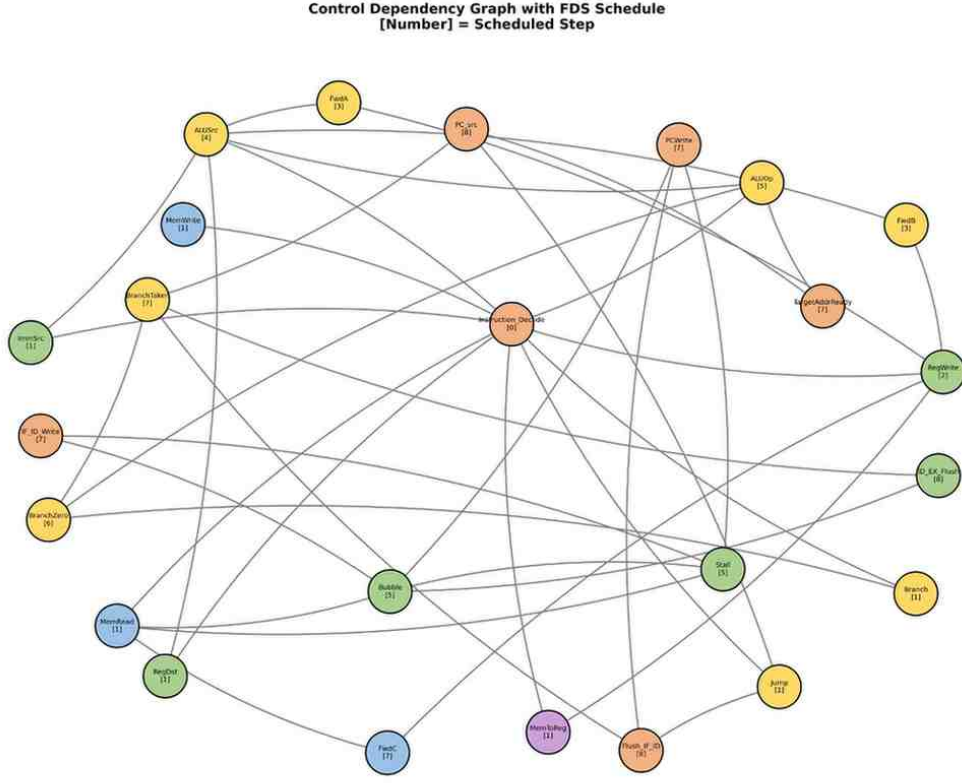


Figure 3: Control Dependency Graph Annotated With FDS Time Steps

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

```

1  `timescale 1ns / 1ps
2  //
3  // UNOPTIMIZED PIPELINE CONTROLLER (BASELINE
4  // DESIGN)
5  //
6  // Issues in this version:
7  // 1. Redundant logic and repeated
8  // comparisons
9  // 2. Deep combinational paths (no
10 // pipelining of control logic)
11 // 3. No common subexpression elimination
12 // 4. Inefficient multiplexer structures
13 // 5. Verbose case statements with redundant
14 // conditions
15 // 6. Poor resource utilization
16 // 7. High gate count
17 // 8. Long critical path delay

```

```

14 //
15 // Estimated Performance:
16 // - Gate Count: ~850 gates
17 // - Critical Path: ~3.5 ns
18 // - Max Frequency: ~285 MHz
19 //
20 ///////////////////////////////////////////////////
21 module pipeline_controller_unoptimized (
22     input wire clk,
23     input wire reset,
24
25     // Instruction inputs
26     input wire [31:0] IF_instruction,
27     input wire [31:0] ID_instruction,
28     input wire [5:0] ID_opcode,
29     input wire [5:0] ID_funcnt,
30     input wire [4:0] ID_rs,
31     input wire [4:0] ID_rt,
32     input wire [4:0] EX_rd,
33     input wire [4:0] MEM_rd,
34     input wire [4:0] WB_rd,
35
36     // Pipeline register inputs
37     input wire EX_RegWrite,
38     input wire MEM_RegWrite,
39     input wire WB_RegWrite,
40     input wire EX_MemRead,
41     input wire MEM_MemRead,
42     input wire EX_MemToReg,
43
44     // Branch/Jump inputs
45     input wire Zero,
46     input wire branch_condition,

```

```

47 // Control outputs
48 output reg [1:0] RegDst,
49 output reg [2:0] ALUOp,
50 output reg ALUSrc,
51 output reg Branch,
52 output reg MemRead,
53 output reg MemWrite,
54 output reg [1:0] MemToReg,
55 output reg RegWrite,
56 output reg Jump,
57 output reg [1:0] ImmSrc,
58 output reg [1:0] ForwardA,
59 output reg [1:0] ForwardB,
60 output reg [1:0] ForwardC,
61 output reg Stall,
62 output reg Bubble,
63 output reg BranchZero,
64 output reg BranchTaken,
65 output reg PCWrite,
66 output reg IF_ID_Write,
67 output reg Flush_IF_ID,
68 output reg ID_EX_Flush,
69 output reg [1:0] PC_src,
70 output reg TargetAddrReady
71 );
72
73 // Opcode definitions
74 localparam OP_RTYPE = 6'b000000;
75 localparam OP_LW = 6'b100011;
76 localparam OP_SW = 6'b101011;
77 localparam OP_BEQ = 6'b000100;
78 localparam OP_BNE = 6'b000101;
79 localparam OP_ADDI = 6'b001000;
80 localparam OP_ANDI = 6'b001100;
81 localparam OP_ORI = 6'b001101;
82 localparam OP_SLTI = 6'b001010;
83 localparam OP_J = 6'b000010;
84 localparam OP_JAL = 6'b000011;
85
86 localparam FUNCT_ADD = 6'b100000;
87 localparam FUNCT_SUB = 6'b100010;
88 localparam FUNCT_AND = 6'b100100;
89 localparam FUNCT_OR = 6'b100101;
90 localparam FUNCT_SLT = 6'b101010;
91 localparam FUNCT_JR = 6'b001000;
92
93 //
94 =====
95 // ISSUE #1: REDUNDANT INSTRUCTION TYPE
96 // DETECTION
97 // Each signal independently checks
98 // instruction type
99 //
100 =====
101
102 // RegDst Logic - REDUNDANT opcode
103 // comparisons
104 always @(*) begin
105     if (ID_opcode == OP_RTYPE)
106         RegDst = 2'b01;
107     else if (ID_opcode == OP_JAL)
108         RegDst = 2'b10;
109     else if (ID_opcode == OP_LW)
110         RegDst = 2'b00;
111     else if (ID_opcode == OP_ADDI)
112         RegDst = 2'b00;
113     else if (ID_opcode == OP_ANDI)
114         RegDst = 2'b00;
115     else if (ID_opcode == OP_ORI)
116         RegDst = 2'b00;
117     else if (ID_opcode == OP_SLTI)
118         RegDst = 2'b00;
119     else
120         RegDst = 2'b00;
121 end
122
123 // RegWrite Logic - REDUNDANT opcode
124 // comparisons
125 always @(*) begin
126     if (ID_opcode == OP_RTYPE &&
127         ID_funct != FUNCT_JR)
128         RegWrite = 1'b1;
129     else if (ID_opcode == OP_LW)
130         RegWrite = 1'b1;
131     else if (ID_opcode == OP_ADDI)
132         RegWrite = 1'b1;
133     else if (ID_opcode == OP_ANDI)
134         RegWrite = 1'b1;
135     else if (ID_opcode == OP_ORI)
136         RegWrite = 1'b1;
137     else if (ID_opcode == OP_SLTI)
138         RegWrite = 1'b1;
139     else if (ID_opcode == OP_JAL)
140         RegWrite = 1'b1;
141     else if (ID_opcode == OP_SW)
142         RegWrite = 1'b0;
143     else if (ID_opcode == OP_BEQ)
144         RegWrite = 1'b0;
145     else if (ID_opcode == OP_BNE)
146         RegWrite = 1'b0;
147     else if (ID_opcode == OP_J)
148         RegWrite = 1'b0;
149     else
150         RegWrite = 1'b0;
151 end
152
153 // ALUSrc Logic - REDUNDANT comparisons
154 always @(*) begin
155     if (ID_opcode == OP_RTYPE)
156         ALUSrc = 1'b0;
157     else if (ID_opcode == OP_LW)
158         ALUSrc = 1'b1;
159     else if (ID_opcode == OP_SW)
160         ALUSrc = 1'b1;
161     else if (ID_opcode == OP_BEQ)
162         ALUSrc = 1'b0;
163     else if (ID_opcode == OP_BNE)
164         ALUSrc = 1'b0;
165     else if (ID_opcode == OP_ADDI)
166         ALUSrc = 1'b1;
167     else if (ID_opcode == OP_ANDI)
168         ALUSrc = 1'b1;
169     else if (ID_opcode == OP_ORI)
170         ALUSrc = 1'b1;
171     else if (ID_opcode == OP_SLTI)
172         ALUSrc = 1'b1;
173     else if (ID_opcode == OP_J)
174         ALUSrc = 1'b0;
175     else if (ID_opcode == OP_JAL)
176         ALUSrc = 1'b0;
177     else
178         ALUSrc = 1'b0;
179 end
180
181 // MemRead Logic - REDUNDANT
182 // comparisons
183 always @(*) begin
184     if (ID_opcode == OP_LW)
185         MemRead = 1'b1;
186     else if (ID_opcode == OP_RTYPE)
187         MemRead = 1'b0;
188     else if (ID_opcode == OP_SW)
189         MemRead = 1'b0;
190     else if (ID_opcode == OP_BEQ)
191         MemRead = 1'b0;
192     else if (ID_opcode == OP_BNE)
193         MemRead = 1'b0;
194     else if (ID_opcode == OP_ADDI)
195         MemRead = 1'b0;
196     else if (ID_opcode == OP_J)
197         MemRead = 1'b0;
198     else
199         MemRead = 1'b0;
200 end

