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 NƠ-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 NƠ-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 TRƯƠNG VĂN CƯƠNG

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 ……
ngày …….. của Hiệu trưởng Trường Đại học Công nghệ Thông tin.

LỜI CẢM ƠN

MỤC LỤC

DANH MỤC HÌNH

DANH MỤC BẢNG

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

TÓM TẮT KHÓA LUẬN

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 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 biết 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 TestbenchQUAN, hỗ trợ người kiểm tra thiết kế có thể tối ưu tính tự động hoá 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 than [5], đồng thời được chuẩn hoá 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 hoá 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 hoá 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

Thực hiện 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. Mục tiêu hướng tới chính là môi trường thiết kế 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 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 SystemVerilogvà 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. Đồng thời, khả năng tái sử dụng của môi trường mô phỏng cũng được nhóm hướng tới thông qua việc sử dụng DPI-C 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++.

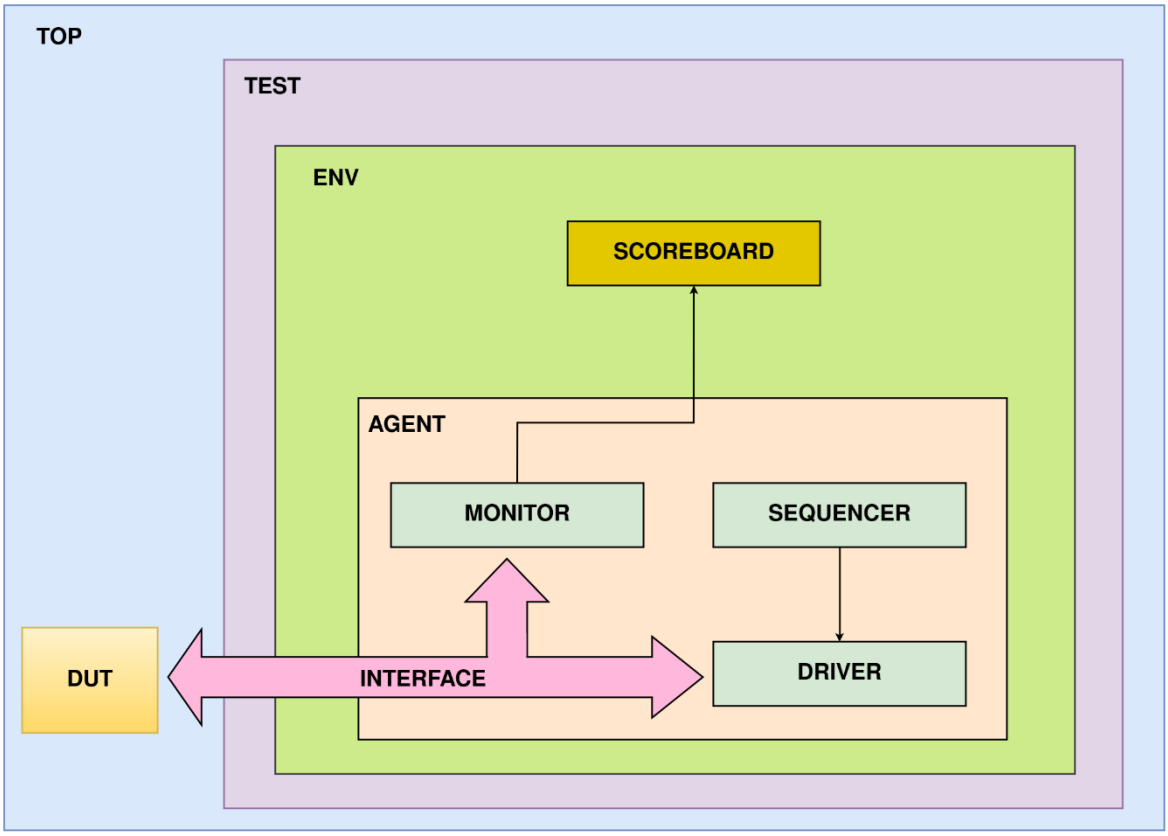
Chương 2. **MÔ HÌNH THIẾT KẾ**

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 công cụ mô phỏng EDA 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 UVM Testbench cơ bản có các thành phần như sau:



Hình 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:
    - Instantiate top-level environment
    - Thực hiện các tùy chỉnh trong môi trường (thông qua factory override hoặc 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 1 SoC design environment chứa PCIe environment, USE environment, Mem Controller environment.

- **UVM Scoreboard**
  - UVM Scoreboard có chức năng kiểm tra hành vi của DUT, UVM Scoareboard nhận transaction chứa các input và output của DUT thông qua Agent's Analysis Port, đưa input qua Referencđếne 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 stimulus được đưa vào DUT, các stimulus này được sinh ra bởi một hoặc các UVM Sequence khác nhau
- **UVM Sequence**
  - UVM Sequence có chức năng tạo ra các stimulus khác nhau.
- **UVM Driver**
  - UVM Driver nhận các gói tin UVM Sequence Item từ UVM Sequencer và lái chúng vào DUT Interface. Driver cũng có một

