

ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA KỸ THUẬT MÁY TÍNH

NGÔ THÁI ANH HÀO

KHÓA LUẬN TỐT NGHIỆP
NGHIÊN CỨU THIẾT KẾ VÀ HIỆN THỰC MÔ HÌNH
KIỂM TRA CHO MỘT THIẾT KẾ MẠNG NO-RON
TÍCH CHẬP

Researching and Implementation of a Verification Environment for
a Convolutional Neural Network Intellectual Property

KỸ SƯ KỸ THUẬT MÁY TÍNH

TP. HỒ CHÍ MINH, 2024

ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA KỸ THUẬT MÁY TÍNH

NGÔ THÁI ANH HÀO – 16520347

KHÓA LUẬN TỐT NGHIỆP
NGHIÊN CỨU THIẾT KẾ VÀ HIỆN THỰC MÔ HÌNH
KIỂM TRA CHO MỘT THIẾT KẾ MẠNG NO-RON
TÍCH CHẬP

**Researching and Implementation of a Verification Environment for
a Convolutional Neural Network Intellectual Property**

KỸ SƯ KỸ THUẬT MÁY TÍNH

GIẢNG VIÊN HƯỚNG DẪN
THẠC SĨ TRƯỜNG VĂN CƯỜNG
THẠC SĨ HỒ NGỌC DIỄM

TP. HỒ CHÍ MINH, 2024

THÔNG TIN HỘI ĐỒNG CHẤM KHÓA LUẬN TỐT NGHIỆP

Hội đồng chấm khóa luận tốt nghiệp, thành lập theo Quyết định số 662/QĐ-ĐHCNTT ngày 5 tháng 7 năm 2024 của Hiệu trưởng Trường Đại học Công nghệ Thông tin.

LỜI CẢM ƠN

Lời đầu tiên, trong quá trình thực hiện Khóa luận tốt nghiệp, em xin gửi lời tri ân sâu sắc đến thầy Trương Văn Cương. Quá trình thực hiện Khóa luận tốt nghiệp và hoàn tất chương trình học của em đã không thể hoàn thành tốt đẹp nếu không có sự dẫn dắt và hỗ trợ hết mình của thầy.

Em cũng xin gửi lời cảm ơn chân thành đến các thầy cô thuộc Khoa Kỹ thuật Máy Tính là người truyền đạt những kiến thức chuyên môn nền tảng giúp em có thể nắm vững trước khi bắt đầu hành trình công việc của mình sau khi tốt nghiệp. Đặc biệt, em xin gửi lời cảm ơn đến thầy Nguyễn Minh Sơn đã hỗ trợ hết mình và trao các cơ hội học tập và làm việc cho em trong quá trình học tập tại trường. Em cũng xin gửi lời cảm ơn đến thầy Lâm Đức Hải là người đã truyền cảm hứng cho em đến với chuyên ngành thiết kế vi mạch.

Cuối cùng, em xin cảm ơn các thầy cô giảng viên thuộc các Khoa, Phòng Đào tạo Đại học, Văn phòng Các chương trình đặc biệt và các thầy cô thuộc phòng ban khác của trường Đại học Công nghệ Thông tin – Đại học Quốc gia Thành phố Hồ Chí Minh đã giúp đỡ và tạo điều kiện học tập cho em trong suốt quá trình học tập và làm việc tại trường.

Thành phố Hồ Chí Minh, tháng 7 năm 2024

Sinh viên thực hiện

Ngô Thái Anh Hào

MỤC LỤC

Chương 1. TÌM HIỂU TỔNG QUAN	2
1.1. Giới thiệu.....	2
1.2. Mục tiêu đề tài	3
1.3. Tổng quan đề tài.....	3
Chương 2. CƠ SỞ LÝ THUYẾT.....	5
2.1. Mô hình môi trường kiểm tra thiết kế UVM	5
2.1.1. Cấu trúc và các thành phần chính bên trong một UVM Testbench.....	5
2.1.2. Transaction-Level Modeling	10
2.1.3. Các Phase bên trong quá trình mô phỏng sử dụng UVM Testbench.	11
2.1.4. UVM Factory, Field Macro và các tiện ích tích hợp	14
2.2. Xây dựng mô hình tin cậy sử dụng SystemVerilog Direct Programming Interface.....	17
2.2.1. Tổng quan về DPI	17
2.2.2. Cách thức hoạt động của DPI.....	18
2.2.3. Xây dựng mô hình tin cậy cho thiết kế kết hợp sử dụng DPI.....	19
2.3. Kiểm tra hành vi thiết kế sử dụng SystemVerilog Assertion.....	21
2.3.1. Tổng quan về SystemVerilog Assertion.....	21
2.3.2. Các loại Assertion	22
2.3.2.1. Immediate Assertion	22
2.3.2.2. Concurrent Assertion	23
2.3.3. Các lớp của Concurrent Assertion.....	24
2.3.4. Các toán tử của Concurrent Assertion	25

2.3.5.	Cú pháp Concurrent Assertion.....	30
2.3.6.	Cấu trúc bind của SystemVerilog	31
Chương 3.	MÔ HÌNH THỰC TẾ MÔI TRƯỜNG KIỂM TRA THIẾT KẾ	33
3.1.	Mô hình tổng quan môi trường kiểm tra thiết kế	33
3.2.	Thiết kế CNN IP được dùng để xây dựng môi trường kiểm tra.....	35
3.2.1.	Tổng quan thiết kế CNN IP	35
3.2.2.	Mô tả input và output của thiết kế	36
3.3.	Đặc trưng cấu trúc môi trường UVM Testbench cho thiết kế tích chập	37
3.3.1.	Sequence Item.....	37
3.3.2.	Sequence.....	38
3.3.3.	Driver.....	39
3.3.4.	Monitor.....	41
3.3.5.	Scoreboard	43
3.3.6.	Agent.....	46
3.3.7.	Env.....	48
3.3.8.	Test	49
3.4.	Mô hình tin cậy cho thiết kế tích chập sử dụng DPI-C	50
3.5.	Kiểm tra hành vi Controller module của thiết kế tích chập sử dụng SystemVerilog Assertion	52
3.6.	Môi trường kiểm tra thiết kế CNN IP ShuffleNetV2.....	55
3.6.1.	Mô tả thiết kế CNN IP ShuffleNetV2	55
3.6.2.	Mô hình tin cậy cho thiết kế tích chập của CNN IP ShuffleNet.....	58
3.6.3.	Đặc trưng cấu trúc môi trường UVM Testbench của thiết kế CNN IP ShuffleNetV2	59

3.6.4. Tối ưu tính tái sử dụng của môi trường kiểm tra.....	59
Chương 4. KIỂM TRA VÀ ĐÁNH GIÁ.....	63
4.1. Tiêu chí đánh giá	63
4.2. Phương pháp đánh giá kết quả.....	64
4.3. Kết quả mô phỏng của UVM Testbench.....	67
4.4. Kết quả kiểm tra hành vi thiết kế	71
4.5. Kết quả so sánh giữa kết quả thực tế và kết quả tin cậy (CNN IP ShuffleNetV2).....	73
Chương 5. TỔNG KẾT.....	75
5.1. Kết luận.....	75
5.2. Giới hạn đề tài	76
5.3. Hướng phát triển.....	76

DANH MỤC HÌNH

Hình 2-1: Cấu trúc của một UVM Testbench cơ bản	6
Hình 2-2: Mô hình các thành phần bên trong thư viện UVM [2]	9
Hình 2-3: Port và Export của Producer và Consumer.....	10
Hình 2-4: Mô hình Port, Export và Analysis Port.....	11
Hình 2-5: Các Phase trong quá trình mô phỏng UVM Testbench.....	12
Hình 2-6: Trình tự thực hiện giữa các phần tử của các Phase	14
Hình 2-7: UVM Factory và các phương thức cung cấp cho người dùng.....	15
Hình 2-8: Giao tiếp giữa SystemVerilog và C	18
Hình 2-9: Cấu trúc và cách thức hoạt động của SystemVerilog sử dụng DPI.....	18
Hình 2-10: Tham chiếu kiểu dữ liệu giữa SystemVerilog và C sử dụng DPI-C	19
Hình 2-11: Mô hình tin cậy kiểm tra kết quả DUT	20
Hình 2-12: Trình tự hoạt động của DPI-C và SystemVerilog trong mô phỏng.....	21
Hình 2-13: Kết quả Assertion violation	23
Hình 2-14: Các lớp bên trong Concurrent Assertion	24
Hình 2-15: Waveform mẫu cho đoạn code bên trên.....	26
Hình 2-16: Kết quả của 2 tiến trình Assertion P1 và P2	27
Hình 2-17: Dạng sóng cho ví dụ toán tử lặp của Concurrent Assertion	29
Hình 2-18: Cú pháp của một mệnh đề Concurrent Assertion	30
Hình 2-19: Cấu trúc bind của SystemVerilog.....	32
Hình 3-1: Tổng quan mô hình môi trường kiểm tra thiết kế	33
Hình 3-2: Kiến trúc thiết kế CNN.....	36
Hình 3-3: Biến enum bên trong Sequence Item.....	37
Hình 3-4: Flow chart của function body() bên trong Sequence	38
Hình 3-5: Các Phase chính bên trong lớp Driver	39
Hình 3-6: Run Phase của lớp Driver.....	40
Hình 3-7: Task drive() của lớp Driver.....	40
Hình 3-8: Giao tiếp giữa lớp Driver và Sequencer	41

Hình 3-9: Các Phase chính bên trong lớp Monitor	41
Hình 3-10: uvm_analysis_port được khai báo bên trong Monitor	42
Hình 3-11: Run Phase của lớp Monitor	42
Hình 3-12: uvm_analysis_imp được khai báo bên trong Scoreboard	43
Hình 3-13: Các Phase chính của lớp Scoreboard	44
Hình 3-14: Cấu trúc bộ nhớ kernel_ram của CNN IP.....	45
Hình 3-15: Bộ nhớ ảo bên trong Scoreboard	46
Hình 3-16: Cấu trúc bộ nhớ ảo được sao chép từ kernel_ram	46
Hình 3-17: Các Phase chính của lớp Agent.....	47
Hình 3-18: Build Phase và Connect Phase của lớp Agent	47
Hình 3-19: Các Phase chính của lớp Env.....	48
Hình 3-20: Build Phase và Connect Phase của lớp Env	49
Hình 3-21: Các Phase chính của lớp Test	49
Hình 3-22: Các hiệu chỉnh cho môi trường test.....	50
Hình 3-23: Hiệu chỉnh về set_drain_time	50
Hình 3-24: Kiến trúc Datapath module của khối tích chập.....	51
Hình 3-25: Các tầng của mô hình DPI	52
Hình 3-26: Concurrent Assertion cho các tín hiệu điều khiển.....	55
Hình 3-27: Tổng quan luồng hoạt động của ShuffleNetV2	56
Hình 3-28: Tổng quan thiết kế phần cứng ShuffleNetV2.....	57
Hình 3-29: Các tầng bên trong môi trường kiểm tra	58
Hình 3-30: Lớp nền Sequence được tham số hóa.....	61
Hình 3-31: Lớp con khởi tạo từ lớp nền	62
Hình 4-1: Một số hình ảnh trong tập dữ liệu đầu vào	64
Hình 4-2: Folder chứa kết quả mô phỏng	65
Hình 4-3: Kết quả tin cậy và kết quả thực tế	65
Hình 4-4: Kết quả so sánh từng trường hợp.....	66
Hình 4-5: Biểu đồ thống kê kết quả so sánh	66
Hình 4-6: Kết quả của quá trình mô phỏng	67

Hình 4-7: Thông tin có ý nghĩa trích xuất từ file báo cáo	68
Hình 4-8: Trạng thái Reset của UVM Testbench.....	69
Hình 4-9: Trạng thái tải trọng số đến DUT của UVM Testbench.....	69
Hình 4-10: Trạng thái thực hiện gửi các gói tin data_in đến DUT	70
Hình 4-11: Kết quả kiểm thử ở Check Phase.....	70
Hình 4-12: Thống kê kết quả ở Report Phase.....	71
Hình 4-13: Kết quả mệnh đề Concurrent Assertion	72
Hình 4-14: Waveform các tín hiệu điều khiển của DUT	72
Hình 4-15: Thông kê kết quả kiểm tra kênh 7	73
Hình 4-16: Đồ thị thống kê kết quả kiểm tra kênh 7	74

DANH MỤC BẢNG

Bảng 2-1: Các lớp con của uvm_object và uvm_component.....	10
Bảng 2-2: Một số Field Macro thông dụng cho các kiểu dữ liệu cơ bản	16
Bảng 3-1: Mô tả input và output của thiết kế CNN.....	36
Bảng 3-2: Mô tả các phép tính và số chu kỳ để hoàn thành của module Datapath ..	53
Bảng 3-3: Các phép tính và chu kỳ tín hiệu điều khiển của chúng tích cực	54
Bảng 4-1: Tiêu chí đánh giá môi trường kiểm tra	63

DANH MỤC TỪ VIẾT TẮT

UVM	Universal Verification Methodology
RTL	Register Transfer Level
SoC	System on Chip
DPI	Direct Programming Interface
SVA	SystemVerilog Assertion
EDA	Electronic Design Automation
DUT	Design Under Test
RAL	Register Abstraction Layer

TÓM TẮT KHÓA LUẬN

Trong đề tài, nhóm tập trung nghiên cứu các cơ sở lý thuyết cần thiết để có thể hiện thực hoá môi trường kiểm tra cho thiết kế CNN IP. Với nội dung trung tâm chính là xây dựng môi trường dựa trên phương pháp và cấu trúc của UVM Framework, nhóm cần hiểu rõ các kiến thức cơ bản của UVM gồm kiến trúc các lớp bên trong UVM và chức năng của chúng, phương thức giao tiếp giữa các lớp sử dụng UVM TLM, quá trình mô phỏng của môi trường dựa trên cơ chế UVM Phasing, thư viện tích hợp UVM cung cấp cho người dùng các lớp nền và các hàm tương ứng cũng như các tiện ích hỗ trợ người dùng trong quá trình xây dựng môi trường UVM Testbench. Việc hiện thực môi trường được nhóm thực hiện trên EDA Tool Xilinx Vivado và môi trường Linux Shell.

Bên cạnh việc sử dụng UVM vào việc xây dựng môi trường kiểm tra, áp dụng Shell, TCL và Perl Scripts cũng hỗ trợ tăng hiệu quả và giảm thời gian cho quá trình kiểm tra thông qua khả năng tự động hóa của Scripts. Đồng thời, một mô hình tin cậy của thiết kế được xây dựng dựa trên ngôn ngữ C và có khả năng giao tiếp với UVM Testbench thông qua DPI-C của SystemVerilog giúp việc xác định tính đúng đắn và độ tin cậy của thiết kế được tối ưu hơn.

Trước khi bắt đầu quá trình xây dựng môi trường kiểm tra cho thiết kế CNN IP, nhóm cũng cần nghiên cứu về mô tả của thiết kế, bao gồm các cổng đầu vào và đầu ra cũng như chức năng của thiết kế và các submodule của nó.

Chương 1. TÌM HIỂU TỔNG QUAN

1.1. Giới thiệu

Trong quá trình thiết kế một mạch RTL, quá trình đánh giá và kiểm tra là một trong những nhân tố then chốt trong việc xác định thiết kế có hoạt động như mong đợi hay không, đồng thời quá trình này cũng cần phải xác định được mức độ hoàn thiện, tính đúng đắn và độ tin cậy của thiết kế.

Mục tiêu chính của việc kiểm tra thiết kế là tìm lỗi của thiết kế đó, quá trình kiểm tra được hiện thực bằng cách đưa các trường hợp đầu vào khác nhau và xác định kết quả đầu ra có chính xác hay không. Độ tin cậy của thiết kế phụ thuộc vào số lượng mẫu thử đầu vào và mức độ chính xác của đầu ra. Do độ phức tạp của quá trình kiểm tra tỉ lệ thuận với độ phức tạp của thiết kế, đặc biệt với các thiết kế như SoC, các phương pháp kiểm tra thiết kế được các tập đoàn lớn lần lượt được phát triển sau sự ra đời của ngôn ngữ SystemVerilog: eRM, RVM, VMM, AVM, OVM và UVM [6]. Các phương pháp này tận dụng điểm mạnh của ngôn ngữ SystemVerilog là áp dụng lập trình hướng đối tượng vào quá trình xây dựng Testbench, hỗ trợ người kiểm tra thiết kế có thể tối ưu tính tự động hóa và khả năng tái sử dụng các phần tử có sẵn bên trong Testbench, giúp tăng hiệu quả và giảm thiểu thời gian kiểm tra. Trong các phương pháp kiểm tra trên, nổi bật nhất chính là UVM (Universal Verification Methodology) được xem như phương pháp kế thừa điểm mạnh của các phương pháp tiền thân [5], UVM cũng được chuẩn hóa và liên tục phát triển bởi Accellera từ 2011 tới nay và được sử dụng rộng rãi trên thế giới ở thời điểm hiện tại. UVM cung cấp cho người dùng một thư viện các lớp có khả năng tự động hóa và tích hợp các tính năng tiện ích hỗ trợ người kiểm tra trong quá trình xây dựng Testbench.

Bên cạnh việc sử dụng UVM vào việc xây dựng môi trường kiểm tra, áp dụng Shell và TCL Scripts cũng hỗ trợ tăng hiệu quả và giảm thời gian cho quá trình kiểm tra thông qua khả năng tự động hóa của Scripts. Đồng thời, một mô hình tin cậy của thiết kế được xây dựng dựa trên ngôn ngữ C và có khả năng giao tiếp với

UVM Testbench thông qua DPI-C của SystemVerilog giúp việc xác định tính đúng đắn và độ tin cậy của thiết kế được tối ưu hơn.

1.2. Mục tiêu đề tài

Mục tiêu của nhóm khi thực hiện đề tài này chính là hiện thực kiểm tra thành công một thiết kế CNN IP có sẵn sử dụng phương pháp UVM. Xây dựng và viết một môi trường UVM Testbench hoàn chỉnh để thực hiện việc kiểm tra thiết kế, độ hoàn chỉnh của UVM Testbench được đánh giá qua các tiêu chí:

- Có đầy đủ các thành phần kiểm tra tiêu chuẩn (uvm_sequence, uvm_driver, uvm_sequencer, uvm_monitor, uvm_agent, uvm_env, uvm_test)
- Các phần tử trong môi trường kiểm tra có khả năng tái sử dụng
- Có thể phân chia để kiểm tra từng phần tử trong thiết kế, mỗi submodule của thiết kế CNN đều có phần tử kiểm tra riêng trong môi trường UVM Testbench
- Có khả năng đánh giá độ chính xác của IP thông qua môi trường UVM Testbench đã viết với nhiều các trường hợp khác nhau (bao gồm trường hợp thông thường và trường hợp góc) và thực hiện thu thập functional coverage.

1.3. Tổng quan đề tài

Ở đề tài này, nhóm quyết định thực hiện việc xây dựng một môi trường kiểm tra thiết kế cho một CNN IP áp dụng phương pháp kiểm tra thiết kế UVM. Môi trường kiểm tra này có khả năng tự động hóa và tính tái sử dụng cao, để đạt được mục tiêu này bên cạnh việc sử dụng UVM nhóm cũng phải áp dụng kỹ thuật Scripting bao gồm Shell, Tcl và Perl vào quá trình hiện thực thiết kế. Một vài kỹ thuật khác được nhóm sử dụng để tăng hiệu suất công việc và chất lượng đầu ra của môi trường kiểm tra đó là DPI sử dụng ngôn ngữ lập trình C/C++ và SVA.

Ưu điểm của môi trường kiểm tra thiết kế mà nhóm xây dựng đó là có thể tự động hóa quá trình kiểm tra theo mong muốn của người sử dụng. Ngôn ngữ đặc tả và kiểm tra phần cứng SystemVerilog và Framework UVM được sử dụng để xây dựng môi trường kiểm tra; các Script Shell và Tcl được sử dụng để điều khiển môi

trường mô phỏng cũng như các chức năng được thực thi bên trong EDA tool; kết quả sau khi thực hiện mô phỏng được tiến hành phân loại thành các file report phục vụ cho quá trình kiểm tra và đánh giá thông qua ngôn ngữ Perl và Python. Đồng thời, khả năng tái sử dụng của môi trường mô phỏng cũng được nhóm hướng tới thông qua việc sử dụng phương pháp tham số hóa cho các thành phần bên trong UVM Testbench. DPI-C được sử dụng để tạo nên mô hình tin cậy cho DUT của thiết kế CNN, nhờ đó việc kiểm tra các thiết kế CNN IP khác nhau có thể được thực hiện thông qua thay đổi mô hình tin cậy được viết bằng ngôn ngữ C/C++.

Chương 2. CƠ SỞ LÝ THUYẾT

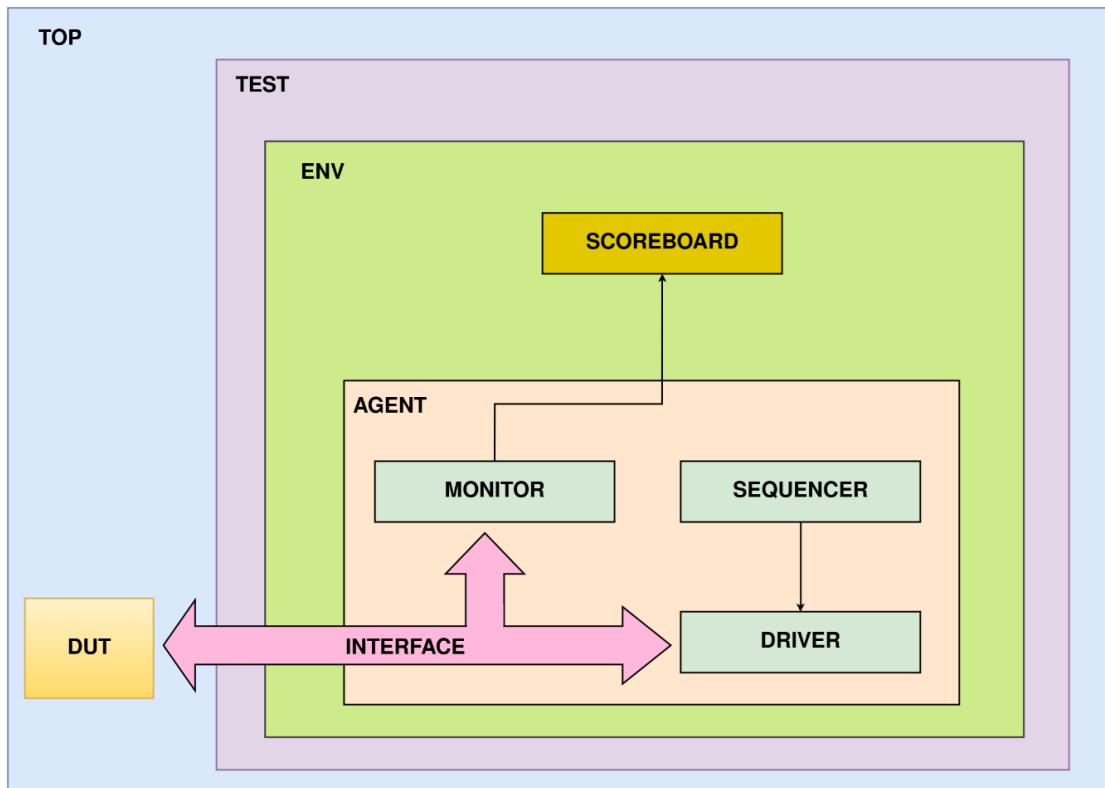
Để hiện thực và đạt được các mục tiêu đã đề ra của đề tài, nhóm thực hiện việc nghiên cứu về lý thuyết và phương pháp xây dựng môi trường kiểm tra sử dụng Framework UVM bao gồm: cấu trúc và các thành phần bên trong môi trường UVM, giao thức TLM, UVM Phase, UVM Factory, Field Macros và các tiện ích cung cấp bởi thư viện tích hợp UVM. Ngoài ra nhóm cũng nghiên cứu và áp dụng tính năng Direct Programming Interface và Assertion được cung cấp bởi SystemVerilog để tăng hiệu suất kiểm tra cho môi trường UVM Testbench. Ở Chương 2, nhóm trình bày về cơ sở lý thuyết được nhóm áp dụng vào đề tài.

2.1. Mô hình môi trường kiểm tra thiết kế UVM

Thông thường, khi tiến hành việc kiểm tra một thiết kế sử dụng SystemVerilog, một mô hình kiểm tra bao gồm các đối tượng của các lớp khác nhau được gọi lên bởi người thiết kế ở tầng cao nhất của Testbench, các lớp này được định nghĩa hoàn toàn do người thiết kế tùy theo mục đích sử dụng, giao tiếp chính của các đối tượng được thực hiện chủ yếu qua giao thức mailbox hoặc semaphore. Phương pháp xây dựng môi trường kiểm tra theo hướng đối tượng này có điểm mạnh giúp người kiểm tra có thể kiểm soát môi trường kiểm tra tốt hơn thông qua việc quản lý các lớp và các đối tượng của chúng, với điểm mạnh là tính kế thừa và tính đa hình hỗ trợ khả năng tái sử dụng các lớp của Testbench. Tận dụng những ưu điểm này của SystemVerilog, UVM là một trong các phương pháp kiểm tra cải tiến và cung cấp các công cụ hỗ trợ mạnh mẽ hơn hỗ trợ cho người thiết kế môi trường kiểm tra có thể tối ưu hóa hiệu suất công việc, nổi bật nhất trong đó chính là khả năng không phụ thuộc vào các công cụ mô phỏng EDA bởi lý do UVM là một phương pháp kiểm tra thiết kế đã được chuẩn hóa.

2.1.1. Cấu trúc và các thành phần chính bên trong một UVM Testbench

Một môi trường kiểm tra thiết kế dựa trên cấu trúc của UVM Testbench cơ bản có các thành phần như sau:



Hình 2-1: Cấu trúc của một UVM Testbench cơ bản

- **UVM Testbench**
 - UVM Testbench bao gồm 2 thành phần chính là UVM Test & DUT (và các config giữa chúng).
- **UVM Test**
 - UVM Test là tầng cao nhất của các UVM components, thực hiện 3 chức năng chính:
 - Khởi tạo top-level environment.
 - Thực hiện các tùy chỉnh trong môi trường (through qua cơ chế Factory Override và Configuration Database).
 - Gửi kích thích đến cho DUT bằng cách gọi UVM Sequence thông qua Environment.
- **UVM Environment**
 - UVM Environment là lớp chứa các thành phần gồm UVM Agent, Scoreboard, hoặc các UVM Environment khác (Top Environment chứa các Environment khác của DUT). Ví dụ:

Một Environment SoC Design chứa PCIe Environment, USE Environment, Mem Controller Environment.

- **UVM Scoreboard**

- UVM Scoreboard có chức năng kiểm tra hành vi của DUT, UVM Scoreboard nhận Transaction chứa các input và output của DUT thông qua Agent Analysis Port, đưa input đến Reference Model (kết quả tin cậy) và so sánh kết quả tin cậy với kết quả của DUT.

- **UVM Agent**

- UVM Agent chứa các lớp tương tác trực tiếp với Interface. Agent chứa:
 - UVM Sequence: kiểm soát kích thích.
 - UVM Driver: đưa các kích thích vào interface.
 - UVM Scoreboard: theo dõi các thay đổi của interface.
 - Agent có thể chứa các thành phần khác như Coverage Collectors và Protocol Checker.

- **UVM Sequencer**

- UVM Sequencer có chức năng kiểm soát các kích thích được đưa vào DUT, các kích thích này được sinh ra bởi một hoặc nhiều UVM Sequence khác nhau.

- **UVM Sequence**

- UVM Sequence có chức năng tạo ra các kích thích khác nhau.

- **UVM Driver**

- UVM Driver nhận các gói tin UVM Sequence Item từ UVM Sequencer và truyền các gói tin này vào Interface. Driver cũng có một TLM port để nhận các gói tin từ Sequencer và có quyền truy xuất vào Interface để giao tiếp với DUT.

- **UVM Monitor**

- UVM Monitor lấy mẫu từ Interface, đóng gói các thông tin thành gói tin để chuyển các gói tin đó tới các thành phần khác.

