Configurable Packet Modifier (CPM) Design Specification - Version 1.0

SystemVerilog and UVM Verification Course

Nadav Talmon

Google Reichman University | Google and Reichman Tech School

1. Introduction

1.1 Purpose

The Configurable Packet Modifier (CPM) is a configurable hardware block that processes single-beat packets received on a streaming input interface and produces corresponding packets on a streaming output interface.

Packet processing behavior is controlled via a register interface and includes:

- deterministic, mode-dependent data transformation
- optional packet drop based on opcode
- deterministic internal processing latency
- ready/valid flow control with backpressure
- observable internal status and counters

The CPM is intentionally compact in functionality while exposing realistic behaviors that require disciplined verification methodologies.

1.2 Scope and Non-Goals

In scope

- Single-beat packet processing
- Configuration via registers
- Deterministic pipeline latency (0–2 cycles)
- Ready/valid flow control with backpressure
- Strict in-order processing
- Status and counters

Out of scope

- Multi-beat packets
- Packet reordering
- Interrupt generation
- DMA or memory interfaces
- Multiple clock domains
- Error response generation

Any behavior not explicitly defined in this specification shall be considered implementation-defined and must not be relied upon by the verification environment.

2. Clocking and Reset

Signal Description

clk	Single system clock
rst	Active-high synchronous reset

Reset behavior

Upon assertion of rst:

- All registers return to reset values
- All internal buffers and pipeline stages are flushed
- All counters are cleared
- The CPM is disabled (CTRL.ENABLE = 0)
- No input packets are accepted
- No output packets are produced

Reset behavior is synchronous to clk. No asynchronous reset behavior is required or expected.

3. Packet Definition

Each packet is single-beat and contains the following fields:

id	4 bits	Transaction identifier
opcode	4 bits	Operation / classification field
payload	16 bits	Data payload

There is no packet framing, burst semantics, or multi-cycle packet structure.

4. Interface Overview

The CPM exposes three logical interfaces:

Streaming Input Interface	Streaming Output Interface	Register Control Interface

Verification of the control/register bus protocol itself is out of scope. The control interface shall be assumed functionally correct.

5. Streaming Input Interface

5.1 Signals

in_valid	Input	1	Input data valid
in_ready	Output	1	CPM ready to accept input
in_id	Input	4	Packet ID
in_opcode	Input	4	Packet opcode
in_payload	Input	16	Packet payload

5.2 Handshake Semantics

A packet is accepted when:

```
in_fire = in_valid && in_ready
```

Acceptance occurs on the rising edge of clk.

The CPM preserves strict packet ordering under all operating conditions.

5.3 Backpressure and Stability Rules

The CPM may deassert in_ready at any time to apply backpressure.

If `in_valid == 1` and `in_ready == 0`, the following signals must remain stable until acceptance:

- in_id
- in_opcode
- in_payload

6. Streaming Output Interface

6.1 Signals

out_valid	Output	1	Output data valid
out_ready	Input	1	Downstream ready
out_id	Output	4	Packet ID
out_opcode	Output	4	Packet opcode
out_payload	Output	16	Packet payload

6.2 Handshake Semantics

A packet is transferred when:

```
out_fire = out_valid && out_ready
```

The CPM preserves strict packet ordering between input and output.

6.3 Stability Rules

If `out_valid == 1` and `out_ready == 0`, the following signals must remain stable:

- out_id
- out_opcode
- out_payload

7. Processing Pipeline and Latency

7.1 Deterministic Latency Model

Packet processing latency is deterministic and depends solely on the configured processing mode.

Latency is defined as the number of cycles between:

- the cycle in which in_fire occurs
- and the earliest cycle in which the corresponding output packet may be presented on the output interface

If out_ready remains asserted, any accepted input packet that is not dropped must result in a valid output packet within at most 2 cycles.

Backpressure on the output interface may delay acceptance beyond the base latency, but the CPM must hold output data stable until accepted.