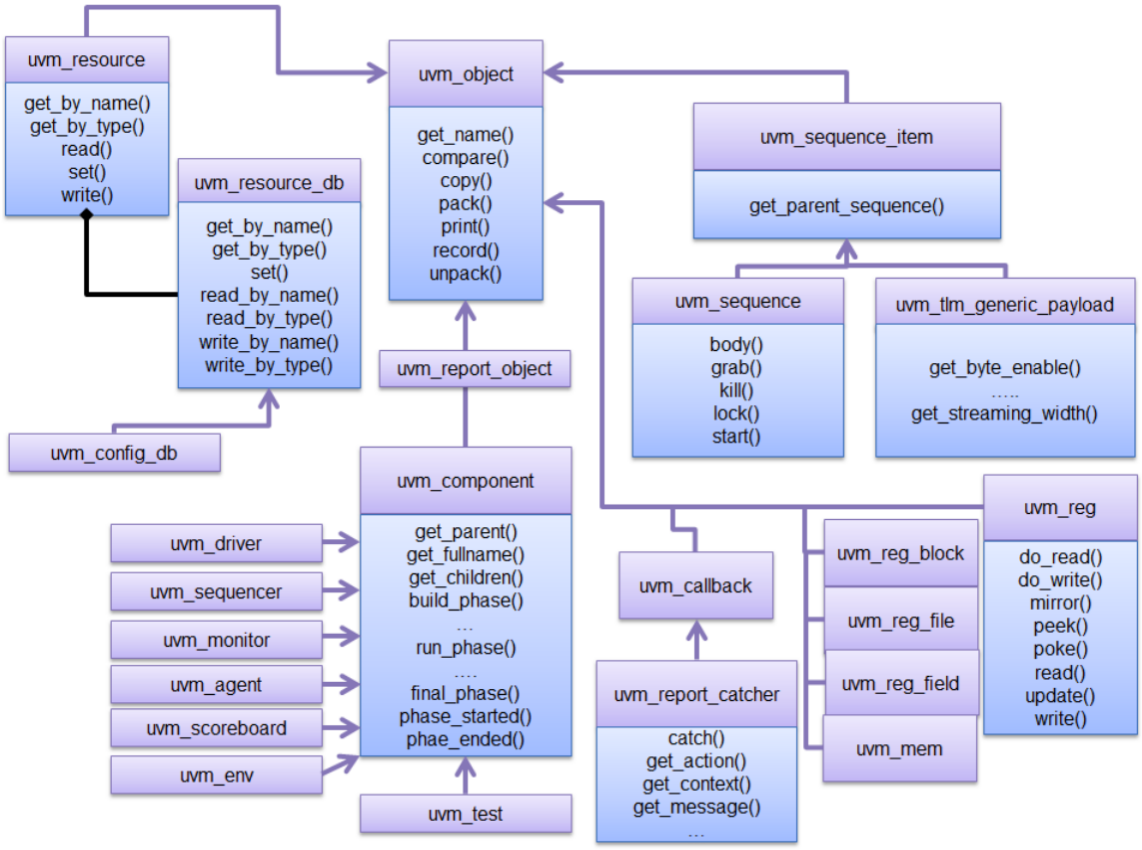
TLM port để nhận các gói tin từ Sequencer và có quyền truy xuất vào Interface để lái các tín hiệu

- **UVM Monitor**
    - UVM Monitor lấy mẫu từ DUT Interface, đóng gói các thông tin thành transaction để chuyển các transaction đó tới các thành phần khác. Tương tự như Driver, Monitor cũng cần có quyền truy xuất trực tiếp tới DUT Interface để lấy mẫu và có một TLM analysis port để broadcast các transaction được khởi tạo từ Monitor.

Quan sát Hình 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 một 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 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 ở lớp Driver và Monitor).

Các lớp trên được cung cấp bởi thư viện tích hợp bên trong UVM, cùng với các phương thức đi kèm riêng 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.

Hình 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.



Hình 2: Mô hình các thành phần bên trong thư viện UVM (Trang 4 uvm_1.2_user_guide)

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(), các test phase, các phương thức Factory 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 1 và Hình 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.

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 theo 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 xuyên 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 1.

uvm_object	uvm_transaction uvm_sequence_item uvm_sequence
uvm_component	uvm_driver uvm_sequencer uvm_monitor uvm_agent uvm_scoreboard uvm_env uvm_test

Bảng 1: Các lớp con của uvm_object và uvm_component

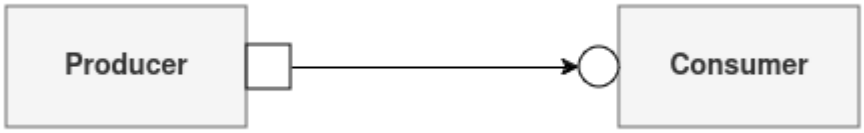
2.1.2. Transaction-Level Modeling

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. Nguyên tắc hoạt động của UVM được trừu 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.

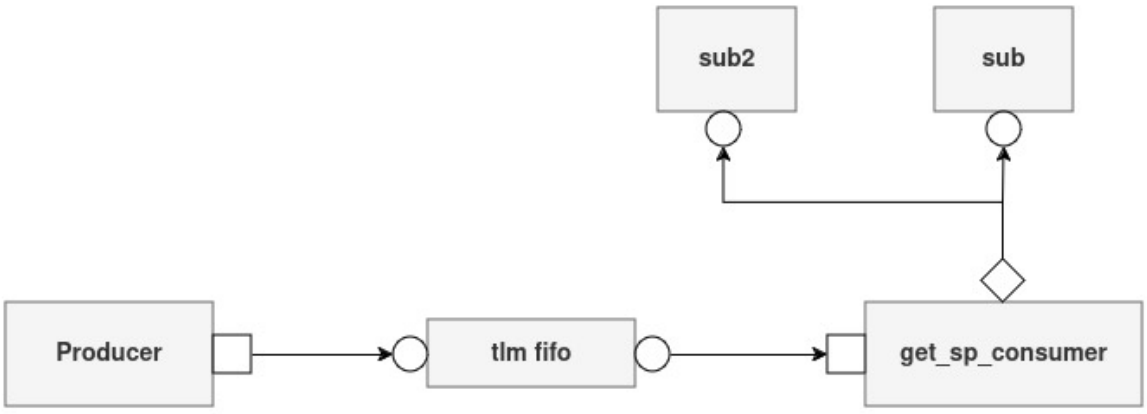
TLM được cung cấp bởi thư viện tích hợp UVM và chứa 3 phần tử chính:

- 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 3: Port và Export của Producer và Consumer

Hình 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 4: Mô hình Port, Export và Analysis Port

Hình 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 của Analysis Port được cung cấp bởi thư viện UVM.