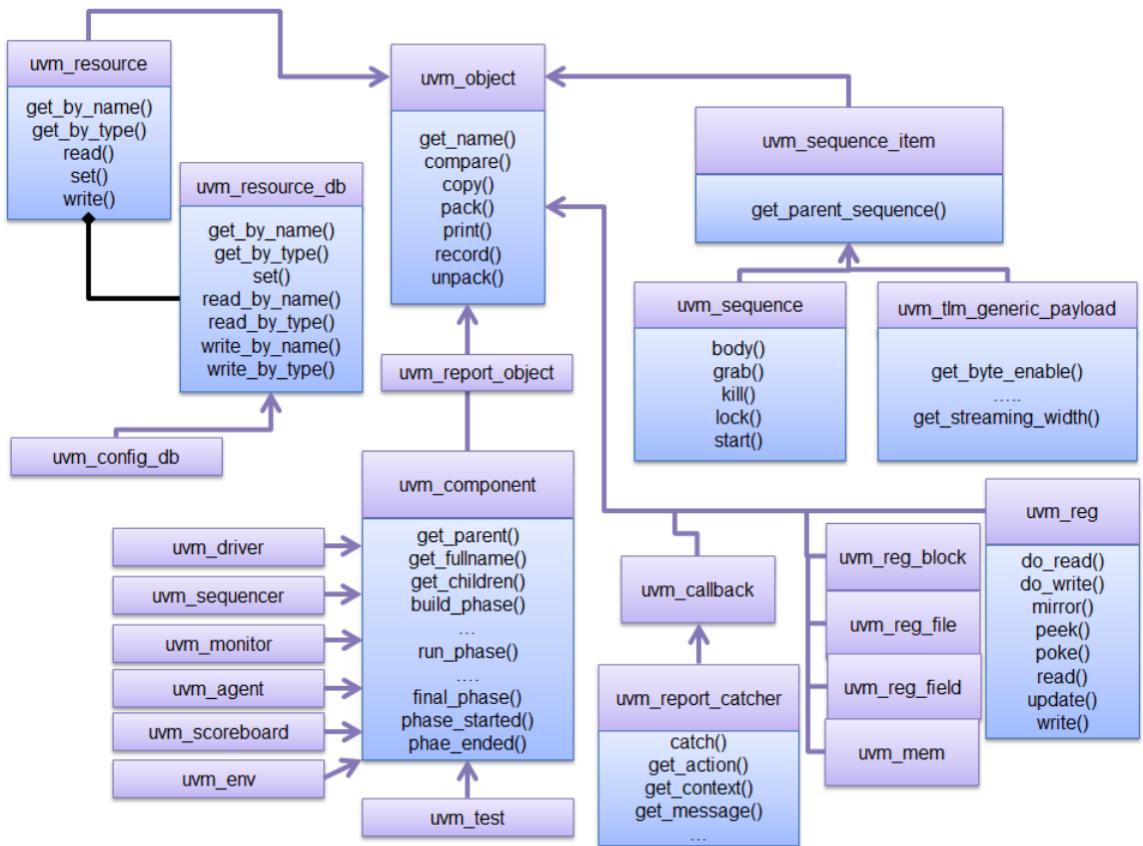
Tương tự như Driver, Monitor cũng cần có quyền truy xuất trực tiếp tới Interface để lấy mẫu và có một TLM Analysis Port để broadcast các gói tin được khởi tạo từ Monitor.

Quan sát Hình 2-1 có thể thấy ở tầng cao nhất, DUT được phân tách ra khỏi UVM Test, UVM Test là lớp cao nhất trong UVM Testbench và là lớp chứa các phần tử kiểm tra khác bên trong nó, việc thay đổi các biến tùy chỉnh của UVM Testbench cũng được thực hiện ở lớp này. Do môi trường kiểm tra và DUT được tách khỏi nhau hoàn toàn, DUT và UVM Test sử dụng Interface để giao tiếp qua lại cũng như truyền và nhận các gói tin trong quá trình mô phỏng (Interface được kết nối trực tiếp với UVM Testbench ở lớp Driver và Monitor).

Các lớp trên được cung cấp bởi thư viện tích hợp bên trong UVM, cùng với các thuộc tính và phương thức đặc trưng của từng lớp, người thiết kế môi trường kiểm tra thực hiện việc gọi các lớp từ thư viện có sẵn và tùy chỉnh chúng để xây dựng một Testbench hoàn chỉnh.

Điểm mạnh của việc sử dụng các lớp từ thư viện tích hợp của UVM bao gồm:

- Đa dạng các tính năng tích hợp sẵn - Thư viện tích hợp UVM cung cấp cho người thiết kế môi trường kiểm tra nhiều tính năng hữu ích cần thiết cho quá trình kiểm tra bao gồm các thao tác hoàn chỉnh như hàm print(), copy(), cơ chế Phasing, các phương thức Factory Override và nhiều tiện ích khác.
- Người thiết kế có thể triển khai mô hình một cách chính xác và nhất quán theo nguyên tắc UVM đề ra - các thành phần bên trong Hình 2-1 và Hình 2-2 đều có thể xây dựng từ lớp cha tương ứng được cung cấp bên trong thư viện tích hợp UVM. Việc tạo lớp con từ các lớp cha được tích hợp bên trong thư viện UVM hỗ trợ khả năng dễ dàng đọc và hiểu code bởi vai trò của các lớp con đã được định nghĩa từ trước bởi lớp cha của chúng.



Hình 2-2: Mô hình các thành phần bên trong thư viện UVM [2]

Hình 2-2 thể hiện cấu trúc của thư viện tích hợp được UVM cung cấp hỗ trợ người thiết kế môi trường kiểm tra có thể sử dụng các thành phần kiểm tra có tính ổn định và tái sử dụng cao để xây dựng một môi trường kiểm tra UVM cho riêng mình.

Các thành phần bên trong môi trường kiểm tra UVM được cấu thành từ hai lớp cơ sở của thư viện tích hợp đó là `uvm_object` và `uvm_component`. Trong quá trình mô phỏng, các gói tin là đối tượng của lớp con được khởi tạo từ `uvm_object`, các gói tin này di chuyển qua lại bên trong môi trường kiểm tra và có dữ liệu thay đổi liên tục tùy từng khoảng thời gian khác nhau, do đó lớp tạo nên các gói tin được gọi là thành phần động (Dynamic Component). Các lớp còn lại có thuộc tính giữ nguyên suốt quá trình mô phỏng kiểm tra được gọi là các thành phần tĩnh (Static Component), các lớp này bao gồm Driver, Sequencer, Monitor, Agent, Scoreboard và Env, chúng đều được khởi tạo từ lớp cơ sở là `uvm_component`. Danh sách các lớp con của `uvm_object` và `uvm_component` được liệt kê qua Bảng 2-1.

Bảng 2-1: Các lớp con của uvm_object và uvm_component

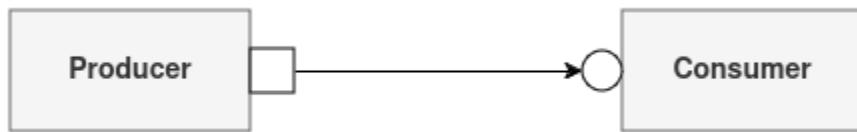
uvm_object	uvm_sequence_item uvm_sequence
uvm_component	uvm_driver uvm_sequencer uvm_monitor uvm_agent uvm_scoreboard uvm_env uvm_test

2.1.2. Transaction-Level Modeling

Các thành phần bên trong UVM Testbench được liên kết với nhau thông qua một phương thức đặc trưng được cung cấp bởi UVM chính là phương thức Transaction-Level Modeling (TLM). Cụ thể, nguyên tắc hoạt động của UVM được trùu tượng hóa bằng việc quản lý và theo dõi các gói tin được truyền tới và lui giữa các phần tử bên trong Testbench, các gói tin chứa các thông tin tín hiệu cụ thể của đầu vào và đầu ra của DUT ở một thời điểm trong quá trình mô phỏng, và các gói tin được truyền và nhận giữa các phần tử bên trong UVM Testbench thông qua TLM.

TLM được cung cấp bởi thư viện tích hợp UVM và chứa 3 phần tử chính:

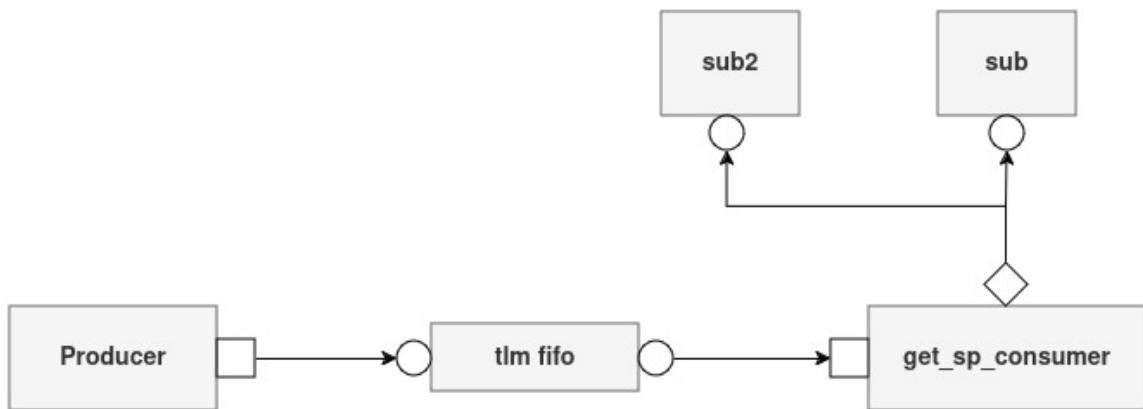
- Port: thành phần nằm trong Producer được dùng để xuất gói tin
- Export: thành phần nằm trong Consumer được dùng để nhận gói tin
- UVM Analysis Port: là một Port đặc biệt có chức năng tương tự như broadcast gói tin từ một Port sang nhiều Export



Hình 2-3: Port và Export của Producer và Consumer

Hình 2-3 thể hiện một liên kết đơn giản nhất giữa 2 thành phần UVM là Producer và Consumer, Producer chứa một Port được biểu diễn bằng hình vuông,

Export nằm trong Consumer biểu diễn bằng hình tròn và mũi tên là hướng di chuyển của gói tin.



Hình 2-4: Mô hình Port, Export và Analysis Port

Hình 2-4 thể hiện một liên kết có chứa cả 3 thành phần của UVM TLM bao gồm Port, Export và Analysis Port. Analysis Port được biểu diễn bởi hình kim cương nằm bên trong thành phần có tên get_sp_consumer. Các gói tin đi từ Producer đến một hàng đợi TLM Fifo và sau đó đi tới get_sp_consumer, lúc này gói tin sẽ được analysis port truyền đến nhiều thành phần khác nhau như sub và sub2. Việc truyền gói tin đến nhiều Export khác nhau được thực hiện bởi hàm write() tích hợp bên trong Analysis Port được cung cấp bởi thư viện UVM.

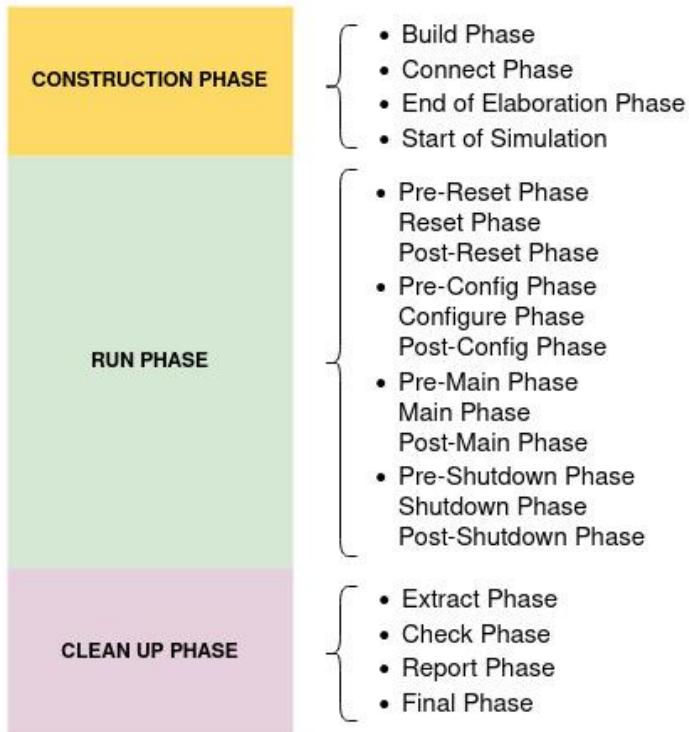
2.1.3. Các Phase bên trong quá trình mô phỏng sử dụng UVM Testbench

Quá trình mô phỏng của một UVM Testbench được chia thành nhiều giai đoạn khác nhau và các giai đoạn này được gọi là Phase (Hình 2-5). Ở mỗi Phase, một tác vụ đặc trưng tương ứng với Phase đó được thực thi, các Phase được tiến hành lần lượt theo một trình tự nhất định được định nghĩa bởi thư viện UVM. Quá trình mô phỏng được xem như hoàn thành khi Phase cuối cùng hoàn tất tác vụ của nó.

Có 2 loại phase bao gồm:

- **Time-consuming Phase:** Phase có sử dụng thời gian mô phỏng bao gồm Run Phase. Vì các Time-consuming Phase sử dụng thời gian mô phỏng nên chúng sẽ phải được thực hiện dưới việc thực thi các task.

- **Non-Time-consuming Phase:** Phase không sử dụng thời gian mô phỏng bao gồm Build Phase, Connect Phase, End of Elaboration Phase, Start of Simulation Phase, Extract Phase, Check Phase, Report Phase, Final Phase. Non-Time-consuming Phase được thực hiện bởi việc thực thi các function.



Hình 2-5: Các Phase trong quá trình mô phỏng UVM Testbench

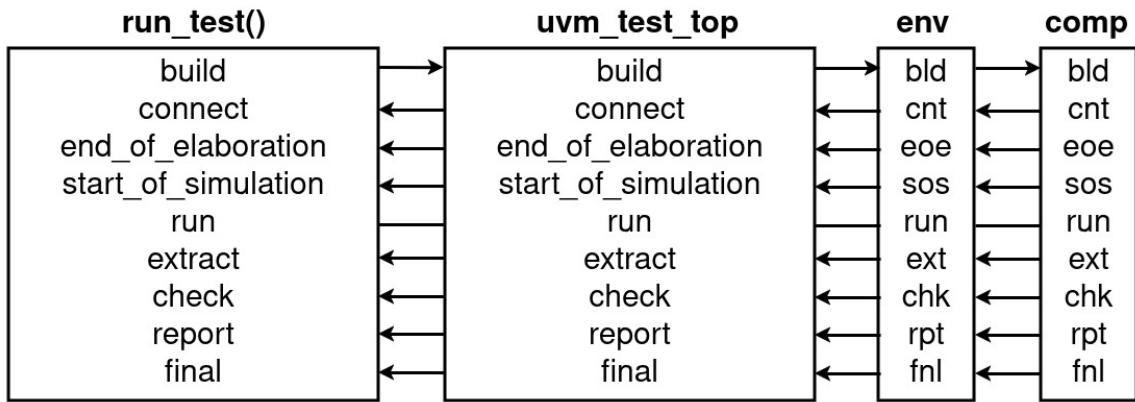
Vai trò của các Phase chính được mô tả như sau:

- **Build Phase** là giai đoạn khởi tạo các thành phần của môi trường kiểm tra. Cụ thể, các đối tượng được khởi tạo từ các lớp mà người kiểm tra đã định nghĩa từ lớp cha của thư viện UVM Testbench. Các thành phần cần thiết cho môi trường kiểm tra được hoàn tất việc xây dựng ở Build Phase.
- **Connect Phase** là giai đoạn kế tiếp sau khi Build Phase hoàn tất. Ở Connect Phase, các TLM Port được khởi tạo để kết nối các thành phần của UVM Testbench với nhau.
- **End of Elaboration** là Phase hỗ trợ người thiết kế môi trường kiểm tra xác định và kiểm tra cấu trúc của UVM Testbench. Công dụng phổ biến của

Phase này chính là in cấu trúc của UVM Testbench để kiểm tra mô hình cây gia phả của môi trường kiểm tra.

- **Reset Phase** là giai đoạn được dành riêng cho DUT và Interface thực hiện việc reset hành vi. Ví dụ: giai đoạn này được dùng để reset mạch về trạng thái mặc định.
- **Configure Phase** là giai đoạn được dùng để điều chỉnh DUT và bất kỳ phần tử nhô nào bên trong Testbench để có thể sẵn sàng bắt đầu thực hiện các Testcase
- **Main Phase** là giai đoạn các thành phần bên trong Testbench hoạt động, các gói tin được truyền từ Driver tới DUT và từ DUT tới Monitor.
- **Shutdown Phase** được dùng để chắc chắn rằng các kích thích được tạo ra trong quá trình mô phỏng đã hoàn toàn được đi qua DUT và không còn tín hiệu nào chưa hoàn tất bên trong DUT.
- **Extract Phase** là giai đoạn trích xuất các tín hiệu từ Scoreboard và giám sát Functional Coverage.
- **Check Phase** là giai đoạn kiểm tra DUT có hoạt động đúng đắn hay không, giai đoạn này đồng thời cũng được dùng để tìm ra các lỗi có thể xảy ra trong quá trình mô phỏng.
- **Report Phase** là giai đoạn chủ yếu được dùng để thông báo kết quả của quá trình mô phỏng ra màn hình và ra các tệp tin
- **Final Phase** dùng để thực hiện các công việc còn lại chưa hoàn thành của quá trình kiểm tra.

Trong quá trình mô phỏng, các Phase được bắt đầu và kết thúc theo một trình tự nhất định, Phase kế tiếp chỉ có thể bắt đầu khi Phase hiện tại đã hoàn thành và kết thúc. Các Phase được áp dụng đối với tất cả các thành phần bên trong môi trường kiểm tra UVM, nghĩa là các phần tử như Driver, Monitor, Agent, Environment hoặc Test đều phải thực hiện tất cả các Phase trong quá trình mô phỏng. Tuy nhiên có sự khác biệt trong thứ tự thực hiện Phase giữa các thành phần tử phụ thuộc vào cây gia phả trong môi trường kiểm tra UVM, được thể hiện qua Hình 2-6.



Hình 2-6: Trình tự thực hiện giữa các phần tử của các Phase

Quan sát Hình 2-6, có thể thấy build phase được thực hiện theo thứ tự top-down, phần tử có thứ bậc lớn nhất trong cây gia phả của môi trường UVM Testbench sẽ thực hiện Build Phase đầu tiên và đi dần xuống phần tử nhỏ nhất. Ví dụ uvm_test_top là lớp cao nhất, do đó sẽ thực hiện Build Phase đầu tiên, kế tiếp env thực hiện Build Phase, sau đó tới các phần tử nhỏ hơn như Agent, và cuối cùng là các phần tử thấp nhất như Driver và Monitor được thực hiện Build Phase.

Các Phase còn lại ngoại trừ Run Phase, có thứ tự thực hiện bottom-up. Ngược lại với Build Phase, các phần tử nhỏ nhất trong cây gia phả được thực hiện trước và uvm_test_top được thực hiện cuối cùng.

Ở Run Phase, đây là giai đoạn các tín hiệu và kích thích di chuyển bên trong môi trường kiểm tra, các phần tử thực hiện phase này song song, nghĩa là ở giai đoạn này mọi phần tử trong UVM Testbench đều phải thực hiện cùng lúc, không theo trình tự trước sau.

2.1.4. UVM Factory, Field Macro và các tiện ích tích hợp

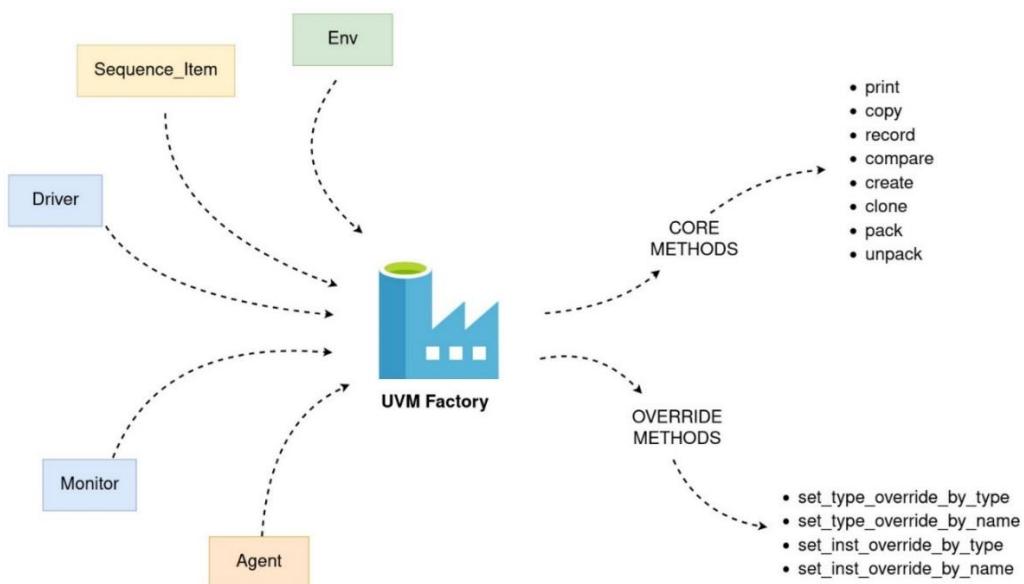
Factory là một cơ chế chính của UVM, cơ chế này hỗ trợ việc tăng tính linh hoạt và khả năng mở rộng Testbench cho người thiết kế môi trường kiểm tra. Có thể xem Factory là một biểu diễn trừu tượng của thư viện tích hợp UVM, tất cả các lớp trong môi trường kiểm tra đều được khởi tạo từ uvm_object và uvm_object đến từ Factory, nghĩa là khi ta thực hiện việc tạo các lớp và đối tượng khác nhau, đều được

thực hiện từ Factory và các lớp này đều phải được đăng ký với Factory với kiểu dữ liệu cụ thể.

Việc đăng ký các lớp tới Factory giúp người thiết kế có thể thực hiện thao tác ghi đè các lớp khi cần thiết, Factory cung cấp cho người dùng các tiện ích liên quan đến việc ghi đè và thay đổi trên toàn bộ môi trường kiểm tra cho các lớp đã được đăng ký trước đó.

Các thuộc tính của lớp cũng có thể đăng ký đến Factory để có thể sử dụng các tiện ích được cung cấp bởi Factory cho các đối tượng của lớp đó. Các tiện ích bao gồm: print, copy, record, compare, create, clone, pack và unpack. Ví dụ để so sánh hai đối tượng ta sử dụng compare(), để copy thuộc tính của một đối tượng sang một đối tượng khác ta sử dụng copy(). Các tiện ích trên nằm trong thư viện tích hợp UVM, người dùng chỉ cần viết lại mà chỉ cần gọi và sử dụng chúng, tuy nhiên cần phải thực hiện việc đăng ký thuộc tính của lớp tới Factory để có thể sử dụng chúng.

Các tiện ích ghi đè lớp và tiện ích cho thuộc tính lớp được thể hiện ở Hình 2-7 bên dưới.



Hình 2-7: UVM Factory và các phương thức cung cấp cho người dùng
Việc đăng ký thuộc tính của lớp tới Factory được thực hiện thông qua Field Macro. Với các biến có kiểu dữ liệu khác nhau sẽ có một Field Macro tương ứng được dùng để đăng ký tới Factory.

Bảng 2-2 là các Field Macro sử dụng cho các biến có kiểu dữ liệu int, string, enum, real, event. UVM cũng cung cấp các Field Macro cho kiểu dữ liệu phức tạp như mảng tĩnh, mảng động, hàng đợi và Associative Array.

Bảng 2-2: Một số Field Macro thông dụng cho các kiểu dữ liệu cơ bản

Utility and Field Macros for Components and Objects	
UTILITY MACROS	The <i>utils</i> macros define the infrastructure needed to enable the object/component for correct factory operation.
<code>`uvm_field_utils_begin</code>	
<code>`uvm_field_utils_end</code>	These macros form a block in which <code>`uvm_field_*</code> macros can be placed.
UVM_FIELD_* MACROS	Macros that implement data operations for scalar properties.
<code>`uvm_field_int</code>	Implements the data operations for any packed integral property.
<code>`uvm_field_object</code>	Implements the data operations for a <u>uvm_object</u> -based property.
<code>`uvm_field_string</code>	Implements the data operations for a string property.
<code>`uvm_field_enum</code>	Implements the data operations for an enumerated property.
<code>`uvm_field_real</code>	Implements the data operations for any real property.
<code>`uvm_field_event</code>	Implements the data operations for an event property.

Bảng 2-2 được lấy từ tài liệu UVM Class Reference Manual 1.2 (trang 454) chính thức từ Accellera, liệt kê một số Field Macro hỗ trợ cho các kiểu dữ liệu cơ bản. UVM Field Macro hỗ trợ hầu hết các kiểu dữ liệu của SystemVerilog (Field Macro kiểu dữ liệu khác cũng được liệt kê trong tài liệu này).

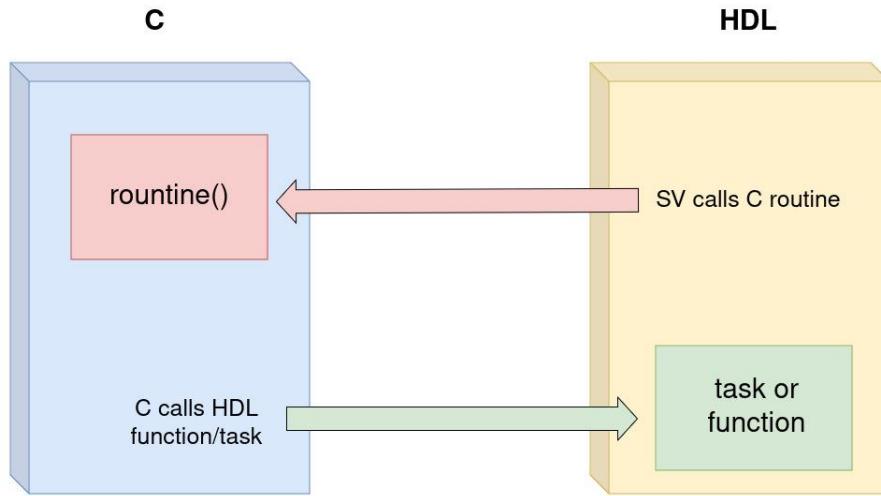
2.2. Xây dựng mô hình tin cậy sử dụng SystemVerilog Direct Programming Interface

Trong quá trình xây dựng môi trường kiểm tra cho một thiết kế phức tạp, để đạt được kết quả kiểm tra chính xác nhất, thông thường người thiết kế môi trường cần có một mô hình tin cậy cho môi trường kiểm tra thiết kế của mình, mô hình tin cậy này được xem như một mô hình đúng đắn phục vụ cho việc kiểm tra chức năng của thiết kế. Kết quả từ mô hình này được gọi là kết quả mong đợi, cũng chính là kết quả mà người thiết kế mong muốn có được từ DUT khi thực hiện kiểm tra chức năng. Mô hình tin cậy này có thể được xây dựng bởi các ngôn ngữ lập trình như C/C++ hoặc Python, và để môi trường mô phỏng (thường được viết bằng SystemVerilog) giao tiếp với các mô hình tin cậy trên được viết bằng các ngôn ngữ khác, SystemVerilog hỗ trợ một tính năng cho người dùng được gọi là Direct Programming Interface.

2.2.1. Tổng quan về DPI

Trong quá trình mô phỏng sử dụng ngôn ngữ đặc tả phần cứng SystemVerilog, người thiết kế môi trường kiểm tra có thể tận dụng chức năng Direct Programming Interface để SystemVerilog có thể giao tiếp với một ngôn ngữ lập trình cấp cao khác như C/C++.

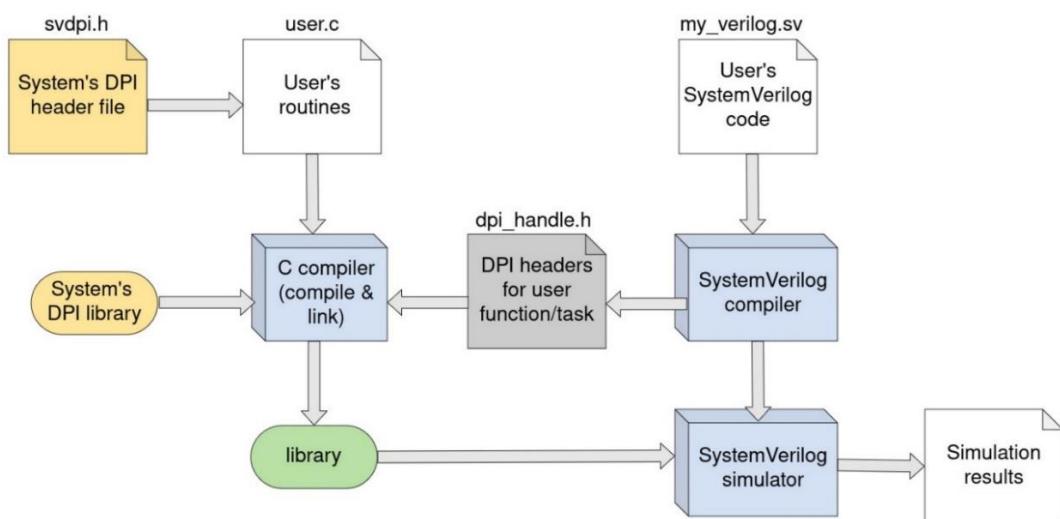
DPI là một cơ chế giao tiếp của SystemVerilog được dùng để ngôn ngữ với các ngôn ngữ lập trình cấp cao khác. Người dùng có thể dùng DPI để gọi một hàm C/C++ trong quá trình thực hiện mô phỏng và ngược lại, C/C++ cũng có thể gọi các phương thức Function hoặc Task từ SystemVerilog, nghĩa là ta có thể lấy tín hiệu từ SystemVerilog Testbench đưa vào C code hoặc truyền bất kỳ tín hiệu nào đến SystemVerilog code từ C code (Hình 2-8).



