VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
Faculty of Electrical and Electronic Engineering

MILESTONE 3 REPORT
COMPUTER ARCHITECTURE

TOPIC

DESIGN OF PIPELINED RISC-V PROCESSORS

Instructional Lecturer:	**TRAN HOANG LINH**
Class:	**L01**
Group:	**23**
Academic Year:	**2025–2026**
Project Source Code:	**Google Drive**

Ho Chi Minh City, January 2026

VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
Faculty of Electrical and Electronic Engineering

MILESTONE 3 REPORT
COMPUTER ARCHITECTURE

TOPIC

DESIGN OF PIPELINED RISC-V PROCESSORS

Group's members table

	Name	**Student ID**	**Contribution**
1	Nguyen Thanh Phong	2312626	100%
2	Nguyen Hoang Tuan	2313746	100%
3	Vu Tien Tuan	2313774	0%

Ho Chi Minh City, January 2026

Contents

1 Introduction

Milestone 3 upgrades our RV32I processor from the single-cycle design (Milestone 2) to a classic 5-stage pipelined microarchitecture consisting of **IF, ID, EX, MEM, WB**. The main objectives are:

- Implement a functional 5-stage pipeline and pass the provided ISA tests.
- Handle pipeline hazards by using **stall/flush** control.
- Evaluate and compare two baseline configurations:
  - **Non-forwarding**: resolve data hazards mainly by stalling until write-back.
  - **Forwarding**: add forwarding paths to reduce stall cycles.

In this baseline implementation, we do **not** include a branch predictor. Instead, we use the standard baseline policy of **predict NOT-TAKEN** (PC increments by 4) and apply **flush** when the branch/jump is resolved as taken. Therefore, the branch misprediction rate is expected to be similar between non-forwarding and forwarding configurations; forwarding mainly improves performance by reducing data-hazard stalls.

2 System Overview

2.1 Pipeline Stages

- **IF (Instruction Fetch)**: update PC, fetch instruction, compute PC+4.
- **ID (Instruction Decode)**: decode, read register file, generate immediate.
- **EX (Execute)**: ALU operations, address generation, branch/jump decision and target calculation.
- **MEM (Memory)**: data memory access for loads/stores (and MMIO if enabled).
- **WB (Writeback)**: select write-back value and write into the register file.

2.2 Top-Level Datapath

Figure 2.1 illustrates the overall pipelined datapath, including pipeline registers, hazard control signals, and the optional forwarding unit.