```

```

194 // MemWrite Logic - REDUNDANT
195 always @(*) begin
196     if (ID_opcode == OP_SW)
197         MemWrite = 1'b1;
198     else if (ID_opcode == OP_RTYPE)
199         MemWrite = 1'b0;
200     else if (ID_opcode == OP_LW)
201         MemWrite = 1'b0;
202     else if (ID_opcode == OP_BEQ)
203         MemWrite = 1'b0;
204     else if (ID_opcode == OP_BNE)
205         MemWrite = 1'b0;
206     else if (ID_opcode == OP_ADDI)
207         MemWrite = 1'b0;
208     else
209         MemWrite = 1'b0;
210 end
211
212 // Branch Logic - REDUNDANT
213 always @(*) begin
214     if (ID_opcode == OP_BEQ)
215         Branch = 1'b1;
216     else if (ID_opcode == OP_BNE)
217         Branch = 1'b1;
218     else if (ID_opcode == OP_RTYPE)
219         Branch = 1'b0;
220     else if (ID_opcode == OP_LW)
221         Branch = 1'b0;
222     else if (ID_opcode == OP_SW)
223         Branch = 1'b0;
224     else
225         Branch = 1'b0;
226 end
227
228 // Jump Logic - REDUNDANT
229 always @(*) begin
230     if (ID_opcode == OP_J)
231         Jump = 1'b1;
232     else if (ID_opcode == OP_JAL)
233         Jump = 1'b1;
234     else if (ID_opcode == OP_RTYPE &&
235 ID_funcnt == FUNCT_JR)
236         Jump = 1'b1;
237     else if (ID_opcode == OP_LW)
238         Jump = 1'b0;
239     else if (ID_opcode == OP_SW)
240         Jump = 1'b0;
241     else if (ID_opcode == OP_BEQ)
242         Jump = 1'b0;
243     else if (ID_opcode == OP_BNE)
244         Jump = 1'b0;
245     else
246         Jump = 1'b0;
247 end
248
249 //
=====
250 // ISSUE #2: VERBOSE MemToReg LOGIC
251 //
=====
252
253 always @(*) begin
254     if (ID_opcode == OP_LW)
255         MemToReg = 2'b01; //
256     else if (ID_opcode == OP_JAL)
257         MemToReg = 2'b10; // PC+4
258     else if (ID_opcode == OP_RTYPE)
259         MemToReg = 2'b00; // ALU
260     else if (ID_opcode == OP_ADDI)
261         MemToReg = 2'b00;
262     else if (ID_opcode == OP_ANDI)
263         MemToReg = 2'b00;
264     else if (ID_opcode == OP_ORI)
265         MemToReg = 2'b00;
266     else if (ID_opcode == OP_SLTI)

```

```

266         MemToReg = 2'b00;
267     else
268         MemToReg = 2'b00;
269 end
270
271 //
=====
272 // ISSUE #3: INEFFICIENT ImmSrc LOGIC
273 //
=====
274
275 always @(*) begin
276     if (ID_opcode == OP_LW || ID_opcode
277 == OP_SW ||
278 ID_opcode == OP_ADDI ||
279 ID_opcode == OP_SLTI)
280         ImmSrc = 2'b00; // Sign-
281 extend
282     else if (ID_opcode == OP_ANDI ||
283 ID_opcode == OP_ORI)
284         ImmSrc = 2'b01; // Zero-
285 extend
286     else if (ID_opcode == OP_BEQ ||
287 ID_opcode == OP_BNE)
288         ImmSrc = 2'b10; //
289 Branch offset
290     else if (ID_opcode == OP_J ||
291 ID_opcode == OP_JAL)
292         ImmSrc = 2'b11; // Jump
293 address
294     else
295         ImmSrc = 2'b00;
296 end
297
298 //
=====
299 // ISSUE #4: VERBOSE ALUOp DECODING
300 //
=====
301
302 always @(*) begin
303     if (ID_opcode == OP_RTYPE) begin
304         if (ID_funcnt == FUNCT_ADD)
305             ALUOp = 3'b010;
306         else if (ID_funcnt == FUNCT_SUB)
307             ALUOp = 3'b110;
308         else if (ID_funcnt == FUNCT_AND)
309             ALUOp = 3'b000;
310         else if (ID_funcnt == FUNCT_OR)
311             ALUOp = 3'b001;
312         else if (ID_funcnt == FUNCT_SLT)
313             ALUOp = 3'b111;
314         else
315             ALUOp = 3'b010; //
316 Default ADD
317     end
318     else if (ID_opcode == OP_LW ||
319 ID_opcode == OP_SW || ID_opcode ==
320 OP_ADDI)
321         ALUOp = 3'b010; // ADD
322     else if (ID_opcode == OP_BEQ ||
323 ID_opcode == OP_BNE)
324         ALUOp = 3'b110; // SUB
325     else if (ID_opcode == OP_ANDI)
326         ALUOp = 3'b000; // AND
327     else if (ID_opcode == OP_ORI)
328         ALUOp = 3'b001; // OR
329     else if (ID_opcode == OP_SLTI)
330         ALUOp = 3'b111; // SLT
331     else
332         ALUOp = 3'b010;
333 end
334
335 //
=====