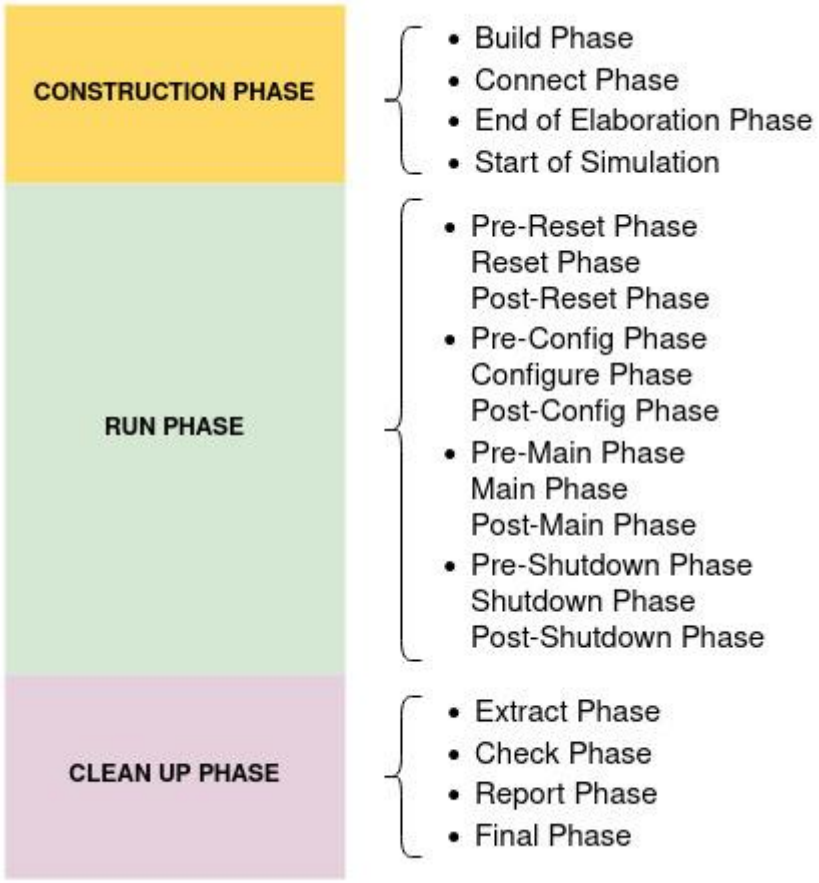
2.1.3. Các Phase bên trong quá trình mô phỏng sử dụng UVM Testbench

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. 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 5 liệt kê các phase được thực thi trong quá trình mô phỏng của một UVM Testbench. Các phase được chia thành 3 giai đoạn chính là Construction Phase, Run Phase và cuối cùng là Clean Up Phase.



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

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

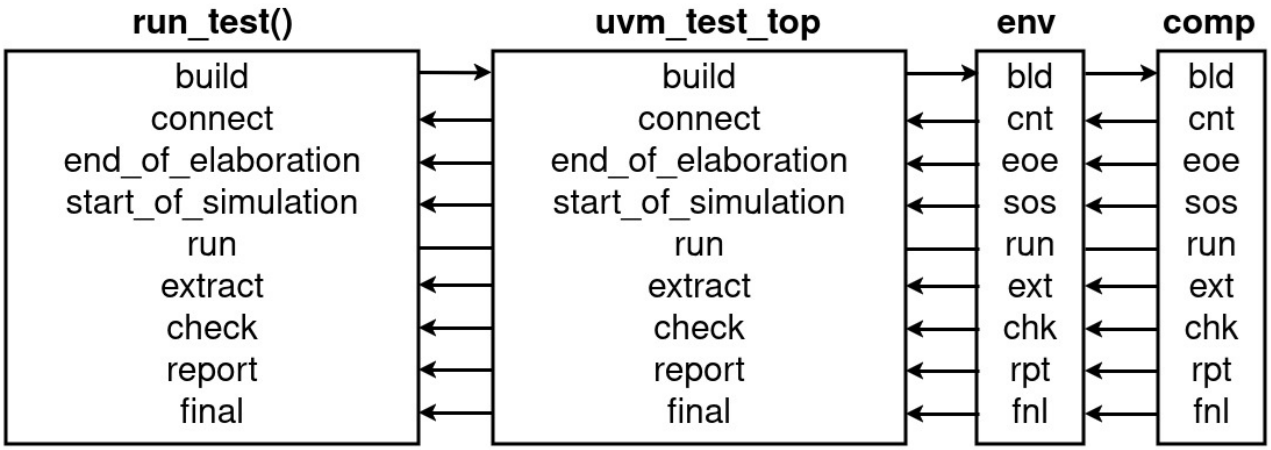
- **Build Phase** là giai đoạn các thành phần của môi trường kiểm tra, cụ thể là các đối tượng được khởi tạo từ các lớp mà người kiểm tra đã định nghĩa từ lớp cha của thư viện UVM Testbench. Các thành phần cần thiết cho môi trường kiểm tra được hoàn tất việc xây dựng ở Build Phase.
- **Connect Phase** 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 nhớ 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 các phần tử phụ thuộc vào cây gia phả trong môi trường kiểm tra UVM, được thể hiện qua Hình 6.