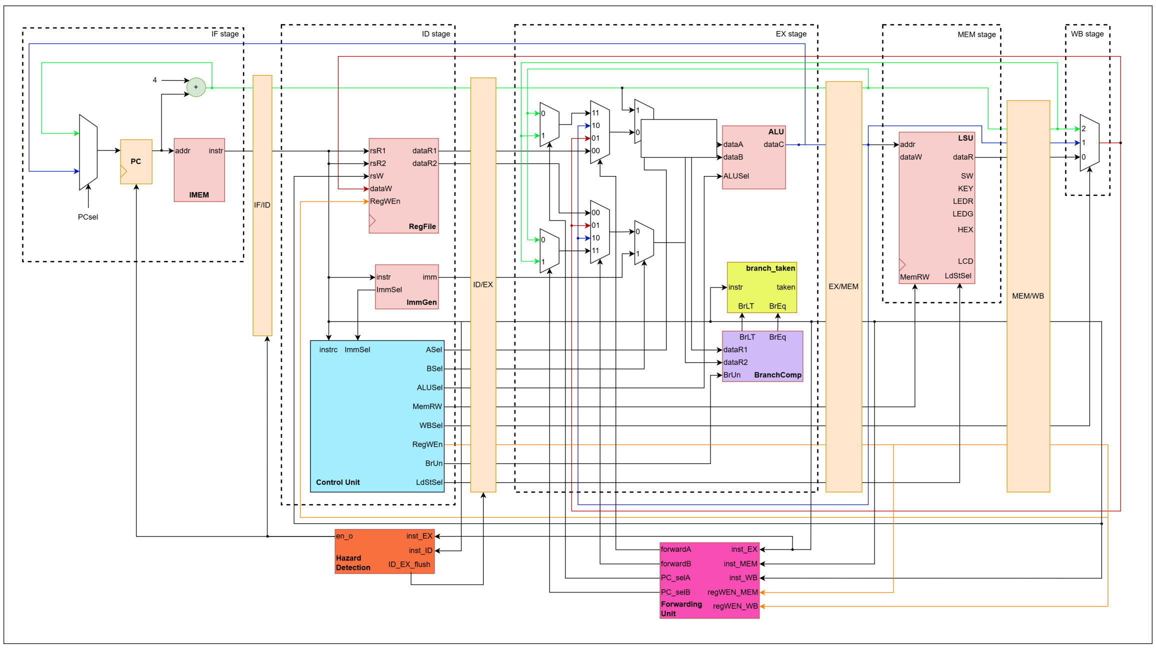


Figure 2.1: *Top-level 5-stage pipelined RV32I processor (hazard unit + optional forwarding).*

2.3 Control Flow (Predict NOT-TAKEN Baseline)

Our baseline control policy is **predict NOT-TAKEN**:

- The fetch stage always continues with PC+4 by default.
- Branch/jump decision is resolved in the EX stage.
- If the control transfer is taken, wrong-path younger instructions are invalid and must be **flushed** (converted into NOP-like behavior by clearing control signals).

Debug/monitor signals (per project interface):

- `o_ctrl`: asserted when a control-transfer instruction (branch/jump) is present in the tracked stage.
- `o_mispred`: asserted when a flush occurs due to a taken control transfer under NOT-TAKEN baseline.

3 Design and Implementation

3.1 Reused Modules From Milestone 2

The following blocks are reused with minimal changes:

- **ALU**, **Immediate Generator**, **Register File**, and **Main Control Decoder**.
- **Load/Store Unit**.

The major Milestone 3 additions are pipeline registers, stage-wise control propagation, and hazard handling logic.

3.2 Pipeline Registers and Control Propagation

We implement four pipeline registers: **IF/ID**, **ID/EX**, **EX/MEM**, **MEM/WB**. Each pipeline register stores both datapath signals and a bundle of control signals required by downstream stages.

General rules:

- **Reset**: initialize stage registers into a safe NOP-like state.
- **Stall**: hold current contents to delay younger instructions (PC and IF/ID typically stall together).
- **Flush**: clear control signals so the flushed instruction produces no architectural side effects.

3.3 Hazard Handling Overview

In a 5-stage pipeline, hazards are categorized into:

- **Data hazards (RAW)**: an instruction needs a register value that is still being produced by an earlier instruction.
- **Control hazards**: the next PC depends on a branch/jump decision resolved later in the pipeline.

This report focuses on data-hazard mechanisms for two configurations:

- Non-forwarding: stall on most RAW hazards until producer writes back.
- Forwarding: forward ALU/WB results when possible and stall only in unavoidable cases (e.g., load-use).

3.4 Non-forwarding Configuration

3.4.1 Hazard Detection Unit Behavior

In the non-forwarding design, the pipeline stalls when the ID-stage instruction depends on a result that is still in-flight in EX or MEM stages. The hazard detection unit compares the source registers of the ID instruction (`rs1_ID`, `rs2_ID`) against the destination registers of the instructions in EX and MEM (`rd_EX`, `rd_MEM`), and uses `regWEn` to filter only instructions that will write back.

To avoid false hazards, the input `uses_rs2` is used:

- If the current ID instruction does not consume `rs2` (e.g., I-type ALU ops, loads), the hazard check against `rs2_ID` is disabled.

When a hazard is detected, the unit forces:

- **PC stall**: `pc_en = 0` (PC is held).
- **IF/ID stall**: `IF_ID_en = 0` (ID instruction is held).
- **Insert bubble**: `ID_EX_flush = 1` (clear control signals into EX so EX gets a NOP).