Hình 2-8: Giao tiếp giữa SystemVerilog và C

2.2.2. Cách thức hoạt động của DPI

Để sử dụng DPI, thư viện svdpi.h cần được thêm vào ở code C. Thư viện này chứa các kiểu dữ liệu đặc biệt và các hàm hỗ trợ việc mapping dữ liệu giữa SystemVerilog và C code. Ở SystemVerilog Code, người thiết kế cần thực hiện cú pháp import hàm từ C Code để trình mô phỏng có thể gọi hàm đó. Đồng thời Code C cũng cần được trình biên dịch tổng hợp thành một file .so trước khi thực hiện mô phỏng để trình mô phỏng của SystemVerilog gọi các hàm từ code C (Hình 2-9).



Hình 2-9: Cấu trúc và cách thức hoạt động của SystemVerilog sử dụng DPI

Khả năng gửi và nhận giá trị của các biến giữa SystemVerilog và C được thực hiện bởi thư viện svdpi.h với các biến tham chiếu như Hình 2-10 (được lấy từ trang 418 sách SystemVerilog for Verification 3rd Edition Chris Spear [4]), cho thấy các

kiểu dữ liệu biến phô biến như int, longint giữ nguyên khi truyền nhận các biến giữa SystemVerilog và C. Tuy nhiên một số kiểu dữ liệu thông dụng của SystemVerilog như kiểu dữ liệu bit được chuyển thành svBit hoặc unsigned char* ở ngôn ngữ lập trình C, kiểu dữ liệu logic cũng được chuyển thành svLogic hoặc unsigned char* ở ngôn ngữ lập trình C.

<i>SystemVerilog</i>	<i>C (input)</i>	<i>C (output)</i>
byte	char	char*
shortint	short int	short int*
int	int	int*
longint	long long int	long int*
shortreal	float	float*
real	double	double*
string	const char*	char**
string [N]	const char**	char**
bit	svBit or unsigned char	svBit* or unsigned char*
logic, reg	svLogic or unsigned char	svLogic* or unsigned char*
bit[N:0]	const svBitVecVal*	svBitVecVal*
reg[N:0] logic[N:0]	const svLogicVecVal*	svLogicVecVal*
unsized array[]	const svOpenArrayHandle	svOpenArrayHandle
chandle	const void*	void*

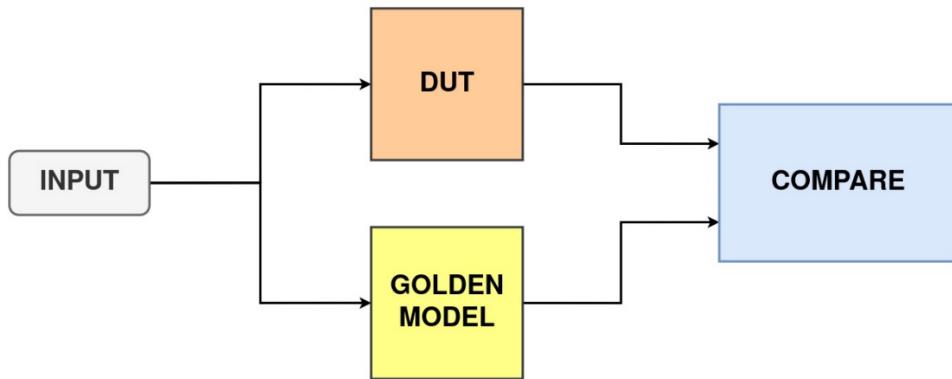
Hình 2-10: Tham chiếu kiểu dữ liệu giữa SystemVerilog và C sử dụng DPI-C

Thư viện svdpi.h cũng cung cấp các hàm xử lý các biến kiểu dữ liệu svBit hoặc svLogic. Tuy nhiên một số tính năng của thư viện svdpi.h bị giới hạn bởi EDA tools, tùy thuộc vào các EDA tool khác nhau mà một số tính năng được hỗ trợ hoặc không.

2.2.3. Xây dựng mô hình tin cậy cho thiết kế kết hợp sử dụng DPI

Tận dụng các điểm mạnh trên của chức năng Direct Programming Interface, nhóm quyết định xây dựng một mô hình tin cậy ở quá trình thiết kế môi trường kiểm tra, mô hình này được sử dụng để so sánh với kết quả thực tế có được, việc so sánh và đánh giá kết quả được thực hiện ở Check Phase, Scoreboard là thành phần chính trong môi trường kiểm tra đảm nhiệm vai trò này. Cụ thể, một mô hình tin cậy được viết bằng ngôn ngữ lập trình C sẽ đảm nhiệm vai trò là golden model, đầu vào của mô hình này tương tự như đầu vào của DUT. Đồng thời, gói tin được đưa

vào DUT cũng được đưa vào golden model, sau khi cả hai mô hình thực hiện tính toán trong quá trình mô phỏng, kết quả của golden model và kết quả có được từ DUT sẽ được so sánh để đánh giá tính chính xác của thiết kế (Hình 2-11).



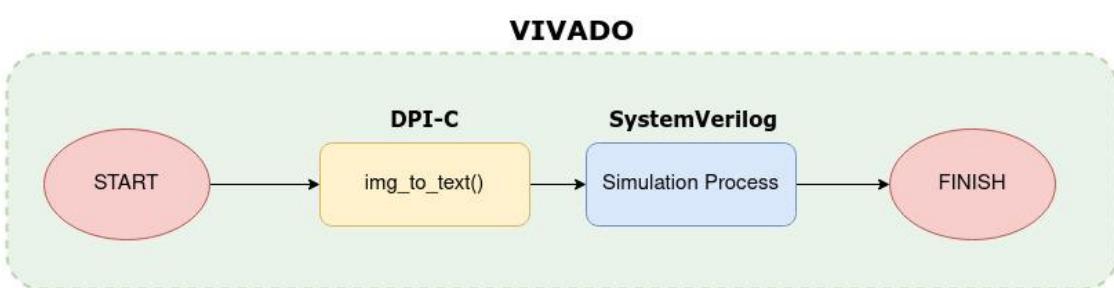
Hình 2-11: Mô hình tin cậy kiểm tra kết quả DUT

Sau khi kiểm tra và đánh giá kết quả, Scoreboard cũng đảm nhiệm vai trò đưa ra thông số độ tin cậy của mô hình sau khi mô phỏng dựa trên số lượng Testcase cũng như tỉ lệ đạt và không đạt của các Testcase cụ thể.

Ưu điểm của việc xây dựng mô hình tin cậy bằng ngôn ngữ C và áp dụng DPI để đưa vào quá trình mô phỏng đó là ta có thể sử dụng các thư viện và hàm tích hợp của C để xây dựng mô hình tin cậy, ưu điểm của ngôn ngữ C đó là có thể thực thi được ở nhiều môi trường khác nhau. Đồng thời nếu xây dựng mô hình tin cậy bằng SystemVerilog, điều này sẽ bị trùng lặp với quá trình thiết kế IP.

Một ưu điểm khác của DPI đó là giảm đi các công đoạn phức tạp trong quá trình tiền xử lý. Cụ thể, nhóm cũng tận dụng DPI vào quá trình tiền xử lý, một đoạn code C++ có nhiệm vụ chuyển file ảnh thành file text được thực hiện bởi thư viện OpenCV được nhóm chuẩn bị, các file text này là đầu vào của DUT và golden model. Theo phương pháp truyền thống, các file text này phải được chuẩn bị trước khi thực hiện mô phỏng bằng việc chạy code Python hoặc Matlab, kết quả của quá trình này chính là đầu vào của quá trình mô phỏng kiểm tra. Với việc áp dụng DPI, bước tiền xử lý này có thể gộp vào quá trình mô phỏng bằng việc dùng SystemVerilog gọi hàm tiền xử lý được viết bằng C++ có sử dụng thư viện OpenCV

để thực hiện việc chuyển hình sang text trước khi bước vào giai đoạn mô phỏng kiểm tra, khi bắt đầu mô phỏng đoạn code C đóng vai trò tiền xử lý sẽ được biên dịch và thực hiện trước, sau đó quá trình mô phỏng mới bắt đầu. Toàn bộ quá trình trên được thực hiện hoàn toàn bằng trình biên dịch và trình mô phỏng của công cụ EDA tool (Hình 2-12) có áp dụng thư viện svdpi.h, điều này giúp toàn bộ quá trình chuẩn bị dữ liệu và mô phỏng trở nên liền mạch và nhất quán.



Hình 2-12: Trình tự hoạt động của DPI-C và SystemVerilog trong mô phỏng

2.3. Kiểm tra hành vi thiết kế sử dụng SystemVerilog Assertion

Với các thiết kế ngày càng phức tạp hơn, nỗ lực để kiểm tra thiết kế cũng trở nên thử thách hơn. Để tăng hiệu suất và chất lượng kiểm tra, người thiết kế môi trường kiểm tra cần phải áp dụng nhiều phương pháp và kỹ thuật kiểm tra khác nhau, và Assertion là một trong những tính năng thiết yếu được hỗ trợ bởi SystemVerilog nhằm hỗ trợ người kiểm tra thiết kế có được một môi trường kiểm tra hoàn thiện nhất có thể.

2.3.1. Tổng quan về SystemVerilog Assertion

Assertion là một đoạn code thường được dùng để kiểm tra hành vi của một thiết kế, đoạn Assertion Code có luận lý phụ thuộc vào hành vi mà người kiểm tra muốn xác nhận ở thiết kế. Nếu hành vi của thiết kế thỏa với đoạn Code mô tả hành vi Assertion thì thông báo (có hoặc không) cho người kiểm tra kiểm tra thành công, ngược lại nếu hành vi của thiết kế không thỏa với đoạn Assertion Code, công cụ EDA tools sẽ báo lỗi cùng với thời gian cụ thể nơi hành vi bị vi phạm cũng như vi phạm nào bị xảy ra.

2.3.2. Các loại Assertion

Có 2 loại Assertion chính là Immediate Assertion và Concurrent Assertion.

2.3.2.1. Immediate Assertion

Immediate Assertion là một mệnh đề tuần tự được dùng chủ yếu trong mô phỏng. Một Assertion là một mệnh đề xác minh luận lý của một phần tử là đúng hay sai, có cách hoạt động tương tự như mệnh đề if else. Điểm khác biệt duy nhất giữa mệnh đề của Immediate Assertion và mệnh đề if else đó là mệnh đề if else chỉ kiểm tra và trả về kết quả true hoặc false, đối với mệnh đề Immediate Assertion khi kiểm tra trả về kết quả false, một thông báo error sẽ được hiển thị thông báo mệnh đề thất bại.

```
module SVA_example;
    logic clk = 0;
    logic a;
    logic b;

    always #5 clk = ~clk;

    always @ (posedge clk)
        assert(a == b);

    always @ (negedge clk)
        assert(a != b);

    initial begin
        repeat(10) begin
            #10 a = $random;
            b = $random;
        end
        #10 $finish;
    end
endmodule
```

Xét kết quả của đoạn code trên (Hình 2-13):

```

Error: Assertion violation
Time: 5 ns Iteration: 0 Process: /SVA_example/Alwaysll_1 File: /
Error: Assertion violation
Time: 15 ns Iteration: 0 Process: /SVA_example/Alwaysll_1 File: /
Error: Assertion violation
Time: 20 ns Iteration: 0 Process: /SVA_example/Alwaysl4_2 File: /
Error: Assertion violation
Time: 30 ns Iteration: 0 Process: /SVA_example/Alwaysl4_2 File: /
Error: Assertion violation
Time: 45 ns Iteration: 0 Process: /SVA_example/Alwaysll_1 File: /
Error: Assertion violation
Time: 50 ns Iteration: 0 Process: /SVA_example/Alwaysl4_2 File: /
Error: Assertion violation
Time: 65 ns Iteration: 0 Process: /SVA_example/Alwaysll_1 File: /
Error: Assertion violation
Time: 75 ns Iteration: 0 Process: /SVA_example/Alwaysll_1 File: /
Error: Assertion violation
Time: 85 ns Iteration: 0 Process: /SVA_example/Alwaysll_1 File: /
Error: Assertion violation
Time: 95 ns Iteration: 0 Process: /SVA_example/Alwaysll_1 File: /
Error: Assertion violation
Time: 100 ns Iteration: 0 Process: /SVA_example/Alwaysl4_2 File: /

```

Hình 2-13: Kết quả Assertion violation

Đoạn Code trên là một ví dụ về Immediate Assertion. Trong đoạn Code có 2 mệnh đề được đưa ra, mệnh đề thứ nhất kiểm tra ở mỗi cạnh lên tín hiệu xung clk tín hiệu a có giống với tín hiệu b hay không, mệnh đề thứ hai kiểm tra ở mỗi cạnh xuống tín hiệu xung clk tín hiệu a có khác với tín hiệu b hay không. Nếu bất kỳ mệnh đề nào thất bại, trình mô phỏng sẽ xuất thông báo “Assertion violation” tại thời gian xảy ra vi phạm Assertion ra màn hình kết quả (Hình 2-13).

2.3.2.2. Concurrent Assertion

Concurrent Assertion là Assertion được dùng chủ yếu trong việc xác định hành vi của thiết kế, đồng thời cũng là Assertion được dùng phổ biến nhất trong quá trình kiểm tra thiết kế. Concurrent Assertion có cơ chế chính là xác định một chuỗi hành vi (được gọi là một sequence) có xảy ra trong quá trình kiểm tra thiết kế hay không. Người thiết kế môi trường kiểm tra có nhiệm vụ thiết lập chuỗi sequence này và tùy theo mong muốn của người kiểm tra mà xác nhận chuỗi hành vi này cần phải xảy ra hoặc không được phép xảy ra trong quá trình mô phỏng. Ví dụ:

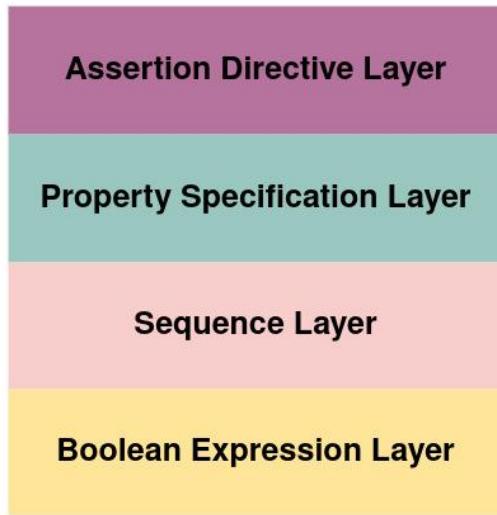
```
assert property (@(posedge clk) req |-> ##[1:2] ack)
```

Đoạn assertion bên trên là một Concurrent Assertion, mệnh đề Assertion có nghĩa tại mỗi cạnh lén xung clk, nếu tín hiệu req đang ở mức cao thì kiểm tra trong vòng một hoặc hai cạnh lén xung clock tiếp theo, tín hiệu ack phải ở mức cao. Nếu tín hiệu ack đều ở mức thấp trong 2 xung clk kế tiếp, mệnh đề được xem như thất bại và một lỗi Assertion violation sẽ được thông báo bởi trình mô phỏng.

Concurrent Assertion là một tính năng mạnh mẽ của SystemVerilog trong việc hỗ trợ người kiểm tra thiết kế xác minh hành vi của DUT. Trong phạm vi của khóa luận, nhóm quyết định lựa chọn vận dụng Concurrent Assertion vào quá trình xây dựng mô hình kiểm tra để kiểm tra hành vi của các tín hiệu bên trong môi trường kiểm tra, giúp hỗ trợ tăng tính chính xác và chất lượng đầu ra sau khi tiến hành mô phỏng.

2.3.3. Các lớp của Concurrent Assertion

Concurrent Assertion có thể được chia thành bốn lớp trừu tượng như sau:



Hình 2-14: Các lớp bên trong Concurrent Assertion

- **Boolean Expression Layer:** là lớp căn bản và thấp nhất của Concurrent Assertion có vai trò đánh giá một mệnh boolean là đúng hoặc sai. Các toán tử cơ bản được sử dụng trong lớp này bao gồm **&&**, **||**, **==**, **!=**
- **Sequence Layer:** là lớp kế tiếp cao hơn Boolean Expression Layer. Nguyên tắc của các lớp bên trong Concurrent Assertion là lớp cao hơn sẽ bao gồm

các lớp thấp hơn, do đó Boolean Expression Layer có thể được sử dụng bên trong lớp Sequence Layer. Sequence Layer có vai trò chính là cho phép người dùng định nghĩa các chuỗi hành vi hoặc sự kiện cụ thể có phụ thuộc và thời gian. Ví dụ đoạn Code sau định nghĩa một chuỗi hành vi “nếu tín hiệu req ở mức cao thì sau đó 2 xung clock tín hiệu ack phải ở mức cao”, nếu mệnh đề trên là đúng thì kết quả trả về của sequence là True, ngược lại nếu mệnh đề không thỏa kết quả trả về của sequence là False. Ví dụ về một chuỗi sequence có cú pháp như sau:

```
sequence s1;
    req ##2 ack;
endsequence
```

Đồng thời ở lớp Sequence Layer, các toán tử delay như **##** và toán tử lặp (**[*]** hoặc **[=]**) và một số toán tử thông dụng khác như **throughout**, **within**, **intersect**, **and**, **or** cũng được sử dụng để định nghĩa các chuỗi.

- **Property Specification Layer:** được xem như lớp chứa tập hợp các chuỗi khác nhau bên trong nó được định nghĩa với từ khóa **property** và **endproperty**, các chuỗi sequence được định nghĩa bên trong 2 từ khóa này. Ở lớp này các toán tử ngầm định được sử dụng để xác định tiền đề và kết quả giữa các chuỗi bên trong, các toán tử này bao gồm toán tử overlapping **| ->** và non-overlapping **| =>**
- **Assertion Directive Layer:** là lớp cao nhất và chứa tất cả các lớp bên dưới. Ở đây người thiết kế môi trường kiểm tra sẽ lựa chọn sử dụng property nào để kiểm tra hành vi của DUT.

2.3.4. Các toán tử của Concurrent Assertion

Toán tử phỏ biến được dùng trong Concurrent Assertion là toán tử ngầm định được dùng để xác định tiền đề và kết quả của một chuỗi sequence, toán tử ngầm định của Concurrent Assertion có 2 loại bao gồm:

- **Overlapping:** là toán tử ngầm định được dùng để xác định tiền đề và kết quả có cùng thỏa trung nhau ngay tại cạnh lên / cạnh xuống xung clock hay không
- **Non-overlapping:** là toán tử ngầm định được dùng để xác định tiền đề và kết quả có cùng thỏa tại hai cạnh lên / cạnh xuống xung clock hay không, thời điểm kiểm tra hai chuỗi không được phép trùng nhau

Sự khác nhau của hai toán tử ngầm định Overlapping và Non-overlapping được thể hiện qua ví dụ sau:

```
module sva_test;

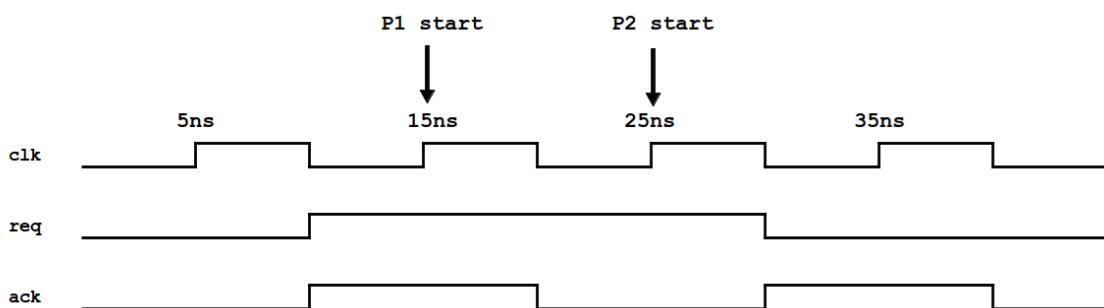
logic req;
logic ack;
logic clk = 0;

always #5 clk = ~clk;

a1: assert property (@(posedge clk) req |-> ack);
a2: assert property (@(posedge clk) req |=> ack);

endmodule
```

Ở ví dụ trên, tín hiệu req được xem như mệnh đề tiền đề và ack được xem như mệnh đề kết quả. Quan sát dạng sóng ứng với đoạn code trên ở Hình 2-15.



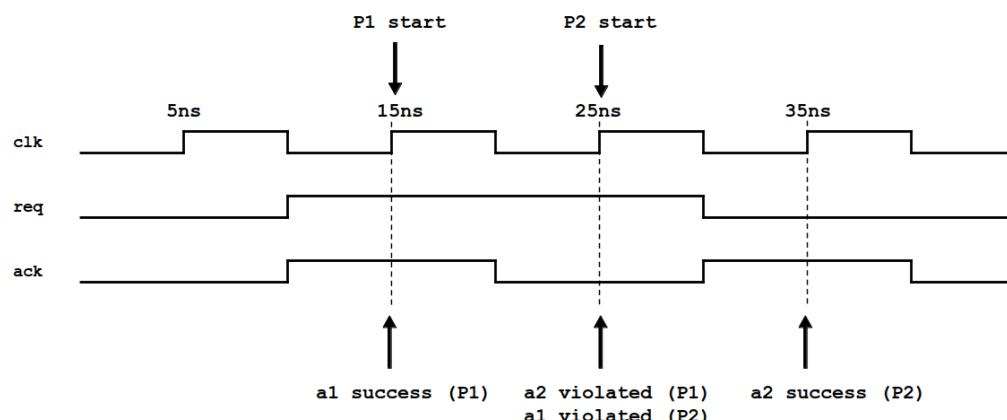
Hình 2-15: Waveform mẫu cho đoạn code bên trên

Cả hai mệnh đề Assertion đều có thời gian tại cạnh lên tín hiệu clk, do đó các thời điểm Assertion được kiểm tra là 5ns, 15ns, 25ns và 35ns. Do cả hai mệnh đề Assertion có chung tiền đề là tại mỗi cạnh lên tín hiệu clk, kiểm tra tín hiệu req có ở mức cao hay không, vì vậy ở 15ns và 25ns kết quả sẽ được kiểm tra do tại hai thời

điểm trên req ở mức cao, còn lại ở 5ns và 35ns req đều ở mức thấp. Hai tiến trình Assertion được khởi tạo ở 15ns và 25ns ta gọi hai tiến trình này là P1 và P2 (Hình 2-15).

Mệnh đề Assertion a1 sử dụng toán tử ngầm định Overlapping. Cụ thể, tại cạnh lên xung clock 15ns và 25ns, tiền đề được kiểm tra trước và nếu tiền đề thỏa ngay lập tức kết quả sẽ được kiểm tra, cả hai đều được kiểm tra ngay tại thời điểm cạnh lên xung clock. Đối với mệnh đề Assertion a2 sử dụng toán tử ngầm định Non-overlapping, việc kiểm tra không được trùng lặp tại một thời điểm, cụ thể tiền đề được kiểm tra tại xung clock chỉ định và nếu tiền đề thỏa thì kết quả sẽ được kiểm tra ở xung clock kế tiếp tại 25ns.

Với tiến trình đầu tiên tại 15ns, tín hiệu req ở mức cao và đồng thời tín hiệu ack cũng ở mức cao, do đó a1 thỏa và Assertion được xem như thành công. Tuy nhiên ở cạnh lên xung clock kế tiếp tín hiệu ack ở mức thấp, do đó a2 không thỏa và Assertion được xem như thất bại. Tương tự đối với tiến trình thứ hai tại 25ns, Assertion a1 thất bại do tín hiệu req ở mức cao trong khi tín hiệu ack lại ở mức thấp, Assertion a2 thành công do ở xung clock kế tiếp tại 35ns tín hiệu ack ở mức cao (Hình 2-16).



Hình 2-16: Kết quả của 2 tiến trình Assertion P1 và P2

Thời gian thông báo Assertion thành công hay thất bại của toán tử ngầm định Overlapping là ngay tại thời xung clock được chỉ định, toán tử Non-overlapping là tại xung clock kế tiếp nơi chuỗi kết quả được kiểm tra.

Bên cạnh toán tử ngầm định có chức năng xác định một chuỗi hành vi của DUT, một toán tử khác được dùng phổ biến trong Concurrent Assertion đó là toán tử định thời (Delay Operator), công dụng của toán tử định thời là dùng để định thời một khoảng thời gian trước khi một sự kiện xảy ra, toán tử định thời sử dụng ký hiệu `##`. Ví dụ ta có một mệnh đề tại mỗi cạnh lén xung clock, nếu tín hiệu req đang ở mức cao, sau đó 4 chu kỳ xung clock kiểm tra tín hiệu ack có ở mức cao hay không, mệnh đề Assertion này có dạng như sau:

```
assert property (@(posedge clk) req |-> ##4 ack);
```

Ta cũng có thể xác định một khoảng thời gian tối thiểu và tối đa cho toán tử định thời. Ví dụ: Tại mỗi cạnh lén xung clock, nếu tín hiệu req đang ở mức cao, kiểm tra trong vòng 3 tới 5 chu kỳ xung clock kế tiếp tín hiệu ack có ở mức cao hay không, mệnh đề Assertion này có dạng như sau:

```
assert property (@(posedge clk) req |-> ##[3:5] ack);
```

Cuối cùng, để tạo một chuỗi sequence hoàn chỉnh, cụ thể để xác định một tín hiệu ở trạng thái mức cao hoặc mức thấp trong một khoảng số lượng xung clock cụ thể, toán tử lặp (Repetition Operator) được sử dụng trong Concurrent Assertion. Có 3 dạng toán tử lặp được sử dụng bằng 3 ký hiệu `[*n]`, `[=n]`, `[->n]`. Các dạng toán tử này được sử dụng ở các trường hợp khác nhau. Cụ thể là:

- Toán tử `[*n]` là toán tử lặp liên tiếp dùng để xác định một tín hiệu ở mức cao / mức thấp liên tiếp trong n chu kỳ xung clock.
- Toán tử `[=n]` và `[->n]` là toán tử lặp không liên tiếp được dùng để xác định một tín hiệu ở mức cao mức / mức thấp trong n chu kỳ xung clock. Đôi với toán tử này, việc tín hiệu phải ở trạng thái cao / thấp liên tiếp không bị bắt buộc.

Ví dụ ta có 3 mệnh đề Assertion như sau:

```

module sva;

    reg clk = 0;
    reg a;
    reg b;
    reg c;

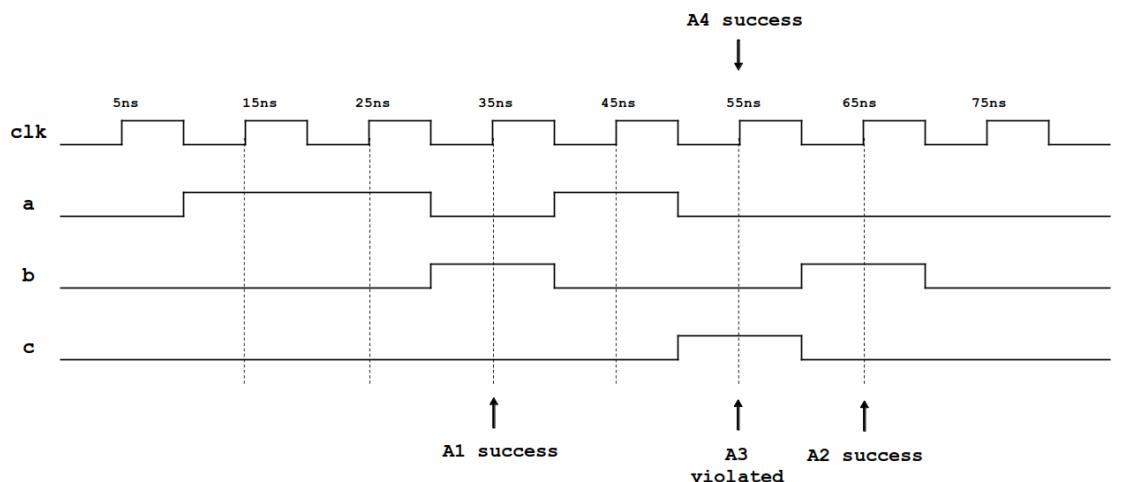
    always #5 clk = ~clk;

    A1: assert property (@(posedge clk) a[*2] ##1 b);
    A2: assert property (@(posedge clk) a[=3] ##1 b);
    A3: assert property (@(posedge clk) a[->3] ##1 b);
    A4: assert property (@(posedge clk) a[->3] ##1 c);

endmodule

```

Xét dạng sóng ứng với đoạn code trên như sau (Hình 2-17):



Hình 2-17: Dạng sóng cho ví dụ toán tử lặp của Concurrent Assertion

Mệnh đề A1 có nghĩa sau khi tín hiệu a ở mức cao trong 2 cạnh lên xung clock liên tiếp, kiểm tra tín hiệu b có ở mức cao trong xung clock kế tiếp hay không. Vì a ở mức cao tại 15ns và 25ns, tín hiệu b có trạng thái mức cao ở xung cạnh lên xung clock kế tiếp tại 35ns nên mệnh đề thỏa, Assertion thành công.

