CPM Final Project - Verification Requirements & Deliverables

OBJECTIVE

Build and close a UVM-based verification environment for the CPM DUT.

The expected workflow is:

1	2
Requirements	Architecture
3	4
Stimulus	Checking
5	6
Coverage	Closure

Verification of the control/register bus protocol is out of scope (assume it works).

CONTROL MODEL (MANDATORY RULE)

This project enforces a strict separation of responsibilities:

Test

chooses scenario + configures the environment

Virtual Sequence

orchestrates execution

Leaf Sequences

generate stimulus only

Violations (examples):

- Test drives transactions directly
- Driver raises objections
- Leaf sequences perform configuration/checking → considered **architectural errors**.

Rules:

- No direct DUT signal access from tests
- Configuration and knobs must go via uvm_config_db
- All components must be factory-registered

REQUIRED SEQUENCES

Minimum required sequences (names may vary, roles must match):

Sequence	Purpose
config_seq (RAL-based)	Program CTRL/MODE/PARAMS/DROP via RAL
base_traffic_seq	Random packet stimulus
stress_seq	Burst traffic to cause stalls/backpressure
drop_seq	Force opcode matching drop configuration
top_virtual_seq	Full scenario orchestration

You must implement at least one main test that:

01

Selects the virtual sequence to run

02

Sets configuration knobs (counts, mode schedule, stress level)

03

Applies factory overrides (see Section 6)

04

Starts the virtual sequence

05

Ends cleanly (no hangs)

Recommended: a single cpm_base_test that can be configured via plusargs / config knobs.

MANDATORY UVM MECHANISMS (MUST DEMONSTRATE)

6.1 RAL (Mandatory)

Implement a CPM register model (matches the DUT spec).

Integrate:

- uvm_reg_adapter
- uvm_reg_predictor

Run at least:

- uvm_reg_hw_reset_seq
- one custom RAL configuration sequence

Rules:

- Do not access registers by writing bus transactions directly from tests
- Register programming must be done via RAL API (reg.write/read/mirror/update)

6.2 Virtual Sequence (Mandatory)

Complete System Flow Orchestration

Your top_virtual_seq must orchestrate a complete system flow:

1. Reset
2. Configure
3. Traffic
4. Reconfigure
5. Stress
6. Drop
7. Readback
8. End

Requirements:

- Raise/drop objections only at the test or top virtual sequence level
- Reconfigure MODE during runtime and confirm behavior follows "sampled at input accept time"

6.3 Factory Override (Mandatory)

You must demonstrate one meaningful factory override, for example:

> Override base_traffic_seq → coverage_traffic_seq (forces rare MODE/opcode combinations or drop event)

Rule:

Override is applied in the test (selecting behavior), not scattered across components.

6.4 Callbacks (Mandatory)

Implement at least one callback mechanism (choose one):

Driver callback

that can modify outgoing transactions

Monitor callback

used for event counting or tagging

Scoreboard callback

for additional reporting (less recommended)

The callback must have a real reason (not a dummy print).

6.5 Functional Coverage (Mandatory)

Must include:

- Coverpoint: MODE
- Coverpoint: OPCODE
- Cross: MODE × OPCODE
- Drop event coverage (at least one bin)
- Stall/backpressure event coverage (at least one bin)

Coverage must be collected in UVM (subscriber recommended).

6.6 SVA (Mandatory)

Assertions must be written in the stream interface (not inside UVM classes).

Required properties:

1

Input stability under stall

If in_valid && !in_ready → input fields stable

2

Output stability under stall

If out_valid && !out_ready → output fields stable

3

Bounded liveness

If an input packet is accepted and not dropped, an output must appear within the bound (bound must be consistent with CPM spec: base latency ≤2 cycles when out_ready stays high; allow additional slack due to buffering/backpressure logic)

4

One meaningful cover property

Examples: a stall event, or mode=ADD exercised, or drop event observed

SCOREBOARD REQUIREMENTS

The scoreboard must:

Core Functions

- Maintain an expected queue of outputs
- Use a reference model:
  - apply transformation based on MODE/PARAMS sampled at input acceptance
  - handle drop rule (no expected output)
- Compare to observed outputs
- Provide actionable mismatch reporting (expected vs actual + context)

End-of-test checks:

- Counter invariant: COUNT_OUT + DROPPED_COUNT == COUNT_IN
- No leftover expected items (queue empty)

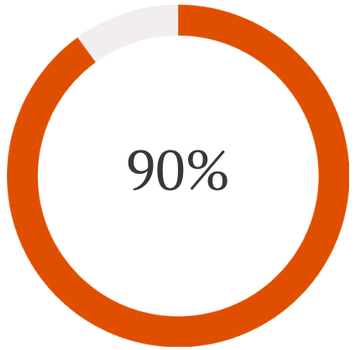
CLOSURE CRITERIA (DEFINITION OF "DONE")

A submission is considered closed only when:

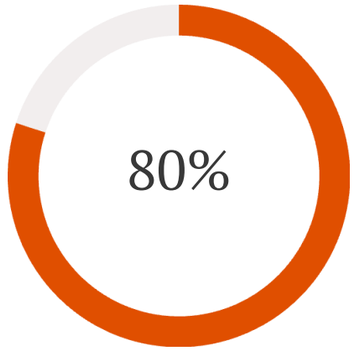
- [ ] All tests pass (no runtime hangs)
- [ ] Scoreboard reports 0 mismatches
- [ ] All assertions pass (no failures)
- [ ] Functional coverage targets are achieved (see below)
- [ ] RAL reset sequence passes cleanly
- [ ] End-of-test invariants are checked and pass

Recommended targets:

MODE coverage



OPCODE coverage



MODE×OPCODE cross

Drop bin: hit at least once

Stall bin: hit at least once

DELIVERABLES

Submit a single ZIP containing:

Full source code

(UVM TB + assertions + RAL code)

A Verification Plan

(see separate template)

Coverage report

(screenshot or text summary)

Assertion report

(tool output summary)

Reflection Report

Challenges, limitations, future work

Professional requirement: Organize code using packages.

COMMON RULES (NON-NEGOTIABLE)

Non-Negotiable Rules

No direct DUT signal access from tests

No bus-level register access from tests (use RAL)

No objections inside driver/monitor

Tests configure what, virtual sequence controls how

Keep the project slim: do not invent extra DUT features