7.2 Mode-Dependent Behavior

0	PASS	payload_out = payload_in	0 cycles
1	XOR	payload_out = payload_in ^ MASK	1 cycle
2	ADD	payload_out = payload_in + ADD_CONST	2 cycles
3	ROT	payload_out = rotate_left(payload_in, ROT_AMT)	1 cycle

Mode and parameter values are sampled at input acceptance time (in_fire). Packets already accepted are processed using the configuration values in effect at their acceptance time.

7.3 Ordering Guarantees

1

Packets are processed strictly in order

2

No reordering is permitted

3

Each output packet corresponds to a unique previously accepted input packet, unless dropped

8. Drop Functionality

8.1 Drop Condition

A packet is dropped if all of the following are true at input acceptance time (in_fire):

- DROP_CFG.DROP_EN == 1
- in_opcode == DROP_CFG.DROP_OPCODE

8.2 Drop Semantics

When a packet is dropped:

- It is counted as an accepted input packet
- No output packet is produced
- DROPPED_COUNT is incremented
- Dropped packets do not occupy output pipeline stages and do not contribute to output latency

9. Backpressure Behavior

The CPM may temporarily stop accepting input by deasserting in_ready

Backpressure may occur due to internal pipeline occupancy or downstream congestion

Internal buffering is finite

Packet ordering is preserved under all backpressure conditions

10. Register Control Interface

All signals are synchronous to clk. Reset is synchronous via rst.

Signal	Direction	Description
req	Input	Request valid
gnt	Output	Grant / accept request
write_en	Input	Write enable (1 = write, 0 = read)
addr[7:0]	Input	Byte address
wdata[31:0]	Input	Write data
rdata[31:0]	Output	Read data

10.1 Transaction Handshake

A register transaction is considered **accepted** in any cycle where:

> `req == 1 && gnt == 1`

The CPM does not introduce wait states on the register interface. The signal gnt is asserted whenever req is asserted.

Therefore, a transaction always completes in a single cycle.

10.3 Write Transaction Semantics

A write transaction is defined by:

```
req == 1 && gnt == 1 && write_en == 1
```

The addressed register is updated on the rising edge of the clock cycle in which the write transaction is accepted.

10.4 Read Transaction Semantics

A read transaction is defined by:

```
req == 1 && gnt == 1 && write_en == 0
```

The value of the addressed register is returned on the signal rdata.

For the purposes of this project, the master shall sample rdata on the rising edge of the clock cycle **following** the acceptance of the read transaction.

10.5 Protocol Scope

The CPM register interface is intended for **functional control and configuration only**. Protocol-level aspects such as arbitration, retries, errors, wait states, or bus contention are out of scope and shall not be verified as part of this project.

10.6 Register Access Semantics

All register writes take effect on the rising edge of clk in which the write transaction completes.

10.7 Register Summary

0x00	CTRL	RW
0x04	MODE	RW
0x08	PARAMS	RW
0x0C	DROP_CFG	RW
0x10	STATUS	RO
0x14	COUNT_IN	RO
0x18	COUNT_OUT	RO
0x1C	DROPPED_COUNT	RO

10.8 CTRL Register (0x00)

0	ENABLE	RW	0	Enables CPM operation
1	SOFT_RST	RW	0	Self-clearing soft reset

Writing SOFT_RST = 1 clears counters and internal state. The bit self-clears to 0 within one cycle.

10.9 MODE Register (0x04)

[1:0]	MODE	RW	0	Processing mode

11.0 PARAMS Register (0x08)

[15:0]	MASK	RW	0x0000	XOR mask
[31:16]	ADD_CONST	RW	0x0000	Add constant

11.1 DROP_CFG Register (0x0C)

0	DROP_EN	RW	0	Enable packet dropping
[7:4]	DROP_OPCODE	RW	0	Opcode to drop

10.7 STATUS Register (0x10)

0	BUSY	RO	Asserted when at least one accepted packet has not yet produced an output or been dropped

10.8 Counter Registers

COUNT_IN	Number of accepted input packets
COUNT_OUT	Number of output packets produced
DROPPED_COUNT	Number of dropped packets

Invariant (when stable):

```
COUNT_OUT + DROPPED_COUNT == COUNT_IN
```