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

Quan sát Hình 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 builld 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 tiền, 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

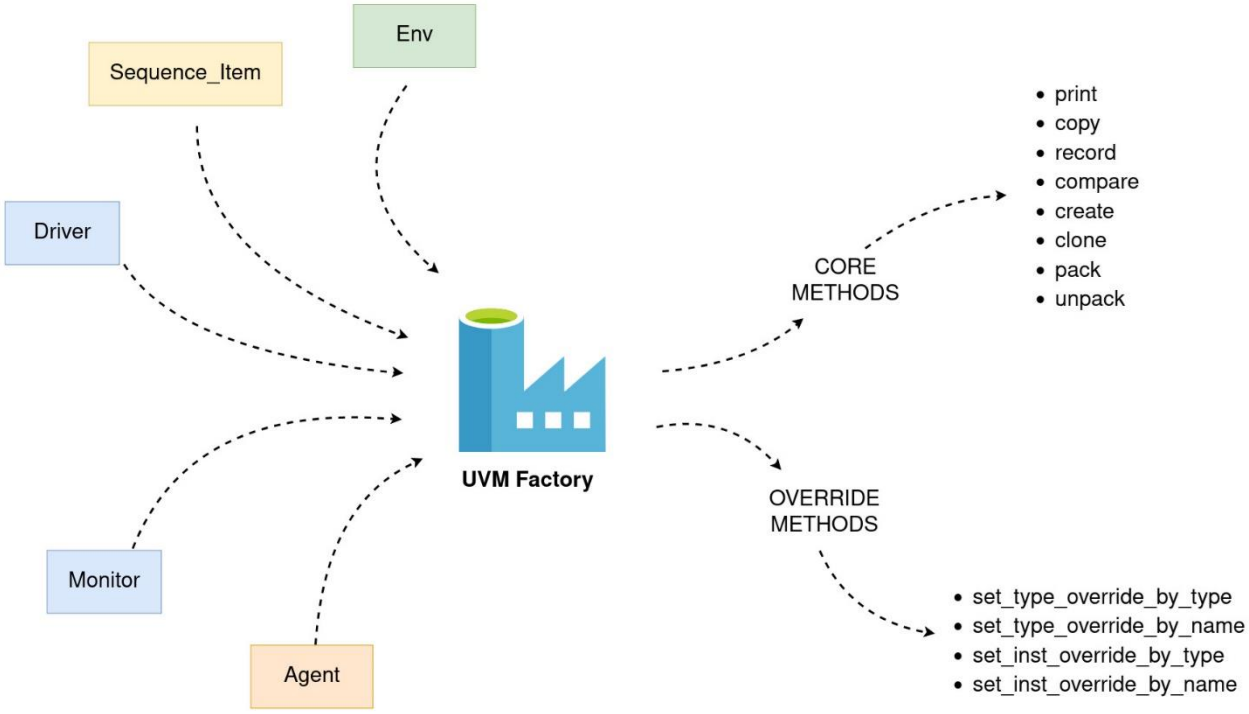
Các phase còn lại ngoại trừ run phase, có thứ tự thực hiện bottom-up. Ngược lại với build phase, các phần tử nhỏ nhất trong cây gia phả được thực hiện trước và uvm_test_top được thực hiện cuối cùng.

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 được cung cấp bởi UVM

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 trượng 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 tới Factory với kiểu dữ liệu cụ thể. Việc đăng ký các lớp tới Factory giúp người thiết kế có thể thực hiện việc 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 như Hình 7.

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.



Hình 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 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.

Utility and Field Macros for Components and Objects	
UTILITY MACROS	The *utils* macros define the infrastructure needed to enable the object/component for correct factory operation.
`uvm_field_utils_begin	
`uvm_field_utils_end	These macros form a block in which `uvm_field_*

	macros can be placed.
`UVM_FIELD_* MACROS	Macros that implement data operations for scalar properties.
`uvm_field_int	Implements the data operations for any packed integral property.
`uvm_field_object	Implements the data operations for a uvm_object-based property.
`uvm_field_string	Implements the data operations for a string property.
`uvm_field_enum	Implements the data operations for an enumerated property.
`uvm_field_real	Implements the data operations for any real property.
`uvm_field_event	Implements the data operations for an event property.

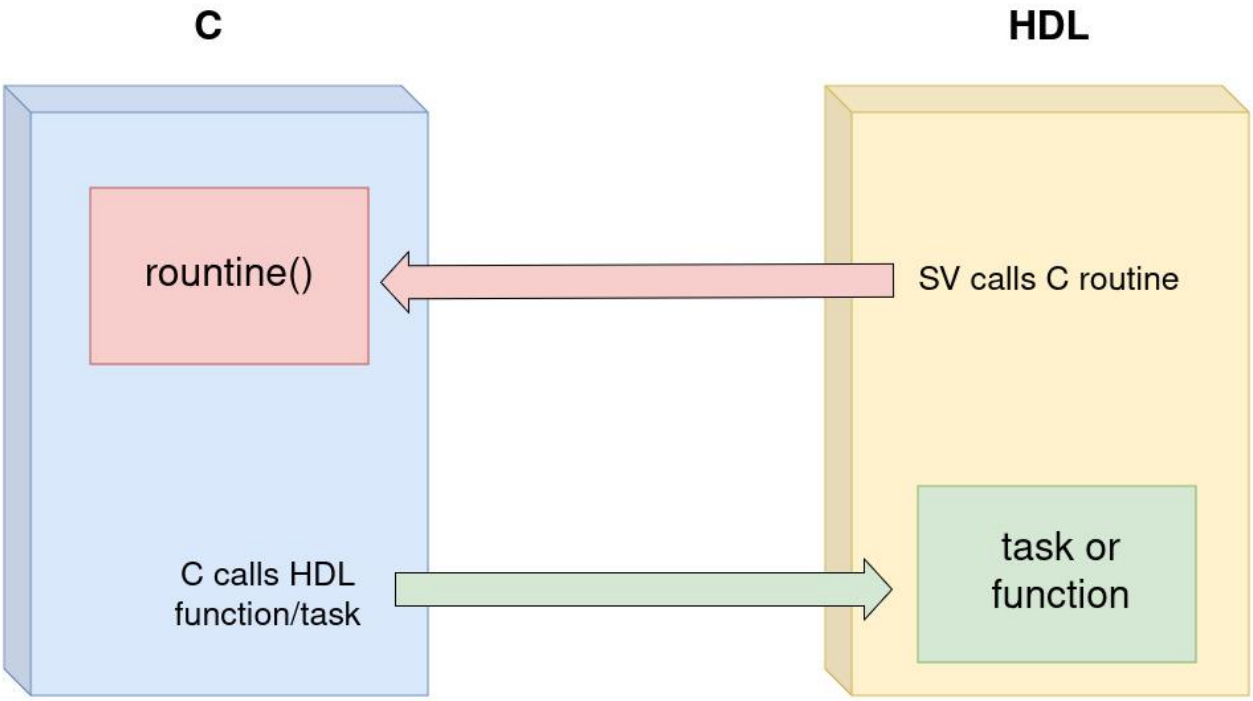
Bảng 2: Một số Field Macro thông dụng cho các kiểu dữ liệu cơ bản

