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

NGUYỄN QUỐC TRƯỜNG AN
DƯƠNG HOÀNG TUẤN

KHÓA LUẬN TỐT NGHIỆP

THIẾT KẾ BỘ ĐIỀU KHIỂN ĐỒNG BỘ CACHE CHO HỆ THỐNG ĐA VI XỬ LÝ DỰA TRÊN LÕI CPU RISC-V RV32I

Designing a cache coherence controller for multiprocessor systems based on RISC-V RV32I core

CỬ NHÂN NGÀNH KỸ THUẬT MÁY TÍNH

TP. HỒ CHÍ MINH, 2025

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

NGUYỄN QUỐC TRƯỜNG AN – 21521810
DƯƠNG HOÀNG TUẤN – 21522750

KHÓA LUẬN TỐT NGHIỆP

THIẾT KẾ BỘ ĐIỀU KHIỂN ĐỒNG BỘ CACHE CHO HỆ THỐNG ĐA VI XỬ LÝ DỰA TRÊN LÕI CPU RISC-V RV32I

Designing a cache coherence controller for multiprocessor systems based on RISC-V RV32I core

CỬ NHÂN NGÀNH KỸ THUẬT MÁY TÍNH

GIẢNG VIÊN HƯỚNG DẪN
ThS. PHẠM MINH QUÂN
KS. TRẦN ĐẠI DƯƠNG

TP. HỒ CHÍ MINH, 2025

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

Lời đầu tiên, nhóm chúng em, gồm Nguyễn Quốc Trường An và Dương Hoàng Tuấn xin gửi lời cảm ơn chân thành đến thầy hướng dẫn là ThS. Phạm Minh Quân và KS. Trần Đại Dương. Thầy đã tận tình dành thời gian quý báu để hỗ trợ, góp ý và chỉnh sửa cho nhóm trong suốt thời gian hoàn thành đề tài.

Bên cạnh đó, nhóm chúng em cũng xin chân thành cảm ơn thầy/cô, bạn bè khoa Kỹ thuật Máy tính nói riêng và trường Đại học Công nghệ Thông tin – Đại học quốc gia Thành phố Hồ Chí Minh nói chung đã tận tình hỗ trợ, giảng dạy những kiến thức bổ ích để nhóm có thể hoàn thành đề tài một cách tốt đẹp.

Trong suốt quá trình thực hiện đề tài, nhóm không thể tránh khỏi những khó khăn, thách thức cũng như thiếu sót do giới hạn về mặt thời gian và kiến thức chuyên sâu. Vì thế mong quý thầy/cô và các bạn thông cảm, và đóng góp ý kiến để đề tài này được phát triển và hoàn thiện hơn trong tương lai.

Một lần nữa nhóm xin chân thành cảm ơn.

TP. Hồ Chí Minh, ngày 13 tháng 06 năm 2025

Sinh viên thực hiện

Nguyễn Quốc Trường An Dương Hoàng Tuấn

MỤC LỤC

DANH MỤC HÌNH

DANH MỤC BẢNG

DANH MỤC TỪ VIẾT TẮT

Viết tắt	Viết đầy đủ	Ý nghĩa
CPU	Central Processing Unit	Bộ xử lý trung tâm
MSI	Modified, Shared, Invalid	Trạng thái Chỉnh sửa - Chia sẻ - Không hợp lệ
MESI	Modified, Exclusive, Shared, Invalid	Trạng thái Chỉnh sửa - Độc quyền - Chia sẻ - Không hợp lệ
MOESI	Modified, Owned, Exclusive, Shared, Invalid	Trạng thái Chỉnh sửa - Sở hữu - Độc quyền - Chia sẻ-Không hợp lệ
PLRUt	Tree-based Pseudo Least Recently Used	Thuật toán "Ít được sử dụng nhất dựa vào cây nhị phân"
RAM	Random-Access Memory	Bộ nhớ truy cập ngẫu nhiên
AXI	Advanced eXtensible Interface	Giao diện mở rộng nâng cao
ACE	AXI Coherency Extensions	Mở rộng tính nhất quán AXI
RISC	Reduced Instruction Set Computer	Máy tính với tập lệnh đơn giản hóa
FPGA	Field-Programmable Gate Array	Vi mạch tích hợp cỡ lớn dùng cấu trúc mảng phần tử logic mà người dùng có thể lập trình được
I-Cache	Instruction Cache	Bộ nhớ đệm lệnh
D-Cache	Data Cache	Bộ nhớ đệm dữ liệu
FCFS	First-Come First-Served	Giải thuật đến trước phục vụ trước
RR	Round-Robin	Giải thuật luân phiên

TÓM TẮT KHÓA LUẬN

Nội dung chính của khóa luận tập trung vào việc thiết kế bộ nhớ đệm theo kiến trúc Multiway Set Associative với các chức năng hoàn chỉnh của một bộ nhớ đệm, sau đó tích hợp bộ điều khiển sử dụng giao thức Cache Coherence MOESI Snoopy hỗ trợ cho hệ thống đa lõi gồm 2 lõi xử lý RISC-V RV32I.