3.4.2 Hazard Logic Summary

Let `rd_EX` and `rd_MEM` be the destination registers of EX/MEM instructions. A stall is asserted when:

- EX produces a register write and its destination matches `rs1_ID`, or matches `rs2_ID` when `uses_rs2=1`.
- MEM produces a register write and its destination matches `rs1_ID`, or matches `rs2_ID` when `uses_rs2=1`.

Because there is no forwarding, the consumer must wait until the value is finally written back, resulting in multiple stall cycles depending on where the producer currently is in the pipeline.

3.5 Forwarding Configuration

3.5.1 Forwarding Unit (EX Operand Selection)

The forwarding unit reduces stalls by selecting the most recent value for EX operands directly from later pipeline stages. For the instruction currently in EX:

- If a matching destination register exists in MEM stage and writes back, forward from MEM (`forward=2'b10`).
- Otherwise, if a matching destination register exists in WB stage and writes back, forward from WB (`forward=2'b01`).
- Else, use the original operand read from register file (`forward=2'b00`).

We apply **priority** MEM over WB because MEM contains newer data for back-to-back dependencies.

3.5.2 Forwarding Control Signals

Two 2-bit control signals are generated:

- `forwardA_EX`: select source for EX operand A (rs1 path).
- `forwardB_EX`: select source for EX operand B (rs2 path).

Destination register `rd=0` is ignored to preserve x0 semantics.

3.5.3 Hazard Detection Unit Behavior

Even with forwarding, the **load-use** hazard typically cannot be resolved in the same cycle because load data is only available after MEM. Therefore, we stall only for the standard load-use condition:

- If EX-stage instruction is a load (opcode `0000011`) and its destination `rd_EX` matches `rs1_ID` or `rs2_ID`, then stall PC and IF/ID and insert a bubble into EX.

This makes the forwarding pipeline much less stall-heavy than the non-forwarding pipeline.

3.5.4 Hazard Logic Summary

With forwarding enabled, most RAW hazards are resolved without stalling by selecting the newest available value for EX operands from later stages. Forwarding decisions are based on register-number matching and write-enable signals:

- **Forward from MEM (highest priority):** if `regWEn_MEM=1`, `rd_MEM`$\neq$`0`, and `rd_MEM = rsX_EX` $\Rightarrow$ select MEM result.
- **Forward from WB:** else if `regWEn_WB=1`, `rd_WB`$\neq$`0`, and `rd_WB = rsX_EX` $\Rightarrow$ select WB result.
- **No forwarding:** otherwise use the register-file value captured in ID/EX.

Despite forwarding, a **load-use** dependency still requires a one-cycle stall because load data becomes available only after the MEM stage. The pipeline stalls when the EX instruction is a load and its destination `rd_EX` matches `rs1_ID` or `rs2_ID`; in that case PC and IF/ID are held, and a bubble is inserted into ID/EX.

3.6 Stall/Flush Policy Summary

Table 3.1 summarizes the stall/flush behavior used in both configurations.

Scenario	**PC enable**	**IF/ID enable**	**ID/EX flush**	**Effect**
No hazard	1	1	0	Normal pipeline flow.
Non-fwd RAW hazard (EX/MEM producer)	0	0	1	Hold PC and IF/ID; insert a bubble in ID/EX until the producer value is available at write-back.
Fwd load-use hazard (EX is load)	0	0	1	One-cycle stall; after that, forwarding resolves the dependency.
Taken branch/jump (predict NOT-TAKEN)	1	*Flush*	*Flush*	Redirect PC to the target and flush younger wrong-path instructions in IF/ID and ID/EX.

Table 3.1: *Stall/flush policy summary for non-forwarding and forwarding configurations.*

4 Verification

4.1 Simulation Methodology

Verification is performed in two layers:

- **Directed unit testcases (Forwarding configuration):** We write short micro-programs to intentionally create specific data-dependency patterns so that the forwarding unit must select the correct source (`EX/MEM` or `MEM/WB`) and obey the priority rule (MEM over WB). For each testcase, we provide a pipeline timeline and a typed *Results* log showing the expected forwarding select codes.
- **Course-provided ISA tests (Non-forwarding and Forwarding configurations):** After validating the forwarding behavior with directed testcases, we run the official ISA regression suite provided by the course. Correctness is determined by the testbench `PASS/FAIL` criteria. At the end

of each run, the testbench reports total executed cycles, total executed instructions, IPC/CPI, and branch misprediction statistics. These ISA results are used for the final comparison between the non-forwarding and forwarding baselines.