Mệnh đề A2 có nghĩa sau khi tín hiệu a ở mức cao trong 3 cạnh lên xung clock (không cần phải liên tiếp như A1), kiểm tra sau 1 cạnh lên xung clock tín hiệu b có ở mức cao trong bất kỳ thời gian nào hay không. Do tín hiệu a có trạng thái mức

cao tại 15ns, 25ns và 45ns, sau đó tín hiệu b có trạng thái mức cao tại 65ns, mệnh đề Assertion A2 thành công.

Mệnh đề A3 có cấu trúc ngữ nghĩa của tiền đề tương tự A2, tuy nhiên ở chuỗi kết quả, toán tử `[->]` bắt buộc chuỗi phải thỏa ngay sau khi tiền đề thành công. Nghĩa là sau khi tiền đề thỏa tại 45ns, ngay sau đó 1 chu kỳ, tín hiệu b phải ở mức cao. Do tại 55ns tín hiệu b ở mức thấp nên mệnh đề Assertion A3 coi như vi phạm.

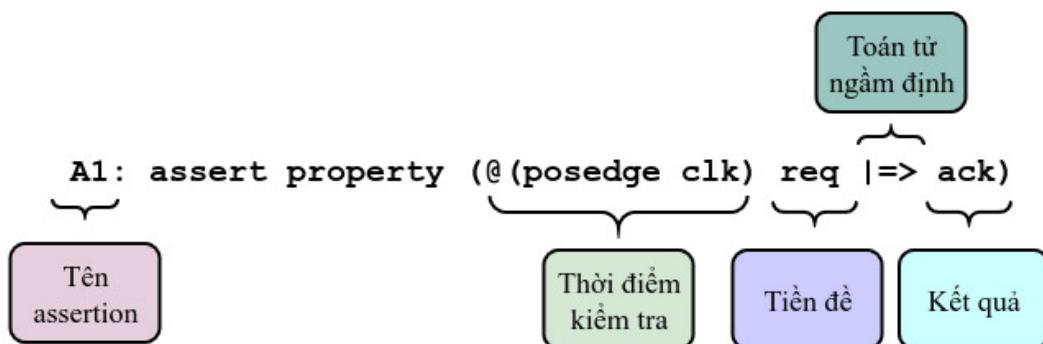
Tương tự ngữ nghĩa mệnh đề A3, mệnh đề A4 kiểm tra tại 55ns tín hiệu c có ở mức cao hay không. Do tín hiệu c mức cao tại cạnh lên xung clock kế tiếp sau khi thỏa tiền đề, mệnh đề A4 được xem như thành công.

Mệnh đề A2 và A3 cũng thể hiện sự khác nhau giữa hai toán tử lặp không liên tiếp sử dụng `[=]` hoặc `[->]`.

Ta cũng có thể sử dụng cú pháp `[*m:n]`, `[=m:n]`, `[->m:n]` để xác định một khoảng thời gian thay vì xác định một thời gian cụ thể cho toán tử lặp tương tự như toán tử định thời.

2.3.5. Cú pháp Concurrent Assertion

Một mệnh đề Concurrent Assertion có cú pháp như sau:



Hình 2-18: Cú pháp của một mệnh đề Concurrent Assertion

Mô tả cú pháp của một mệnh đề Concurrent Assertion như sau:

- **Tên Assertion** là tên của mệnh đề Concurrent Assertion, mỗi Assertion phải có một tên riêng

- **Thời điểm kiểm tra** là thời điểm mệnh đề được kiểm tra, thông thường cạnh lén hoặc cạnh xuống xung clock sẽ được chọn làm thời điểm kiểm tra
- **Tiền đề** là một chuỗi hành vi và là điều kiện kiểm tra đầu tiên của mệnh đề, chuỗi hành vi được kiểm tra đầu tiên, nếu tiền đề thỏa tại thời điểm kiểm tra kết quả sẽ được kiểm tra, nếu tiền đề không thỏa tại thời điểm kiểm tra, kết quả sẽ không được kiểm tra và mệnh đề sẽ được bỏ qua
- **Toán tử ngầm định** là một trong hai toán tử Overlapping $| ->$ hoặc Non-overlapping $| =>$
- **Kết quả** là chuỗi hành vi thứ hai, chuỗi hành vi này được kiểm tra nếu chuỗi hành vi ở tiền đề thỏa, nếu chuỗi hành vi của kết quả thỏa mệnh đề được xem như thành công, ngược lại nếu chuỗi hành vi của kết quả không thỏa mệnh đề được xem như vi phạm

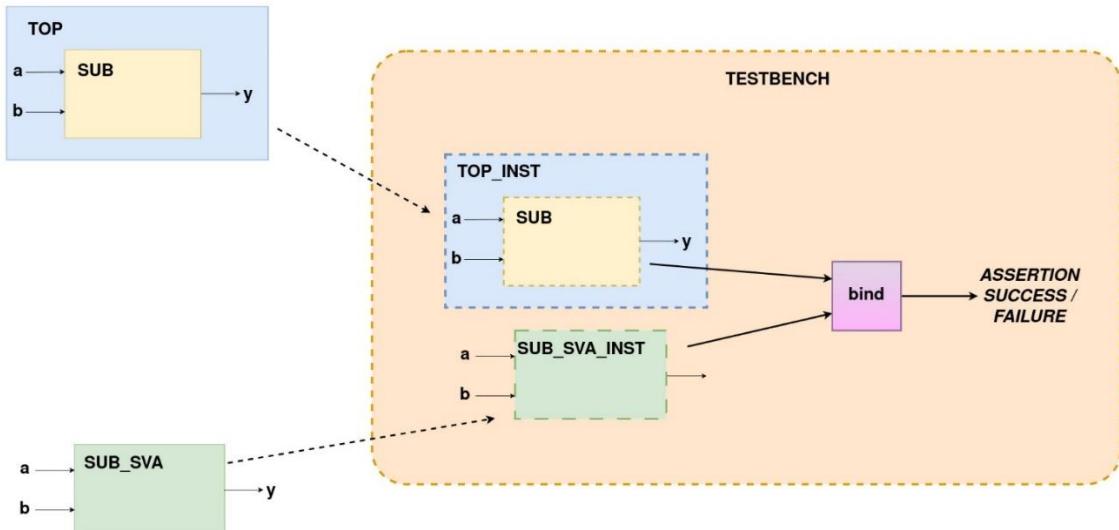
2.3.6. Cấu trúc bind của SystemVerilog

Thông thường trong một thiết kế, các Assertion sẽ phải nằm trong RTL code nhằm xác định hành vi của thiết kế. Tuy nhiên người thiết kế môi trường kiểm tra không được chỉnh sửa RTL code nhằm giữ độ chính xác và nhất quán cho DUT trong quá trình kiểm tra. Để hỗ trợ người kiểm tra đưa các Assertion vào Testbench mà không phải chỉnh sửa RTL code, SystemVerilog cung cấp cấu trúc bind được hiện thực qua từ khóa “bind”.

Cấu trúc bind hỗ trợ người thiết kế môi trường kiểm tra tạo nên một module rỗng có các input và output tương tự như DUT. Sử dụng bind để liên kết module này với DUT, người kiểm tra có khả năng trích xuất tín hiệu của DUT trong quá trình mô phỏng thông qua input và output của module này, đồng thời, các Assertion cũng được thêm vào module này để kiểm tra hành vi của DUT. Khi thực hiện việc bind hai module với nhau, một module là DUT và module còn lại đóng vai trò nắm giữ Assertion của DUT, trong quá trình mô phỏng, các tín hiệu ra vào DUT đồng thời cũng sẽ ra vào module rỗng đó.

Ở Hình 2-19, module TOP có chứa một module con có tên là SUB. Để xác định hành vi của module SUB, ta tạo một module rỗng tên SUB_SVA, SUB_SVA

có input và output giống hệt SUB và bên trong SUB_SVA, ta đưa các mệnh đề Assertion cần kiểm tra vào. Sau đó ở Testbench, thực hiện việc bind module SUB và SUB_SVA, khi thực hiện mô phỏng, các Assertion nằm trong SUB_SVA đóng vai trò kiểm tra hành vi của module SUB sẽ được thực thi.

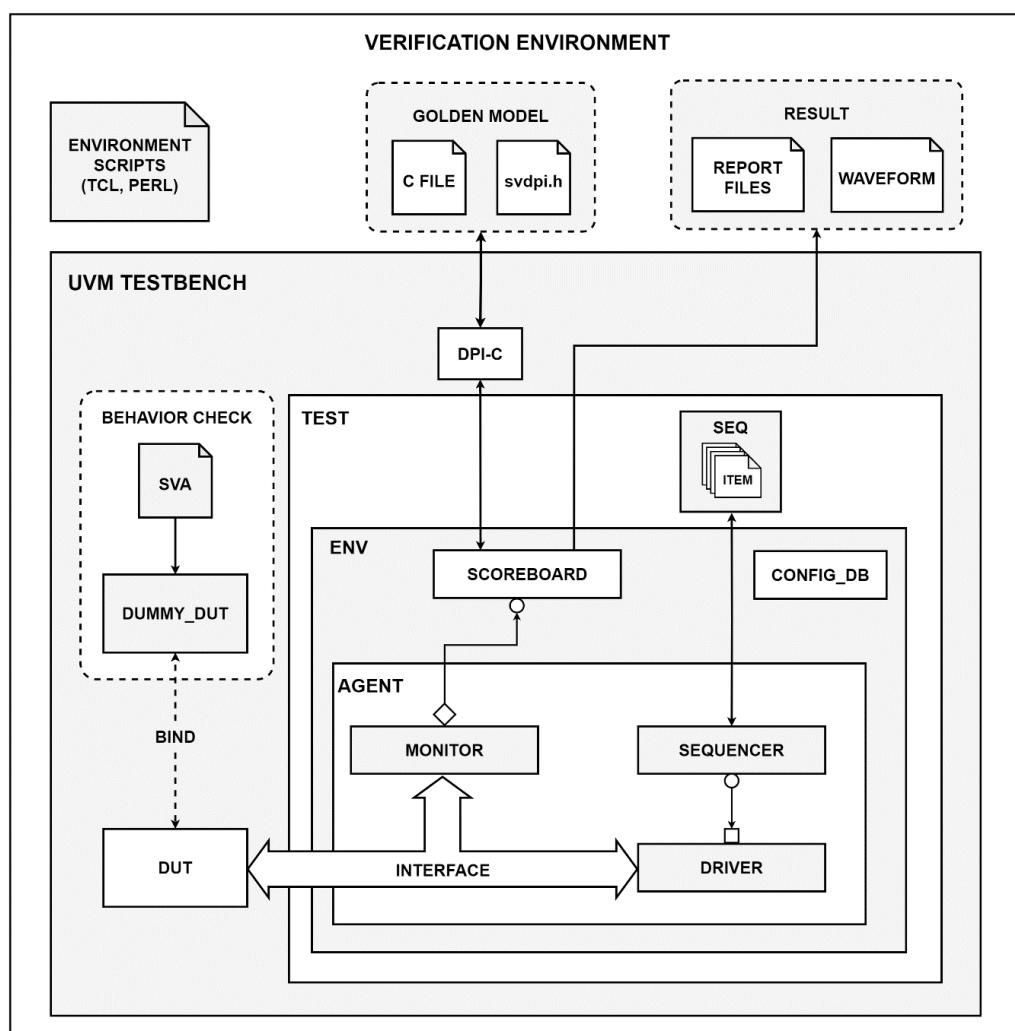


Hình 2-19: Cấu trúc bind của SystemVerilog

Chương 3. MÔ HÌNH THỰC TẾ MÔI TRƯỜNG KIỂM TRA THIẾT KẾ

Ở Chương 3, nhóm trình bày về quá trình hiện thực mô hình môi trường kiểm tra cho thiết kế CNN IP được nhóm xây dựng và hiện thực, Chương 3 bao gồm các giai đoạn trong việc nghiên cứu tìm hiểu về mô tả của thiết kế cần kiểm tra, từ đó xây dựng và thiết kế môi trường kiểm tra thực tế cho thiết kế này.

3.1. Mô hình tổng quan môi trường kiểm tra thiết kế



Hình 3-1: Tổng quan mô hình môi trường kiểm tra thiết kế

Hình 3-1 là kiến trúc tổng quan của mô hình kiểm tra mà nhóm hiện thực. Mục tiêu của nhóm ở phần đầu quá trình thực hiện đề tài khóa luận này chính là xây

dựng một môi trường kiểm tra có khả năng tự động hóa và tái sử dụng cho một CNN IP. Sau khi áp dụng những kiến thức nền tảng và cơ sở lý thuyết ở Chương 2, nhóm thực hiện xây dựng môi trường kiểm tra thiết kế có cấu trúc gốc như Hình 2-1. Cụ thể Các phần tử chính của môi trường bao gồm DUT, UVM Testbench, mô hình tin cậy, SystemVerilog Assertion và các Script tự động hóa được nhóm tích hợp vào môi trường kiểm tra.

Phương pháp xây dựng môi trường kiểm tra chính của nhóm sử dụng đó là ứng dụng UVM để mô hình hóa các phần tử bên trong môi trường, kết hợp với một số tùy chỉnh và các tính năng khác nhau được cung cấp bởi SystemVerilog và EDA Tool (Xilinx Vivado 2022.2) nhằm đạt được các mục tiêu đã đề ra.

UVM Testbench có nhiệm vụ hỗ trợ quản lý các thành phần bên trong Testbench thông qua các lớp bên trong nó. Sau khi nghiên cứu về cơ sở lý thuyết, nhóm quyết định xây dựng môi trường UVM dựa theo khung thiết kế chung được chuẩn hóa bởi Accellera [2]. Việc xây dựng các thành phần chính của UVM Tesbench được thực hiện nhờ các lớp nền và những phương thức được cung cấp bởi thư viện tích hợp UVM. Việc điều khiển và hiệu chỉnh UVM Testbench được thực hiện thông qua cơ chế Phasing của UVM.

Để xác định tính đúng đắn của thiết kế cần kiểm tra, mô hình tin cậy dùng để đánh giá và kiểm tra kết quả thực tế có được trong quá trình mô phỏng chức năng mạch được nhóm tích hợp bên trong môi trường kiểm tra, nhóm xây dựng mô hình này dựa trên chức năng của thiết kế gốc và được hiện thực hóa bằng tính năng Direct Programming Interface được cung cấp bởi SystemVerilog.

Đồng thời trong quá trình thực hiện mô phỏng, để tiến hành kiểm tra hành vi của thiết kế cũng như tính đúng đắn của các tín hiệu điều khiển bên trong DUT, nhóm cũng sử dụng cơ chế SystemVerilog Assertion để tạo ra các chuỗi hành vi cần kiểm tra cho DUT, các chuỗi này được lưu trong một module rỗng và được liên kết với thiết kế gốc thông qua cơ chế “bind”.

Cuối cùng, để tối ưu tự động hóa cho môi trường kiểm tra, nhóm cũng viết các script TCL, Perl để hiệu chỉnh môi trường cũng như các input cần thiết trong từng

quá trình cụ thể trong môi trường mô phỏng UVM. Script TCL được dùng để biên dịch và tổng hợp mô hình tin cậy viết bằng ngôn ngữ C để có được thư viện .so sử dụng trong quá trình kiểm tra kết quả sử dụng DPI-C. Script Perl được dùng để xử lý chuỗi, lọc kết quả ý nghĩa của UVM report file phục vụ cho việc theo dõi quá trình mô phỏng đồng thời hỗ trợ cho việc kiểm tra, đánh giá cũng như tìm lỗi của kết quả mô phỏng.

Chi tiết về cấu trúc và chức năng cụ thể của từng thành phần bên trong môi trường kiểm tra thiết kế (UVM Testbench, mô hình tin cậy, Assertion) mà nhóm xây dựng được trình bày ở các phần sau.

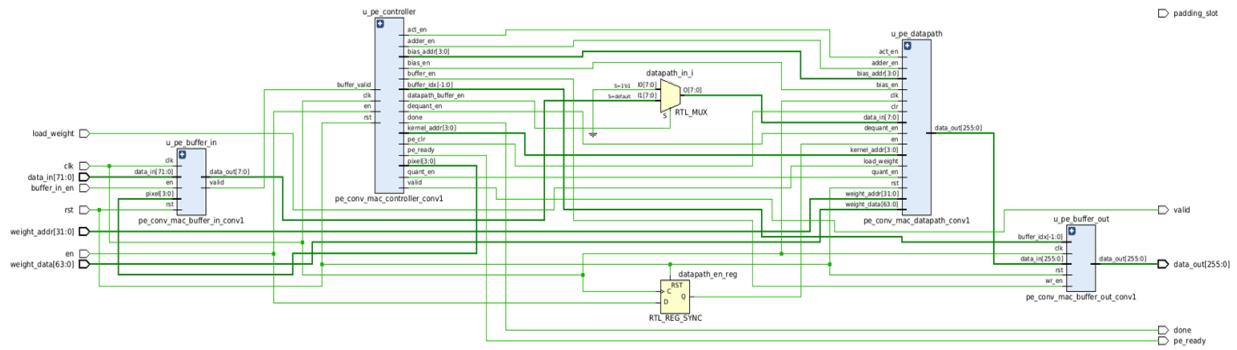
3.2. Thiết kế CNN IP được dùng để xây dựng môi trường kiểm tra

Để thực hiện kiểm tra thiết kế tích chập, nhóm quyết định chọn phương pháp kiểm tra white-box, cụ thể nhóm được cung cấp thiết kế hoàn chỉnh và mô tả thiết kế từ nhóm thiết kế, từ đây nhóm cần hiểu rõ cấu trúc và cách thức hoạt động của các phần tử bên trong thiết kế tích chập, số chu kỳ để hoàn thành và các đầu vào đầu ra của thiết kế, cuối cùng để hiện thực môi trường thiết kế kiểm tra, nhóm cần lên ý tưởng, xây dựng những thành phần và chức năng cần thiết cho môi trường dựa trên thiết kế tích chập, sau khi hoàn thành xây dựng môi trường, nhóm tiến hành mô phỏng, trích xuất và xử lý kết quả để thu lại các thông kê kết quả kiểm tra thiết kế.

3.2.1. Tổng quan thiết kế CNN IP

Đặc trưng của CNN đó chính là tuy có nhiều mô hình khác nhau, khói tính toán chính của mọi CNN IP là khói tích chập, khói tích chập có nhiệm vụ thực hiện phép tính toán tích chập với tập dữ liệu đầu vào với trọng số và một số tính toán khác như dequantize, activation và quantize. Ở giai đoạn đầu của khóa luận, nhóm quyết định thực hiện việc xây dựng mô hình kiểm tra cho một khói tích chập của CNN IP mô hình LeNet5 có sẵn đã được thiết kế và hiện thực bởi một nhóm nghiên cứu khác [13].

Mô hình thiết kế có các phần tử chính đó là khói controller, khói datapath và hai buffer bao gồm buffer_in và buffer_out. Kiến trúc của khói tính toán tích chập của CNN IP có dạng như sau:



Hình 3-2: Kiến trúc thiết kế CNN

Thiết kế trên (Hình 3-2) có vai trò thực hiện phép tính tích chập trong một mô hình CNN: khối controller có chức năng thực hiện điều khiển hoạt động của mạch, khối datapath thực hiện các phép tính toán của quá trình tích chập, một bộ nhớ buffer_in và một bộ nhớ buffer_out để lưu các dữ liệu đầu vào và kết quả đầu ra.

Để thực hiện phép tính tích chập, thiết kế cần có dữ liệu đầu vào và các giá trị trọng số. Giá trị đầu vào được đảm nhiệm bởi bus tín hiệu **data_in** và được lưu vào bộ nhớ đệm **buffer_in**, trọng số được lưu bằng bus **weight_data** vào bộ nhớ riêng nằm trong datapath. Trước khi giai đoạn tính toán tích chập được thực thi, bộ nhớ trọng số cần được lưu giá trị trước, quá trình lưu trọng số được điều khiển bằng tín hiệu **load_weight**. Kết quả của quá trình tính toán tích chập được lưu bên trong bộ nhớ đệm **buffer_out**.

3.2.2. Mô tả input và output của thiết kế

Các tín hiệu đầu vào và đầu ra của thiết kế cũng như mô tả chức năng của chúng được trình bày qua Bảng 3-1.

Bảng 3-1: Mô tả input và output của thiết kế CNN

Signal	I/O	Description
clk	IN	Tín hiệu xung clock
rst	IN	Tín hiệu reset cho toàn bộ thiết kế
load_weight	IN	Tín hiệu điều khiển dùng cho giai đoạn load trọng số
weight_addr	IN	Địa chỉ nơi trọng số được lưu

weight_data	IN	Giá trị của trọng số
data_in	IN	Dữ liệu đầu vào dùng cho tích chập
buffer_in_en	IN	Tín hiệu enable dùng cho buffer_in
en	IN	Tín hiệu cho phép thiết kế hoạt động
valid	OUT	Tín hiệu thông báo dữ liệu đầu ra hợp lệ
data_out	OUT	Dữ liệu đầu ra của phép tích chập
done	OUT	Tín hiệu thông báo một chu trình tích chập cho một ảnh 28x28 đã hoàn thành
pe_ready	OUT	Tín hiệu thông báo thiết kế sẵn sàng hoạt động

3.3. Đặc trưng cấu trúc mô trường UVM Testbench cho thiết kế tích chập

Ở phần này, nhóm mô tả về các thành phần cụ thể của mô trường UVM được nhóm xây dựng cho mô trường kiểm tra thiết kế tích chập. Các điểm chính của từng thành phần được nhóm liệt kê thông qua các thuộc tính và phương thức của từng thành phần, đặc biệt cách hoạt động của chúng được thể hiện qua các Phase chính của từng thành phần trong UVM Testbench được nhóm xây dựng.

3.3.1. Sequence Item

Sequence Item của mô trường UVM có các tín hiệu tương tự như input và output được liệt kê ở Bảng 3-1. Tất cả tín hiệu thuộc lớp Sequence Item đều có kiểu dữ liệu logic, tuy nhiên tín hiệu weight_data và data_in sẽ có từ khóa “rand” được dùng để khởi tạo giá trị ngẫu nhiên cho chúng. Đồng thời một biến có kiểu dữ liệu enum được định nghĩa bên trong lớp sequence item, biến enum này có vai trò là cờ thông báo cho các thành phần bên trong mô trường kiểm tra hiện tại đang ở trạng thái hoạt động nào.

```
typedef enum bit[3:0] {RESET,
                      MEM_KERNEL_LOAD,
                      MEM_BIAS_LOAD,
                      MEM_SCALE_LOAD,
                      RUNNING} oper_mode;
```

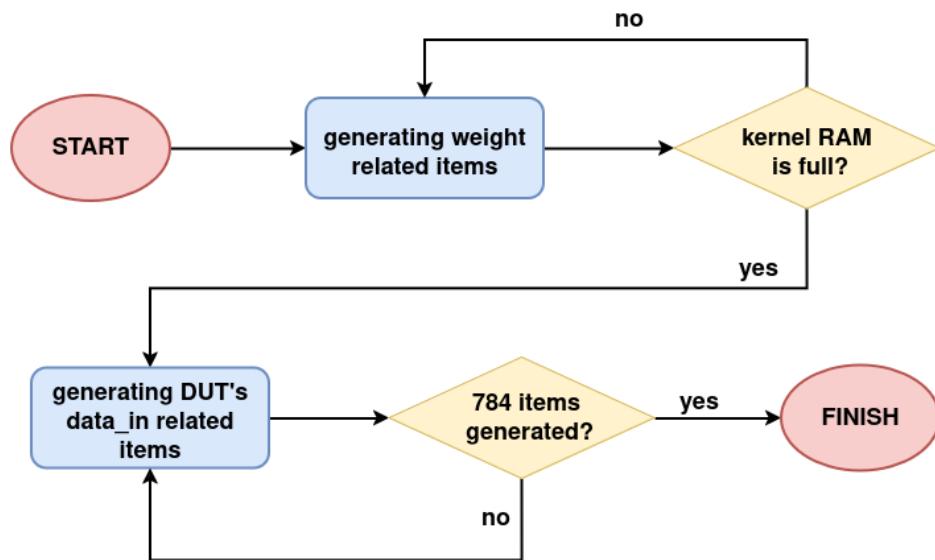
Hình 3-3: Biến enum bên trong Sequence Item

Các trạng thái của biến enum trong Hình 3-3 có ý nghĩa như sau:

- **RESET:** DUT đang thực hiện việc Reset toàn bộ hệ thống
- **MEM_KERNEL_LOAD:** DUT thực hiện load trọng số vào bộ nhớ
- **MEM_BIAS_LOAD:** DUT thực hiện load giá trị bias vào bộ nhớ
- **MEM_SCALE_LOAD:** DUT thực hiện load giá trị scale thực hiện cho quá trình dequantize
- **RUNNING:** Quá trình tính toán tính chập đang được thực hiện

3.3.2. Sequence

Sequence đầu tiên được nhóm hiện thực đó là tạo ra các dữ liệu ngẫu nhiên vào DUT để kiểm tra hoạt động của thiết kế. Cụ thể giá trị của weight_data và data_in sẽ được khởi tạo ngẫu nhiên bằng hàm randomize() kết hợp random constraint để kiểm soát khoảng giá trị được khởi tạo ngẫu nhiên.



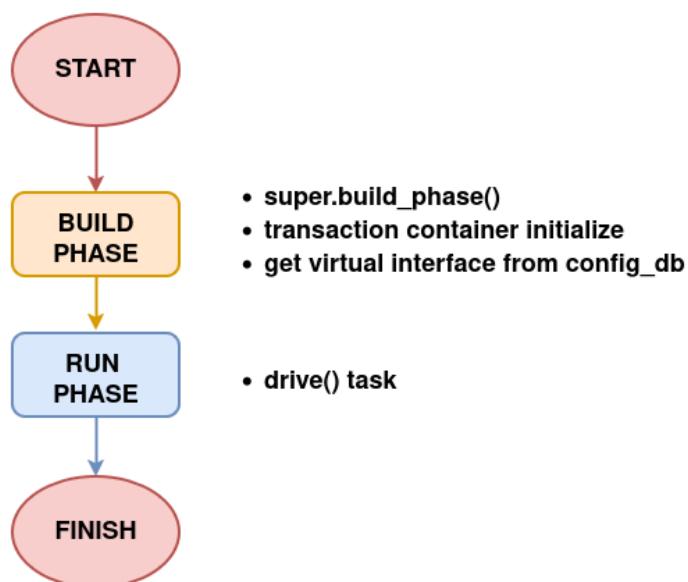
Hình 3-4: Flow chart của function body() bên trong Sequence

Đặc điểm của lớp Sequence đó là lớp được khởi tạo từ uvm_object và là một thành phần động (Dynamic Component), do đó Sequence không có các Phase như các lớp được khởi tạo từ uvm_component. Việc tạo ra các gói tin được thực hiện

bên trong task body() của lớp. Để thực hiện việc kiểm tra cho thiết kế tích chập, Sequence cần tạo ra các gói tin cho việc tải trọng số và các gói tin đóng vai trò như data_in cho DUT. Các gói tin trọng số sẽ được khởi tạo trước cho tới khi bộ nhớ lưu trọng số bên trong DUT đầy, lúc này các gói tin data_in được khởi tạo để DUT thực hiện tính toán tích chập. Hình 3-4 là quá trình Sequence tạo ra các gói tin để gửi đến Sequencer. Tổng số gói tin data_in đưa vào DUT là 784 tương ứng với số lượng pixel trong một ảnh 28x28, là kích cỡ ảnh thiết kế tích chập sử dụng cho đầu vào, được định nghĩa ở mô tả thiết kế.

3.3.3. Driver

Các Phase của lớp Driver nhóm thiết kế có cấu trúc và nhiệm vụ như Hình 3-5. Cụ thể, Driver thực hiện việc truyền các kích thích tới DUT thông qua Interface ở Run Phase. Cụ thể, bên trong Run Phase có một task drive() thực hiện công việc truyền kích thích đến DUT.