Bảng 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

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 code vào C code hoặc lái bất kỳ tín hiệu nào đến SystemVerilog code từ C code (Hình 8).

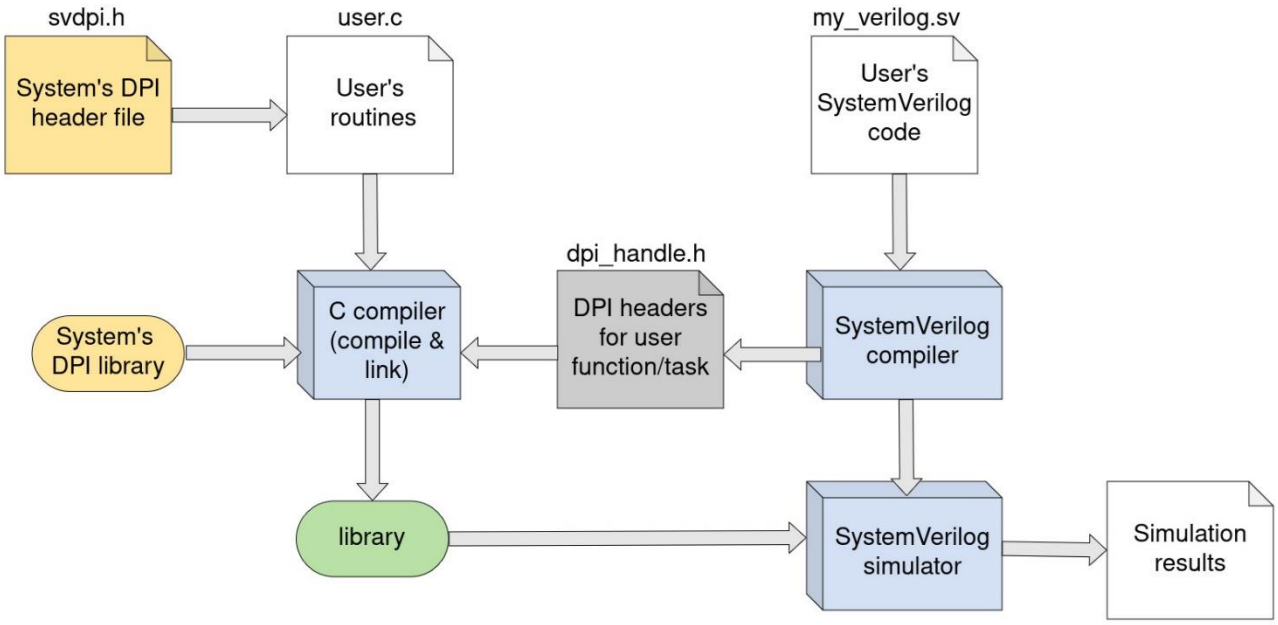


Hình 8: Giao tiếp giữa SystemVerilog và C

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 9).



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

Việc truyền các biến giữa SystemVerilog và C được thực hiện bởi thư viện svdpi.h với các biến tham chiếu như sau:

SystemVerilog	*C (input)*	*C (output)*
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 10: Tham chiếu kiểu dữ liệu giữa SystemVerilog và C sử dụng thư viện svdpi.h

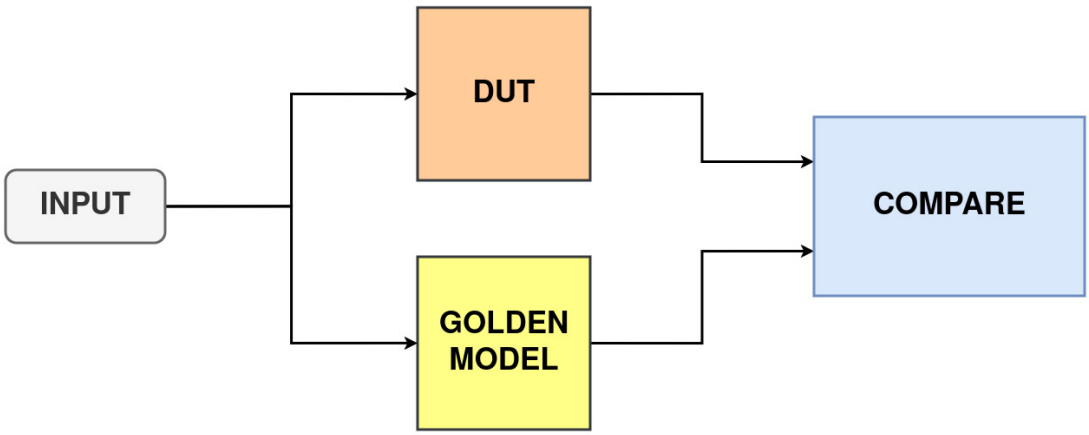
Hình 10 lấy từ trang 418 sách SystemVerilog for Verification 3rd Edition Chris Spear, cho thấy các kiểu dữ liệu biến phổ biến như int, longint giữ nguyên khi truyền dữ liệu giữa SystemVerilog. Tuy nhiên một số kiểu dữ liệu thông dụng của SystemVerilog như kiểu dữ liệu bit được chuyển thành svBit hoặc unsigned char* ở C, kiểu dữ liệu logic cũng được chuyển thành svLogic hoặc unsigned char* ở 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 điểm mạnh này của DPI, nhóm quyết định xây dựng một mô hình tin cậy 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 và Scoreboard là thành phần 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 và 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 DUT (Hình 11).