Đầu tiên nhóm sẽ nghiên cứu cấu trúc, cách tổ chức và hoạt động của kiến trúc Multiway Set Associative Cache sau đó thực hiện thiết kế hoàn chỉnh với ngôn ngữ Verilog, đồng thời cài đặt giải thuật thay thế PLRUt, chính sách Write-back và chính sách cấp phát Read/Write Allocate. Tiếp đến là thiết kế trình đồng nhất dữ liệu giữa các bộ nhớ dữ liệu (D-Cache) của các bộ xử lý trong hệ thống với giao thức Cache Coherence MOESI Snoopy, sử dụng kênh giao tiếp chung AXI Bus mở rộng ACE. Bên cạnh đó nhóm cũng thực hiện thiết kế lõi xử lý theo kiến trúc RISC-V RV32I pipeline 5 tầng, có tích hợp I-Cache, D-Cache và AXI wrapper để phục vụ cho việc thực thi chương trình hiệu quả.

Sau khi hoàn thiện thiết kế phần cứng, nhóm sử dụng ngôn ngữ Python tiến hành thiết kế phần mềm kiểm thử là hệ thống đa vi xử lý, bao gồm: 2 CPU RISC-V RV32I, 2 I-Cache, 2 D-Cache, AXI Bus mở rộng ACE và bộ nhớ chính (Main Memory). Tiếp đến tiến hành tích hợp mô phỏng phần cứng và phần mềm, đánh giá hiệu năng và tính chính xác của thiết kế một cách khách quan.

Công việc sau cùng là hiện thực thiết kế trên FPGA với kit Xilinx Virtex-7 VC707 và tiến hành đánh giá hiệu năng hoạt động của thiết kế.

CHƯƠNG 1. GIỚI THIỆU TỔNG QUAN ĐỀ TÀI

1.1. Tổng quan đề tài

Giải pháp lưu trữ vào bộ nhớ đệm (hay còn gọi là Caching) là giải pháp hiệu quả trong việc sử dụng bộ nhớ đệm để lưu trữ dữ liệu có khả năng sử dụng trong tương lai nhằm giảm thiểu tối đa số lượng truy cập bộ nhớ chính, từ đó có thể tăng tốc độ truy cập trung bình vào bộ nhớ của toàn hệ thống.

Việc bộ nhớ được tổ chức phân cấp và kiến trúc bộ xử lý bên trên sử dụng đa luồng, đa lõi dẫn đến việc chia sẻ bộ nhớ giữa các luồng hoặc lõi của vi xử lý. Vấn đề phát sinh khi thực hiện chia sẻ bộ nhớ đệm đó là sự đảm bảo tính nhất quán của hệ thống bộ nhớ.

Việc ghi dữ liệu vào vùng nhớ chia sẻ chung giữa các bộ xử lý làm nảy sinh một vấn đề nghiêm trọng trong hệ thống đa lõi, góc nhìn của bộ nhớ (Memory View) lên bản sao của dữ liệu được giữ bởi các lõi (nằm trong bộ nhớ đệm của các lõi xử lý) sẽ khác nhau vì không theo dõi được toàn bộ các tương tác hay truy cập dữ liệu đó. Nếu không có biện pháp xử lý nào thì 2 lõi có thể thấy 2 giá trị khác nhau của cùng một vị trí bộ nhớ. Bảng 1.1 minh họa vấn đề và trình bày cách 2 lõi khác nhau có thể nhận 2 giá trị khác nhau cho cùng một vị trí trong bộ nhớ. Vấn đề này được gọi là nhất quán bộ nhớ đệm (Cache Coherence).

Bảng 1.1: Minh họa dữ liệu trong hệ thống đa lõi không hỗ trợ Cache Coherence

Bước	Sự kiện	Nội dung ô nhớ X tại Cache A	Nội dung ô nhớ X tại Cache B	Nội dung ô nhớ X tại Bộ nhớ chính
0				0
1	Lõi A đọc dữ liệu tại X	0		0
2	Lõi B đọc dữ liệu tại X	0	0	0
3	Lõi A ghi “1” vào X	1	0	0

Dựa vào Bảng 1.1, vấn đề cụ thể như sau: giả sử ban đầu các bộ nhớ đệm của cả 2 lõi rỗng và giá trị vùng nhớ tại vị trí X là 0. Sử dụng chính sách Write-through trong trường hợp này để minh họa. Dễ dàng nhận thấy sau khi lõi A ghi giá trị vào X, bộ nhớ đệm của A và bộ nhớ đều chứa giá trị mới, trong khi đó bộ nhớ đệm B vẫn chứa giá trị cũ, và nếu B đọc giá trị của X thì sẽ nhận được giá trị cũ là 0.

Để duy trì tính chính xác của dữ liệu cũng như sự nhất quán trong hệ thống bộ nhớ, giữa các bộ nhớ đệm trong hệ thống đa lõi sử dụng bộ nhớ chia sẻ chung cần có các giao thức để thực hiện việc này, các giao thức này được gọi là giao thức đảm bảo nhất quán bộ nhớ đệm - Cache Coherence. Có hai loại giao thức phổ biến là "Directory-based" và "Snoopy". Directory-based thích hợp dùng cho hệ thống với hàng trăm bộ xử lý kết nối với nhau thành mạng lưới. Snoopy sử dụng dựa trên hệ thống Bus của bộ nhớ và thích hợp cho hệ thống với lõi chia sẻ chung bộ nhớ có số lượng bộ xử lý vừa và nhỏ.