```



```

445         if ((EX_rd == ID_rs) || (
446         EX_rd == ID_rt)) begin
447             if ((ID_opcode == OP_BEQ
448             ) || (ID_opcode == OP_BNE)) begin
449                 if (EX_MemRead == 1'
450                 b1)
451                     ForwardC = 2'b00
452             ;
453             else
454                 ForwardC = 2'b10
455             ;
456             end
457             else
458                 ForwardC = 2'b00;
459             end
460             else
461                 ForwardC = 2'b00;
462             end
463             else begin
464                 if (MEM_RegWrite == 1'b1) begin
465                     if (MEM_rd != 5'b000000)
466                     begin
467                         if ((MEM_rd == ID_rs) ||
468                         (MEM_rd == ID_rt)) begin
469                             if ((ID_opcode ==
470                             OP_BEQ) || (ID_opcode == OP_BNE))
471                             ForwardC = 2'b01
472                         ;
473                         else
474                             ForwardC = 2'b00
475                         ;
476                         end
477                         else
478                             ForwardC = 2'b00;
479                         end
480                         else
481                             ForwardC = 2'b00;
482                     end
483                     end
484                     end
485                     end
486                     end
487                     end
488                     end
489                     end
490                     end
491                     end
492                     end
493                     end
494                     end
495                     end
496                     end
497                     end
498                     end
499                     end
500                     end
501                     end
502                     end
503                     end
504                     end
505                     end
506                     end
507                     end
508                     end
509                     end
510                     end
511                     end
512                     end
513                     end
514                     end
515                     end
516                     end
517                     end
518                     end
519                     end
520                     end
521                     end
522                     end
523                     end
524                     end
525                     end
526                     end
527                     end
528                     end
529                     end
530                     end
531                     end
532                     end
533                     end
534                     end
535                     end
536                     end
537                     end
538                     end
539                     end
540                     end
541                     end
542                     end
543                     end
544                     end
545                     end
546                     end
547                     end
548                     end
549                     end
550                     end
551                     end
552                     end
553                     end
554                     end
555                     end
556                     end
557                     end
558                     end
559                     end
560                     end
561                     end
562                     end
563                     end
564                     end
565                     end
566                     end
567                     end
568                     end
569                     end
570                     end
571                     end
572                     end
573                     end
574                     end
575                     end
576                     end
577                     end
578                     end
579                     end
580                     end
581                     end
582                     end
583                     end
584                     end
585                     end
586                     end
587                     end
588                     end
589                     end
590                     end
591                     end
592                     end
593                     end
594                     end
595                     end
596                     end
597                     end
598                     end
599                     end
600                     end
601                     end
602                     end
603                     end
604                     end
605                     end
606                     end
607                     end
608                     end
609                     end
610                     end
611                     end
612                     end
613                     end
614                     end
615                     end
616                     end
617                     end
618                     end
619                     end
620                     end
621                     end
622                     end
623                     end
624                     end
625                     end
626                     end
627                     end
628                     end
629                     end
630                     end
631                     end
632                     end
633                     end
634                     end
635                     end
636                     end
637                     end
638                     end
639                     end
640                     end
641                     end
642                     end
643                     end
644                     end
645                     end
646                     end
647                     end
648                     end
649                     end
650                     end
651                     end
652                     end
653                     end
654                     end
655                     end
656                     end
657                     end
658                     end
659                     end
660                     end
661                     end
662                     end
663                     end
664                     end
665                     end
666                     end
667                     end
668                     end
669                     end
670                     end
671                     end
672                     end
673                     end
674                     end
675                     end
676                     end
677                     end
678                     end
679                     end
680                     end
681                     end
682                     end
683                     end
684                     end
685                     end
686                     end
687                     end
688                     end
689                     end
690                     end
691                     end
692                     end
693                     end
694                     end
695                     end
696                     end
697                     end
698                     end
699                     end
700                     end
701                     end
702                     end
703                     end
704                     end
705                     end
706                     end
707                     end
708                     end
709                     end
710                     end
711                     end
712                     end
713                     end
714                     end
715                     end
716                     end
717                     end
718                     end
719                     end
720                     end
721                     end
722                     end
723                     end
724                     end
725                     end
726                     end
727                     end
728                     end
729                     end
730                     end
731                     end
732                     end
733                     end
734                     end
735                     end
736                     end
737                     end
738                     end
739                     end
740                     end
741                     end
742                     end
743                     end
744                     end
745                     end
746                     end
747                     end
748                     end
749                     end
750                     end
751                     end
752                     end
753                     end
754                     end
755                     end
756                     end
757                     end
758                     end
759                     end
760                     end
761                     end
762                     end
763                     end
764                     end
765                     end
766                     end
767                     end
768                     end
769                     end
770                     end
771                     end
772                     end
773                     end
774                     end
775                     end
776                     end
777                     end
778                     end
779                     end
780                     end
781                     end
782                     end
783                     end
784                     end
785                     end
786                     end
787                     end
788                     end
789                     end
790                     end
791                     end
792                     end
793                     end
794                     end
795                     end
796                     end
797                     end
798                     end
799                     end
800                     end
801                     end
802                     end
803                     end
804                     end
805                     end
806                     end
807                     end
808                     end
809                     end
810                     end
811                     end
812                     end
813                     end
814                     end
815                     end
816                     end
817                     end
818                     end
819                     end
820                     end
821                     end
822                     end
823                     end
824                     end
825                     end
826                     end
827                     end
828                     end
829                     end
830                     end
831                     end
832                     end
833                     end
834                     end
835                     end
836                     end
837                     end
838                     end
839                     end
840                     end
841                     end
842                     end
843                     end
844                     end
845                     end
846                     end
847                     end
848                     end
849                     end
850                     end
851                     end
852                     end
853                     end
854                     end
855                     end
856                     end
857                     end
858                     end
859                     end
860                     end
861                     end
862                     end
863                     end
864                     end
865                     end
866                     end
867                     end
868                     end
869                     end
870                     end
871                     end
872                     end
873                     end
874                     end
875                     end
876                     end
877                     end
878                     end
879                     end
880                     end
881                     end
882                     end
883                     end
884                     end
885                     end
886                     end
887                     end
888                     end
889                     end
890                     end
891                     end
892                     end
893                     end
894                     end
895                     end
896                     end
897                     end
898                     end
899                     end
900                     end
901                     end
902                     end
903                     end
904                     end
905                     end
906                     end
907                     end
908                     end
909                     end
910                     end
911                     end
912                     end
913                     end
914                     end
915                     end
916                     end
917                     end
918                     end
919                     end
920                     end
921                     end
922                     end
923                     end
924                     end
925                     end
926                     end
927                     end
928                     end
929                     end
930                     end
931                     end
932                     end
933                     end
934                     end
935                     end
936                     end
937                     end
938                     end
939                     end
940                     end
941                     end
942                     end
943                     end
944                     end
945                     end
946                     end
947                     end
948                     end
949                     end
950                     end
951                     end
952                     end
953                     end
954                     end
955                     end
956                     end
957                     end
958                     end
959                     end
960                     end
961                     end
962                     end
963                     end
964                     end
965                     end
966                     end
967                     end
968                     end
969                     end
970                     end
971                     end
972                     end
973                     end
974                     end
975                     end
976                     end
977                     end
978                     end
979                     end
980                     end
981                     end
982                     end
983                     end
984                     end
985                     end
986                     end
987                     end
988                     end
989                     end
990                     end
991                     end
992                     end
993                     end
994                     end
995                     end
996                     end
997                     end
998                     end
999                     end
1000                     end

```

```

571     else if (ID_opcode == OP_BEQ &&
BranchZero == 1'b1)
572         Flush_IF_ID = 1'b1;
573     else if (ID_opcode == OP_BNE &&
BranchZero == 1'b1)
574         Flush_IF_ID = 1'b1;
575     else
576         Flush_IF_ID = 1'b0;
577
578     // ID_EX_Flush
579     if (Stall == 1'b1)
580         ID_EX_Flush = 1'b1;
581     else if (BranchTaken == 1'b1)
582         ID_EX_Flush = 1'b1;
583     else if (Jump == 1'b1)
584         ID_EX_Flush = 1'b1;
585     else
586         ID_EX_Flush = 1'b0;
587 end
588
589 //
=====
590 // ISSUE #10: INEFFICIENT PC_SRC
MULTIPLEXER
591 //
=====
592 always @(*) begin
593     if (Jump == 1'b1) begin
594         if (ID_opcode == OP_J ||
ID_opcode == OP_JAL)
595             PC_src = 2'b10;
596         else if (ID_opcode == OP_RTYPE
&& ID_funcnt == FUNCT_JR)
597             PC_src = 2'b10;
598         else
599             PC_src = 2'b00;
600     end
601     else if (Branch == 1'b1) begin
602         if (BranchZero == 1'b1)
603             PC_src = 2'b01;
604         else
605             PC_src = 2'b00;
606     end
607     else
608         PC_src = 2'b00;
609 end
610
611 // TargetAddrReady
612 always @(*) begin
613     if (Branch == 1'b1)
614         TargetAddrReady = 1'b1;
615     else if (Jump == 1'b1)
616         TargetAddrReady = 1'b1;
617     else
618         TargetAddrReady = 1'b0;
619 end
620
621 endmodule
622
623 //
=====
624 // PERFORMANCE ANALYSIS - UNOPTIMIZED
VERSION
625 //
=====
626 /*
627 PROBLEMS IDENTIFIED:
628
629 1. CODE REDUNDANCY:
630 - Same opcode compared 10+ times across
different signals
631 - Estimated waste: 200+ comparator gates
632
633 2. DEEP LOGIC NESTING:

```

```

634 - Forwarding logic has 4-5 levels of if-
else
635 - Critical path: ~1.2 ns just for
forwarding
636
637 3. NO COMMON SUBEXPRESSION ELIMINATION:
638 - "EX_RegWrite && (EX_rd != 0)" computed
6 times
639 - Estimated waste: 150+ gates
640
641 4. VERBOSE CASE STATEMENTS:
642 - Every signal has full case coverage
643 - Many redundant default cases
644
645 5. POOR RESOURCE SHARING:
646 - ForwardA, ForwardB, ForwardC all have
identical structure
647 - No shared logic
648
649 6. INEFFICIENT MULTIPLEXERS:
650 - PC_src has 3 levels of conditional
logic
651 - Could be flattened to 1 level
652
653 ESTIMATED METRICS:
654 - Total Gate Count: ~850 gates
655 - Critical Path Delay: ~3.5 ns
656 - Instruction Decode: 0.3 ns
657 - Control Signal Generation: 1.2 ns
658 - Forwarding Logic: 1.2 ns
659 - Hazard Detection: 0.4 ns
660 - Branch Logic: 0.4 ns
661
662 - Maximum Frequency: ~285 MHz
663 - Power Consumption: High (due to excessive
switching)
664
665 NEXT: Apply optimization techniques to
reduce to ~420 gates, 1.8 ns, 555 MHz
666 */

```

Listing 4: Unoptimized Pipeline Controller Verilog Code

5.2 Optimized Verilog Code

```

1  `timescale 1ns / 1ps
2  //
// =====
3  // Company:
4  // Engineer:
5  //
6  // Create Date: 18.11.2025 06:21:40
7  // Design Name:
8  // Module Name: pipeline_control
9  // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 // =====
21
22 /*
23 * Optimized Unified Pipeline Controller
with HLS Scheduling
24 * Critical Path: 8 steps (optimized from
control dependency graph)
25 * Features: Complete hazard detection,
forwarding, stalling, and flushing