Hình 3-5: Các Phase chính bên trong lớp Driver

Ở Hình 3-6, bên trong task drive(), có một forever block thực hiện việc chờ gói tin từ Sequencer và gửi gói tin tới Interface. Đối với thiết kế tích chập, ở giai đoạn

bắt đầu mô phỏng, Driver sẽ thực hiện truyền tín hiệu reset tới DUT để đưa mạch trở về trạng thái mặc định.

```
// Run Phase
virtual task run_phase(uvm_phase phase);
    drive();
endtask
```

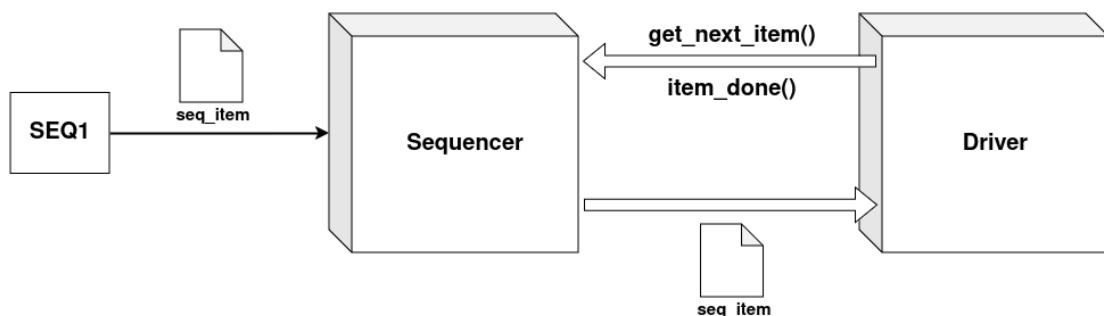
Hình 3-6: Run Phase của lớp Driver

Sau khi hoàn thành giai đoạn reset là giai đoạn tải trọng số vào thiết kế. Giai đoạn này được quyết định bằng tín hiệu load_weight được khởi tạo bởi Sequence. Cụ thể sau khi nhận được gói tin từ Sequencer, Driver kiểm tra tín hiệu load_weight của gói tin, nếu tín hiệu load_weight ở có giá trị 1, gói tin gồm hai thành phần quan trọng nhất là weight_addr và weight_data được truyền tới DUT. Ngược lại nếu load_weight của gói tin có giá trị 0, nghĩa là giai đoạn tích chập đang được thực hiện, các tín hiệu quan trọng cho quá trình này bên trong gói tin là buffer_in_en, data_in được truyền tới DUT, sau đó Driver chờ 10 chu kỳ xung clock trước khi gửi thông báo gói tin hoàn thành và chờ gói tin mới từ Sequence.

```
task drive();
    reset_DUT();
    forever begin
        seq_item_port.get_next_item(tr);
        if(tr.load_weight) begin
            // send weight related items
        end
        else if(!tr.load_weight) begin
            // send data related items
            repeat(10) @ (posedge vif.clk);
        end
        seq_item_port.item_done(tr);
    end
endtask
```

Hình 3-7: Task drive() của lớp Driver

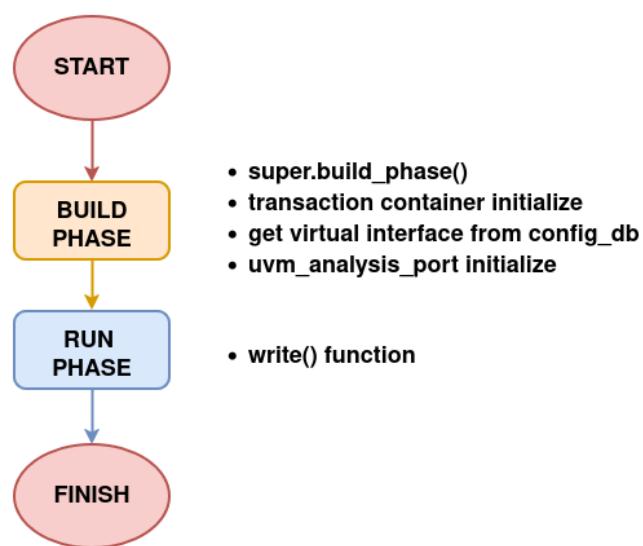
Ở Hình 3-7, việc gửi và nhận gói tin giữa Driver và Sequencer được thực hiện thông qua seq_item_port và hàm get_next_item() / item_done(), cụ thể TLM được thiết lập giữa Driver và Sequencer ở Connect Phase, seq_item_port nằm trong Driver được liên kết với seq_item_export của Sequence. Hàm get_next_item() và item_done() là các hàm tích hợp được cung cấp bởi UVM cho phép thực thi trên seq_item_export, các hàm này đóng vai trò gửi tín hiệu giao tiếp và truyền nhận gói tin giữa Driver và Sequencer. Mô hình giao tiếp và gửi nhận gói tin của Sequencer và Driver được thể hiện qua Hình 3-8.



Hình 3-8: Giao tiếp giữa lớp Driver và Sequencer

3.3.4. Monitor

Các Phase của lớp Monitor nhóm thiết kế có cấu trúc và nhiệm vụ như sau:



Hình 3-9: Các Phase chính bên trong lớp Monitor

Bên trong Monitor, một uvm_analysis_port tên “send” được khởi tạo để thiết lập liên kết TLM với Scoreboard (Hình 3-10), uvm_analysis_port này đóng vai trò dùng để truyền gói tin bắt được từ Interface bởi Monitor đến Scoreboard để thực hiện việc đánh giá và kiểm tra kết quả thực tế.

```

uvm_analysis_port #(transaction) send;

// Build Phase
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    send = new("send", this);
endfunction

```

Hình 3-10: uvm_analysis_port được khai báo bên trong Monitor

Monitor thực hiện vai trò của mình song song với Driver ở Run Phase. Bên trong forever block của Monitor, Monitor đóng gói thông tin lấy được từ Interface ở mỗi cạnh lén xung clock. Monitor kiểm tra gói tin nếu tin hiệu load_weight có giá trị 1 thì gói tin với weight_addr và weight_data được gửi đến Scoreboard để sao chép bộ nhớ của DUT.

```

// Run phase
virtual task run_phase(uvm_phase phase);
    forever begin
        @ (posedge vif.clk);
        if(vif.load_weight) begin
            // capture weights related item
        end
        else if(vif.op == RUNNING) begin
            @ (posedge vif.pe_ready)
            // capture DUT result
        end
        send.write(tr); // Send to Scoreboard
    end
endtask: run_phase

```

Hình 3-11: Run Phase của lớp Monitor

Hình 3-11 mô tả quá trình hoạt động của Monitor, nếu DUT đang ở trạng thái tính toán tích chập, Monitor chờ đến khi output pe_ready của DUT tích cực thì bắt gói tin từ Interface có chứa kết quả hợp lệ từ DUT để gửi đến Scoreboard. Việc gửi gói tin đến Scoreboard được thực hiện thông qua Analysis Port và hàm write().

3.3.5. Scoreboard

Scoreboard có nhiệm vụ nhận gói tin kết quả DUT từ Monitor thông qua giao thức TLM. Cụ thể Monitor gửi gói tin thông qua uvm_analysis_port và Scoreboard nhận gói tin bằng uvm_analysis_imp, do đó trong lớp Scoreboard nhóm định nghĩa một uvm_analysis_imp có tên “recv”.

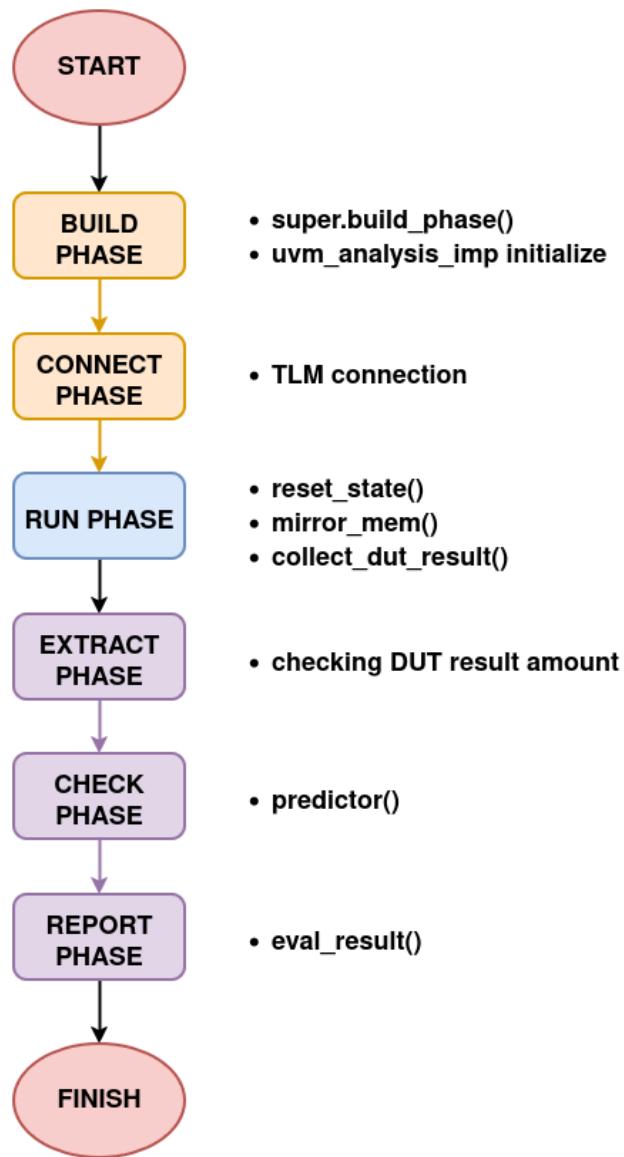
```
uvm_analysis_imp #(transaction, scoreboard) recv;  
  
// Build Phase  
virtual function void build_phase(uvm_phase phase);  
    super.build_phase(phase);  
    recv = new("recv", this);  
endfunction
```

Hình 3-12: uvm_analysis_imp được khai báo bên trong Scoreboard

Ở Hình 3-12, tương tự uvm_analysis_port của Monitor, uvm_analysis_imp của Scoreboard cũng cần phải được khai báo và khởi tạo tại Build Phase, sau đó giao thức TLM sẽ được thực hiện ở Connect Phase bởi UVM.

Sau khi giao thức TLM được khởi tạo, các gói tin được Monitor gửi đến Scoreboard thông qua hàm write(), hàm write() được định nghĩa bên trong Scoreboard. Hàm write() thực hiện 3 phần tương tự thứ tự hoạt động của khối tính toán tích chập bao gồm trạng thái reset, trạng thái tải trọng số vào bộ nhớ và trạng thái tính toán tích chập. Khi nhận được gói tin có trạng thái reset, Scoreboard thực hiện việc chuẩn bị các biến và môi trường mô phỏng cho trước khi chuyển sang trạng thái tải trọng số và tính toán tích chập. Khi nhận được gói tin có trạng thái tải trọng số, Scoreboard thực hiện việc sao chép bộ nhớ của DUT. Khi nhận được gói

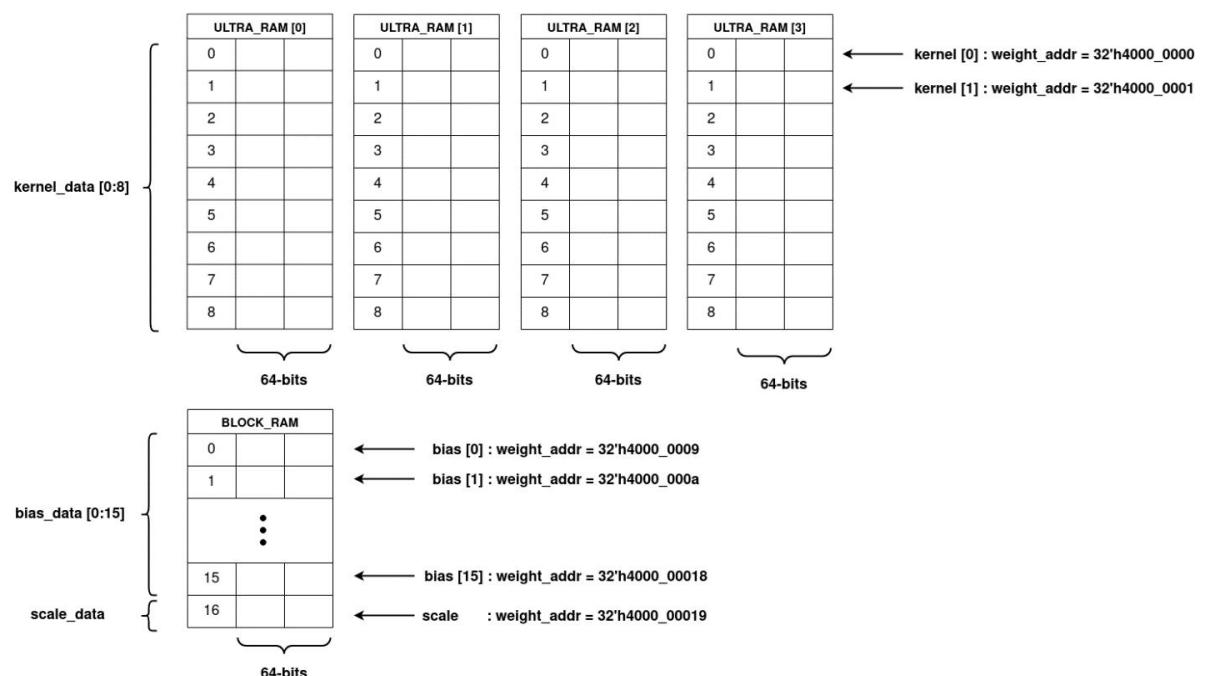
tin có trạng thái kết quả của phép tính tích chập, Scoreboard thực hiện việc thu thập, kiểm tra và đánh giá kết quả có được từ Monitor thông qua các UVM Phase bao gồm Extract Phase, Check Phase và Report Phase. Quá trình hoạt động của Scoreboard trong môi trường kiểm tra được thể hiện qua Hình 3-13.



Hình 3-13: Các Phase chính của lớp Scoreboard

Ở Hình 3-13, hàm `predictor()` bên trong Check Phase là hàm thực hiện việc kiểm tra kết quả và so sánh kết quả thực tế với mô hình tin cậy. `eval_result()` là hàm thực hiện việc thống kê số lượng kết quả chính xác so với số lượng mẫu thử.

Ở Run Phase, sau trạng thái reset là trạng thái tải trọng số vào bộ nhớ, ở trạng thái này, Scoreboard thực hiện việc sao chép bộ nhớ của DUT vào một bộ nhớ ảo bên trong Scoreboard, việc này để chuẩn bị cho quá trình kiểm tra kết quả ở hàm predictor(), kết quả thực tế sẽ được so sánh với kết quả tin cậy. Để có thể sao chép bộ nhớ, người thiết kế môi trường kiểm tra cần phải hiểu được cấu trúc bộ nhớ của DUT.



Hình 3-14: Cấu trúc bộ nhớ kernel_ram của CNN IP

Hình 3-14 là cấu trúc bộ nhớ kernel_ram bên trong datapath của khối tích chập. Bộ nhớ gồm 4 Ultra RAM dùng để lưu trọng số kernel và 17 Block RAM được dùng để lưu trọng số bias và scale. Mỗi ô nhớ của Ultra Ram và Block Ram có độ rộng dữ liệu là 64 bit. Địa chỉ ô nhớ đầu tiên là 0x40000000 và ô nhớ cuối cùng là 0x40000019.

Để sao chép cấu trúc bộ nhớ kernel_ram. Nhóm sử dụng kiểu dữ liệu Associative Array với Key của mảng là địa chỉ và Value là một hàng đợi có độ lớn tối đa là 4 (tương ứng với 4 Ultra RAM), hàng đợi này có phần tử là kiểu dữ liệu

logic có độ lớn 64-bits (tương ứng với độ lớn 64-bits của mỗi ô nhớ). Đặc điểm của Associative Array đó là các Key của mảng là riêng biệt và không trùng nhau, tương tự như địa chỉ mỗi ô nhớ là riêng biệt. Để hiện thực bộ nhớ ảo, nhóm thực hiện typedef để tạo một kiểu dữ liệu riêng là hàng đợi có độ lớn tối đa là 4, hàng đợi này cũng sau đó được khai báo là Value của Associative Array (Hình 3-15).

```

typedef logic [63:0] ultra_ram_queue [$:4];
class scoreboard extends uvm_monitor;
    // class's attributes
    ultra_ram_queue virtual_mem [int];

    // class's methods
    function mirror_mem();
        // ...
    endfunction
endclass

```

Hình 3-15: Bộ nhớ ảo bên trong Scoreboard

Sau khi hoàn thành trạng thái tải trọng số, bộ nhớ ảo của Scoreboard có cấu trúc như sau:

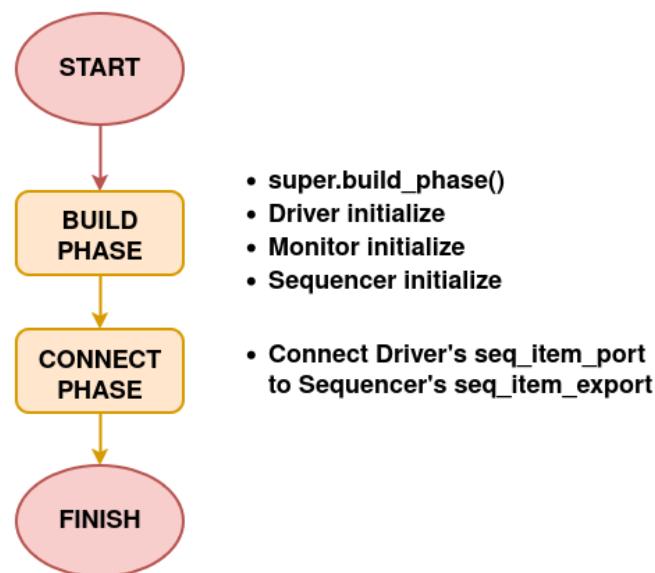
SCOREBOARD'S VIRTUAL MEMORY					
address: 40000000 , { pos : 0 data = e4c698a5_1d8b8e52 }	{ pos : 1 data = 1c4c07d4_ca5b9f68 }	{ pos : 2 data = 0b56e605_23fbc15a }	{ pos : 3 data = 31e73307_2a6bf40c }		
address: 40000001 , { pos : 0 data = 0ffcef29_9dacfb6b }	{ pos : 1 data = 25981a92_3dbcc896 }	{ pos : 2 data = f2bb0d48_8a9cf243 }	{ pos : 3 data = b75ebef4_c44de9ea }		
address: 40000002 , { pos : 0 data = 338a36d_eacd0f0f }	{ pos : 1 data = e1351e14_be1e804 }	{ pos : 2 data = 9271a846_3eef3221 }	{ pos : 3 data = 352d3a10_eb4cd935 }		
address: 40000003 , { pos : 0 data = cf66e6d2_840bb7ef }	{ pos : 1 data = e1351e14_0b81a44 }	{ pos : 2 data = e122546_e117172 }	{ pos : 3 data = 352d3a10_f137f90 }		
address: 40000004 , { pos : 0 data = f001b15f_ec14889c }	{ pos : 1 data = d385f59b_e5e5ae94 }	{ pos : 2 data = fae7c6b6_cc66df8f }	{ pos : 3 data = d9cf04a9_9ff6217f }		
address: 40000005 , { pos : 0 data = f001b15f_ec30766c }	{ pos : 1 data = f3ddcd65_c100d820 }	{ pos : 2 data = 83af5996_a72c1c65 }	{ pos : 3 data = 8052532c_3ddd4d80 }		
address: 40000006 , { pos : 0 data = f524bc78_215f6934 }	{ pos : 1 data = a12c94bb_c1b110c4 }	{ pos : 2 data = c4b9dc3b_ced2c95e }	{ pos : 3 data = 9fcc92b1_c8c492ed }		
address: 40000007 , { pos : 0 data = b264ab7db_ef8066f6 }	{ pos : 1 data = 3c824c85_c3185879 }	{ pos : 2 data = fe255922_c37a54cb }	{ pos : 3 data = b74d4c2ba_b0ac61f5 }		
address: 40000008 , { pos : 0 data = e7fb4a49_caae7fffa }	{ pos : 1 data = 102ef510_91808r3b }	{ pos : 2 data = ere7b4e3_c522ef42 }	{ pos : 3 data = c725e3ce_e5954066 }		
address: 40000009 , { pos : 0 data = 000181dc_0002a284 }					
address: 4000000a , { pos : 0 data = 00018ebb_fffff144 }					
address: 4000000b , { pos : 0 data = fffff0a83_00029bdd }					
address: 4000000c , { pos : 0 data = 0002f5df_00022e08 }					
address: 4000000d , { pos : 0 data = 00014fd8_0000d393 }					
address: 4000000e , { pos : 0 data = fffe18e8_00018a1b }					
address: 4000000f , { pos : 0 data = 0002511b_000051be }					
address: 40000010 , { pos : 0 data = 0001f850_00022a28 }					
address: 40000011 , { pos : 0 data = ffffe0e91_fffff1395 }					
address: 40000012 , { pos : 0 data = 00029400_00020e17 }					
address: 40000013 , { pos : 0 data = 00028874_fffff1987 }					
address: 40000014 , { pos : 0 data = 0000bebf_e000135e2 }					
address: 40000015 , { pos : 0 data = 0000bebf_e00026341 }					
address: 40000016 , { pos : 0 data = fffff00c_0000a156 }					
address: 40000017 , { pos : 0 data = 0001b6bf_0000f07c }					
address: 40000018 , { pos : 0 data = 0000bd07_0000562d }					
address: 40000019 , { pos : 0 data = 00005f10_0002c187 }					

Hình 3-16: Cấu trúc bộ nhớ ảo được sao chép từ kernel_ram

3.3.6. Agent

Lớp Agent có vai trò là lớp chứa Driver, Monitor và Sequencer. Đồng thời giao thức TLM giữa Driver và Sequencer cũng được khởi tạo ở Agent. Cụ thể, trong

Connect Phase, seq_item_port của Driver được kết nối với seq_item_export của Sequencer thông qua hàm connect() của UVM. Các Phases của lớp Monitor nhóm thiết kế có cấu trúc và nhiệm vụ như sau:



Hình 3-17: Các Phase chính của lớp Agent

```

// Build Phase
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    d    = driver::type_id::create("d", this);
    m    = monitor::type_id::create("m", this);
    seqr =
uvm_sequencer#(transaction)::type_id::create("seqr",
this);
endfunction

// Connect Phase
virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    d.seq_item_port.connect(seqr.seq_item_export);
endfunction

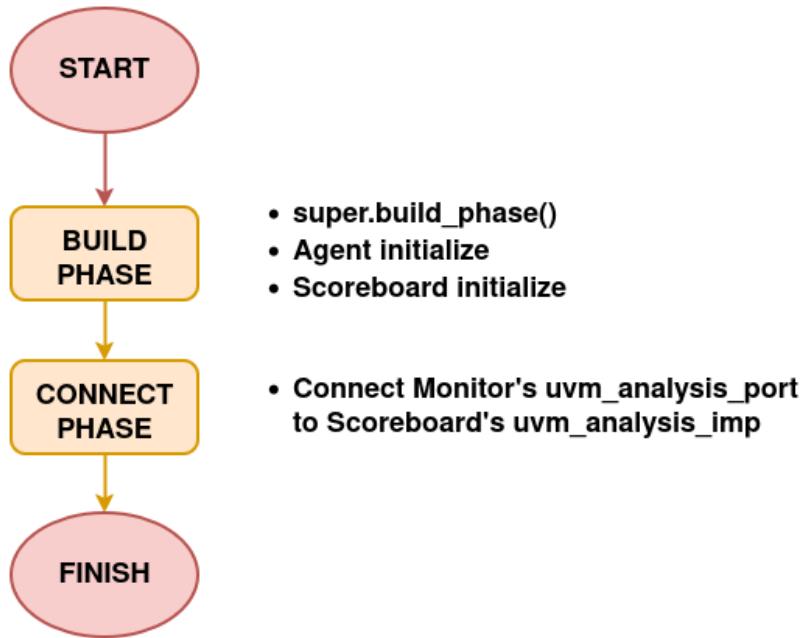
```

Hình 3-18: Build Phase và Connect Phase của lớp Agent

Hình 3-18 là cú pháp của Build Phase và Connect Phase trong lớp Agent. Bên trong Build Phase của Agent, đối tượng thuộc các lớp bao gồm Driver, Monitor và Sequencer được khởi tạo. Kế tiếp ở Connect Phase, giao thức TLM giữa Driver và Sequencer được khởi tạo sử dụng hàm connect(), đây là một hàm được định nghĩa sẵn của lớp Sequencer bên trong thư viện tích hợp UVM, cụ thể sau khi Sequencer đã được khai báo và khởi tạo ở Build Phase, ta có thể sử dụng hàm connect() này để tiến hành việc liên kết giữa Driver và Sequencer bằng cú pháp được định nghĩa bởi tài liệu “UVM 1.2 Class Reference” [1].

3.3.7. Env

Các Phases của lớp Monitor nhóm thiết kế có cấu trúc và nhiệm vụ như sau:



Hình 3-19: Các Phase chính của lớp Env

Lớp Env là lớp đóng gói các lớp Agent và Scoreboard. Đồng thời ở Connect Phase của Env, giao thức TLM giữa Monitor và Scoreboard được thực hiện, `uvm_analysis_port` bên trong Monitor nằm trong Agent được kết nối với `uvm_analysis_imp` của Scoreboard. Hình 3-20 là cú pháp của Build Phase và Connect Phase trong lớp Env. Các đối tượng của Agent và Scoreboard được khởi tạo bên trong Build Phase. Trong Connect Phase, giao thức TLM giữa Agent và Scoreboard được khởi tạo sử dụng hàm `connect()`.

```

// Build Phase
virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
    s = scoreboard::type_id::create("s", this);
    a = agent::type_id::create("a", this);
endfunction

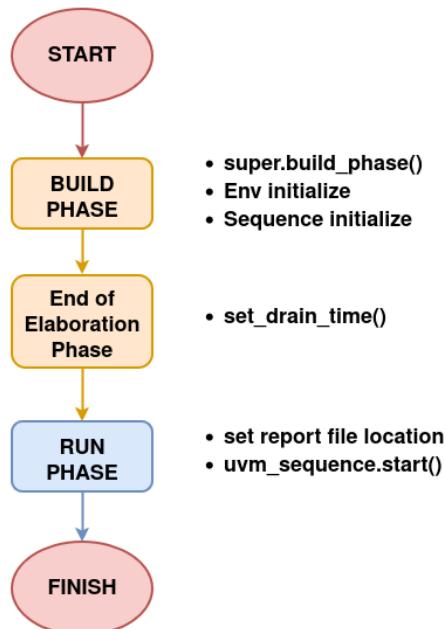
// Connect Phase
virtual function void connect_phase(uvm_phase phase);
    a.m.send.connect(s.recv);
endfunction

```

Hình 3-20: Build Phase và Connect Phase của lớp Env

3.3.8. Test

Lớp Test là lớp cao nhất của môi trường UVM. Test đóng vai trò là lớp chứa Env và Sequence, các hiệu chỉnh cho môi trường UVM cũng được nhóm thực hiện ở lớp Test. Các hiệu chỉnh này bao gồm định nghĩa Report File cho quá trình mô phỏng (định nghĩa ở Run Phase) và kéo dài thời gian mô phỏng để gói tin cuối có thể hoàn thành bên trong DUT và được Monitor bắt được (định nghĩa ở End of Elaboration Phase). Các Phases của lớp Test nhóm thiết kế có cấu trúc và nhiệm vụ như Hình 3-21.



Hình 3-21: Các Phase chính của lớp Test

Ở Build Phase, các đối tượng của lớp Env và Scoreboard được khởi tạo. Ở End of Elaboration Phase, phương thức set_drain_time() được sử dụng để kéo dài Run Phase thêm một khoảng thời gian xác định, đây là một phương thức được tích hợp bên trong thư viện UVM, người thiết kế môi trường kiểm tra có thể gọi và sử dụng theo cú pháp được định nghĩa bởi UVM. Ở Run Phase, đường dẫn dùng để lưu kết quả mô phỏng được nhóm xác định ở đầu Run Phase, kế tiếp start() được gọi để lớp Test bắt đầu cho phép Sequence khởi tạo các gói tin.