Có 2 cách tiếp cận chính để thực hiện giao thức Snoopy là Write-update và Write-invalidate, các giải thuật phổ biến thường dùng để hiện thực giao thức Snoopy có thể kể tên như: MSI (Modified, Shared, Invalid), MESI (Modified, Exclusive, Shared, Invalid) và MOESI (Modified, Owned, Exclusive, Shared, Invalid).

1.2. Lý do thực hiện đề tài

Trong kiến trúc máy tính ngày nay thì xu hướng kiến trúc bộ xử lý song song, bộ xử lý đa lõi chia sẻ dùng chung bộ nhớ ngày càng nổi trội. Nhưng CPU tốn rất nhiều chu kỳ để truy cập bộ nhớ chính trong kiến trúc đa lõi do sự chênh lệch về tốc độ giữa bộ xử lý và bộ nhớ. Vì vậy, để tối ưu hiệu suất thực thi của kiến trúc đa lõi, các bộ nhớ đệm được sử dụng trong hệ thống nhằm giảm thời gian truy cập bộ nhớ của bộ xử lý. Tuy nhiên, bất chấp tất cả những ưu điểm này, vấn đề Cache Coherence vẫn nảy sinh khi truy cập dữ liệu, chỉnh sửa dữ liệu nằm trong vùng bộ nhớ chia sẻ chung giữa các bộ xử lý. Do vậy, giải quyết được vấn đề khi chia sẻ bộ

nhớ dùng chung với bộ nhớ đệm trong hệ thống đa lõi sẽ giúp tốc độ xử lý cũng như hiệu suất của hệ thống được cải thiện đáng kể.

Giao thức MOESI Snoopy nổi trội lên trong số các giao thức Cache Coherence Snoopy phổ biến có thể giúp cải thiện đáng kể trong việc giảm tỉ lệ Cache Miss, giảm độ trễ khi truy cập bộ nhớ, giảm mức tiêu thụ điện năng, và tăng tốc độ xử lý của tổng thể hệ thống. Giao thức MOESI Snoopy giúp đảm bảo tính nhất quán của dữ liệu và chính xác của toàn bộ hệ thống bộ nhớ.

1.3. Khảo sát các đề tài đã thực hiện

Thông qua quá trình tìm kiếm thông tin và khảo sát, nhóm tìm thấy hiện tại có các công trình nghiên cứu và khóa luận tốt nghiệp nổi bật thực hiện nội dung liên quan đến việc nghiên cứu thiết kế bộ nhớ đệm (hay còn gọi là Cache).

Năm 2021, sinh viên Nguyễn Thế Đạt đã thực hiện thiết kế bộ điều khiển Cache 2 mức [1]. Thiết kế này hoạt động ở tốc độ khá cao tuy nhiên không hỗ trợ Cache Coherence và tỉ lệ truy cập thất bại khá cao do sử dụng giải thuật FIFO (First-In First-Out). Bộ nhớ chính được sử dụng khi đánh giá thiết kế cũng còn đơn giản, chưa đáp ứng về các tham số định thời thực tế.

Năm 2022, nhóm sinh viên Trần Quốc Trường và Lê Phước Nhật Nam đã thực hiện thiết kế bộ xử lý RISC-V 32-bit sử dụng kiến trúc Superscalar tích hợp bộ điều khiển 4-Way Set Associative Cache [2]. Tuy nhiên mục tiêu chính của đề tài này là thiết kế bộ xử lý RISC-V 32-bit nên không chú trọng thiết kế bộ nhớ đệm, dẫn đến thiết kế bộ nhớ đệm chưa được tối ưu và không hỗ trợ Cache Coherence.

Năm 2023, nhóm sinh viên Cao Thanh Bình và Đặng Phi Hùng đã thiết kế vi xử lý RISC-V RV32IF hỗ trợ 4-Way Set Associative Cache và bộ tạo số ngẫu nhiên trên FPGA [3]. Đề tài này hoạt động ở tần số khá cao, tuy nhiên không tập trung thiết kế bộ nhớ đệm.

Cùng năm 2023, nhóm sinh viên Trần Tuấn Khanh và Phạm Thanh Lâm đã thực hiện thiết kế bộ xử lý RISC-V có hỗ trợ 4-Way Set Associative Cache [4]. Tuy

nhiên mục tiêu chính của đề tài này là thiết kế bộ xử lý RV64IM theo kiến trúc Superscalar nên không chú trọng việc thiết kế bộ nhớ đệm hoàn chỉnh và cũng không hỗ trợ Cache Coherence.

Tất cả các đề tài nhóm liệt kê phía trên đều có thực hiện thiết kế bộ nhớ đệm tuy nhiên đều có nhược điểm chung là không hỗ trợ Cache Coherence và thiết kế chưa được tối ưu. Riêng đối với đề tài của sinh viên Nguyễn Thế Đạt, bộ nhớ đệm được thiết kế khá đầy đủ so với các nhóm còn lại. Để giải quyết nhược điểm này, nhóm cải tiến kết quả đề tài của sinh viên Nguyễn Thế Đạt và tích hợp thêm tính năng Cache Coherence cho hệ thống gồm hai lõi. Cùng với đó, nhóm cũng nghiên cứu tích hợp giao thức AXI với mở rộng ACE sử dụng giao thức Cache Coherence MOESI Snoopy vào thiết kế bộ nhớ đệm, thiết kế mô hình kiểm thử của hệ thống đa vi xử lý tích hợp cả phần mềm và phần cứng có khả năng tự động xác minh kết quả nhằm tối ưu quá trình kiểm tra và đánh giá.