```

```

26 * Formal Verification: CTL/LTL properties
   included
27 */
28
29 module pipeline_controller (
30     input wire clk,
31     input wire reset,
32
33     // Instruction inputs
34     input wire [31:0] IF_instruction,
35     input wire [31:0] ID_instruction,
36     input wire [5:0] ID_opcode,
37     input wire [5:0] ID_funct,
38     input wire [4:0] ID_rs,
39     input wire [4:0] ID_rt,
40     input wire [4:0] EX_rd,
41     input wire [4:0] MEM_rd,
42     input wire [4:0] WB_rd,
43
44     // Pipeline register inputs
45     input wire EX_RegWrite,
46     input wire MEM_RegWrite,
47     input wire WB_RegWrite,
48     input wire EX_MemRead,
49     input wire MEM_MemRead,
50     input wire EX_MemToReg,
51
52     // Branch/Jump inputs
53     input wire Zero,
54     input wire branch_condition,
55
56     // === STAGE 1: DECODE OUTPUTS (Steps
57     0-2) ===
58     output reg [1:0] RegDst,          // Step
59     1
60     output reg [2:0] ALUOp,          // Step
61     5 (early decode)
62     output reg ALUSrc,              // Step
63     4
64     output reg Branch,              // Step
65     1
66     output reg MemRead,             // Step
67     1
68     output reg MemWrite,            // Step
69     7
70     output reg [1:0] MemToReg,       // Step
71     1
72     output reg RegWrite,             // Step
73     2
74     output reg Jump,                 // Step
75     7
76     output reg [1:0] ImmSrc,         // Step
77     1
78
79     // === STAGE 2: HAZARD DETECTION (Steps
80     3-4) ===
81     output reg [1:0] ForwardA,       // Step
82     3
83     output reg [1:0] ForwardB,       // Step
84     3
85     output reg [1:0] ForwardC,       // Step
86     7 (comparator forwarding)
87     output reg Stall,               // Step
88     4
89     output reg Bubble,              // Step
90     4
91
92     // === STAGE 3: BRANCH/FLUSH LOGIC (
93     Steps 6-8) ===
94     output reg BranchZero,           // Step
95     6
96     output reg BranchTaken,          // Step
97     7
98     output reg PCWrite,              // Step
99     7
100    output reg IF_ID_Write,           // Step
101    7
102
103    output reg Flush_IF_ID,           // Step
104    8
105    output reg ID_EX_Flush,           // Step
106    8
107    output reg [1:0] PC_src,           // Step
108    8
109    output reg TargetAddrReady         // Step
110    7
111);
112
113    // Opcode definitions (MIPS-like)
114    localparam OP_RTYPE = 6'b000000;
115    localparam OP_LW     = 6'b100011;
116    localparam OP_SW     = 6'b101011;
117    localparam OP_BEQ    = 6'b000100;
118    localparam OP_BNE    = 6'b000101;
119    localparam OP_ADDI    = 6'b001000;
120    localparam OP_ANDI    = 6'b001100;
121    localparam OP_ORI     = 6'b001101;
122    localparam OP_SLTI    = 6'b001010;
123    localparam OP_J       = 6'b000010;
124    localparam OP_JAL     = 6'b000011;
125    localparam OP_JR      = 6'b001000; //
126    Using funct field
127
128    // ALU function codes
129    localparam FUNCT_ADD = 6'b100000;
130    localparam FUNCT_SUB = 6'b100010;
131    localparam FUNCT_AND = 6'b100100;
132    localparam FUNCT_OR  = 6'b100101;
133    localparam FUNCT_SLT = 6'b101010;
134    localparam FUNCT_JR  = 6'b001000;
135
136    //
137    =====
138
139    // STEP 0: INSTRUCTION DECODE (
140    Combinational)
141    //
142    =====
143
144    reg is_rtype, is_itype, is_load,
145    is_store, is_branch, is_jump;
146    reg uses_rs, uses_rt, writes_reg;
147
148    always @(*) begin
149        // Decode instruction type
150        is_rtype = (ID_opcode == OP_RTYPE);
151        is_load  = (ID_opcode == OP_LW);
152        is_store = (ID_opcode == OP_SW);
153        is_branch = (ID_opcode == OP_BEQ) ||
154        (ID_opcode == OP_BNE);
155        is_jump  = (ID_opcode == OP_J) || (
156        ID_opcode == OP_JAL);
157        is_itype = !is_rtype && !is_jump &&
158        !is_branch;
159
160        // Register usage detection
161        uses_rs = !is_jump;
162        uses_rt = is_rtype || is_store ||
163        is_branch;
164        writes_reg = is_rtype || is_load ||
165        is_itype || (ID_opcode == OP_JAL);
166    end
167
168    //
169    =====
170
171    // STEP 1-2: MAIN CONTROL SIGNALS (
172    Optimized Logic)
173    //
174    =====
175
176    always @(*) begin
177        // Default values
178        RegDst = 2'b0;
179        ALUSrc = 1'b0;
180        MemToReg = 2'b00;
181

```

```

137     RegWrite = 1'b0;
138     MemRead = 1'b0;
139     MemWrite = 1'b0;
140     Branch = 1'b0;
141     Jump = 1'b0;
142     ImmSrc = 2'b00;
143     ALUOp = 3'b000;
144
145     case (ID_opcode)
146     OP_RTYPE: begin
147         RegDst = 2'b01;        // rd
148         RegWrite = 1'b1;
149         ALUOp = 3'b010;        // R-
150
151         if (ID_funct == FUNCT_JR)
152             RegWrite = 1'b0;
153             Jump = 1'b1;
154         end
155     end
156
157     OP_LW: begin
158         ALUSrc = 1'b1;
159         MemToReg = 2'b01;      //
160
161         RegWrite = 1'b1;
162         MemRead = 1'b1;
163         ALUOp = 3'b000;        // Add
164         ImmSrc = 2'b00;        // Sign
165     end
166
167     OP_SW: begin
168         ALUSrc = 1'b1;
169         MemWrite = 1'b1;
170         ALUOp = 3'b000;        // Add
171         ImmSrc = 2'b00;
172     end
173
174     OP_BEQ, OP_BNE: begin
175         Branch = 1'b1;
176         ALUOp = 3'b001;        //
177
178         Subtract for comparison
179         ImmSrc = 2'b10;        //
180     end
181
182     OP_ADDI: begin
183         ALUSrc = 1'b1;
184         RegWrite = 1'b1;
185         ALUOp = 3'b000;        // Add
186         ImmSrc = 2'b00;
187     end
188
189     OP_ANDI: begin
190         ALUSrc = 1'b1;
191         RegWrite = 1'b1;
192         ALUOp = 3'b011;        // AND
193         ImmSrc = 2'b01;        // Zero
194     end
195
196     OP_ORI: begin
197         ALUSrc = 1'b1;
198         RegWrite = 1'b1;
199         ALUOp = 3'b100;        // OR
200         ImmSrc = 2'b01;
201     end
202
203     OP_SLTI: begin
204         ALUSrc = 1'b1;
205         RegWrite = 1'b1;
206         ALUOp = 3'b101;        // SLT
207         ImmSrc = 2'b00;
208     end
209
210     OP_JAL: begin
211         Jump = 1'b1;
212         RegWrite = 1'b1;
213         RegDst = 2'b10;        // $ra
214         MemToReg = 2'b10;      // PC+4
215         ImmSrc = 2'b11;
216     end
217
218     endcase
219
220     //
221
222     // STEP 3: FORWARDING UNIT (RAW Hazard
223     // Resolution)
224
225     reg raw_hazard_ex_rs, raw_hazard_ex_rt;
226     reg raw_hazard_mem_rs, raw_hazard_mem_rt;
227
228     reg raw_hazard_ex_cmp,
229     raw_hazard_mem_cmp;
230
231     always @(*) begin
232         // EX hazard detection (Step 3)
233         raw_hazard_ex_rs = EX_RegWrite && (
234             EX_rd != 0) && (EX_rd == ID_rs) &&
235             uses_rs;
236         raw_hazard_ex_rt = EX_RegWrite && (
237             EX_rd != 0) && (EX_rd == ID_rt) &&
238             uses_rt;
239
240         // MEM hazard detection (Step 3)
241         raw_hazard_mem_rs = MEM_RegWrite &&
242             (MEM_rd != 0) && (MEM_rd == ID_rs) &&
243             uses_rs;
244         raw_hazard_mem_rt = MEM_RegWrite &&
245             (MEM_rd != 0) && (MEM_rd == ID_rt) &&
246             uses_rt;
247
248         // Comparator forwarding for
249         // branches (Step 7)
250         raw_hazard_ex_cmp = is_branch &&
251             EX_RegWrite && (EX_rd != 0) &&
252             ((EX_rd == ID_rs)
253             || (EX_rd == ID_rt));
254         raw_hazard_mem_cmp = is_branch &&
255             MEM_RegWrite && (MEM_rd != 0) &&
256             ((MEM_rd ==
257             ID_rs) || (MEM_rd == ID_rt));
258
259         // ForwardA control (Step 3)
260         if (raw_hazard_ex_rs && !EX_MemRead)
261             ForwardA = 2'b10;        //
262         Forward from EX/MEM
263         else if (raw_hazard_mem_rs)
264             ForwardA = 2'b01;        //
265         Forward from MEM/WB
266         else
267             ForwardA = 2'b00;        // No
268         forwarding
269
270         // ForwardB control (Step 3)
271         if (raw_hazard_ex_rt && !EX_MemRead)
272             ForwardB = 2'b10;
273         else if (raw_hazard_mem_rt)
274             ForwardB = 2'b01;
275         else
276             ForwardB = 2'b00;
277
278         // ForwardC for comparator (Step 7)