4.2 Directed Unit Testcases

4.2.1 Forwarding Unit Testcases (Forwarding configuration)

To validate the forwarding unit, we create directed micro-programs that force different forwarding sources: **EX/MEM (MEM stage)** and **MEM/WB (WB stage)**. In our design, the selection codes are: `2'b10` = forward from MEM (`alu_mem`), `2'b01` = forward from WB (`wb_data`), and `2'b00` = no forwarding (use ID/EX values). MEM has higher priority than WB.

4.2.1.1 TC-F1: Forward from MEM (EX/MEM) for rs1

```
add  x5, x1, x2
sub  x6, x5, x3    # needs x5 immediately -> forwardA_EX = 2'b10
```

Listing 1: TC-F1 (MEM forwarding): back-to-back dependency

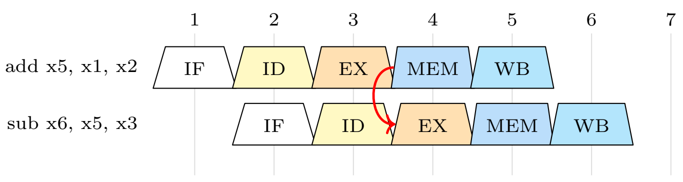


Figure 4.1: *TC-F1 pipeline timeline (forward from MEM to EX for rs1).*

Results

```
3: inst_EX=add x5,x1,x2, ForwardA: 00, ForwardB: 00, PC_fw_A: 0, PC_fw_B: 0
4: inst_EX=sub x6,x5,x3, ForwardA: 10, ForwardB: 00, PC_fw_A: 0, PC_fw_B: 0
```

4.2.1.2 TC-F2: Forward from WB (MEM/WB) for rs2

```
add  x5, x1, x2
addi x0, x0, 0      # nop to create 1-cycle gap
sub  x6, x3, x5     # x5 available in WB -> forwardB_EX = 2'b01
```

Listing 2: TC-F2 (WB forwarding): one-cycle gap dependency

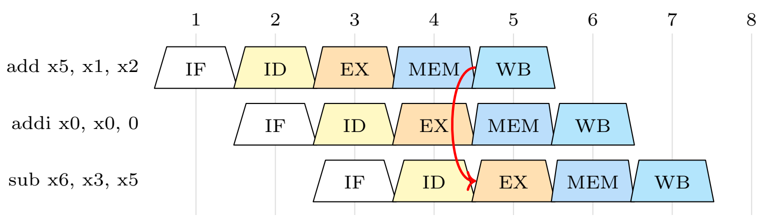


Figure 4.2: *TC-F2 pipeline timeline (forward from WB to EX for rs2).*

Results

```
3: inst_EX=add x5,x1,x2, ForwardA: 00, ForwardB: 00, PC_fw_A: 0, PC_fw_B: 0
4: inst_EX=addi x0,x0,0, ForwardA: 00, ForwardB: 00, PC_fw_A: 0, PC_fw_B: 0
5: inst_EX=sub x6,x3,x5, ForwardA: 00, ForwardB: 01, PC_fw_A: 0, PC_fw_B: 0
```

4.2.1.3 TC-F3: MEM has priority over WB (mixed sources)

```
add  x5, x1, x2
add  x6, x3, x4
sub  x7, x5, x6     # at EX: x6 in MEM, x5 in WB -> rs2 from MEM, rs1 from WB
```

Listing 3: TC-F3 (priority): MEM over WB

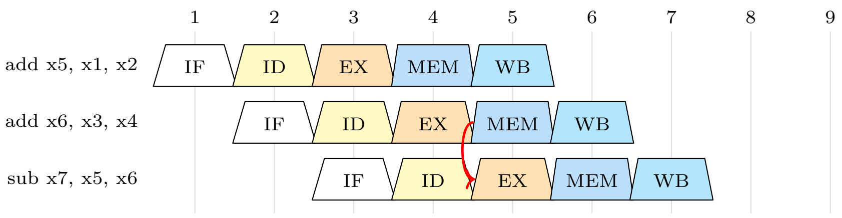


Figure 4.3: *TC-F3 pipeline timeline (mixed forwarding sources; MEM has higher priority than WB).*