Sau quá trình khảo sát, cân nhắc thời gian và quy mô thực hiện, nhóm quyết định sẽ thực hiện thiết kế bộ nhớ đệm với đầy đủ các tính năng:

- Multiway Set Associative Cache.
- Giải thuật thay thế PLRUt.
- Hỗ trợ 2-Core RISC-V RV32I.
- Chính sách Write-back.
- Chính sách Read/Write Allocate.
- Giao thức đảm bảo Cache Coherence MOESI Protocol.

1.4. Mục tiêu của đề tài

Từ thực tiễn phát triển và ứng dụng rộng rãi của hệ thống đa vi xử lý và nhu cầu cấp bách giải quyết vấn đề Cache Coherence đã đặt ra, khóa luận đặt mục tiêu tổng quan thiết kế bộ điều khiển Cache Coherence sử dụng giao thức MOESI Snoopy để giải quyết được các vấn đề về Cache Coherence trong hệ thống đa vi xử lý.

Kết quả đề tài có thể mô phỏng được trên phần mềm Vivado và thu được kết quả mong muốn, tối ưu được tốc độ, hiệu suất và khắc phục được các hạn chế của các thiết kế tham khảo sử dụng các giao thức Snoopy khác.

Mục tiêu chi tiết của đề tài được liệt kê cụ thể như sau:

- Thiết kế bộ nhớ đệm theo kiến trúc Multiway Set Associative với 4-Way và 16-Set riêng biệt cho từng lõi.
- Thiết kế bộ điều khiển Cache Coherence sử dụng giao thức Cache Coherence MOESI Snoopy trong bộ nhớ đệm, hỗ trợ cho hai lõi chạy cùng một chương trình với nhiều luồng (Multi-thread).
- Thiết kế lõi xử lý theo kiến trúc RISC-V RV32I được pipeline 5 tầng và có tích hợp AXI wrapper.
- Thiết kế phần cứng bằng ngôn ngữ Verilog với hệ thống đa vi xử lý, gồm: 2 CPU RISC-V RV32I, 2 I-Cache, 2 D-Cache, AXI Bus mở rộng ACE và bộ nhớ chính (Main Memory) được tích hợp là IP BRAM của Xilinx.
- Thiết kế mô hình kiểm thử bằng phần mềm với ngôn ngữ Python có khả năng tính toán cho ra kết quả mong đợi với hệ thống đa vi xử lý, gồm: 2 CPU RISC-V RV32I, 2 I-Cache, 2 D-Cache, AXI Bus mở rộng ACE và bộ nhớ chính.
- Thiết kế trình tự động hóa đồng xác minh kết quả giữa phần cứng và phần mềm.
- Hiện thực thiết kế trên FPGA với kit Xilinx Virtex-7 VC707 sau đó đánh giá độ chính xác và hiệu năng của thiết kế.

1.5. Thách thức của đề tài

Để thực hiện đề tài cần nắm rõ cách thức hoạt động của các cơ chế như Write-back, Write-through, Write-update hay Write-invalidate. Cần nắm rõ sự phân cấp trong việc tổ chức bộ nhớ, cách mà các cấp bộ nhớ giao tiếp với nhau, tổ chức và các hoạt động của hệ thống đa vi xử lý.

Để thực hiện thiết kế trên một hệ thống bộ nhớ với nhiều bộ nhớ đệm một cách hoàn chỉnh, cần nắm rõ cách cài đặt các cấp bộ nhớ, đồng thời còn phải đảm bảo các chức năng như bảo mật, cấp quyền cho hệ điều hành khi truy cập bộ nhớ đệm, ngăn chặn các truy cập trái phép, cài đặt giải thuật thay thế (Cache Replacement Algorithm)…

Để thực hiện đánh giá hiệu suất và tính chính xác của hệ thống, cần xây dựng và kết nối nhiều thành phần với nhau, đòi hỏi kiến thức toàn diện về hệ thống đa vi xử lý và cách xử lý các xung đột nếu có khi thực thi chương trình.