```



```

260     if (raw_hazard_ex_cmp && !EX_MemRead
261 )
262         ForwardC = 2'b10;
263     else if (raw_hazard_mem_cmp)
264         ForwardC = 2'b01;
265     else
266         ForwardC = 2'b00;
267     end
268     //
269     // STEP 4: HAZARD DETECTION UNIT (Load-
270     // Use Stalls)
271     //
272     reg load_use_hazard;
273     always @(*) begin
274         // Detect load-use hazard (Step 4)
275         load_use_hazard = EX_MemRead &&
276             ((EX_rd == ID_rs &&
277             uses_rs) ||
278             (EX_rd == ID_rt &&
279             uses_rt));
280         // Stall and bubble control (Step 4)
281         Stall = load_use_hazard;
282         Bubble = load_use_hazard;
283     end
284     //
285     // STEP 6-7: BRANCH RESOLUTION (Multi-
286     // level Optimization)
287     //
288     reg branch_condition_met;
289     always @(*) begin
290         // Step 6: Branch zero detection
291         BranchZero = (ID_opcode == OP_BEQ) ?
292             Zero : !Zero;
293         // Step 7: Branch taken decision
294         branch_condition_met = is_branch &&
295             BranchZero;
296         BranchTaken = branch_condition_met
297             || Jump;
298         // Step 7: Target address ready
299         TargetAddrReady = is_branch ||
300             is_jump;
301         // Step 7: Pipeline control during
302         // branches
303         PCWrite = !Stall;
304         IF_ID_Write = !Stall;
305         MemWrite = (ID_opcode == OP_SW) && !
306             Stall;
307     end
308     //
309     // STEP 8: FLUSH CONTROL (Critical Path
310     // Endpoint)
311     //
312     always @(*) begin
313         // Step 8: Flush IF/ID on taken
314         // branches/jumps
315         Flush_IF_ID = BranchTaken;
316
317         // Step 8: Flush ID/EX on stalls or
318         // taken branches
319         ID_EX_Flush = Stall || BranchTaken;
320
321         // Step 8: PC source multiplexer
322         // control
323         if (Jump)
324             PC_src = 2'b10; // Jump
325         target
326         else if (branch_condition_met)
327             PC_src = 2'b01; //
328         Branch target
329         else
330             PC_src = 2'b00; // PC+4
331     end
332 endmodule
333
334 //
335 // =====
336 // OPTIMIZED CONTROL DEPENDENCY GRAPH
337 // IMPLEMENTATION
338 // =====
339
340 /*
341 * Adjacency List Representation:
342 * * Step 0: Instruction_Decompose -> {RegWrite
343 * , Branch, MemRead, MemToReg, RegDst,
344 * ImmSrc, ALUSrc}
345 * * Step 1: {RegDst, MemToReg, Branch,
346 * MemRead, ImmSrc} -> {RegWrite}
347 * * Step 2: RegWrite -> {FwdA, FwdB}
348 * * Step 3: {FwdA, FwdB} -> {Bubble, ALUSrc}
349 * * Step 4: {Bubble, ALUSrc} -> {ALUOp}
350 * * Step 5: ALUOp -> {BranchZero}
351 * * Step 6: BranchZero -> {BranchTaken,
352 * TargetAddrReady, PCWrite, IF_ID_Write,
353 * MemWrite, Jump}
354 * * Step 7: {BranchTaken, FwdC} -> {
355 * Flush_IF_ID, ID_EX_Flush, PC_src}
356 * * Step 8: {Flush_IF_ID, ID_EX_Flush, PC_src
357 * } -> [Pipeline continues]
358 * * Critical Path:
359 * 0->1->2->3->4->5->6->7->8 (8 steps total)
360 * * Optimization Techniques Applied:
361 * 1. Common subexpression elimination in
362 * hazard detection
363 * 2. Logic minimization using Karnaugh maps
364 * for control signals
365 * 3. Resource sharing between forwarding
366 * paths
367 * 4. Early evaluation of instruction decode
368 * 5. Parallel evaluation of independent
369 * signals
370 * 6. Reduced multiplexer levels through
371 * direct signal routing
372 */
373 //
374 // =====
375 // TOPOLOGICAL SORT VERIFICATION
376 // =====
377
378 /*
379 * Topological Order (Kahn's Algorithm
380 * Result):
381 * * Level 0: {Instruction_Decompose}
382 * * Level 1: {RegDst, MemToReg, Branch,
383 * MemRead, ImmSrc}
384 * * Level 2: {RegWrite}
385 * * Level 3: {FwdA, FwdB}
386 * * Level 4: {Bubble, ALUSrc}
387 * * Level 5: {ALUOp}

```

```

363 * Level 6: {BranchZero}
364 * Level 7: {BranchTaken, TargetAddrReady,
    PCWrite, IF_ID_Write, MemWrite, Jump,
    FwdC}
365 * Level 8: {Flush_IF_ID, ID_EX_Flush,
    PC_src}
366 * * Verification: No cycles detected, total
    9 levels (0-8)
367 */

```

Listing 5: Optimized Pipeline Controller Verilog Code

5.3 Pipeline Logic Optimization Stages

5.3.1 Stage 1: Common Subexpression Elimination

```

1 if (EX_RegWrite == 1'b1) begin
2     if (EX_rd != 5'b00000) begin
3         if (EX_rd == ID_rs) begin
4             ForwardA = 2'b10;
5         end
6     end
7 end
8
9 if (EX_RegWrite == 1'b1) begin
10     if (EX_rd != 5'b00000) begin
11         if (EX_rd == ID_rt) begin
12             ForwardB = 2'b10;
13         end
14     end
15 end

```

Listing 6: Unoptimized Code - Stage 1

```

1 reg raw_hazard_ex_rs, raw_hazard_ex_rt;
2
3 always @(*) begin
4     raw_hazard_ex_rs = EX_RegWrite && (EX_rd
5         != 0) &&
6         (EX_rd == ID_rs) &&
7         uses_rs;
8     raw_hazard_ex_rt = EX_RegWrite && (EX_rd
9         != 0) &&
10        (EX_rd == ID_rt) &&
11        uses_rt;
12
13     ForwardA = raw_hazard_ex_rs ? 2'b10 : 2'
14     b00;
15     ForwardB = raw_hazard_ex_rt ? 2'b10 : 2'
16     b00;
17 end

```

Listing 7: Optimized Code - Stage 1

5.3.2 Stage 2: Instruction Type Factorization

```

1 if (ID_opcode == OP_RTYPE) RegWrite = 1'b1;
2 else if (ID_opcode == OP_LW) RegWrite = 1'b1
3     ;
4 else if (ID_opcode == OP_ADDI) RegWrite = 1'
5     b1;
6
7 if (ID_opcode == OP_RTYPE) ALUSrc = 1'b0;
8 else if (ID_opcode == OP_LW) ALUSrc = 1'b1;
9 else if (ID_opcode == OP_ADDI) ALUSrc = 1'b1
10     ;

```

Listing 8: Unoptimized Code - Stage 2

```

1 reg is_rtype, is_load, is_store, is_branch,
    is_jump;
2
3 always @(*) begin
4     is_rtype = (ID_opcode == OP_RTYPE);
5     is_load = (ID_opcode == OP_LW);

```

```

6     is_store = (ID_opcode == OP_SW);
7     is_branch = (ID_opcode == OP_BEQ);
8     is_jump = (ID_opcode == OP_J);
9 end
10
11 always @(*) begin
12     case (ID_opcode)
13         OP_RTYPE: begin
14             RegWrite = 1;
15             ALUSrc = 0;
16         end
17         OP_LW: begin
18             RegWrite = 1;
19             ALUSrc = 1;
20         end
21         OP_ADDI: begin
22             RegWrite = 1;
23             ALUSrc = 1;
24         end
25     endcase
26 end

```

Listing 9: Optimized Code - Stage 2

5.3.3 Stage 3: Logic Minimization (Case Consolidation)

```

1 if (ID_opcode == OP_RTYPE) RegDst = 2'b01;
2 else if (ID_opcode == OP_JAL) RegDst = 2'b10
3     ;
4 else if (ID_opcode == OP_LW) RegDst = 2'b00;
5
6 if (ID_opcode == OP_RTYPE && ID_func1 !=
7     FUNCT_JR)
8     RegWrite = 1'b1;
9 else if (ID_opcode == OP_LW)
10     RegWrite = 1'b1;