Bên trong Run Phase của lớp Test, ngoài việc khởi tạo và thực hiện phương thức khởi động cho Sequence, một số hiệu chỉnh cho môi trường Test cũng được nhóm thực hiện ở đây. Hình 3-22 là các hiệu chỉnh để lưu tất cả các thông báo của `uvm_info vào một Text File.

```
log_file = $fopen(report_path);
uvm_top.set_report_default_file_hier(log_file);
uvm_top.set_report_severity_action_hier
```

Hình 3-22: Các hiệu chỉnh cho môi trường test

```
main_phase = phase.find_by_name("main", 0);
main_phase.phase_done.set_drain_time(this, 500);
```

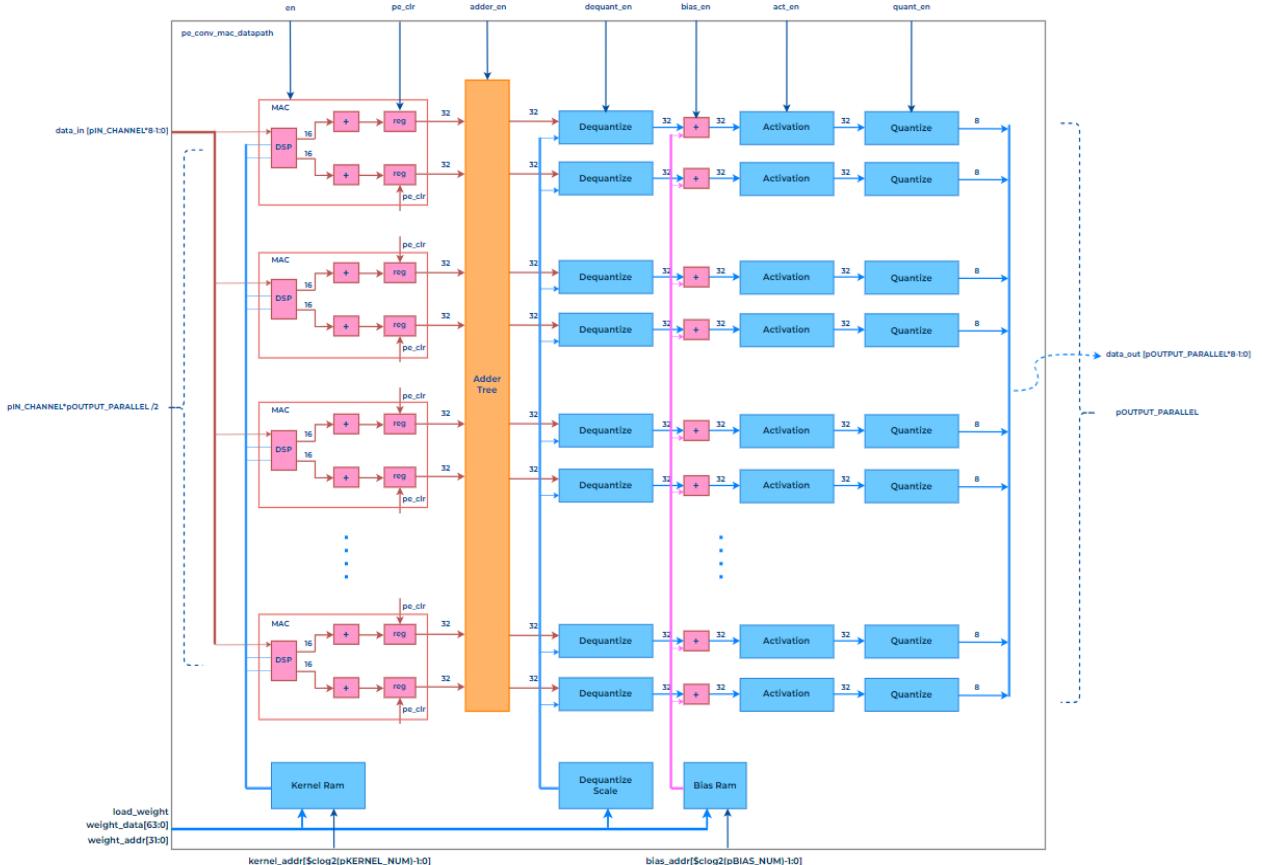
Hình 3-23: Hiệu chỉnh về set_drain_time

Đồng thời ở End of Elaboration Phase, các hiệu chỉnh về kéo dài thời gian mô phỏng (Hình 3-23) được nhóm thêm vào thông qua phương thức set_drain_time được cung cấp bởi thư viện tích hợp UVM.

3.4. Mô hình tin cậy cho thiết kế tích chập sử dụng DPI-C

Để xây dựng mô hình tin cậy phục vụ cho việc kiểm tra tính đúng đắn của thiết kế tích chập, nhóm quyết định sử dụng ngôn ngữ C với sự hỗ trợ của thư viện svdpi.h để hiện thực mô hình tin cậy này. Điểm mạnh của mô hình được viết bằng C đó là có thể thực thi mô hình này trên các môi trường khác nhau.

Phương pháp kiểm tra của nhóm thực hiện để kiểm tra thiết kế tích chập đó là phương pháp white-box, do đó nhóm cần hiểu rõ cách thức hoạt động cụ thể của từng khối thực hiện việc tính toán tích chập bên trong DUT, cũng chính là module Datapath của thiết kế.



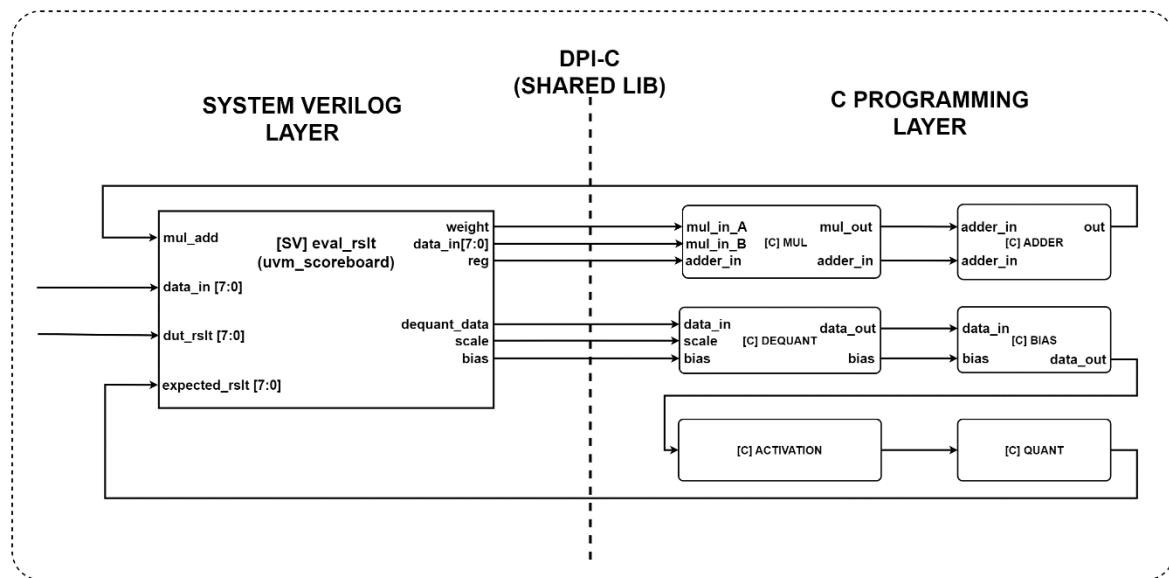
Hình 3-24: Kiến trúc Datapath module của khối tích chập

Hình 3-24 là module datapath của khối tích chập, các phép tính toán được thực hiện ở module này, do đó mô hình tin cậy cho DUT được nhóm xây dựng dựa trên hoạt động của nó. Các phần tử chính của module bao gồm các khối DSP thực hiện nhân hai số nguyên, Adder Tree, Dequantize, Activation và Quantize.

Tương tự, một mô hình tin cậy được viết bằng ngôn ngữ lập trình C được xây dựng chứa các hàm thực hiện những chức năng trên. Trong quá trình mô phỏng, SystemVerilog Testbench sẽ thực hiện việc gọi các hàm C này thông qua DPI-C. Như đã đề cập ở Chương 2, dữ liệu đầu vào của DUT và mô hình tin cậy có giá trị

giống nhau, đầu ra bao gồm kết quả thực tế và kết quả mong đợi sẽ được so sánh sau đó.

Mô hình giữa lớp ngôn ngữ lập trình C và SystemVerilog Testbench được thể hiện qua Hình 3-25, các hàm ở lớp ngôn ngữ lập trình C sẽ được xem như mô hình tin cậy để thực hiện các công việc tính toán chính bên trong module datapath phía trên. Kết quả mô phỏng của SystemVerilog Testbench sẽ được lưu trong một hàng đợi được khởi tạo ở uvm_scoreboard, sau khi hoàn thành việc đưa tất cả sequence item vào DUT, môi trường UVM chuyển qua Check Phase, tại đây việc kiểm tra kết quả sẽ được thực thi, các kết quả được lưu bên trong hàng đợi sẽ được so sánh với kết quả của mô hình tin cậy để đánh giá tính chính xác của Testbench. Các hàm với các chức năng tương tự datapath của khối tích chập được viết trên ngôn ngữ C với sự hỗ trợ của thư viện svdpi.h bao gồm: hàm nhân và cộng, hàm bias, hàm tính sigmoid và hàm thực hiện lượng tử hóa.



Hình 3-25: Các tầng của mô hình DPI

3.5. Kiểm tra hành vi Controller module của thiết kế tích chập sử dụng SystemVerilog Assertion

Để điều khiển module datapath, khối tích chập sử dụng module controller điều khiển các tín hiệu adder_en, bias_en, act_en, dequant_en, quant_en, buffer_en. Các

tín hiệu này là đầu ra của controller và cũng là đầu vào của datapath. Đồng thời controller cũng có các tín hiệu thông báo như valid và done. Tùy thuộc vào từng giai đoạn tính toán bên trong datapath mà controller sẽ tích cực các tín hiệu điều khiển tương ứng. Để kiểm tra hành vi của controller có tích cực các tín hiệu điều khiển đúng với mong muốn của thiết kế ban đầu hay không, nhóm sử dụng SystemVerilog Assertion và cấu trúc “bind” để kiểm tra hành vi của controller. Cụ thể, một module rỗng có input và output giống hệt controller, nắm vai trò giữ các Concurrent Assertion của module controller được định nghĩa và được khởi tạo bên trong UVM Testbench. Áp dụng cấu trúc “bind”, việc kiểm tra hành vi của module controller bằng Concurrent Assertion được hiện thực bên trong Testbench mà không cần chỉnh sửa RTL code gốc.

Bảng 3-2: Mô tả các phép tính và số chu kỳ để hoàn thành của module Datapath

Thứ tự	Phép tính	Số chu kỳ cần để hoàn thành
1	DSP	5 chu kỳ
2	Adder	1 chu kỳ
3	Dequantize	9 chu kỳ
4	Bias	1 chu kỳ
5	Act	4 chu kỳ
6	Quantize	1 chu kỳ
7	buffer_in	1 chu kỳ
8	buffer_out	1 chu kỳ

Bảng 3-2 là mô tả thời gian thực hiện của khối datapath trong thiết kế tích hợp. Ứng với mỗi phép tính, controller sẽ gửi tín hiệu “enable” trong một chu kỳ cho phép khối phép tính tương ứng được thực hiện. Để xác định tính đúng đắn controller, nhóm định nghĩa các mệnh đề Concurrent Assertion cho các tín hiệu enable của controller với các chuỗi sequence tương ứng với số chu kỳ hoạt động của các khối phép tính ở bảng này.

Chuỗi tiền đề của các Concurrent Assertion được nhóm chọn là tín hiệu buffer_valid, tín hiệu này là output của bộ nhớ buffer_in để thông báo đến

controller dữ liệu đã được tải hoàn chỉnh vào bộ nhớ buffer_in. Cụ thể, khi buffer_in tích cực mức cao trong 1 chu kỳ, tiền đề được xem thỏa và các mệnh đề kết quả sẽ được kiểm tra. Sau khi buffer_valid, lần lượt các tín hiệu điều khiển tích cực với số chu kỳ theo bảng sau.

Bảng 3-3: Các phép tính và chu kỳ tín hiệu điều khiển của chúng tích cực

Thứ tự	Phép tính	Chu kỳ tín hiệu tích cực
1	DSP	Chu kỳ 5
2	Adder	Chu kỳ 6
3	Dequantize	Chu kỳ 15
4	Bias	Chu kỳ 16
5	Act	Chu kỳ 20
6	Quantize	Chu kỳ 21
7	buffer_in	Chu kỳ 22
8	buffer_out	Chu kỳ 23

Từ Bảng 3-3 phía trên, trên các Assertion cho các tín hiệu điều khiển có dạng như sau:

```

sequence s_buffer_valid;
  @ (posedge clk) $rose(buffer_valid) ##1
  $fell(buffer_valid);
endsequence

A_adder_en: assert property (@(posedge clk)
  s_buffer_valid |-> ##4 $rose(adder_en) ##1
  $fell(adder_en))

A_dequant_en: assert property (@(posedge clk)
  s_buffer_valid |-> ##5 $rose(dequant_en) ##1
  $fell(dequant_en))

A_bias_en: assert property (@(posedge clk)
  s_buffer_valid |-> ##14 $rose(bias_en) ##1
  $fell(bias_en))

```

```

A_act_en: assert property (@(posedge clk)
    s_buffer_valid |-> ##15 $rose(act_en) ##1
    $fell(act_en))

A_quant_en: assert property (@(posedge clk)
    s_buffer_valid |-> ##19 $rose(quant_en) ##1
    $fell(quant_en))

A_buffer_en: assert property (@(posedge clk)
    s_buffer_valid |-> ##20 $rose(buffer_en) ##1
    $fell(buffer_en))

A_valid: assert property (@(posedge clk)
    s_buffer_valid |-> ##21 $rose(valid) ##1
    $fell(valid))

```

Hình 3-26: Concurrent Assertion cho các tín hiệu điều khiển

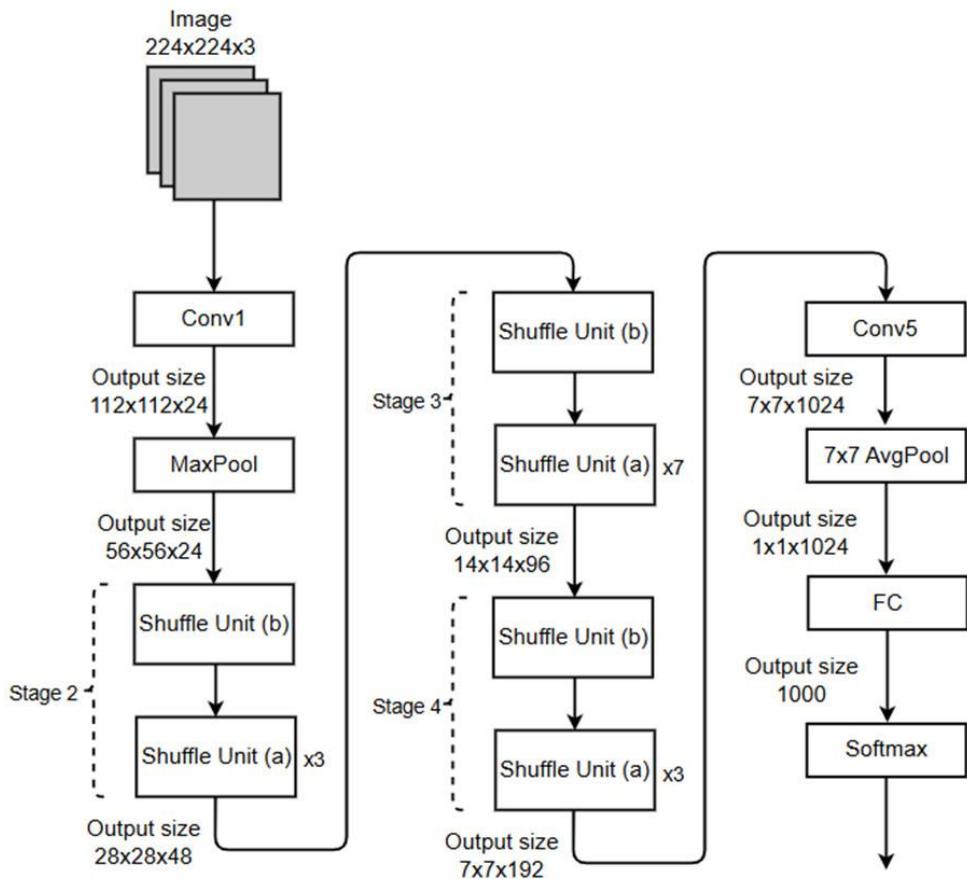
Chuỗi sequence “s_buffer_valid” được định nghĩa là tiền đề để sử dụng cho các Assertion bên dưới. Các Assertion có vai trò xác định các tín hiệu điều khiển có tích cực đúng theo Hình 3-26 và kéo dài trong 1 chu kỳ hay không.

3.6. Môi trường kiểm tra thiết kế CNN IP ShuffleNetV2

Sau khi hoàn thành xây dựng môi trường kiểm tra cho khối tính toán tích chập của thiết kế CNN IP LeNet5, nhóm áp dụng những kiến thức từ cơ sở lý thuyết trên để tiến hành xây dựng môi trường kiểm tra thiết kế cho CNN IP ShuffleNetV2 [13].

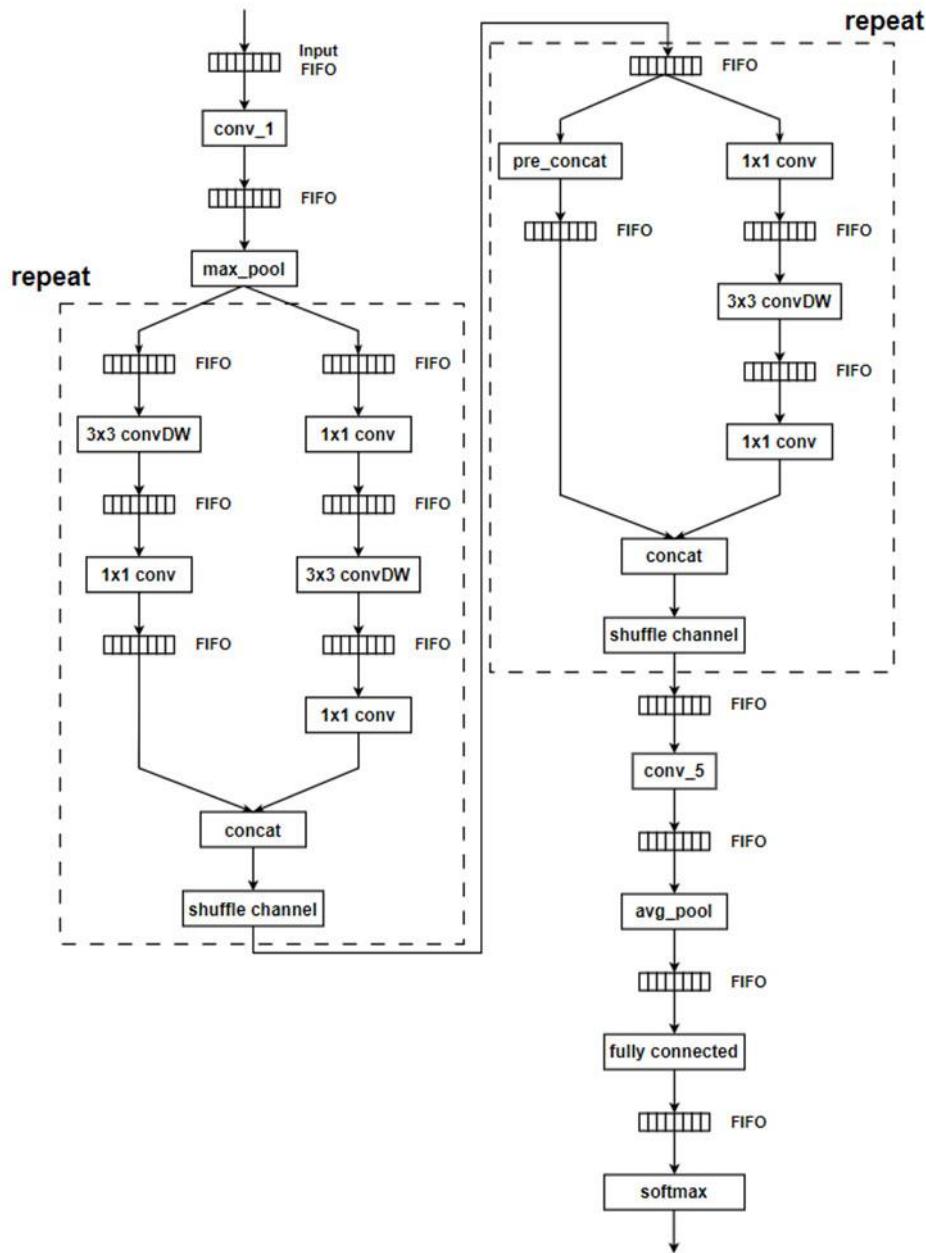
3.6.1. Mô tả thiết kế CNN IP ShuffleNetV2

Thiết kế CNN IP ShuffleNetV2 là đề tài Khóa luận tốt nghiệp được thiết kế bởi nhóm Trần Đăng Hậu và Lê Quang Phong. Hình 3-27 là mô tả tổng quan luồng hoạt động của mô hình ShuffleNetV2. Các khối tính toán chính của mô hình này bao gồm nhiều các lớp tích chập với các thuộc tính khác nhau theo mô tả của mô hình ShuffleNet bao gồm: đặc trưng đầu vào và đầu ra, stride, padding và hàm kích hoạt của từng lớp. Các thành phần chính của mô hình này bao gồm Conv1, MaxPool, Stage2, Stage3, Stage4, Conv5, AvgPool, FC và Softmax.



Hình 3-27: Tổng quan luồng hoạt động của ShuffleNetV2

CNN IP LeNet5 và ShuffleNetV2 có sự tương đồng trong luồng hoạt động. Cụ thể trước khi các phép tính tích chập được thực hiện, bộ nhớ lưu trọng số cần phải được nạp giá trị, thiết kế ShuffleNet cũng áp dụng cơ chế lượng tử hóa nên các giá trị trọng số của IP bao gồm kernel, bias, dequantize scale và quantize scale cần được lưu vào bộ nhớ trước khi thực hiện tính toán tích chập. Từ luồng hoạt động tổng quan này, mô hình thiết kế phần cứng của CNN IP ShuffleNetV2 có dạng như Hình 3-28. Các module chính bên trong thiết kế phần cứng bao gồm conv_1, max_pool, convDW, conv1x1, conv_5, avg_pool, fully_connected. Điểm chung của các khối này là bên trong đều chứa một khối tính toán tích chập. Đồng thời, giữa các module này có các module fifo để lưu và truy xuất đặc trưng đầu vào và đầu ra cho mỗi khối. Độ lớn của từng fifo có sự khác nhau do đặc trưng đầu vào và đầu ra của từng khối tích chập có sự khác nhau.



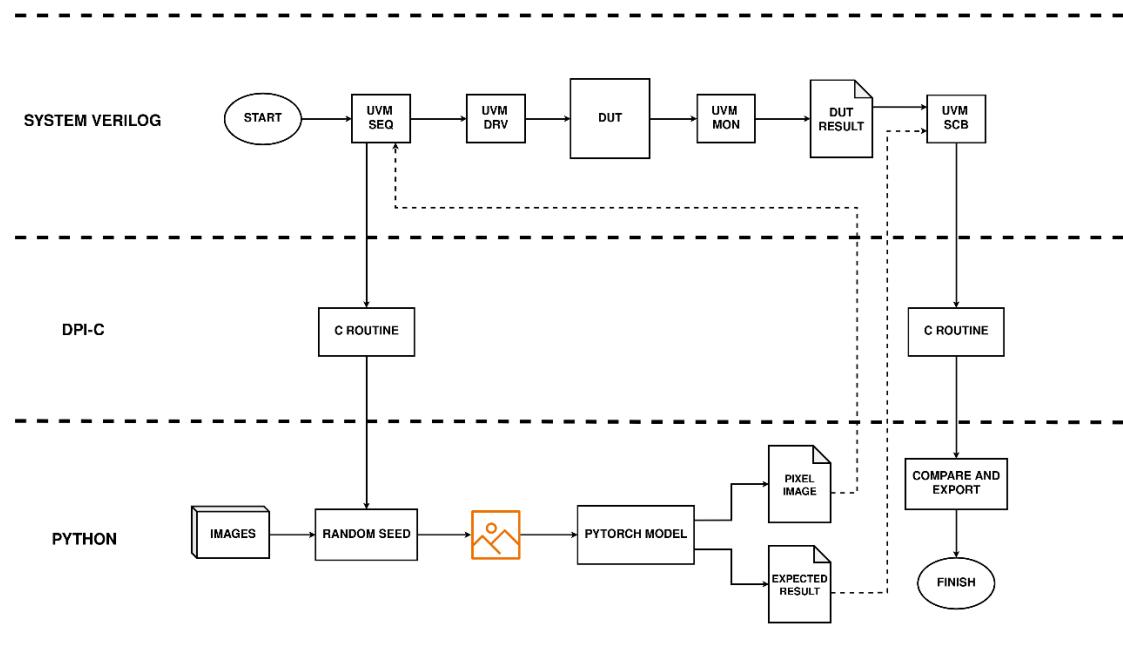
Hình 3-28: Tổng quan thiết kế phần cứng ShuffleNetV2

CNN IP LeNet5 và ShuffleNetV2 có sự tương đồng trong luồng hoạt động. Cụ thể, trước khi các phép tính tích chập được thực hiện, bộ nhớ lưu trọng số phải được nạp giá trị, thiết kế ShuffleNetV2 cũng áp dụng cơ chế lượng tử hóa nên các giá trị trọng số bao gồm kernel, bias, dequantize scale và quantize scale cần được lưu vào ô nhớ trước khi thực hiện tính toán tích chập.

3.6.2. Mô hình tin cậy cho thiết kế tích chập của CNN IP ShuffleNet

Đối với việc kiểm tra thiết kế tích chập cho CNN IP ShuffleNet, nhóm quyết định lựa chọn hướng tiếp cận black-box. Lý do để kiểm tra một cách chính xác nhất nhóm quyết định chọn mô hình tin cậy ShuffleNetV2 được cung cấp bởi Pytorch [14], Pytorch là một Framework mã nguồn mở cung cấp các mô hình học sâu với đặc điểm dễ sử dụng và được chấp nhận rộng rãi bởi cộng đồng, do đó tính đúng đắn của mô hình ShuffleNetV2 cung cấp bởi Pytorch đảm bảo được độ tin cậy.

Cụ thể, với cùng tập đầu vào, kết quả của DUT và mô hình tin cậy từ Pytorch được so sánh với nhau. Việc này được thực hiện thông qua ba lớp chính bao gồm: SystemVerilog, DPI-C và Python. Các giao tiếp giữa các lớp được thể hiện qua Hình 3-29. Ở lớp SystemVerilog, các Phase bên trong UVM được thực hiện theo thứ tự đã đề ra. Tuy nhiên ở giai đoạn tạo ra các gói tin đầu vào bởi uvm_sequence, SystemVerilog code sẽ gọi đến mô hình Pytorch để cùng tạo ra kết quả đầu ra từ mô hình tin cậy. Sau khi quá trình Run Phase hoàn thành ở UVM Testbench, ở Scoreboard, việc so sánh kết quả sẽ được thực hiện giữa kết quả của DUT và kết quả của mô hình cung cấp bởi Pytorch.



Hình 3-29: Các tầng bên trong môi trường kiểm tra

3.6.3. Đặc trưng cấu trúc mô trường UVM Testbench của thiết kế CNN IP ShuffleNetV2

Cấu trúc UVM Testbench của mô trường kiểm tra thiết kế CNN IP ShuffleNetV2 có sự tương đồng với LeNet5 ở phần lớn các thành phần bao gồm Transaction, Driver, Monitor, Agent và Env. Tuy nhiên, do đặc trưng thiết kế và cơ chế hoạt động khác nhau và phương pháp kiểm tra khác nhau (white-box và black-box), nhóm cần phải điều chỉnh hai thành phần trong UVM Testbench là Sequence và Scoreboard.

Cụ thể ở lớp Sequence, sau khi task chịu trách nhiệm tải trọng số vào bộ nhớ hoàn thành, để tạo ra các gói tin chứa data_in, lớp Sequence gọi hàm một hàm C thông qua DPI-C, hàm C này có chức năng gọi đến Python Code chứa thư viện Pytorch để tiến hành tiền xử lý ảnh đầu vào thành dạng Text File, Text File này được Sequence đọc và đóng gói thành các gói tin để gửi đến Driver. Python Code trên cũng đưa dữ liệu tiền xử lý vào thư viện Pytorch để tạo ra kết quả của mô hình tin cậy, kết quả này sẽ được so sánh với kết quả thực tế từ DUT ở Check Phase của lớp Scoreboard.

Ở lớp Scoreboard, sau khi Extract Phase hoàn thành, việc so sánh kết quả tin cậy và kết quả thực tế được thực hiện ở Check Phase thông qua một hàm C khác. Quá trình này được biểu diễn ở Hình 3-29.

3.6.4. Tối ưu tính tái sử dụng của mô trường kiểm tra

Khác với mô trường kiểm tra thiết kế CNN IP LeNet5, trong giai đoạn thiết kế mô trường kiểm tra cho thiết kế CNN IP ShuffleNetV2 nhóm có thực hiện một cải tiến đó là tập trung vào việc tối ưu tính tái sử dụng của các thành phần bên trong mô trường bằng cách sử dụng cơ chế tham số hóa cho các lớp của UVM Testbench. Mục tiêu của việc áp dụng tham số hóa các lớp chính là tăng hiệu suất trong việc xây dựng mô trường, rút gọn code giúp dễ quan sát và chỉnh sửa khi cần.