1.6. Giới hạn của đề tài

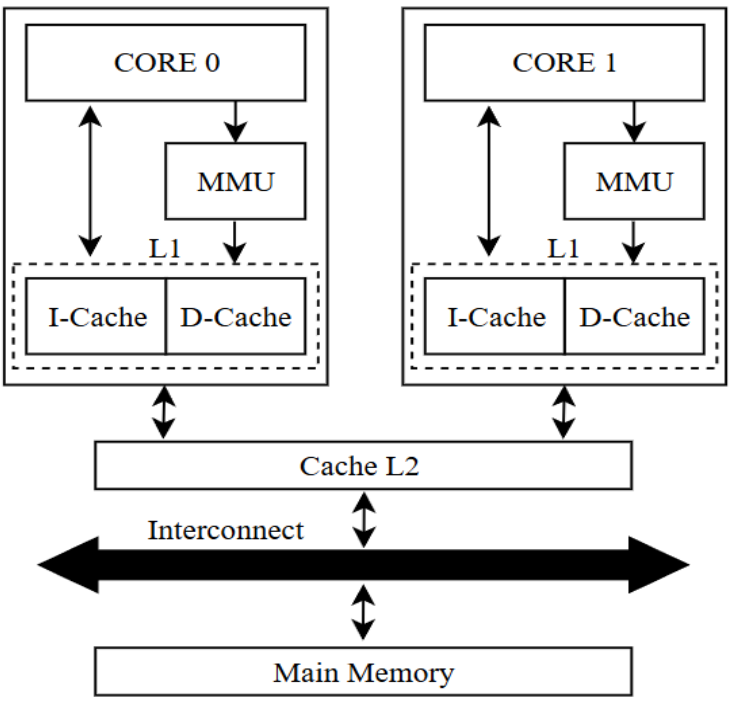
Đề tài hiện dừng ở mức thực hiện giao thức Cache Coherence đảm bảo đồng nhất dữ liệu giữa các bộ nhớ đệm và chỉ hỗ trợ cho hệ thống đa vi xử lý gồm 2 lõi.

Để tạo thành một thiết kế hệ thống hoàn chỉnh, cần thiết kế thêm nhiều thành phần khác như đơn vị quản lý bộ nhớ MMU (Memory Management Unit), bộ truy cập bộ nhớ trực tiếp DMA (Direct Memory Access), thiết lập các cơ chế ngắt (Interrupt) ...

CHƯƠNG 2. CƠ SỞ LÝ THUYẾT

2.1. Khái niệm và tổng quan về bộ nhớ đệm

Bộ nhớ đệm là bộ nhớ lưu trữ tạm thời các lệnh và dữ liệu được sử dụng thường xuyên để bộ xử lý trung tâm (hay còn gọi là CPU) của hệ thống có thể truy cập nhanh hơn. Trong các vi xử lý hiện đại, thông thường bộ nhớ đệm được chia thành hai loại chính: I-Cache (Instruction Cache) và D-Cache (Data Cache). I-Cache lưu trữ các lệnh của chương trình để CPU có thể truy xuất nhanh hơn khi thực hiện các thao tác điều khiển, còn D-Cache lưu trữ dữ liệu mà chương trình đang xử lý nhằm giảm độ trễ khi đọc/ghi dữ liệu từ bộ nhớ chính. Việc tách biệt I-Cache và D-Cache giúp CPU thực hiện song song việc nạp lệnh và xử lý dữ liệu, từ đó tối ưu hóa hiệu suất tổng thể của hệ thống. Vị trí bộ nhớ đệm trong hệ thống đa lõi thể hiện trong Hình 2.1.



Hình 2.1: Ví dụ mô tả vị trí của bộ nhớ đệm trong hệ thống đa vi xử lý

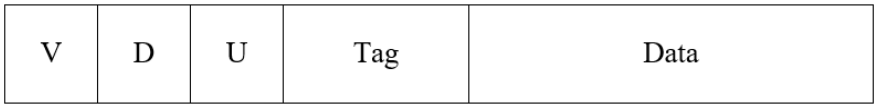
Thông thường, bộ nhớ đệm (Cache) sẽ nằm giữa đơn vị quản lý bộ nhớ (MMU) và bộ nhớ chính (Main Memory). Địa chỉ ảo được dịch thành địa chỉ vật lý thông qua khối MMU, sau đó địa chỉ vật lý sẽ được đưa vào truy cập bộ nhớ đệm,

nếu dữ liệu CPU cần có trong bộ nhớ đệm, ta gọi là Cache Hit, ngược lại là Cache Miss. Khi một truy cập được cho là miss ở bộ nhớ đệm Ln, CPU sẽ tiếp tục tìm kiếm dữ liệu cần ở bộ nhớ đệm Ln+1, tương tự cho đến bộ nhớ chính và sau cùng là bộ nhớ thứ cấp (SSD, Hard Drive).

2.1.1. Các trường trong bộ nhớ đệm và trường địa chỉ

Các trường của một bộ nhớ đệm tổng quát bao gồm các trường:

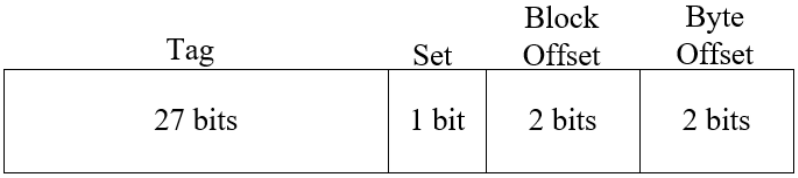
- Tag: dùng để so sánh với trường Tag trong địa chỉ vật lý để xác định xem dữ liệu tại Block đó có ứng với địa chỉ vật lý đó hay không.
- Bit V (Valid): chỉ định dữ liệu tại Block này có tồn tại hay không.
- Data: chứa dữ liệu cần ghi đệm.
- Ngoài ra còn các bit khác như bit Dirty (D), bit Usage (U) [5] … để phục vụ cho việc Write-back và chạy các giải thuật thay thế.