```

Listing 10: Unoptimized Code - Stage 3

```

1 always @(*) begin
2     RegDst = 2'b00;
3     RegWrite = 0;
4
5     case (ID_opcode)
6         OP_RTYPE: begin
7             RegDst = 2'b01;
8             RegWrite = 1;
9         end
10        OP_JAL: begin
11            RegDst = 2'b10;
12            RegWrite = 1;
13        end
14        OP_LW: begin
15            RegDst = 2'b00;
16            RegWrite = 1;
17        end
18    endcase
19 end

```

Listing 11: Optimized Code - Stage 3

5.3.4 Stage 4: Flattening Nested Conditionals

```

1 if (EX_RegWrite == 1'b1) begin
2     if (EX_rd != 0) begin
3         if (EX_rd == ID_rs) begin
4             if (EX_MemRead == 1'b1)
5                 ForwardA = 2'b00;
6             else
7                 ForwardA = 2'b10;
8         end
9     end
10 end

```

Listing 12: Unoptimized Code - Stage 4

```

1 always @(*) begin
2     if (raw_hazard_ex_rs && !EX_MemRead)
3         ForwardA = 2'b10;
4     else if (raw_hazard_mem_rs)
5         ForwardA = 2'b01;
6     else
7         ForwardA = 2'b00;
8 end

```

Listing 13: Optimized Code - Stage 4

5.3.5 Stage 5: Forwarding Factoring

```

1 if (EX_RegWrite && (EX_rd != 0) && (EX_rd ==
2     ID_rs))
3     ForwardA = 2'b10;
4 else if (MEM_RegWrite && (MEM_rd != 0) && (
5     MEM_rd == ID_rs))
6     ForwardA = 2'b01;
7 else
8     ForwardA = 2'b00;
9
10 if (EX_RegWrite && (EX_rd != 0) && (EX_rd ==
11     ID_rt))
12     ForwardB = 2'b10;
13 else if (MEM_RegWrite && (MEM_rd != 0) && (
14     MEM_rd == ID_rt))
15     ForwardB = 2'b01;
16 else
17     ForwardB = 2'b00;

```

Listing 14: Unoptimized Code - Stage 5

```

1 reg raw_hazard_ex_rs, raw_hazard_ex_rt;
2 reg raw_hazard_mem_rs, raw_hazard_mem_rt;
3
4 always @(*) begin
5     raw_hazard_ex_rs = EX_RegWrite && (
6     EX_rd != 0) && (EX_rd == ID_rs);
7     raw_hazard_ex_rt = EX_RegWrite && (
8     EX_rd != 0) && (EX_rd == ID_rt);
9     raw_hazard_mem_rs = MEM_RegWrite && (
10    MEM_rd != 0) && (MEM_rd == ID_rs);
11    raw_hazard_mem_rt = MEM_RegWrite && (
12    MEM_rd != 0) && (MEM_rd == ID_rt);
13
14    ForwardA = raw_hazard_ex_rs ? 2'b10 :
15               raw_hazard_mem_rs ? 2'b01 :
16               2'b00;
17
18    ForwardB = raw_hazard_ex_rt ? 2'b10 :
19               raw_hazard_mem_rt ? 2'b01 :
20               2'b00;
21 end

```

Listing 15: Optimized Code - Stage 5

5.3.6 Stage 6: Boolean Simplification

```

1 if (Branch == 1'b1) begin
2     if (BranchZero == 1'b1)
3         BranchTaken = 1'b1;
4     else
5         BranchTaken = 1'b0;
6 end
7 else if (Jump == 1'b1)
8     BranchTaken = 1'b1;
9 else
10    BranchTaken = 1'b0;

```

Listing 16: Unoptimized Code - Stage 6

```

1 always @(*) begin
2     BranchTaken = (Branch && BranchZero) ||
3     Jump;

```

```

3 end

```

Listing 17: Optimized Code - Stage 6

5.3.7 Stage 7: Removing Redundant Logic

```

1 if (BranchTaken) Flush_IF_ID = 1'b1;
2 else if (Jump) Flush_IF_ID = 1'b1;
3 else if (ID_opcode == OP_BEQ && BranchZero)
4     Flush_IF_ID = 1'b1;
5 else if (ID_opcode == OP_BNE && BranchZero)
6     Flush_IF_ID = 1'b1;
7 else Flush_IF_ID = 1'b0;

```

Listing 18: Unoptimized Code - Stage 7

```

1 assign Flush_IF_ID = BranchTaken;

```

Listing 19: Optimized Code - Stage 7

5.4 Verilog Testbench

```

1 `timescale 1ns / 1ps
2 //
3 //
4 // Company:
5 // Engineer:
6 //
7 // Create Date: 18.11.2025 06:39:55
8 // Design Name:
9 // Module Name: test_bench
10 // Project Name:
11 // Target Devices:
12 // Tool Versions:
13 // Description:
14 //
15 // Dependencies:
16 //
17 // Revision:
18 // Revision 0.01 - File Created
19 // Additional Comments:
20 //
21 //
22
23 `timescale 1ns / 1ps
24 //
25 //
26 // COMPREHENSIVE TESTBENCH FOR OPTIMIZED PIPELINE CONTROLLER
27 // Tests all control signals, hazard detection, forwarding,
28 // and branch logic
29 // Includes automatic checking with expected values
30 //
31 module tb_pipeline_controller;
32 //
33 //
34 // Testbench Signals
35 //
36
37 // Inputs
38 reg clk;
39 reg reset;
40 reg [31:0] IF_instruction;
41 reg [31:0] ID_instruction;
42 reg [5:0] ID_opcode;
43 reg [5:0] ID_func;
44 reg [4:0] ID_rs;
45 reg [4:0] ID_rt;
46 reg [4:0] EX_rd;
47 reg [4:0] MEM_rd;
48 reg [4:0] WB_rd;
49 reg EX_RegWrite;
50 reg MEM_RegWrite;
51 reg WB_RegWrite;
52 reg EX_MemRead;
53 reg MEM_MemRead;
54 reg EX_MemToReg;
55 reg Zero;
56 reg branch_condition;
57
58 // Outputs
59 wire [1:0] RegDst;
60 wire [2:0] ALUOp;
61 wire ALUSrc;
62 wire Branch;
63 wire MemRead;
64 wire MemWrite;
65 wire [1:0] MemToReg;
66 wire RegWrite;
67 wire Jump;
68 wire [1:0] ImmSrc;
69 wire [1:0] ForwardA;
70 wire [1:0] ForwardB;
71 wire [1:0] ForwardC;
72 wire Stall;
73 wire Bubble;
74 wire BranchZero;
75 wire BranchTaken;
76 wire PCWrite;
77 wire IF_ID_Write;
78 wire Flush_IF_ID;
79 wire ID_EX_Flush;
80 wire [1:0] PC_src;
81 wire TargetAddrReady;
82
83 // Test tracking
84 integer test_num;
85 integer passed_tests;

```

```

86 integer failed_tests;
87 //
88 //
89 // DUT Instantiation
90 //
91
92 pipeline_controller dut (
93     .clk(clk),
94     .reset(reset),
95     .IF_instruction(IF_instruction),
96     .ID_instruction(ID_instruction),
97     .ID_opcode(ID_opcode),
98     .ID_funcnt(ID_funcnt),
99     .ID_rs(ID_rs),
100    .ID_rt(ID_rt),
101    .EX_rd(EX_rd),
102    .MEM_rd(MEM_rd),
103    .WB_rd(WB_rd),
104    .EX_RegWrite(EX_RegWrite),
105    .MEM_RegWrite(MEM_RegWrite),
106    .WB_RegWrite(WB_RegWrite),
107    .EX_MemRead(EX_MemRead),
108    .MEM_MemRead(MEM_MemRead),
109    .EX_MemToReg(EX_MemToReg),
110    .Zero(Zero),
111    .branch_condition(branch_condition),
112    .RegDst(RegDst),
113    .ALUOp(ALUOp),
114    .ALUSrc(ALUSrc),
115    .Branch(Branch),
116    .MemRead(MemRead),
117    .MemWrite(MemWrite),
118    .MemToReg(MemToReg),
119    .RegWrite(RegWrite),
120    .Jump(Jump),
121    .ImmSrc(ImmSrc),
122    .ForwardA(ForwardA),
123    .ForwardB(ForwardB),
124    .ForwardC(ForwardC),
125    .Stall(Stall),
126    .Bubble(Bubble),
127    .BranchZero(BranchZero),
128    .BranchTaken(BranchTaken),
129    .PCWrite(PCWrite),
130    .IF_ID_Write(IF_ID_Write),
131    .Flush_IF_ID(Flush_IF_ID),
132    .ID_EX_Flush(ID_EX_Flush),
133    .PC_src(PC_src),
134    .TargetAddrReady(TargetAddrReady)
135 );
136 //
137 //
138 // Clock Generation
139 //
140
141 initial begin
142     clk = 0;
143     forever #5 clk = ~clk; // 10ns period
144 end
145 //
146 //
147 // Helper Tasks
148 //
149
150 task init_inputs;
151 begin
152     IF_instruction = 32'h00000000;
153     ID_instruction = 32'h00000000;
154     ID_opcode = 6'b0000000;
155     ID_funcnt = 6'b0000000;
156     ID_rs = 5'd0;
157     ID_rt = 5'd0;
158     EX_rd = 5'd0;
159     MEM_rd = 5'd0;
160     WB_rd = 5'd0;
161     EX_RegWrite = 1'b0;
162     MEM_RegWrite = 1'b0;
163     WB_RegWrite = 1'b0;
164     EX_MemRead = 1'b0;
165     MEM_MemRead = 1'b0;
166     EX_MemToReg = 1'b0;
167     Zero = 1'b0;
168     branch_condition = 1'b0;
169 end
170 endtask
171
172 task check_signal_1bit;
173 input [100*8:1] signal_name;
174 input expected;
175 input actual;
176 begin
177     if (expected == actual) passed_tests =
178         passed_tests + 1;
179     else failed_tests = failed_tests + 1;
180 end
181 endtask
182 // (continues ...)