Results

```
3: inst_EX=add x5,x1,x2, ForwardA: 00, ForwardB: 00, PC_fw_A: 0, PC_fw_B: 0
4: inst_EX=add x6,x3,x4, ForwardA: 00, ForwardB: 00, PC_fw_A: 0, PC_fw_B: 0
5: inst_EX=sub x7,x5,x6, ForwardA: 01, ForwardB: 10, PC_fw_A: 0, PC_fw_B: 0
```

4.2.2 Hazard Unit Testcases (Forwarding configuration)

With forwarding enabled, the hazard unit stalls only for the classic load-use case because load data is available after MEM. During a stall, PC and IF/ID are held and one bubble (NOP) is injected into EX. Bubbles are shown as gray patterned cells without text.

4.2.2.1 TC-H1: Load-use stall (lw → add)

```
lw   x5, 0(x1)
add  x6, x5, x2     # uses loaded x5 -> must stall 1 cycle
```

Listing 4: TC-H1 (load-use): one-cycle stall then forward from WB

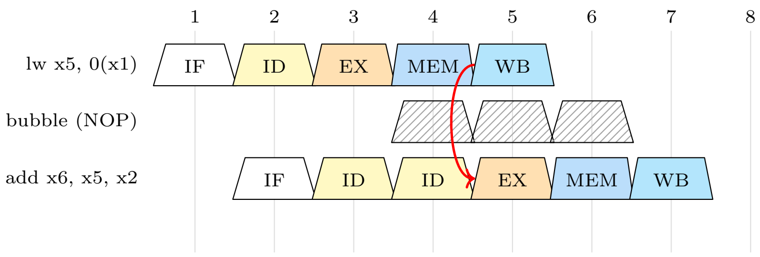


Figure 4.4: *TC-H1 pipeline timeline (one-cycle load-use stall; bubble injected into EX).*

Results

```
3: HZD=1 (pc_en=0, IF_ID_en=0, ID_EX_flush=1)
4: inst_EX=bubble (NOP)
5: inst_EX=add x6,x5,x2, ForwardA: 01, ForwardB: 00, PC_fw_A: 0, PC_fw_B: 0
```

4.2.2.2 TC-H2: Load-use stall (lw → sw uses rs2)

```
lw   x5, 0(x1)
sw   x5, 0(x2)      # store-data uses x5 (rs2) -> must stall 1 cycle
```