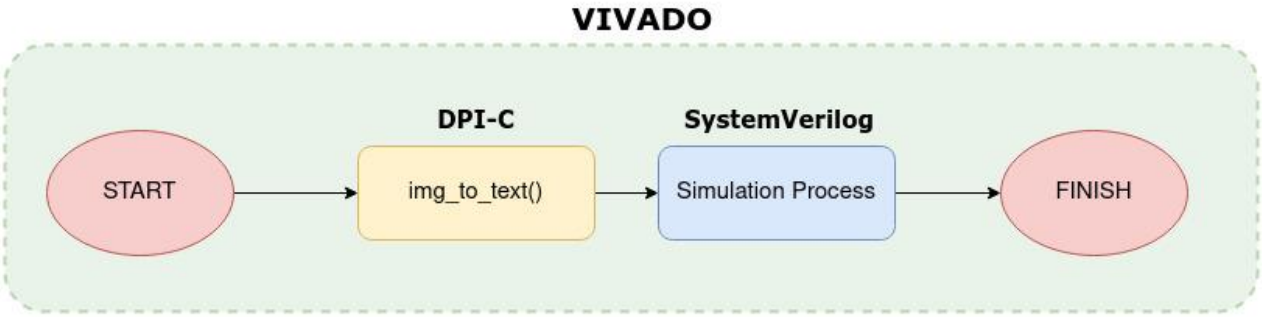
Hình 11: Mô hình tin cậy kiểm tra kết quả DUT

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 chạy được ở nhiều môi trường khác nhau. Đồng thời việc xây dựng mô hình tin cậy bằng SystemVerilog có nhược điểm đó là bị trùng lặp với quá trình thiết kế.

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 giao 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 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 12: Trình tự hoạt động của DPI-C và SystemVerilog trong quá trình mô phỏng

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

2.3.1. Tổng quan về 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.

Assertion là một đoạn code thường được dùng để kiểm tra hành vi của một thiết kế, đoạn code Assertion 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 code Assertion, 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
```

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 13).