Cụ thể, như đã đề cập phía trên, các module trong thiết kế CNN IP ShuffleNetV2 có một điểm chung đó là đều có một khối tính toán tích chập bên trong, các khối này có cơ chế hoạt động tương tự nhau và sự khác nhau duy nhất

của chúng đó là các tham số bao gồm độ lớn đặc trưng đầu vào và đầu ra, tham số padding, stride, activation. Với đặc điểm này, nhóm thực hiện xây dựng các lớp nền có chứa các tham số cho phép người dùng thay đổi khi tạo ra các lớp con từ các lớp nền này. Các lớp con chứa các thuộc tính của lớp nền, các thuộc tính này tùy thuộc vào tham số được truyền vào bởi người dùng, người dùng có thể ghi đè hoặc điều chỉnh các thuộc tính của lớp con nhờ cơ chế kế thừa và đa hình trong các lớp của SystemVerilog.

```

class convl_std_seq #((
    parameter pDATA_WIDTH          = shared_pkg::pDATA_WIDTH,
    Parameter pWEIGHT_DATA_WIDTH = shared_pkg::pWEIGHT_DATA_WIDTH,
    parameter pWEIGHT_BASE_ADDR   = shared_pkg::pWEIGHT_BASE_ADDR,
    parameter pIN_CHANNEL         = convl_pkg::pIN_CHANNEL,
    parameter pOUT_CHANNEL        = convl_pkg::pOUT_CHANNEL,
    parameter pINPUT_WIDTH        = convl_pkg::pINPUT_WIDTH,
    parameter pINPUT_HEIGHT       = convl_pkg::pINPUT_HEIGHT,
    parameter pULTRA_RAM_NUM     = convl_pkg::pULTRA_RAM_NUM,
    parameter pBLOCK_RAM_NUM     = convl_pkg::pBLOCK_RAM_NUM,
    parameter pKERNEL_NUM         = convl_pkg::pKERNEL_NUM,
    parameter pBIAS_NUM           = convl_pkg::pBIAS_NUM,
    parameter pDEQUANT_SCALE_NUM = convl_pkg::pDEQUANT_SCALE_NUM,
    parameter pWEIGHTS_NUM        = convl_pkg::pWEIGHTS_NUM,
    parameter weight_path         = convl_pkg::weight_path,
    parameter image_path          = convl_pkg::image_path,
    parameter pZEROPOINT_LOAD    = 1,
)
type T_TRANS = convl_transaction) extends uvm_sequence#(T_TRANS);

// Register to Factory
`uvm_object_utils(convl_std_seq)

```

```

// Properties

T_TRANS tr;

int trans_amount = pINPUT_WIDTH * pINPUT_HEIGHT;

logic [pDATA_WIDTH*pIN_CHANNEL-1:0] in_fm [0:pINPUT_WIDTH*pINPUT_HEIGHT-1];

logic [pWEIGHT_DATA_WIDTH-1:0] weights [0:pWEIGHTS_NUM-1];

// Constructor

function new(input string path = "std_seq");
    super.new(path);
endfunction

// body task and other methods

endclass : convl_std_seq

```

Hình 3-30: Lớp nền Sequence được tham số hóa

Hình 3-30 là lớp nền của được khởi tạo từ uvm_sequence, lớp nền này chứa các tham số cần thiết để khởi tạo các class con từ nó. Các tham số của từng lớp được lưu vào các package riêng biệt để dễ dàng quản lý và thay đổi. Các tham số này được sử dụng trong các thuộc tính và phương thức của lớp được khởi tạo từ lớp nền. Nhờ đó ta không cần phải tái định nghĩa các thuộc tính và phương thức có cấu trúc tương tự với lớp nền ở lớp con.

```

class convDW_std_seq extends convl_std_seq#(
    .pWEIGHT_BASE_ADDR (shared_pkg::pWEIGHT_BASE_ADDR),
    .pIN_CHANNEL      (convDW_pkg::pIN_CHANNEL),
    .pOUT_CHANNEL     (convDW_pkg::pOUT_CHANNEL),
    .pINPUT_WIDTH     (convDW_pkg::pINPUT_WIDTH),
    .pINPUT_HEIGHT    (convDW_pkg::pINPUT_HEIGHT),

```

```

.pULTRA_RAM_NUM      (convDW_pkg::pULTRA_RAM_NUM) ,
.pBLOCK_RAM_NUM      (convDW_pkg::pBLOCK_RAM_NUM) ,
.pKERNEL_NUM          (convDW_pkg::pKERNEL_NUM) ,
.pBIAS_NUM            (convDW_pkg::pBIAS_NUM) ,
.pDEQUANT_SCALE_NUM  (convDW_pkg::pDEQUANT_SCALE_NUM) ,
.pWEIGHTS_NUM         (convDW_pkg::pWEIGHTS_NUM) ,
.weight_path          (convDW_pkg::weight_path) ,
.image_path           (convDW_pkg::image_path) ,
.pZEROPOINT_LOAD     (2) ,
.T_TRANS              (convDW_transaction)) ;

// Register to Factory
`uvm_object_utils(convDW_std_seq)

// Constructor
function new(input string path = "convDW_std_seq");
    super.new(path);
endfunction

endclass : convDW_std_seq

```

Hình 3-31: Lớp con khởi tạo từ lớp nền

Hình 3-31 là lớp con được khởi tạo từ lớp nền ở Hình 3-30. Ở lớp con các tham số thuộc package convDW_pkg được truyền vào để khởi tạo các thuộc tính và phương thức phù hợp với các tham số ấy.

Phương pháp tham số hóa các lớp được nhóm áp dụng vào phần lớn các thành phần trong UVM Testbench bao gồm uvm_sequence_item, uvm_sequence, uvm_agent, uvm_env để tối ưu khả năng tái sử dụng của môi trường kiểm tra, giúp code trở nên ngắn gọn, dễ dàng trong khả năng theo dõi và quản lý các lớp.

Chương 4. KIỂM TRA VÀ ĐÁNH GIÁ

Ở Chương 4, nhóm sẽ trình bày về tiêu chí đánh giá môi trường kiểm tra được nhóm xây dựng, phương pháp đánh giá kết quả và kết quả thực tế sau khi hiện thực môi trường kiểm tra và tiến hành mô phỏng cho thiết kế CNN IP.

4.1. Tiêu chí đánh giá

Dựa trên các mục tiêu được đề ra ban đầu, nhóm đưa ra các tiêu chí đánh giá cho môi trường kiểm tra thiết kế như sau.

Bảng 4-1: Tiêu chí đánh giá môi trường kiểm tra

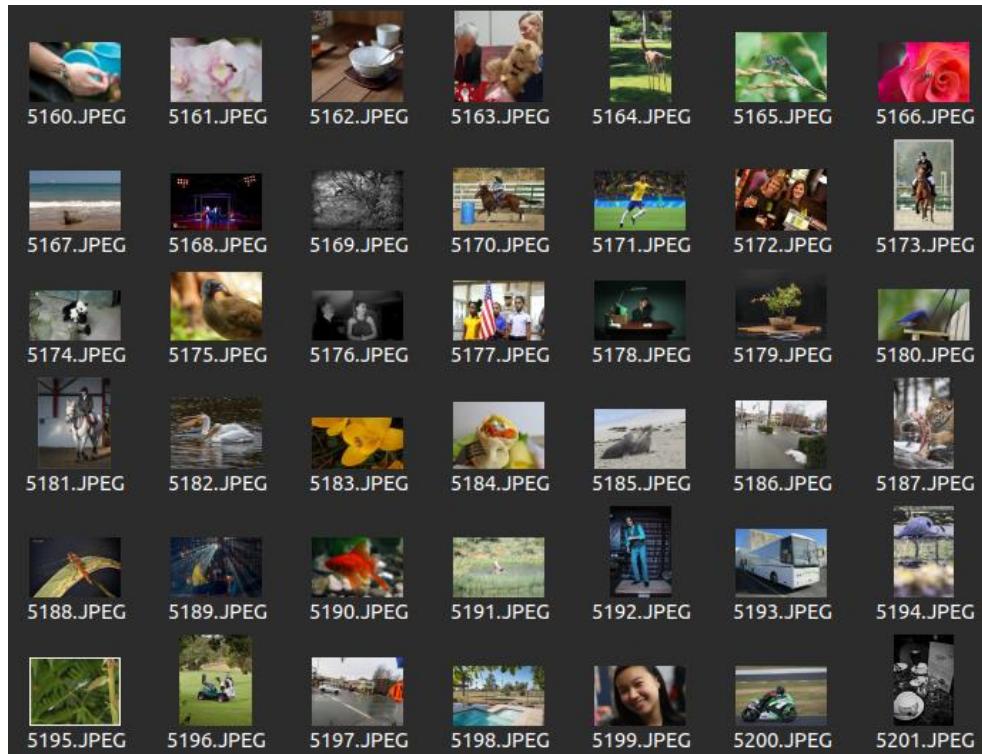
STT	Tiêu chí đánh giá	Đạt
1	Môi trường kiểm tra được xây dựng dựa trên cấu trúc và phương pháp kiểm tra UVM	X
2	Môi trường kiểm tra tối ưu về khả năng tự động hóa và tái sử dụng	X
3	Đa dạng số trường hợp kiểm tra đầu vào	X
4	Kiểm tra thiết kế theo phương pháp White-box và Black-box	X
5	Xây dựng mô hình tin cậy cho thiết kế sử dụng DPI-C	X
6	Sử dụng SystemVerilog Assertion để kiểm tra hành vi thiết kế	X
7	Kết quả kiểm thử của thiết kế được lưu lại và thông kê dưới dạng các file báo cáo	X
8	Tích hợp Functional Coverage	
9	Ứng dụng tính năng RAL của UVM vào việc kiểm tra bộ nhớ cho thiết kế	

4.2. Phương pháp đánh giá kết quả

Để đánh giá kết quả của môi trường kiểm tra thiết kế CNN IP, nhóm lựa chọn phương pháp lựa chọn ngẫu nhiên đầu vào cho thiết kế và kiểm tra kết quả dựa trên đầu vào ngẫu nhiên đó.

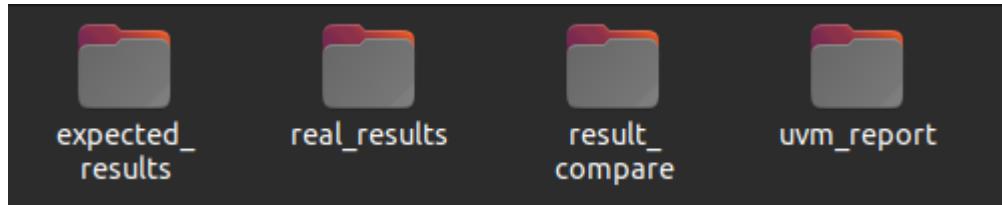
Ở giai đoạn xây dựng môi trường kiểm tra cho thiết kế CNN IP LeNet5, việc lựa chọn đầu vào ngẫu nhiên được hiện thực nhờ vào việc sử dụng cơ chế randomize() của SystemVerilog. Các tín hiệu được thực hiện ngẫu nhiên bao gồm tín hiệu weight_data và data_in cùng với một số ràng buộc cho hàm ngẫu nhiên.

Tiếp theo, ở giai đoạn xây dựng môi trường kiểm tra cho thiết kế CNN IP ShuffleNetV2. Nhóm lựa chọn tập đầu vào sử dụng tập dữ liệu “ImageNet Large Scale Visual Recognition Challenge 2017 (ILSVRC2017)” thay cho đầu vào của thiết kế. Tập dữ liệu gồm 5500 ảnh (Hình 4-1), UVM Testbench sử dụng DPI-C để gọi Python Code chọn ngẫu nhiên một ảnh và thực hiện việc tiền xử lý cho ảnh đó, kết quả là một Textfile để đưa vào quá trình mô phỏng cho DUT.



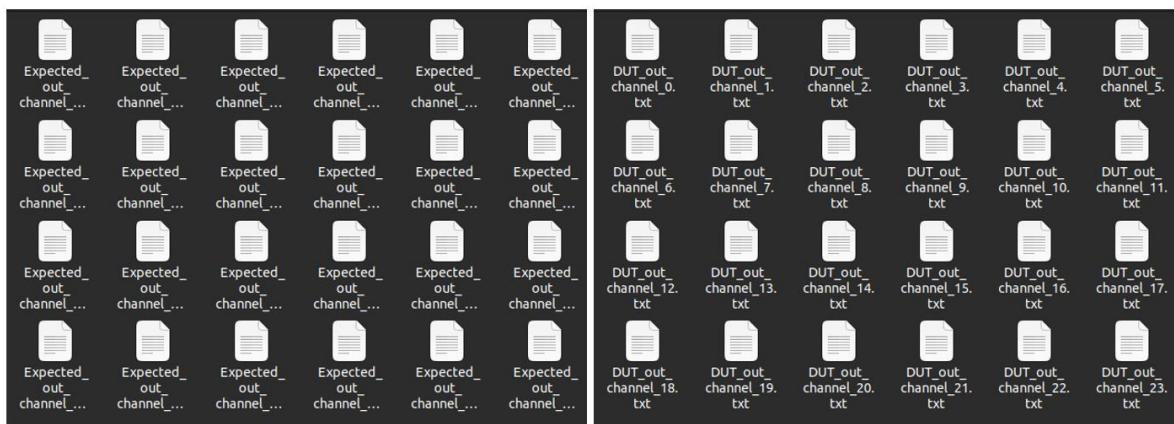
Hình 4-1: Một số hình ảnh trong tập dữ liệu đầu vào

Trong quá trình mô phỏng của UVM Testbench, sau khi Run Phase hoàn tất, việc trích xuất kết quả và lưu thành các file báo cáo bao gồm text file chứa các thông báo của `uvm_info macro và kết quả từ DUT được thực hiện ở Extract Phase, việc kiểm tra kết quả đã trích xuất được thực hiện ở Check Phase.



Hình 4-2: Folder chứa kết quả mô phỏng

Hình 4-2 là các thư mục chứa kết quả mô phỏng bao gồm kết quả tin cậy, kết quả thực tế từ DUT, kết quả so sánh và text file chứa các báo cáo của `uvm_info macro. Hình 4-3 là kết quả của thư mục chứa kết quả tin cậy và thư mục chứa kết quả thực tế của DUT.



Hình 4-3: Kết quả tin cậy và kết quả thực tế

Cuối cùng, kết quả tin cậy và kết quả thực tế được so sánh bằng Python Code để có được các file thống kê dưới dạng HTML. Các file thống kê này bao gồm số lượng kết quả, chi tiết so sánh kết quả (Hình 4-4) và độ lệch của kết quả so với mô

hình tin cậy, độ lệch trung bình và thống kê số lượng trường hợp dựa trên từng giá trị sai số (Hình 4-5).

[CONV1] Comparison Report Channel 21

Number of Matches: 9745

Number of Mismatches: 2799

Average Mismatch Difference: 1.49

Comparison Detail: [Expand]

Pixel	DUT Result	Pytorch Result	Difference
1	74	75	-1
2	79	79	0
3	79	79	0
4	79	79	0
5	79	79	0
6	79	79	0
7	79	79	0
8	79	79	0
9	78	79	-1
10	79	79	0
11	79	79	0
12	78	79	-1
13	79	79	0
14	79	79	0
15	79	79	0
16	79	79	0
17	79	80	-1
18	80	80	0
19	80	81	-1
20	80	81	-1
..

Hình 4-4: Kết quả so sánh từng trường hợp

Total of Difference Chart: [Expand]



Hình 4-5: Biểu đồ thống kê kết quả so sánh

4.3. Kết quả mô phỏng của UVM Testbench

Sau khi quá trình thực hiện mô phỏng, kết quả của quá trình mô phỏng được nhóm thực hiện lưu vào text file sử dụng phương thức set_report_default_file_hier() của thư viện tích hợp UVM, các report của UVM được lưu vào file có dạng như sau:

```
1 UVM_INFO /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/thesis_HauPhong/my_verification_HAO/
CONV_verification/CONV_verification.srccs/sim_pe_conv_mac_conv1/new/monitor.sv(39) @ 10000:
uvm_test_top.e.a.m [MON] SYSTEM RESET DETECTED
2 UVM_INFO /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/thesis_HauPhong/my_verification_HAO/
CONV_verification/CONV_verification.srccs/sim_pe_conv_mac_conv1/new/scoreboard.sv(123) @ 10000:
uvm_test_top.e.s [SCB] SYSTEM RESET
3 UVM_INFO /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/thesis_HauPhong/my_verification_HAO/
CONV_verification/CONV_verification.srccs/sim_pe_conv_mac_conv1/new/scoreboard.sv(126) @ 10000:
uvm_test_top.e.s [SCB] VIRTUAL MEMORY FILE READY TO WRITE: /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/
log_dir/virtual_mem.txt
4 UVM_INFO /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/thesis_HauPhong/my_verification_HAO/
CONV_verification/CONV_verification.srccs/sim_pe_conv_mac_conv1/new/monitor.sv(39) @ 30000:
uvm_test_top.e.a.m [MON] SYSTEM RESET DETECTED
5 UVM_INFO /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/thesis_HauPhong/my_verification_HAO/
CONV_verification/CONV_verification.srccs/sim_pe_conv_mac_conv1/new/scoreboard.sv(123) @ 30000:
uvm_test_top.e.s [SCB] SYSTEM RESET
6 UVM_INFO /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/thesis_HauPhong/my_verification_HAO/
CONV_verification/CONV_verification.srccs/sim_pe_conv_mac_conv1/new/scoreboard.sv(126) @ 30000:
uvm_test_top.e.s [SCB] VIRTUAL MEMORY FILE READY TO WRITE: /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/
log_dir/virtual_mem.txt
7 UVM_INFO /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/thesis_HauPhong/my_verification_HAO/
CONV_verification/CONV_verification.srccs/sim_pe_conv_mac_conv1/new/monitor.sv(39) @ 50000:
uvm_test_top.e.a.m [MON] SYSTEM RESET DETECTED
8 UVM_INFO /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/thesis_HauPhong/my_verification_HAO/
CONV_verification/CONV_verification.srccs/sim_pe_conv_mac_conv1/new/scoreboard.sv(123) @ 50000:
uvm_test_top.e.s [SCB] SYSTEM RESET
9 UVM_INFO /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/thesis_HauPhong/my_verification_HAO/
CONV_verification/CONV_verification.srccs/sim_pe_conv_mac_conv1/new/scoreboard.sv(126) @ 50000:
uvm_test_top.e.s [SCB] VIRTUAL MEMORY FILE READY TO WRITE: /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/
log_dir/virtual_mem.txt
10 UVM_INFO /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/thesis_HauPhong/my_verification_HAO/
CONV_verification/CONV_verification.srccs/sim_pe_conv_mac_conv1/new/monitor.sv(39) @ 70000:
uvm_test_top.e.a.m [MON] SYSTEM RESET DETECTED
11 UVM_INFO /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/thesis_HauPhong/my_verification_HAO/
CONV_verification/CONV_verification.srccs/sim_pe_conv_mac_conv1/new/scoreboard.sv(123) @ 70000:
uvm_test_top.e.s [SCB] SYSTEM RESET
12 UVM_INFO /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/thesis_HauPhong/my_verification_HAO/
CONV_verification/CONV_verification.srccs/sim_pe_conv_mac_conv1/new/scoreboard.sv(126) @ 70000:
uvm_test_top.e.s [SCB] VIRTUAL MEMORY FILE READY TO WRITE: /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/
log_dir/virtual_mem.txt
13 UVM_INFO /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/thesis_HauPhong/my_verification_HAO/
CONV_verification/CONV_verification.srccs/sim_pe_conv_mac_conv1/new/driver.sv(71) @ 90000:
uvm_test_top.e.a.d [DRV] SYSTEM RESET: START OF SIMULATION
14 UVM_INFO /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/thesis_HauPhong/my_verification_HAO/
CONV_verification/CONV_verification.srccs/sim_pe_conv_mac_conv1/new/monitor.sv(39) @ 90000:
uvm_test_top.e.a.m [MON] SYSTEM RESET DETECTED
15 UVM_INFO /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/thesis_HauPhong/my_verification_HAO/
CONV_verification/CONV_verification.srccs/sim_pe_conv_mac_conv1/new/scoreboard.sv(123) @ 90000:
```

Hình 4-6: Kết quả của quá trình mô phỏng

Ở Hình 4-6, do trong file có nhiều thông tin không cần thiết, nhóm sử dụng Perl Script để lọc ra các thông tin có ý nghĩa, kết quả của quá trình này nhóm thu được kết quả mô phỏng như sau:

TIME	HIERARCHY	MESSAGE
10	uvm_test_top.e.a.m	[MON] SYSTEM RESET DETECTED
10	uvm_test_top.e.s	[SCB] SYSTEM RESET
10	uvm_test_top.e.s	[SCB] VIRTUAL MEMORY FILE READY TO WRITE: /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/log_dir/virtual_mem.txt
30	uvm_test_top.e.m	[MON] SYSTEM RESET DETECTED
30	uvm_test_top.e.s	[SCB] SYSTEM RESET
30	uvm_test_top.e.s	[SCB] VIRTUAL MEMORY FILE READY TO WRITE: /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/log_dir/virtual_mem.txt
50	uvm_test_top.e.m	[MON] SYSTEM RESET DETECTED
50	uvm_test_top.e.s	[SCB] SYSTEM RESET
50	uvm_test_top.e.s	[SCB] VIRTUAL MEMORY FILE READY TO WRITE: /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/log_dir/virtual_mem.txt
70	uvm_test_top.e.m	[MON] SYSTEM RESET DETECTED
70	uvm_test_top.e.s	[SCB] SYSTEM RESET
70	uvm_test_top.e.s	[SCB] VIRTUAL MEMORY FILE READY TO WRITE: /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/log_dir/virtual_mem.txt
90	uvm_test_top.e.d	[DRV] SYSTEM RESET: START OF SIMULATION
90	uvm_test_top.e.m	[MON] SYSTEM RESET DETECTED
90	uvm_test_top.e.s	[SCB] SYSTEM RESET
90	uvm_test_top.e.s	[SCB] VIRTUAL MEMORY FILE READY TO WRITE: /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/log_dir/virtual_mem.txt
90	uvm_test_top.e.a.seqr@ss	[STD_SEQ] -----[KERNEL LOADING]-----
110	uvm_test_top.e.a.d	[DRV] [MEMORY LOADING] Weight Loaded: weight_addr = 40000000 , weight_data = 1d8b8e52_e4c698a5
110	uvm_test_top.e.m	[MON] [MEMORY LOADING] Weight Loaded: weight_addr = 40000000 , weight_data = 1d8b8e52_e4c698a5
110	uvm_test_top.e.s	[SCB] [MEMORY LOADING] Weight Loaded: weight_addr = 40000000 , weight_data = 1d8b8e52_e4c698a5
110	uvm_test_top.e.a.seqr@ss	[STD_SEQ] -----[KERNEL LOADING]-----
130	uvm_test_top.e.a.d	[DRV] [MEMORY LOADING] Weight Loaded: weight_addr = 40000000 , weight_data = ca5b9f68_1c4c07d4
130	uvm_test_top.e.m	[MON] [MEMORY LOADING] Weight Loaded: weight_addr = 40000000 , weight_data = ca5b9f68_1c4c07d4
130	uvm_test_top.e.s	[SCB] [MEMORY LOADING] Weight Loaded: weight_addr = 40000000 , weight_data = ca5b9f68_1c4c07d4
130	uvm_test_top.e.a.seqr@ss	[STD_SEQ] -----[KERNEL LOADING]-----
150	uvm_test_top.e.a.d	[DRV] [MEMORY LOADING] Weight Loaded: weight_addr = 40000000 , weight_data = 23fbcl5a_0b5ee005
150	uvm_test_top.e.m	[MON] [MEMORY LOADING] Weight Loaded: weight_addr = 40000000 , weight_data = 23fbcl5a_0b5ee005
150	uvm_test_top.e.s	[SCB] [MEMORY LOADING] Weight Loaded: weight_addr = 40000000 , weight_data = 23fbcl5a_0b5ee005
150	uvm_test_top.e.a.seqr@ss	[STD_SEQ] -----[KERNEL LOADING]-----
170	uvm_test_top.e.a.d	[DRV] [MEMORY LOADING] Weight Loaded: weight_addr = 40000000 , weight_data = 2a6bf40c_31e73307
170	uvm_test_top.e.m	[MON] [MEMORY LOADING] Weight Loaded: weight_addr = 40000000 , weight_data = 2a6bf40c_31e73307
170	uvm_test_top.e.s	[SCB] [MEMORY LOADING] Weight Loaded: weight_addr = 40000000 , weight_data = 2a6bf40c_31e73307
170	uvm_test_top.e.a.seqr@ss	[STD_SEQ] -----[KERNEL LOADING]-----
190	uvm_test_top.e.a.d	[DRV] [MEMORY LOADING] Weight Loaded: weight_addr = 40000001 , weight_data = 9dac37b0_0ffcef29
190	uvm_test_top.e.m	[MON] [MEMORY LOADING] Weight Loaded: weight_addr = 40000001 , weight_data = 9dac37b0_0ffcef29
190	uvm_test_top.e.s	[SCB] [MEMORY LOADING] Weight Loaded: weight_addr = 40000001 , weight_data = 9dac37b0_0ffcef29
190	uvm_test_top.e.a.seqr@ss	[STD_SEQ] -----[KERNEL LOADING]-----
210	uvm_test_top.e.a.d	[DRV] [MEMORY LOADING] Weight Loaded: weight_addr = 40000001 , weight_data = 3dbc8c96_25981a92
210	uvm_test_top.e.m	[MON] [MEMORY LOADING] Weight Loaded: weight_addr = 40000001 , weight_data = 3dbc8c96_25981a92
210	uvm_test_top.e.s	[SCB] [MEMORY LOADING] Weight Loaded: weight_addr = 40000001 , weight_data = 3dbc8c96_25981a92
210	uvm_test_top.e.a.seqr@ss	[STD_SEQ] -----[KERNEL LOADING]-----
230	uvm_test_top.e.a.d	[DRV] [MEMORY LOADING] Weight Loaded: weight_addr = 40000001 , weight_data = 8a9cf243_f2bb4e2
230	uvm_test_top.e.m	[MON] [MEMORY LOADING] Weight Loaded: weight_addr = 40000001 , weight_data = 8a9cf243_f2bb4e2

Hình 4-7: Thông tin có ý nghĩa trích xuất từ file báo cáo

Hình 4-7 là kết quả của Perl Script, kết quả mô phỏng được chia thành 3 cột chính bao gồm:

- **TIME:** thời gian thông báo được thực hiện
- **HIERARCHY:** Cây gia phả của phần tử UVM mà thông báo được thực hiện. Ví dụ: “uvm_test_top.e.a.m” là cây gia phả của Monitor (uvm_test_top.env.agent.monitor)
- **MESSAGE:** Chi tiết thông tin của thông báo

Các thông báo được thực hiện ở từng phần tử bên trong UVM Testbench và ở từng giai đoạn cụ thể bao gồm UVM Phase và các bước của quá trình tính toán tích chập bao gồm: reset, tải các trọng số vào ô nhớ, thực hiện tính toán.

Hình 4-8 thể hiện quá trình Reset đang được thực hiện. Tín hiệu reset được truyền tới DUT từ lớp Driver, do đó khi các lớp Monitor và Scoreboard nhận được gói tin có tín hiệu reset, các thông báo của hai lớp này được in ra màn hình.

TIME	HIERARCHY	MESSAGE
10	uvm_test_top.e.a.m	[MON] SYSTEM RESET DETECTED
10	uvm_test_top.e.s	[SCB] SYSTEM RESET
10	uvm_test_top.e.s	[SCB] VIRTUAL MEMORY FILE READY TO WRITE: /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/log_dir/virtual_mem.txt
30	uvm_test_top.e.a.m	[MON] SYSTEM RESET DETECTED
30	uvm_test_top.e.s	[SCB] SYSTEM RESET
30	uvm_test_top.e.s	[SCB] VIRTUAL MEMORY FILE READY TO WRITE: /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/log_dir/virtual_mem.txt
50	uvm_test_top.e.a.m	[MON] SYSTEM RESET DETECTED
50	uvm_test_top.e.s	[SCB] SYSTEM RESET
50	uvm_test_top.e.s	[SCB] VIRTUAL MEMORY FILE READY TO WRITE: /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/log_dir/virtual_mem.txt
70	uvm_test_top.e.a.m	[MON] SYSTEM RESET DETECTED
70	uvm_test_top.e.s	[SCB] SYSTEM RESET
70	uvm_test_top.e.s	[SCB] VIRTUAL MEMORY FILE READY TO WRITE: /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/log_dir/virtual_mem.txt
90	uvm_test_top.e.a.d	[DRV] SYSTEM RESET: START OF SIMULATION
90	uvm_test_top.e.a.m	[MON] SYSTEM RESET DETECTED
90	uvm_test_top.e.s	[SCB] SYSTEM RESET
90	uvm_test_top.e.s	[SCB] VIRTUAL MEMORY FILE READY TO WRITE: /home/hao/Documents/0.KHOA_LUAN_TOT_NGHIEP/log_dir/virtual_mem.txt

Hình 4-8: Trạng thái Reset của UVM Testbench

Hình 4-9 thể hiện quá trình tải trọng số đang được thực hiện, trên hình là trọng số kernel và trọng số bias đang được tải, giá trị weight_addr và weight_data được hiển thị ở thông báo từ các lớp Driver, Monitor và Scoreboard. Mỗi gói tin trọng số có ba thông báo từ ba bộ phận trong UVM Testbench là Driver, Monitor và Scoreboard.