Listing 5: TC-H2 (load-use): lw then sw depends on rs2

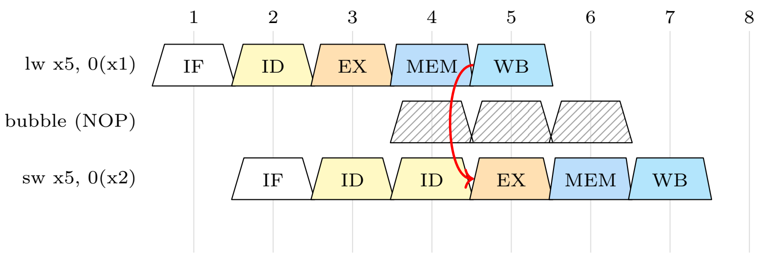


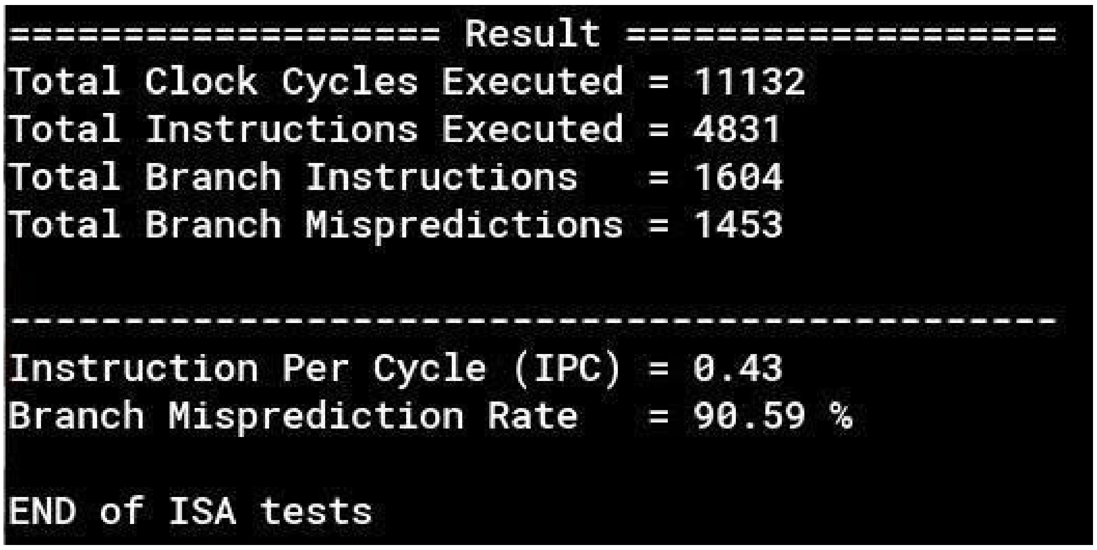
Figure 4.5: *TC-H2 pipeline timeline (lw→sw load-use stall; bubble injected into EX).*

Results

```
3: HZD=1 (pc_en=0, IF_ID_en=0, ID_EX_flush=1)
4: inst_EX=bubble (NOP)
5: inst_EX=sw x5,0(x2), ForwardA: 00, ForwardB: 01, PC_fw_A: 0, PC_fw_B: 0
```

4.3 ISA Test Results

Figures 4.6 and 4.7 show the result summaries for non-forwarding and forwarding configurations.

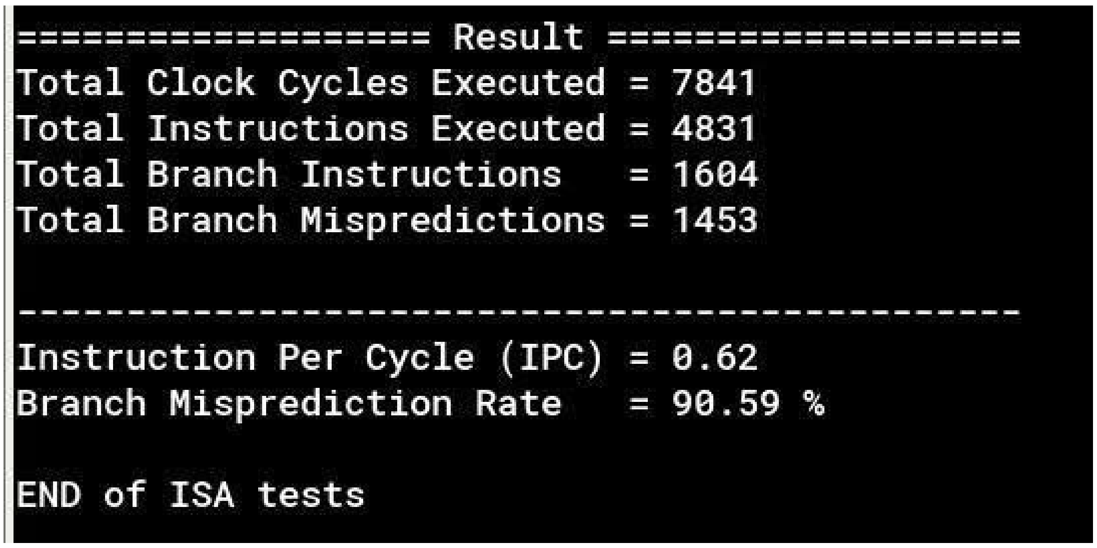


```
==================== Result ====================
Total Clock Cycles Executed = 11132
Total Instructions Executed = 4831
Total Branch Instructions   = 1604
Total Branch Mispredictions = 1453

-----------------------------------------------
Instruction Per Cycle (IPC) = 0.43
Branch Misprediction Rate   = 90.59 %

END of ISA tests
```