```
Error: Assertion violation
Time: 5 ns  Iteration: 0  Process: /SVA_example/Always11_1  File: /
Error: Assertion violation
Time: 15 ns  Iteration: 0  Process: /SVA_example/Always11_1  File:
Error: Assertion violation
Time: 20 ns  Iteration: 0  Process: /SVA_example/Always14_2  File:
Error: Assertion violation
Time: 30 ns  Iteration: 0  Process: /SVA_example/Always14_2  File:
Error: Assertion violation
Time: 45 ns  Iteration: 0  Process: /SVA_example/Always11_1  File:
Error: Assertion violation
Time: 50 ns  Iteration: 0  Process: /SVA_example/Always14_2  File:
Error: Assertion violation
Time: 65 ns  Iteration: 0  Process: /SVA_example/Always11_1  File:
Error: Assertion violation
Time: 75 ns  Iteration: 0  Process: /SVA_example/Always11_1  File:
Error: Assertion violation
Time: 85 ns  Iteration: 0  Process: /SVA_example/Always11_1  File:
Error: Assertion violation
Time: 95 ns  Iteration: 0  Process: /SVA_example/Always11_1  File:
Error: Assertion violation
Time: 100 ns  Iteration: 0  Process: /SVA_example/Always14_2  File:
```

Hình 13: Kết quả Assertion violation

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:

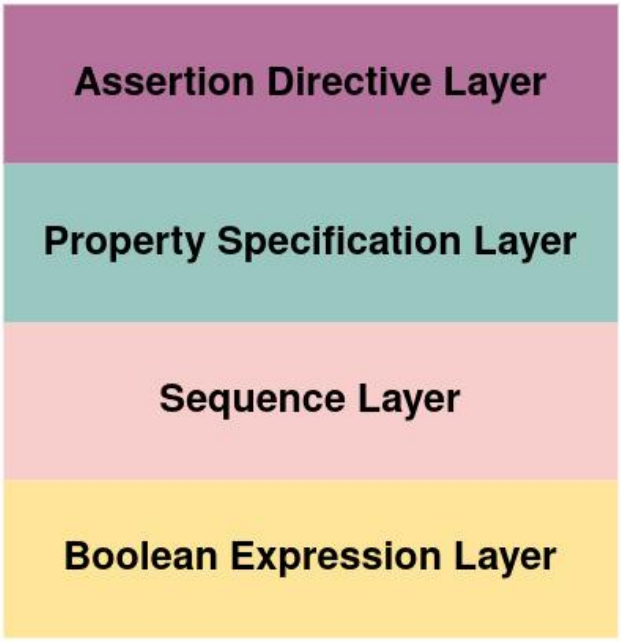
```
assert property (@(posedge clk) req |-> ##[1:2] ack)
```

Đoạn assertion bên trên là một Concurrent Assertion, mệnh đề Assertion có nghĩa tại mỗi cạnh lên xung clk, nếu tín hiệu req đang ở mức cao thì kiểm tra trong vòng một hoặc hai cạnh lên 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 sẽ được thông báo bởi trình mô phỏng.

Concurrent 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 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 phổ 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:

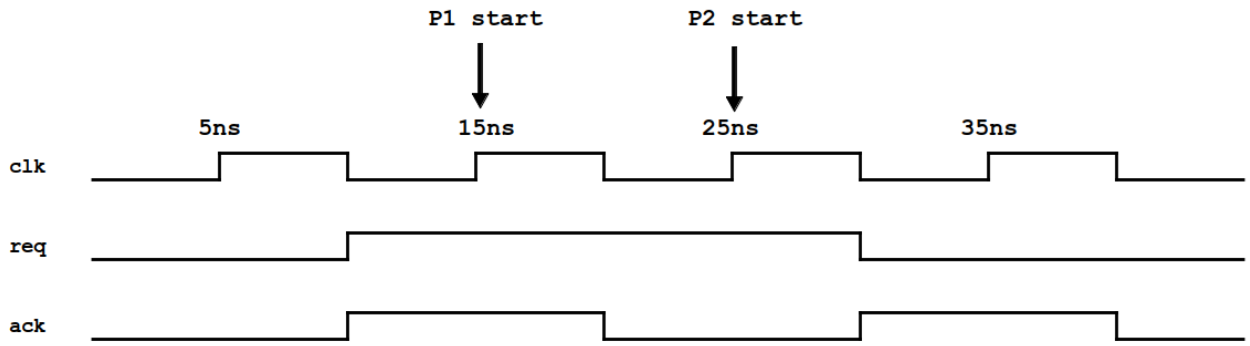
- **Overlapping:** là toán tử ngầm định được dùng để xác định tiền đề và kết quả có cùng thỏa trùng nhau ngay tại cạnh lên / cạnh xuống xung clock hay không
- **Non-overlapping:** là toán tử ngầm định được dùng để xác định tiền đề và kết quả có cùng thỏa tại hai cạnh lên / cạnh xuống xung clock hay không,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