Hình 2.2: Các trường trong một Cache Block [5]

Được thể hiện rõ trong Hình 2.2, để truy cập vào bộ nhớ đệm, địa chỉ vật lý được chia thành các trường địa chỉ:

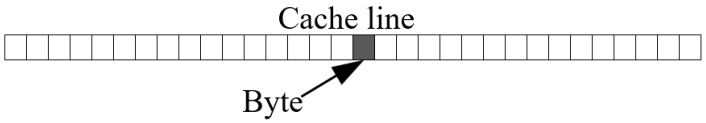
- Tag: dùng để so sánh với trường Tag trong bộ nhớ đệm để xác định xem dữ liệu tại Block đó có ứng với địa chỉ hiện tại hay không.
- Set: dùng để xác định địa chỉ vật lý ánh xạ với Set nào trong bộ nhớ đệm.
- Block Offset: xác định địa chỉ ánh xạ với word nào trong Cache Block.
- Byte Offset: xác định byte dữ liệu cần trong một word, nếu địa chỉ vật lý truy cập theo word, trường này sẽ luôn bằng “00”.



Hình 2.3: Địa chỉ 32-bit với Direct Mapped Cache 2-Set và 4-word Block [5]

2.1.2. Cache Line size

Trong thực tế, độ rộng của một Cache Line (Cache Block) là 64-byte liền kề nhau như Hình 2.4. Kích thước Cache Line 64-byte phổ biến trong các CPU Intel/AMD. Đối với kiến trúc 32-bit, một Cache Line sẽ chứa 16-word (16x32 bit), còn đối với kiến trúc 64-bit, một Cache Line chứa 8-word (8x64 bit).



Hình 2.4: Minh họa một Cache Line

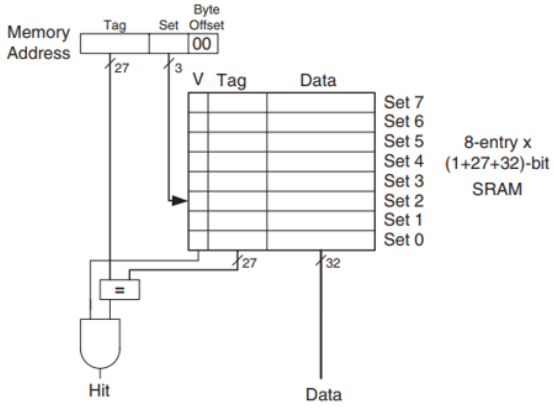
2.2. Phân loại bộ nhớ đệm

Nếu dựa vào mục đích sử dụng thì ta có Data Cache (D-Cache) và Instruction Cache (I-Cache). Còn dựa vào kiến trúc, ta có 3 loại kiến trúc bộ nhớ đệm chính:

- Direct Mapped Cache.
- Multiway Set Associative Cache.
- Fully Associative Cache.

2.2.1. Direct Mapped Cache

Direct Mapped Cache có cấu tạo mỗi Set chỉ chứa duy nhất một Block, nên mỗi địa chỉ vật lý trong bộ nhớ chính ánh xạ với duy nhất một Block trong bộ nhớ đệm Hình 2.5.

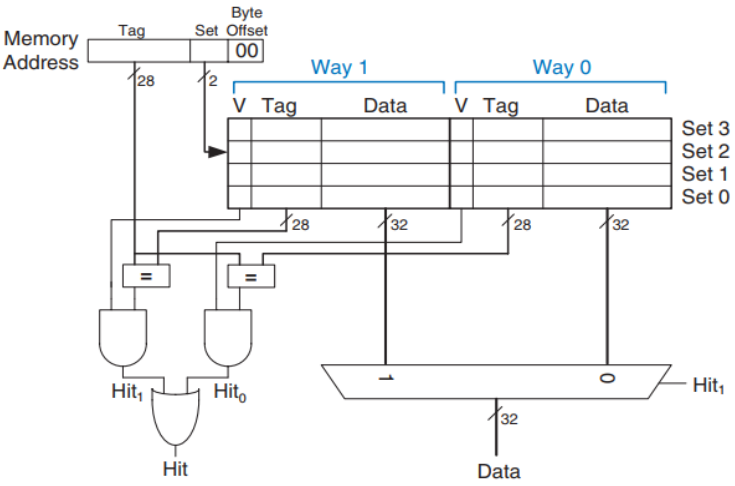


Hình 2.5: Direct Mapped Cache với 8-Set [5]

Ưu điểm của Direct Mapped Cache là đơn giản, dễ hiện thực, chi phí thấp, tốc độ truy cập nhanh, chỉ cần một phép so sánh tag. Tuy nhiên, Direct Mapped Cache cũng có nhược điểm không nhỏ là dễ gây ra xung đột (Cache Block Conflict).

2.2.2. Multiway Set Associative

Multiway Set Associative Cache có cấu tạo mỗi Set bao gồm N Cache Block, địa chỉ vật lý ánh xạ với một Set duy nhất trong bộ nhớ đệm, nhưng dữ liệu tại địa chỉ vật lý này có thể nằm trong một trong các Block thuộc Set này, minh họa trong Hình 2.6.