730	uvm_test_top.e.a.seqr@ss	[STD_SEQ] -----[KERNEL LOADING]-----
750	uvm_test_top.e.a.d	[DRV] [MEMORY LOADING] Weight Loaded: weight_addr = 40000008 , weight_data = caae7ffa_e7fba449
750	uvm_test_top.e.a.m	[MON] [MEMORY LOADING] Weight Loaded: weight_addr = 40000008 , weight_data = caae7ffa_e7fba449
750	uvm_test_top.e.s	[SCB] [MEMORY LOADING] Weight Loaded: weight_addr = 40000008 , weight_data = caae7ffa_e7fba449
750	uvm_test_top.e.a.seqr@ss	[STD_SEQ] -----[KERNEL LOADING]-----
770	uvm_test_top.e.a.d	[DRV] [MEMORY LOADING] Weight Loaded: weight_addr = 40000008 , weight_data = 9180af3b_102ef510
770	uvm_test_top.e.a.m	[MON] [MEMORY LOADING] Weight Loaded: weight_addr = 40000008 , weight_data = 9180af3b_102ef510
770	uvm_test_top.e.s	[SCB] [MEMORY LOADING] Weight Loaded: weight_addr = 40000008 , weight_data = 9180af3b_102ef510
770	uvm_test_top.e.a.seqr@ss	[STD_SEQ] -----[KERNEL LOADING]-----
790	uvm_test_top.e.a.d	[DRV] [MEMORY LOADING] Weight Loaded: weight_addr = 40000008 , weight_data = c522ef42_efe7b4e3
790	uvm_test_top.e.a.m	[MON] [MEMORY LOADING] Weight Loaded: weight_addr = 40000008 , weight_data = c522ef42_efe7b4e3
790	uvm_test_top.e.s	[SCB] [MEMORY LOADING] Weight Loaded: weight_addr = 40000008 , weight_data = c522ef42_efe7b4e3
790	uvm_test_top.e.a.seqr@ss	[STD_SEQ] -----[KERNEL LOADING]-----
810	uvm_test_top.e.a.d	[DRV] [MEMORY LOADING] Weight Loaded: weight_addr = 40000008 , weight_data = e5954066_c725e3ce
810	uvm_test_top.e.a.m	[MON] [MEMORY LOADING] Weight Loaded: weight_addr = 40000008 , weight_data = e5954066_c725e3ce
810	uvm_test_top.e.s	[SCB] [MEMORY LOADING] Weight Loaded: weight_addr = 40000008 , weight_data = e5954066_c725e3ce
810	uvm_test_top.e.a.seqr@ss	[STD_SEQ] -----[BIAS LOADING]-----
830	uvm_test_top.e.a.d	[DRV] [MEMORY LOADING] Weight Loaded: weight_addr = 40000009 , weight_data = 0002a284_000181dc
830	uvm_test_top.e.a.m	[MON] [MEMORY LOADING] Weight Loaded: weight_addr = 40000009 , weight_data = 0002a284_000181dc
830	uvm_test_top.e.s	[SCB] [MEMORY LOADING] Weight Loaded: weight_addr = 40000009 , weight_data = 0002a284_000181dc
830	uvm_test_top.e.a.seqr@ss	[STD_SEQ] -----[BIAS LOADING]-----
850	uvm_test_top.e.a.d	[DRV] [MEMORY LOADING] Weight Loaded: weight_addr = 4000000a , weight_data = fffff1544_00018eb
850	uvm_test_top.e.a.m	[MON] [MEMORY LOADING] Weight Loaded: weight_addr = 4000000a , weight_data = fffff1544_00018eb
850	uvm_test_top.e.s	[SCB] [MEMORY LOADING] Weight Loaded: weight_addr = 4000000a , weight_data = fffff1544_00018eb
850	uvm_test_top.e.a.seqr@ss	[STD_SEQ] -----[BIAS LOADING]-----
870	uvm_test_top.e.a.d	[DRV] [MEMORY LOADING] Weight Loaded: weight_addr = 4000000b , weight_data = 000298dd_fffff0a83
870	uvm_test_top.e.a.m	[MON] [MEMORY LOADING] Weight Loaded: weight_addr = 4000000b , weight_data = 000298dd_fffff0a83
870	uvm_test_top.e.s	[SCB] [MEMORY LOADING] Weight Loaded: weight_addr = 4000000b , weight_data = 000298dd_fffff0a83
870	uvm_test_top.e.a.seqr@ss	[STD_SEQ] -----[BIAS LOADING]-----

Hình 4-9: Trạng thái tải trọng số đến DUT của UVM Testbench

Sau khi hoàn tất quá trình tải trọng số vào bộ nhớ, các gói tin chứa đầu vào data_in được đưa vào DUT từ UVM Testbench, kết quả của quá trình mô phỏng được thu lại thông qua lớp Monitor (Hình 4-10). Chi tiết thông tin gói tin được in ra thông báo thực hiện bởi hàm tr_display() bên trong lớp Transaction.

1150	uvm_test_top.e.a.seqr@ss	[STD_SEQ] Memory Weights Load Finished. Start Generating Transaction Data!
1150	uvm_test_top.e.a.seqr@ss	[STD_SEQ] [No.0] Transaction Generated: data_in = 5b4ff9212f1e1b8b4
1150	uvm_test_top.e.a.seqr@ss.tr	[DRV] -----Transaction Info-----
1150	uvm_test_top.e.a.seqr@ss.tr	[DRV] op = RUNNING
1150	uvm_test_top.e.a.seqr@ss.tr	[DRV] en = 1
1150	uvm_test_top.e.a.seqr@ss.tr	[DRV] buffer_in_en = 1
1150	uvm_test_top.e.a.seqr@ss.tr	[DRV] data_in = 5b4ff9212f1e1b8b4
1150	uvm_test_top.e.a.seqr@ss.tr	[DRV] load_weight = 0
1150	uvm_test_top.e.a.seqr@ss.tr	[DRV] weight_addr = d4455b41 - dunkare
1150	uvm_test_top.e.a.seqr@ss.tr	[DRV] weight_data = 434d000005ca8 - dunkare
1150	uvm_test_top.e.a.seqr@ss.tr	[DRV] -----
1170	uvm_test_top.e.a.d	[DRV] Deassert buffer_in_en : 1
1350	reporter@tr	[MON] -----Transaction Info-----
1350	reporter@tr	[MON] op = RUNNING
1350	reporter@tr	[MON] en = 1
1350	reporter@tr	[MON] buffer_in_en = 1
1350	reporter@tr	[MON] data_in = 5b4ff9212f1e1b8b4
1350	reporter@tr	[MON] load_weight = 0
1350	reporter@tr	[MON] weight_addr = 40000019 - dunkare
1350	reporter@tr	[MON] weight_data = 2c18700005f10 - dunkare
1350	reporter@tr	[MON] data_out = 0
1350	reporter@tr	[MON] valid = 0
1350	reporter@tr	[MON] done = 0
1350	reporter@tr	[MON] pe_ready = 1
1350	reporter@tr	[MON] -----
1370	uvm_test_top.e.a.seqr@ss	[STD_SEQ] [No.1] Transaction Generated: data_in = fc1254418aacdacffcc
1370	uvm_test_top.e.a.seqr@ss.tr	[DRV] -----Transaction Info-----
1370	uvm_test_top.e.a.seqr@ss.tr	[DRV] op = RUNNING
1370	uvm_test_top.e.a.seqr@ss.tr	[DRV] en = 1
1370	uvm_test_top.e.a.seqr@ss.tr	[DRV] buffer_in_en = 1
1370	uvm_test_top.e.a.seqr@ss.tr	[DRV] data_in = fc1254418aacdacffcc
1370	uvm_test_top.e.a.seqr@ss.tr	[DRV] load_weight = 0
1370	uvm_test_top.e.a.seqr@ss.tr	[DRV] weight_addr = 1fa45dc9 - dunkare
1370	uvm_test_top.e.a.seqr@ss.tr	[DRV] weight_data = 1d6330002c97a - dunkare
1370	uvm_test_top.e.a.seqr@ss.tr	[DRV] -----
1390	uvm_test_top.e.a.d	[DRV] Deassert buffer_in_en : 1
1530	reporter	[SVA] @1530000 A_adder_en success
1550	reporter	[SVA] @1550000 A_dequant_en success
1570	reporter@tr	[MON] -----Transaction Info-----

Hình 4-10: Trạng thái thực hiện gửi các gói tin data_in đến DUT

Sau khi Run Phase hoàn thành, kết quả thực tế được trích xuất bởi Extract Phase để chuẩn bị tiến hành đánh giá kết quả. Ở Check Phase, kết quả thực tế và kết quả mong đợi được so sánh, quá trình và kết quả so sánh được in ra thông báo bởi lớp Scoreboard. Thông tin đầu vào data_in và data_out được hiển thị ở thông báo cùng với trạng thái “PASS” hoặc “FAIL” (Hình 4-11).

174290	uvm_test_top.e.s	[SCB_RPT] +----- PASS -----+ [SCB_RPT] DATA IN = 65d87ff29bcbe803d1 [SCB_RPT] EXPECTED DATA OUT = 80b1c4cf8744e5abb5ae3cefd5e64120ddcd26ce7cc25acb3d8eeee3f3dd3ebd1 [SCB_RPT] REAL DATA OUT = 80b1c4cf8744e5abb5ae3cefd5e64120ddcd26ce7cc25acb3d8eeee3f3dd3ebd1 [SCB_RPT] +-----+
174290	uvm_test_top.e.s	[SCB_RPT] +----- FAIL -----+ [SCB_RPT] DATA IN = c6e4aaaf7e9ea3d76e1 [SCB_RPT] EXPECTED DATA OUT = 7db5c9d08949e9acb5b33eed6ea461ddad765ebd022b5add6eced3c3ad2e8ca [SCB_RPT] REAL DATA OUT = 7db5c9d08949e9acb5b33eed6ea461ddad766ebd022b5add6eced3c3ad2e8ca [SCB_RPT] +-----+
174290	uvm_test_top.e.s	[SCB_RPT] +-----+
174290	uvm_test_top.e.s	[SCB_RPT] [576] [SCB_RPT] +-----
174290	uvm_test_top.e.s	[SCB_RPT] +----- FAIL -----+ [SCB_RPT] DATA IN = b9b031a1bf2f9786ef [SCB_RPT] EXPECTED DATA OUT = 83aec0d89048e5a3bab33deedceb3e20dbd478ebcf20b3b6dcoded3e3ad2e8d0 [SCB_RPT] REAL DATA OUT = 83aec0d89048e5a3bab33deedceb3e20dbd478ebcf20b3b6dcoded3e3ad2e8d0 [SCB_RPT] +-----+
174290	uvm_test_top.e.s	[SCB_RPT] +-----+
174290	uvm_test_top.e.s	[SCB_RPT] [577] [SCB_RPT] +-----
174290	uvm_test_top.e.s	[SCB_RPT] +----- PASS -----+ [SCB_RPT] DATA IN = b9b031a1bf2f9786ef [SCB_RPT] EXPECTED DATA OUT = 83aec0d89048e5a3bab33deedceb3e20dbd478ebcf20b3b6dcoded3e3ad2e8d0 [SCB_RPT] REAL DATA OUT = 83aec0d89048e5a3bab33deedceb3e20dbd478ebcf20b3b6dcoded3e3ad2e8d0 [SCB_RPT] +-----+
174290	uvm_test_top.e.s	[SCB_RPT] +-----+
174290	uvm_test_top.e.s	[SCB_RPT] [578] [SCB_RPT] +----- PASS -----+ [SCB_RPT] DATA IN = b9b113eff9dbf633df [SCB_RPT] EXPECTED DATA OUT = 80aac1d4873de6a1b8b03eed8e74022d8d574ebcb26a9b3ddefed4838d4e6cf [SCB_RPT] REAL DATA OUT = 80aac1d4873de6a1b8b03eed8e74022d8d574ebcb26a9b3ddefed4838d4e6cf [SCB_RPT] +-----+
174290	uvm_test_top.e.s	[SCB_RPT] +-----

Hình 4-11: Kết quả kiểm thử ở Check Phase

```

| 174298 | uvm_test_top.e.s | [SCB_RPT] ###### LIST OF FAIL CASES #####
| 174299 | uvm_test_top.e.s | [SCB_RPT] # !!! FAIL AT [ 25] - DATA IN = 5b11a96717ed9cd66a #
| 174299 | uvm_test_top.e.s | [SCB_RPT] # !!! FAIL AT [ 156] - DATA IN = eb828e2c88d184a6f1 #
| 174299 | uvm_test_top.e.s | [SCB_RPT] # !!! FAIL AT [ 261] - DATA IN = cec486770fffb3b6bf #
| 174299 | uvm_test_top.e.s | [SCB_RPT] # !!! FAIL AT [ 262] - DATA IN = b61a49cbe7fde3befa #
| 174299 | uvm_test_top.e.s | [SCB_RPT] # !!! FAIL AT [ 576] - DATA IN = c6e4aaftc9ea3d76e1 #
| 174299 | uvm_test_top.e.s | [SCB_RPT] #####
| 174299 | uvm_test_top.e.s | [SCB_RPT] ****
| 174299 | uvm_test_top.e.s | [SCB_RPT] q_real_data_in size : 784
| 174299 | uvm_test_top.e.s | [SCB_RPT] q_real_data_out size : 784
| 174299 | uvm_test_top.e.s | [SCB_RPT] q_expected_data_out size : 784
| 174299 | uvm_test_top.e.s | [SCB_RPT] Pass cases : 779 / 784 (99.362245 %)
| 174299 | uvm_test_top.e.s | [SCB_RPT] Fail cases : 5 / 784 (0.637755 %)
| 174299 | uvm_test_top.e.s | [SCB_RPT] ****
| 174299 | uvm_test_top.e.s | [SCB_RPT]
| 174299 | reporter          | [UVM/REPORT/SERVER] [UVM/RELENOTES] 1
| 174299 | reporter          | [UVM/COMP/NAMECHECK] 1
| 174299 | reporter          | [TEST_DONE] 1
| 174299 | reporter          | [SVA] 5494
| 174299 | reporter          | [STD_SEQ] 841
| 174299 | reporter          | [SCB_RPT] 5504
| 174299 | reporter          | [SCB] 67
| 174299 | reporter          | [RNTST] 1
| 174299 | reporter          | [MON] 10289
| 174299 | reporter          | [DRV] 7924
| 174299 | reporter          | ** Report counts by id
| 174299 | reporter          | UVM_FATAL : 0
| 174299 | reporter          | UVM_ERROR : 0
| 174299 | reporter          | UVM_WARNING : 0
| 174299 | reporter          | UVM_INFO :30123
| 174299 | reporter          | ** Report counts by severity
| 174299 | reporter          | --- UVM Report Summary ---

```

Hình 4-12: Thông kê kết quả ở Report Phase

Cuối cùng ở Report Phase, các thông tin về đánh giá kết quả của lớp Scoreboard sau khi thực hiện kiểm tra (Hình 4-12) được in ra thông báo bao gồm: các trường hợp sai và vị trí của chúng, số lượng mẫu thử, tỉ lệ chính xác của thiết kế dựa trên số lượng mẫu thử và thống kê về số lượng các thông báo của UVM Testbench. Một số thống kê mặc định về số lượng của từng macro báo cáo tích hợp bên trong UVM Testbench bao gồm UVM_FATAL, UVM_ERROR, UVM_WARNING và UVM_INFO được in ở cuối báo cáo.

4.4. Kết quả kiểm tra hành vi thiết kế

Việc kiểm tra hành vi thiết kế được nhóm thực hiện thông qua các mệnh đề Concurrent Assertion định nghĩa ở Hình 3-26, kết quả các mệnh đề Concurrent Assertion này cũng được in ra text file chứa thông báo của môi trường UVM Testbench.

Ở Hình 4-13, kết quả của các mệnh đề được in ra ở text file chứa các thông báo của `uvm_info macro. Tên của mệnh đề Assertion và thời điểm kết quả mệnh đề được xem như thành công hay vi phạm cũng được hiển thị ở đầu thông báo.

Hình 4-13: Kết quả mệnh đề Concurrent Assertion

Quan sát kết quả dạng sóng tại thời điểm các kết quả mệnh đề Concurrent Assertion được xem như thành công, việc quan sát này dùng để kiểm tra hành vi của tín hiệu điều khiển và kết quả mệnh đề có khớp với nhau hay không thông qua Hình 4-14.



Hình 4-14: Waveform các tín hiệu điều khiển của DUT

Xét tín hiệu điều khiển adder_en, bias_en và kết quả mệnh đề Assertion của 2 tín hiệu này ở Hình 4-13 và Hình 4-14, có thể thấy mệnh đề Assertion của adder_en thành công ở hai thời điểm là 1530ns và 1750ns – một chu kỳ sau khi tín hiệu adder_en tích cực và sau khi buffer_valid tích cực 5 chu kỳ. Tương tự đối với tín

hiệu bias_en tích cực sau buffer_valid 15 chu kỳ, do đó mệnh đề Assertion của bias_en được xem như thành công tại thời điểm 1730ns – sau khi bias_en tích cực một chu kỳ.

4.5. Kết quả so sánh giữa kết quả thực tế và kết quả tin cậy (CNN IP ShuffleNetV2)

Sau khi tiến hành mô phỏng môi trường kiểm tra thiết kế CNN IP ShuffleNetV2 với khối CONV1 nhóm thu được kết quả như sau.

[CONV1] Comparison Report Channel 7

Number of Matches: 2294

Number of Mismatches: 10250

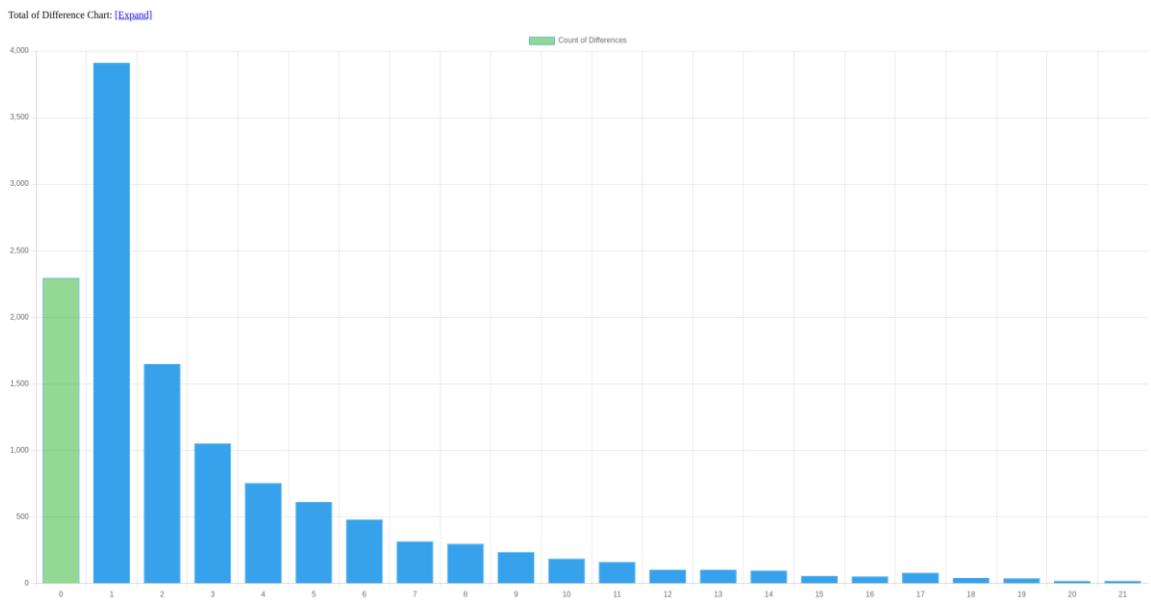
Average Mismatch Difference: 3.78

Comparison Detail: [Expand]

Pixel	DUT Result	Pytorch Result	Difference
1	0	14	-14
2	0	14	-14
3	0	14	-14
4	0	14	-14
5	13	14	-1
6	13	14	-1
7	13	13	0
8	13	14	-1
9	13	14	-1
10	13	13	0
11	13	14	-1
12	13	13	0
13	13	14	-1
14	13	14	-1
15	13	14	-1
16	13	14	-1
17	14	14	0
18	14	14	0
19	14	15	-1
20	14	13	1
21	14	14	0

Hình 4-15: Thông kê kết quả kiểm tra kênh 7

Hình 4-15 là một trong các kết quả so sánh đầu ra của DUT và đầu ra từ mô hình tin cậy. Khối CONV1 có 24 kênh đầu ra, do đó kết quả kiểm tra cần phải so sánh 24 kênh này với mô hình tin cậy từ Pytorch. Hình 4-15 là kết quả so sánh của kênh thứ bảy. Bên cạnh thông kê giá trị từng trường hợp giữa mô hình tin cậy và DUT, file HTML cũng bao gồm đồ thị thống kê số lượng trường hợp ứng với từng giá trị lệch().



Hình 4-16: Đồ thị thống kê kết quả kiểm tra kênh 7

Chương 5. TỔNG KẾT

5.1. Kết luận

Từ khi ra đời, UVM đã được chứng minh là một Framework mạnh mẽ, đóng vai trò thiết yếu cho người kỹ sư kiểm tra thiết kế, đặc biệt trong quá trình xây dựng một môi trường kiểm tra có tính nhất quán, dễ quản lý và hỗ trợ khả năng hiệu chỉnh và điều khiển môi trường. Điểm nổi bật của UVM đó là hỗ trợ tối đa về khả năng tự động hóa cũng như tái sử dụng môi trường thông qua các lớp và đa dạng phương thức tích hợp của thư viện cũng như các tiện ích được cung cấp bởi UVM Factory, nhờ đó hỗ trợ việc xây dựng một môi trường kiểm tra cho một thiết kế mạch số phức tạp trở nên hệ thống hoá hơn.

Qua quá trình tìm hiểu, nghiên cứu và thực hiện đề tài, nhóm đã hoàn thành việc xây dựng một môi trường kiểm tra dựa trên phương pháp và kiến trúc môi trường kiểm tra UVM, cùng với một số hiệu chỉnh và áp dụng các tính năng, cơ chế kiểm tra như SVA và DPI của SystemVerilog để hoàn thiện môi trường cho thiết kế CNN IP với các tính năng cụ thể. Qua quá trình tiến hành mô phỏng môi trường kiểm tra, môi trường kiểm tra thiết kế CNN IP được nhóm thiết kế đã đảm bảo được các tính năng cơ bản của từng thành phần bên trong, quá trình và kết quả mô phỏng của môi trường được nhóm điều khiển và theo dõi thông qua các Script tự động hóa môi trường và text file lưu quá trình thực hiện và kết quả mô phỏng.

Điểm mạnh của môi trường được nhóm xây dựng đó là khả năng kiểm tra hành vi của thiết kế trong quá trình hoạt động thông qua việc sử dụng cơ chế Assertion để kiểm tra các tín hiệu điều khiển. Việc ứng dụng tính năng Direct Programming Interface cũng hỗ trợ Testbench có khả năng giao tiếp với một lớp tính toán được viết bằng ngôn ngữ lập trình khác như C/C++ và Python, từ đó hỗ trợ việc xây dựng mô hình tin cậy có khả năng chạy trên nhiều nền tảng khác nhau.

5.2. Giới hạn đề tài

Trong quá trình thực hiện đề tài, nhóm gặp phải một số giới hạn bao gồm chưa đủ thời gian để nghiên cứu sâu về các tính năng cũng như kỹ thuật nâng cao trong việc xây dựng và hiệu chỉnh môi trường UVM, bao gồm kỹ thuật xây dựng nhiều Interface, Agent hoặc Environment khác bên trong UVM Testbench để hỗ trợ việc kiểm tra nhiều submodule lớn bên trong thiết kế cần kiểm tra; sử dụng config_db để hiệu chỉnh môi trường UVM Testbench. Nhóm cũng đang trong quá trình xây dựng nhiều Sequence khác nhau để kiểm tra thiết kế thay vì chỉ đang sử dụng Sequence có chức năng khởi tạo các giá trị ngẫu nhiên với ràng buộc “constraint”. Nhóm cũng chưa đủ thời gian và tài liệu nghiên cứu về cơ chế UVM Register Abstract Layer (RAL), còn được gọi là UVM Register Model, đây là cơ chế hỗ trợ người thiết kế môi trường kiểm tra thực hiện việc kiểm tra các phần tử nhớ bên trong DUT theo dạng xây dựng các lớp với các cơ chế truy cập bộ nhớ front-door và back-door, đây cũng chính là một trong những tính năng mạnh mẽ được cung cấp bởi UVM.

5.3. Hướng phát triển

Hiện tại trước khi bảo vệ khóa luận, nhóm vẫn đang nghiên cứu để phát triển hoàn chỉnh môi trường kiểm tra. Cụ thể nhóm đang thực hiện phát triển môi trường kiểm tra cho một mô hình CNN IP Shuffle Net V2 dựa trên môi trường của thiết kế Lenet5, để hiện thực điều này nhóm cần chỉnh sửa một số lớp để phù hợp với thiết kế mới bao gồm Interface, Sequence Item, Sequence, Driver, Monitor và Scoreboard. Tuy đã hoàn thành giai đoạn xây dựng các thành phần cơ bản của môi trường, nhóm vẫn đang nghiên cứu để xây dựng mô hình tin cậy cho Scoreboard của thiết kế Shuffle Net V2. Ở mô hình tin cậy này nhóm lựa chọn phương pháp kiểm tra black-box, từ mô tả đầu vào và đầu ra của thiết kế và một mô hình tin cậy có sẵn uy tín (Mô hình Pytorch ShuffleNet_v2_x0_5) được sự ủng hộ rộng rãi từ cộng đồng, nhóm sử dụng mô hình này cho quá trình kiểm tra kết quả của UVM Testbench, việc này giúp đảm bảo độ chính xác của quá trình kiểm tra chức năng mạch.

TÀI LIỆU THAM KHẢO

- [1] Accellera Systems Initiative (Accellera), “Universal Verification Methodology (UVM) 1.2 Class Reference”, June 2014
- [2] Accellera Systems Initiative (Accellera), “Universal Verification Methodology (UVM) 1.2 User’s Guide”, June 2014
- [3] “IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language”, IEEE Std 1800-2005
- [4] Chris Spear, “SystemVerilog for Verification: A Guide to Learning the Testbench Language”, 3rd Edition, Synopsys, Inc, 2012
- [5] N.Bheema Rao, Srikanth Konale, “C-based Predictor for Scoreboard in Universal Verification Methodology”, IEEE International Conference on Advances in Engineering & Technology Research (ICAETR – 2014), 2014
- [6] N B Harshitha, Praveen Kumar Y G, M Z Kurian, “An Introduction to Universal Verification Methodology for the digital design of Integrated circuits (IC’s); A Review”, 2021 International Conference on Artificial Intelligence and Smart Systems (ICAIS), 2021
- [7] Siemens Digital Industries Software, “Universal Verification Methodology – UVM Cookbook”, SEIMEN Verification Academy, 2021
- [8] Ando Ki, “Tutorial on DPI (Direct Programming Interface)”, <https://github.com>
- [9] Kumar Khandagle, “UVM for Verification”, <https://www.udemy.com>
- [10] Kumar Khandagle, “SystemVerilog Assertion (SVA) for Newbie”, <https://www.udemy.com>
- [11] David Rich, “Siemens Digital Industries Software - The Missing Link: The Testbench to DUT Connection”, SEIMEN Verification Academy, 2018

- [12] Dr. A. K. Kureshi, “UVM Architecture for Verification”, International Journal of Electronics and Communication Engineering & Technology (IJECE), 2016
- [13] Trần Đăng Hậu, Lê Quang phong, “Nghiên cứu và hiện thực IP ShuffleNet”, Khóa luận Tốt nghiệp Khoa Kỹ thuật Máy tính Học Kỳ 2, 2024
- [14] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, Jian Sun, “ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design”, Jul 2018
- [15] Pytorch Team, “ShuffleNetV2 - An efficient ConvNet optimized for speed and memory, pre-trained on ImageNet”,
https://pytorch.org/hub/pytorch_vision_shufflenet_v2