Đối với dạng sóng có dạng như Hình 15



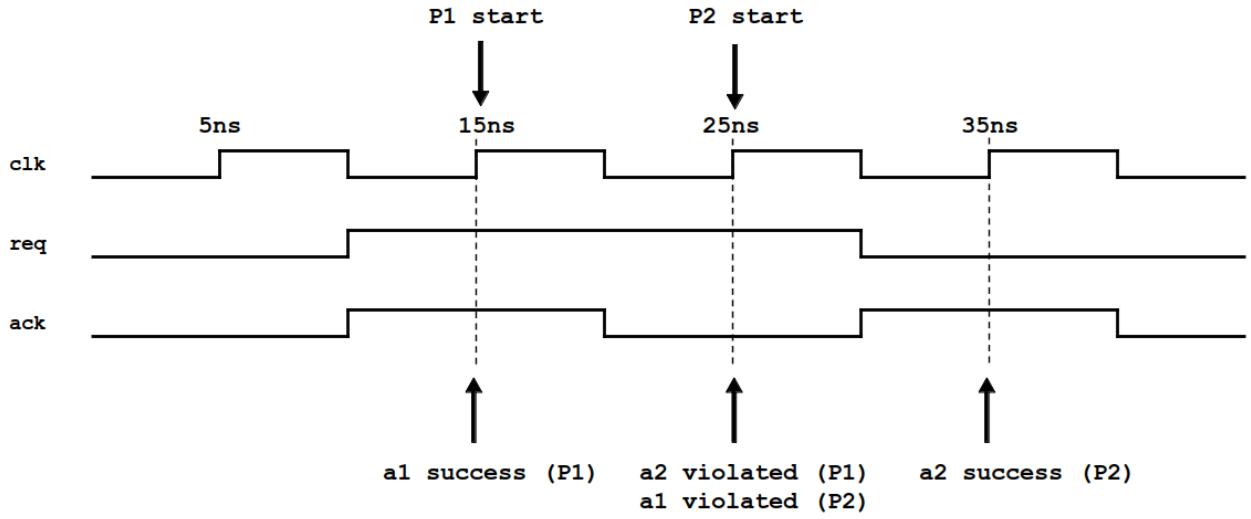
Hình 15: Waveform mẫu cho đoạn code bên trên

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 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 điể 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 16).



Hình 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 lên 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 lên 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].**

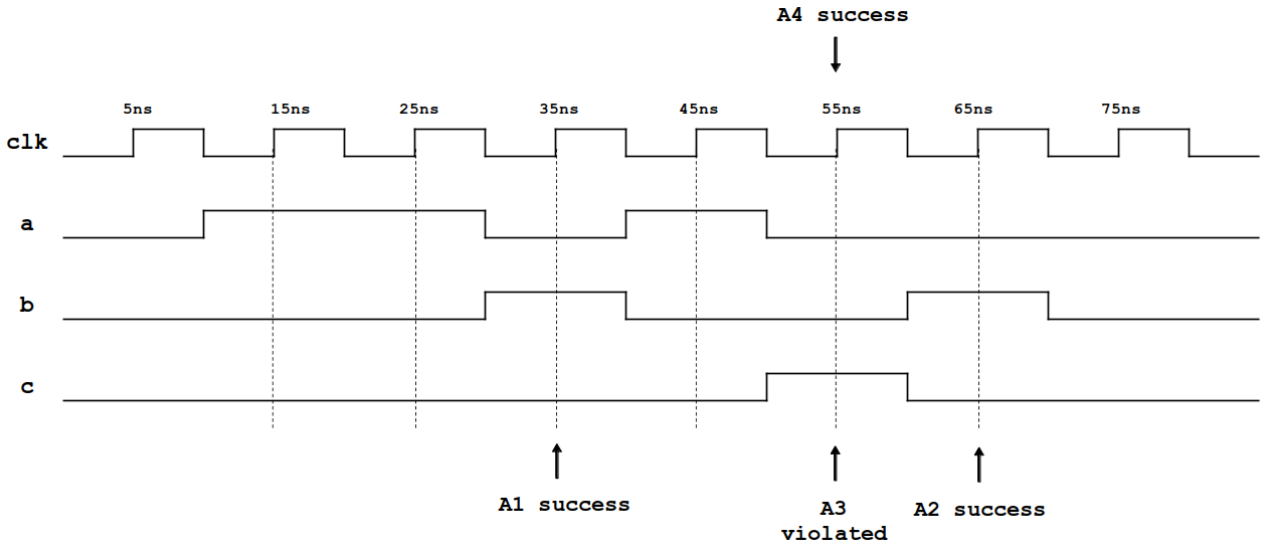
- Toán tử **[*n]** là toán tử lặp liên tiếp dùng để xác định một tín hiệu ở mức cao / mức thấp liên tiếp trong n chu kỳ xung clock.
- Toán tử **[=n]** và **[->n]** là toán tử lặp không liên tiếp được dùng để xác định một tín hiệu ở mức cao mức / mức thấp trong n chu kỳ xung clock. Đối với toán tử này, 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 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 mệnh đề 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.

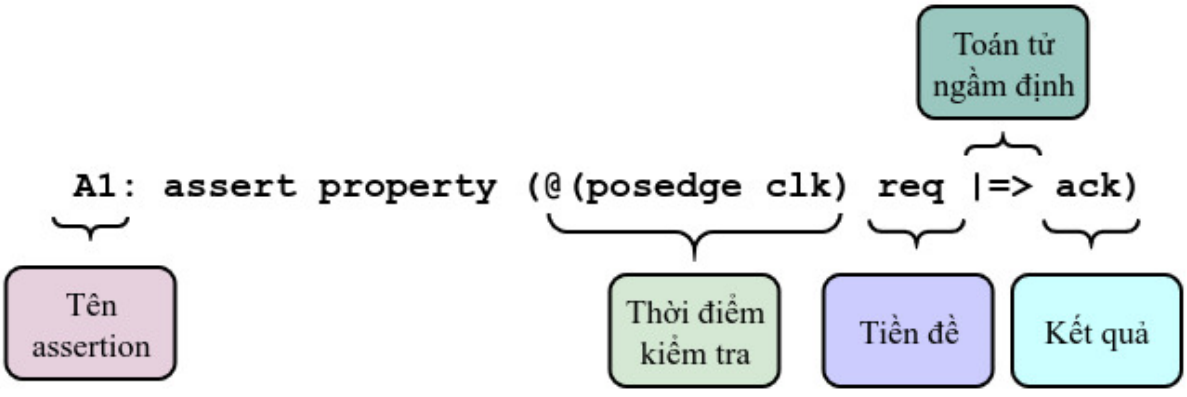
Tương tự ngữ nghĩa mệnh đề A3, mệnh đề A4 kiểm tra tại 55ns tín hiệu c có ở mức cao hay không. Do tín hiệu c mức cao tại cạnh lên xung clock kế tiếp sau khi thỏa tiền đề, mệnh đề A4 được xem như thành công.

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 18: Cú pháp của một mệnh đề Concurrent Assertion

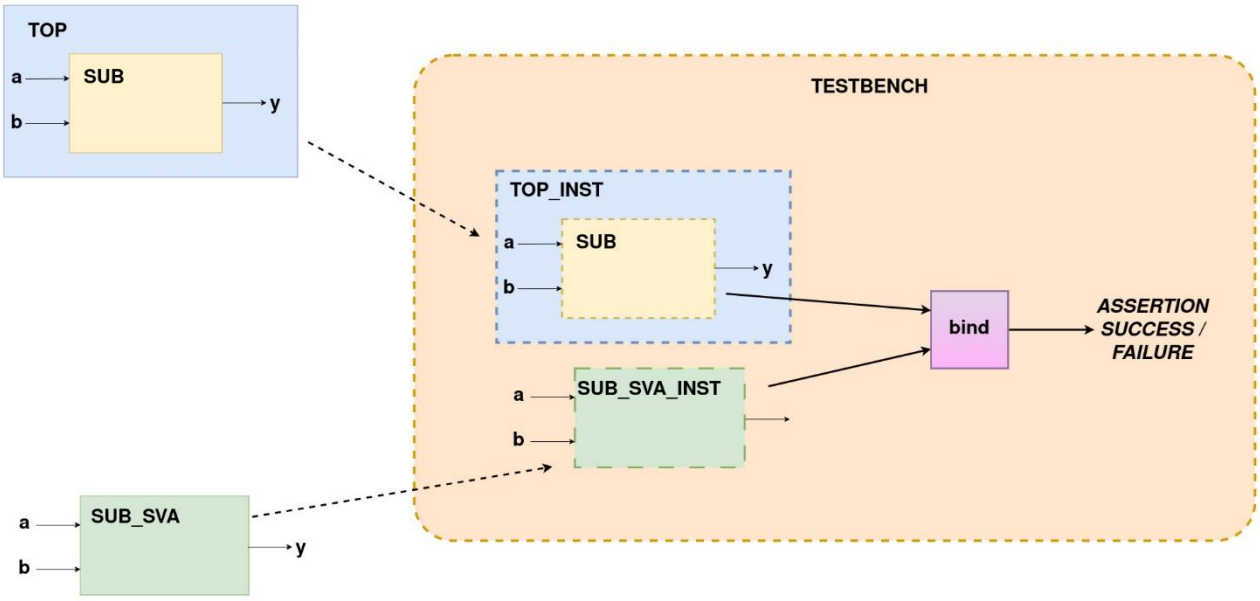
- **Tên Assertion** là tên của mệnh đề Concurrent Assertion, mỗi Assertion phải có một tên riêng
- **Thời điểm kiểm tra** là thời điểm mệnh đề được kiểm tra, thông thường cạnh lên hoặc cạnh xuống xung clock sẽ được chọn làm thời điểm kiểm tra
- **Tiền đề** là một chuỗi hành vi và là điều kiện kiểm tra 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.



Hình 19: Cấu trúc bind của SystemVerilog

Ở Hình 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.

Chương 3. **TÊN CHƯƠNG 3**

3.1. Chủ đề cấp độ 2

Nội dung ……

Nội dung…….

3.1.1. Chủ đề cấp độ 3

3.1.1.1. Chủ đề cấp độ 4

3.2. Chủ đề cấp độ 2

TÀI LIỆU THAM KHẢO

Theo chuẩn IEEE