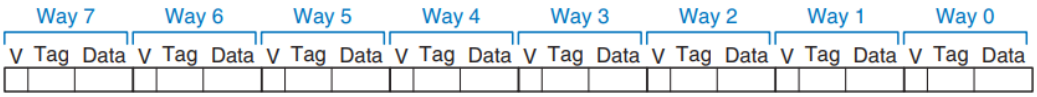


Hình 2.6: Two-Way Set Associative Cache [5]

Ưu điểm của Multiway Set Associative Cache là có tỉ lệ miss thấp hơn so với Direct Mapped Cache (So sánh khi cùng kích thước bộ nhớ đệm).

2.2.3. Fully Associative Cache

Fully Associative Cache có cấu trúc gồm một Set duy nhất (S = 1) và bao gồm nhiều Block, dữ liệu có thể nằm trong 1 trong các Block trên Set duy nhất, kiến trúc thể hiện trong Hình 2.7.



Hình 2.7: Fully Associative Cache với 8 Way [5]

Fully Associative Cache có ưu điểm là tỉ lệ miss thấp nhất trong 3 loại bộ nhớ đệm do giảm thiểu khả năng gây xung đột. Tuy nhiên nó có nhược điểm là chi phí thiết kế cao do yêu cầu phần cứng nhiều hơn để phục vụ cho việc so sánh nhiều Tag nên chỉ thích hợp cho các bộ nhớ đệm có dung lượng nhỏ và tốc độ truy cập chậm do phải so sánh nhiều.

2.3. Lý do lựa chọn 4-Way Set Associative

Để đưa ra một thiết kế tối ưu cho hệ thống cần so sánh và cân nhắc nhiều yếu tố, sau đây sẽ trình bày lý do đề tài chọn kiến trúc 4-Way Set Associative Cache.

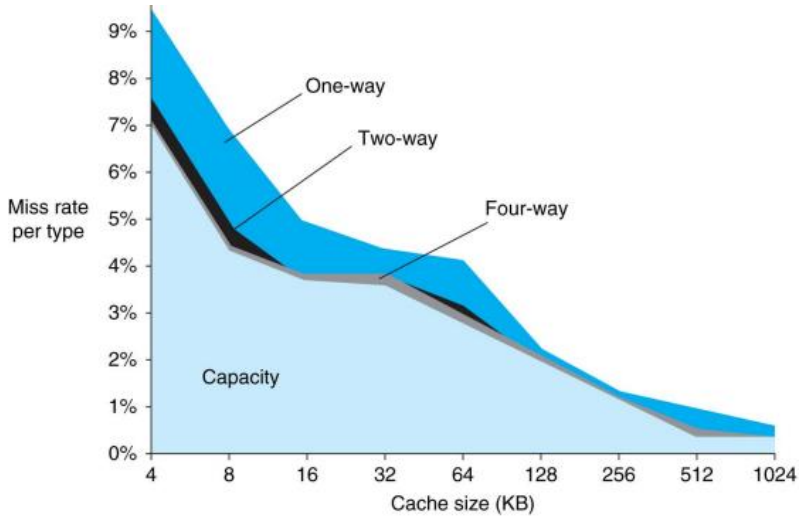
2.3.1. Phương pháp cải thiện hiệu suất Cache

Tỉ lệ Cache Miss có thể được giảm xuống bằng cách thay đổi Capacity (C), Block Size (b) và Associativity (N). Ta phân Cache Miss thành ba loại:

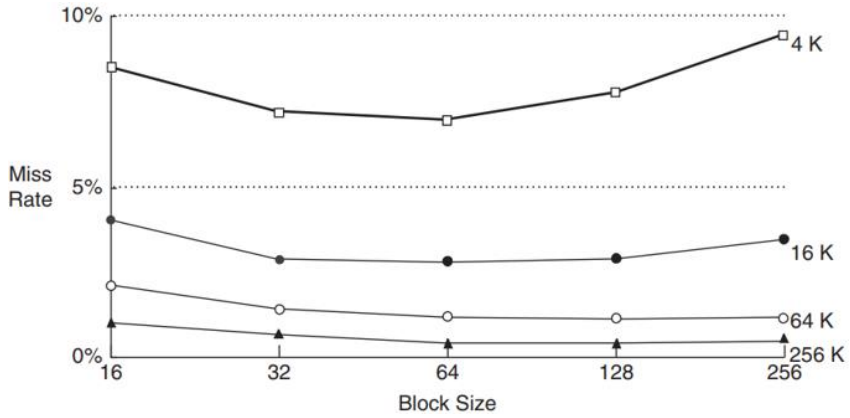
- Compulsory Miss: xảy ra khi CPU lần đầu truy cập Cache Block.
- Capacity Miss: xảy ra khi bộ nhớ đệm quá nhỏ để chứa tất cả dữ liệu được sử dụng đồng thời.
- Conflict Miss: xảy ra khi nhiều địa chỉ ánh xạ tới cùng một Cache Block và các khối bị loại bỏ vẫn còn cần được sử dụng trong tương lai gần.

Một số giải pháp để cải thiện hiệu suất của bộ nhớ đệm thông qua việc giảm tỉ lệ Cache Miss:

- Tăng Capacity (C): giảm Conflict Miss và Capacity Miss nhưng không làm giảm Compulsory Miss.
- Tăng Block Size (b): giảm Compulsory Miss nhưng có thể tăng Conflict Miss. Còn có thể tăng thời gian thực thi do Miss Penalty.
- Tăng Associativity (N): có thể giảm Conflict Miss.