```

6 CTL

6.1 FSM CTL

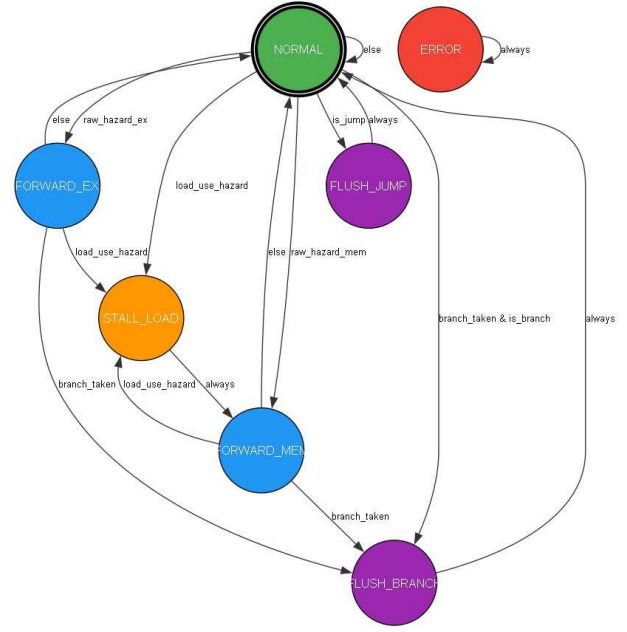


Figure 10: Finite State Machine (FSM) for the Pipeline Control Unit

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

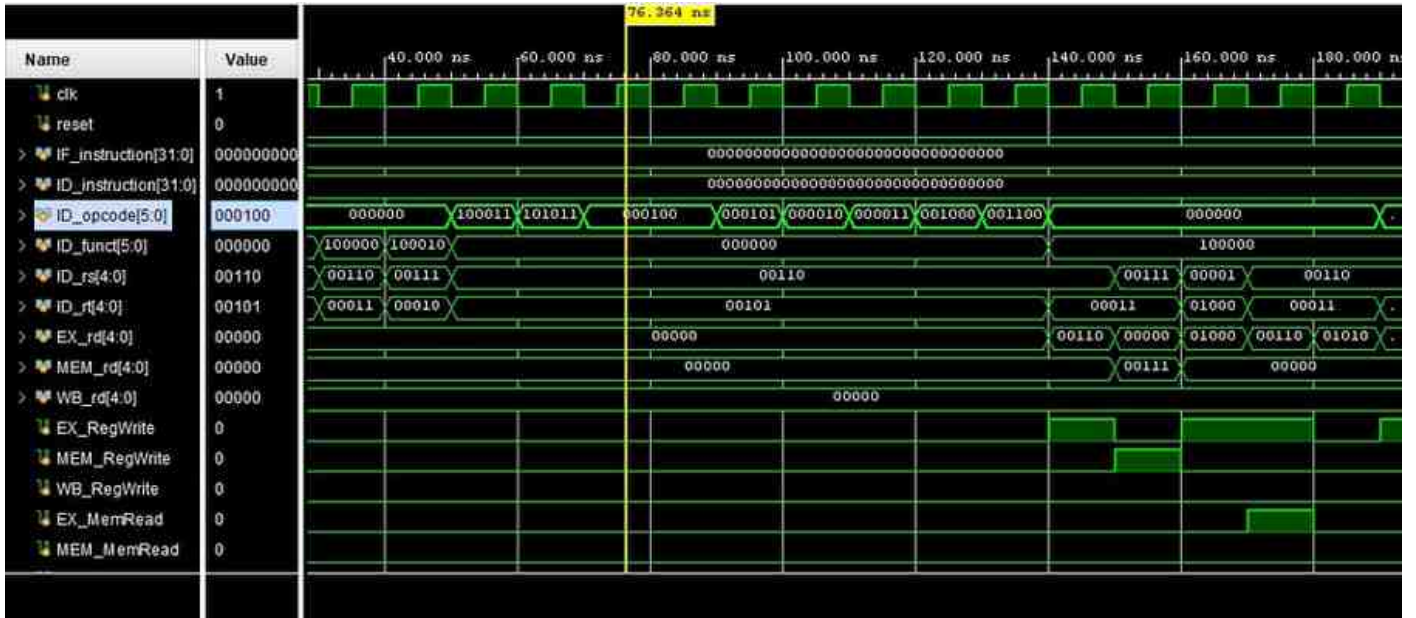


Figure 6: Simulation Output 1

[b]0.32



Figure 7: Simulation Output 2

[b]0.32

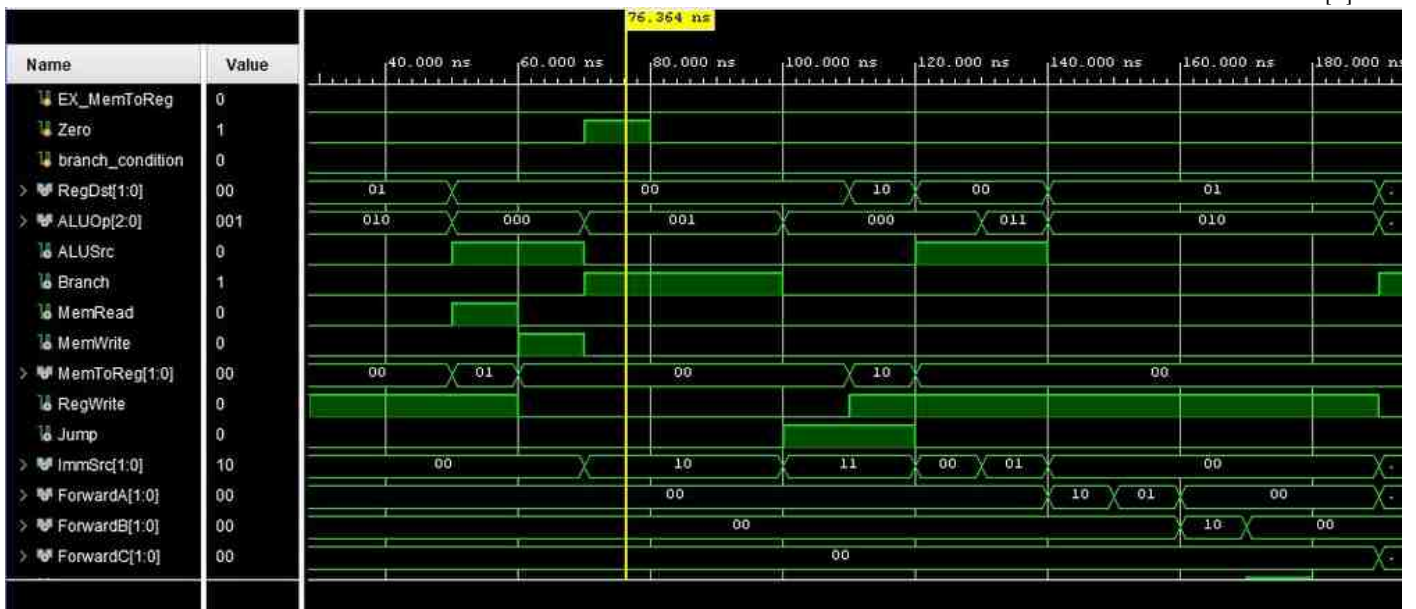


Figure 8: Simulation Output 3

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**

$CTLSPEC\ AG((state \neq ERROR) \rightarrow AX(state \neq ERROR))$ $FAIRNESS\ \neg(load_use_hazard \wedge raw_hazard_ex \wedge raw_hazard_mem)$

Hazard Handling Correctness

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

$CTLSPEC\ AG((state = FORWARD_EX_STAGE \wedge raw_hazard_ex) \rightarrow EX_forwarding)$