Figure 4.6: *ISA test summary (Non-forwarding configuration).*



```
=================== Result ===================
Total Clock Cycles Executed = 7841
Total Instructions Executed = 4831
Total Branch Instructions   = 1604
Total Branch Mispredictions = 1453

----------------------------------------------
Instruction Per Cycle (IPC) = 0.62
Branch Misprediction Rate   = 90.59 %

END of ISA tests
```

Figure 4.7: *ISA test summary (Forwarding configuration).*

5 Performance and Results

5.1 Metrics

We report three key performance metrics:

- **IPC (Instructions Per Cycle)**: higher IPC indicates fewer stalls and better pipeline utilization.
- **CPI (Cycles Per Instruction)**: reciprocal of IPC; lower CPI is better.
- **Branch Misprediction Rate**: with predict NOT-TAKEN baseline, taken control transfers cause flushes and increase mispredictions.

5.2 ISA Test Performance (Non-forwarding vs Forwarding)

We report ISA test results only. Table 5.1 summarizes the measured results.

Metric	Non-forwarding	Forwarding
Total Clock Cycles Executed	11132	7841
Total Instructions Executed	4831	4831
IPC	0.43	0.62
CPI	2.30	1.62
Total Branch Instructions	1604	1604
Total Branch Mispredictions	1453	1453
Branch Misprediction Rate	90.59%	90.59%

Table 5.1: *ISA test performance comparison between non-forwarding and forwarding configurations.*

5.3 Discussion

Forwarding provides a clear speedup on the ISA tests. Compared to the non-forwarding design, the forwarding design reduces total cycles from 11132 to 7841, i.e., 3291 fewer cycles (about 29.6% reduction). This corresponds to an overall speedup of approximately 1.42×.

The IPC improves from 0.43 to 0.62 because forwarding eliminates many stall cycles caused by data hazards. Consequently, CPI drops from 2.30 to 1.62.

The branch misprediction rate remains the same at 90.59% for both configurations because both designs use the same **predict NOT-TAKEN** baseline policy. Forwarding improves data-hazard handling but does not change the control-hazard behavior.

5.4 Quartus Timing and Resource Utilization

For Quartus post-fit evaluation, both the instruction memory (IMEM) and data memory (DMEM) are configured to **2K 32-bit words** each. Timing results (Fmax, setup slack, hold slack) are taken from TimeQuest reports, while logic/register/memory utilization is taken from the Fitter summary.

Metric	Non-forwarding	Forwarding
Fmax (MHz)	52.41	51.98
Worst Setup Slack (ns)	0.921	0.762
Worst Hold Slack (ns)	0.215	0.236
Total Logic Elements (count or %)	9,219 / 14,448 (64%)	9,141 / 14,448 (63%)
Total Registers (count or %)	1,858	1,743
Total Memory Bits (count or %)	65,536 / 239,616 (27%)	65,536 / 239,616 (27%)

Table 5.2: *Quartus post-fit timing and resource summary.*

6 Conclusion

We successfully implemented a 5-stage pipelined RV32I processor and verified correctness using the course ISA tests. Two baseline configurations were evaluated:

- **Non-forwarding**: correct execution is achieved by stalling on RAW hazards until results are written back.
- **Forwarding**: forwarding paths significantly reduce stall cycles and improve pipeline utilization.

On the ISA tests, forwarding reduces total cycles by about 29.6% and achieves roughly 1.42× speedup, increasing IPC from 0.43 to 0.62. The branch misprediction rate remains unchanged because both configurations use the same predict NOT-TAKEN control policy.

7 Future Work

Possible improvements include:

- Implement a branch predictor to reduce control-hazard penalty and lower misprediction rate.
- Optimize critical paths to improve Fmax and timing slack in Quartus.
- Add additional benchmarks beyond ISA tests for broader performance evaluation.

References

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: RISC-V Edition*, Morgan Kaufmann, 2017.