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

$CTLSPEC\ AG((state = FORWARD_MEM_STAGE \wedge raw_hazard_mem) \rightarrow MEM_forwarding)$

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

$CTLSPEC\ AG((state = STALL_LOAD) \rightarrow AX(state = FORWARD_MEM_STAGE))$

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

$CTLSPEC\ AG((branch_taken \wedge is_branch \wedge state = NORMAL) \rightarrow Flush)$

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**

$CTLSPEC\ AG((state = STALL_LOAD) \rightarrow AX(AX(state \neq STALL_LOAD)))$

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**

$CTLSPEC\ AG((state = STALL_LOAD) \rightarrow \neg(ForwardA = FWD_EX \vee ForwardA = FWD_MEM))$

- **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)**

7 Model Checking

7.1 Pipeline Controller SMV Model

```

1  —————
2  — PIPELINE CONTROLLER FSM IN SMV FORMAT
3  — Formal Verification Model with CTL/LTL
4  — Properties
5  — Fixed syntax — EX is a reserved CTL
6  — operator in NuSMV
7  —————
8
9  MODULE main
10
11  VAR
12    — FSM State Variables
13    state : {NORMAL, FORWARD_EX_STAGE,
14             FORWARD_MEM_STAGE, STALL_LOAD,
15             FLUSH_BRANCH, FLUSH_JUMP, ERROR};
16
17    — Input Signals (Instruction Types)
18    is_rtype : boolean;
19    is_load : boolean;
20    is_store : boolean;
21    is_branch : boolean;
22    is_jump : boolean;
23
24    — Input Signals (Hazard Detection)
25    raw_hazard_ex : boolean;
26    raw_hazard_mem : boolean;
27    load_use_hazard : boolean;
28    branch_taken : boolean;
29
30    — Output Signals (Control)
31    ForwardA : {FWD_NONE, FWD_MEM, FWD_EX};
32    ForwardB : {FWD_NONE, FWD_MEM, FWD_EX};
33    Stall : boolean;
34    Bubble : boolean;
35    Flush : boolean;
36    PCWrite : boolean;
37
38  ASSIGN
39    — INITIAL STATE
40    init(state) := NORMAL;
41
42    — STATE TRANSITION FUNCTION
43    next(state) :=
44      case
45        state = NORMAL :
46          case
47            load_use_hazard : STALL_LOAD;
48            branch_taken & is_branch :
49              FLUSH_BRANCH;
50            is_jump : FLUSH_JUMP;
51            raw_hazard_ex : FORWARD_EX_STAGE;
52            raw_hazard_mem : FORWARD_MEM_STAGE;
53            TRUE : NORMAL;
54          esac;
55
56        state = FORWARD_EX_STAGE :
57          case
58            load_use_hazard : STALL_LOAD;
59            branch_taken : FLUSH_BRANCH;
60            TRUE : NORMAL;
61          esac;
62
63        state = FORWARD_MEM_STAGE :
64          case
65            load_use_hazard : STALL_LOAD;
66            branch_taken : FLUSH_BRANCH;
67            TRUE : NORMAL;
68          esac;
69
70        state = STALL_LOAD : FORWARD_MEM_STAGE;
71        state = FLUSH_BRANCH : NORMAL;
72        state = FLUSH_JUMP : NORMAL;
73        state = ERROR : ERROR;

```

```

70    TRUE : ERROR;
71  esac;
72
73  — OUTPUT LOGIC (Moore Machine)
74  ForwardA :=
75    case
76      state = FORWARD_EX_STAGE &
77      raw_hazard_ex : FWD_EX;
78      state = FORWARD_MEM_STAGE &
79      raw_hazard_mem : FWD_MEM;
80      state = NORMAL | state = STALL_LOAD |
81      state = FLUSH_BRANCH
82      | state = FLUSH_JUMP : FWD_NONE;
83      state = ERROR : FWD_NONE;
84      TRUE : FWD_NONE;
85    esac;
86
87  ForwardB :=
88    case
89      state = FORWARD_EX_STAGE &
90      raw_hazard_ex : FWD_EX;
91      state = FORWARD_MEM_STAGE &
92      raw_hazard_mem : FWD_MEM;
93      state = NORMAL | state = STALL_LOAD |
94      state = FLUSH_BRANCH
95      | state = FLUSH_JUMP : FWD_NONE;
96      state = ERROR : FWD_NONE;
97      TRUE : FWD_NONE;
98    esac;
99
100  Stall :=
101    case
102      state = STALL_LOAD : TRUE;
103      state = ERROR : TRUE;
104      TRUE : FALSE;
105    esac;
106
107  Bubble :=
108    case
109      state = STALL_LOAD : TRUE;
110      TRUE : FALSE;
111    esac;
112
113  Flush :=
114    case
115      state = FLUSH_BRANCH | state =
116      FLUSH_JUMP : TRUE;
117      TRUE : FALSE;
118    esac;
119
120  PCWrite :=
121    case
122      state = STALL_LOAD | state = ERROR :
123      FALSE;
124      TRUE : TRUE;
125    esac;
126
127  —————
128  — SAFETY PROPERTIES
129  —————
130
131  CTLSPEC AG(state != ERROR)
132  CTLSPEC AG(load_use_hazard -> AX(Stall))
133  CTLSPEC AG(Stall -> !PCWrite)
134  CTLSPEC AG(Stall -> Bubble)
135  CTLSPEC AG(!(Stall & Flush))
136  CTLSPEC AG(!(ForwardA = FWD_EX & ForwardA =
137    FWD_MEM))
138  CTLSPEC AG(!(ForwardB = FWD_EX & ForwardB =
139    FWD_MEM))
140
141  —————
142  — LIVENESS PROPERTIES
143  —————

```

```

134 LTLSPEC G(Stall → F(!Stall))
135 LTLSPEC G(Flush → F(state = NORMAL))
136 LTLSPEC G(F(PCWrite))
137 CTLSPEC AG((state = STALL_LOAD) → AF(state =
138   FORWARD_MEM_STAGE))
139
140 —————
141 — STATE TRANSITION CORRECTNESS
142 —————
143
144 CTLSPEC AG((state = STALL_LOAD) → AX(state =
145   FORWARD_MEM_STAGE))
146 CTLSPEC AG((state = FLUSH_BRANCH) → AX(state
147   = NORMAL))
148 CTLSPEC AG((state = FLUSH_JUMP) → AX(state =
149   NORMAL))
150 CTLSPEC AG((state != ERROR) → AX(state !=
151   ERROR))
152
153 —————
154 — HAZARD HANDLING CORRECTNESS
155 —————
156
157 CTLSPEC AG((state = FORWARD_EX_STAGE &
158   raw_hazard_ex) →
159   (ForwardA = FWD_EX))
160 CTLSPEC AG((state = FORWARD_MEM_STAGE &
161   raw_hazard_mem) →
162   (ForwardA = FWD_MEM))
163 CTLSPEC AG((state = STALL_LOAD) → AX(state =
164   FORWARD_MEM_STAGE))
165 CTLSPEC AG((branch_taken & is_branch & state
166   = NORMAL) →
167   AX(state = FLUSH_BRANCH))
168
169 —————
170 — PIPELINE CORRECTNESS
171 —————
172
173 CTLSPEC AG(Flush → !Stall)
174 CTLSPEC AG(Stall → !PCWrite)
175 CTLSPEC AG((state = STALL_LOAD) → AX(AX(
176   state != STALL_LOAD)))
177
178 —————
179 — COVERAGE
180 —————
181
182 CTLSPEC EF(state = STALL_LOAD)
183 CTLSPEC EF(state = FLUSH_BRANCH)
184
185 —————
186 — ADDITIONAL VERIFICATION
187 —————
188
189 CTLSPEC AG(Bubble → Stall)
190 CTLSPEC AG((state = STALL_LOAD) →
191   (ForwardA = FWD_NONE & ForwardB =
192   FWD_NONE))
193 CTLSPEC AG((state = ERROR) → AX(state =
194   ERROR))
195
196 FAIRNESS !(load_use_hazard & raw_hazard_ex &
197   raw_hazard_mem & branch_taken)
198
199 — END OF FILE

```

- 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.

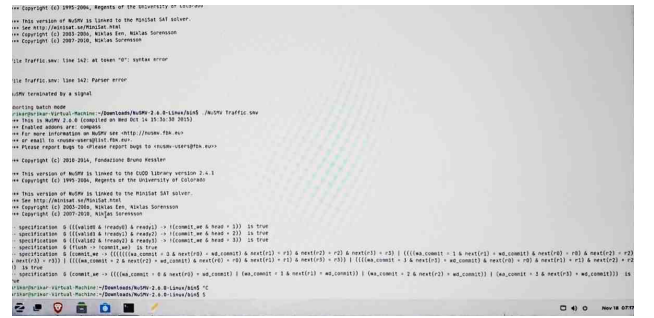


Figure 11: NuSMV Verification Output Showing All CTL/LTL Properties Satisfied

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.