Hình 2.8: Tỉ lệ Cache Miss tương quan giữa Associativity và Cache Size [5]



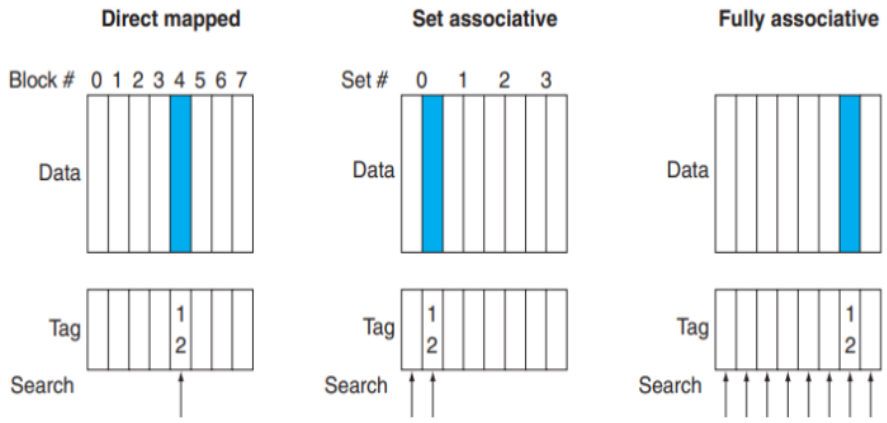
Hình 2.9: Tỉ lệ Cache Miss tương quan giữa Cache Block và Capacity [5]

Hình 2.8 và Hình 2.9 minh họa rõ sự ảnh hưởng của các thông số thiết kế Cache đến tỉ lệ Cache Miss. Hình 2.8 cho thấy khi tăng kích thước Cache và mức độ kết hợp (Associativity) từ 1-Way lên 4-Way, tỉ lệ Cache Miss, đặc biệt là Conflict Miss, giảm đáng kể. Trong khi đó, Hình 2.9 thể hiện rằng việc điều chỉnh

kích thước Block cũng ảnh hưởng đến tỉ lệ miss, nhưng sự giảm này không lớn bằng so với việc tăng Associativity hoặc kích thước Cache. Như vậy, để tối ưu hiệu quả Cache, cần đồng thời cân nhắc cả mức độ kết hợp, kích thước Cache và Block Size.

2.3.2. Ưu điểm của 4-Way Set Associative Cache

So với hai kiến trúc Direct Mapped Cache và Fully Associative Cache, kiến trúc Multiway Set Associative Cache cho thấy sự cân bằng giữa việc tìm kiếm khối và cách tổ chức khối trong bộ nhớ đệm nên mang lại hiệu quả cao hơn cho hệ thống máy tính. Hình 2.10 chỉ ra sự khác nhau trong cách tìm kiếm dữ liệu trong bộ nhớ đệm với từng loại kiến trúc: Direct Mapped Cache, Set Associative Cache, Fully Associative Cache.



Hình 2.10: Tìm kiếm trên 3 loại kiến trúc bộ nhớ đệm [5]

2.4. Các chính sách bộ nhớ đệm

Chính sách bộ nhớ đệm (hay Cache Policy) là các quy tắc xác định cách dữ liệu được ghi vào bộ nhớ đệm và ghi từ bộ nhớ đệm vào bộ nhớ chính (Main Memory).

2.4.1. Chính sách thay thế

Vì bộ nhớ đệm luôn có kích thước, khả năng lưu trữ nhỏ hơn bộ nhớ chính, nên cần có các chính sách thay thế (hay còn gọi là Cache Replacement) khi những

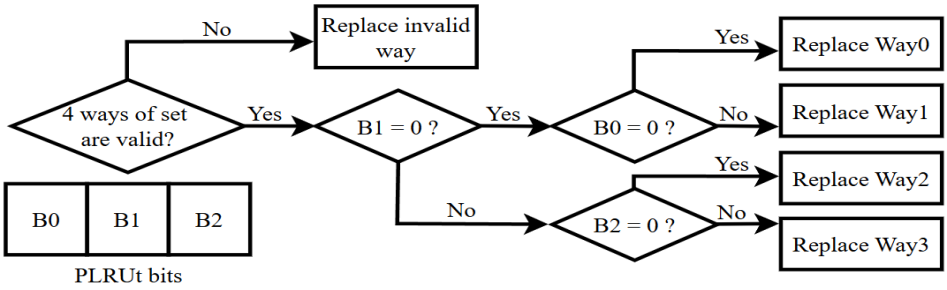
dữ liệu đó không còn được sử dụng hoặc cần tìm chỗ thay thế để ghi dữ liệu mới hơn khi Set đã đầy. Có khá nhiều giải thuật thay thế như:

- Least Recently Used (LRU).
- Pseudo Least Recently Used Policy based on MRU bits (PLRUm).
- Tree-based Pseudo Least Recently Used (PLRUt).
- Static Re-Reference Interval Prediction (SRRIP).
- …

Trong đó, PLRUt [6] và PLRUm [7] là các giải thuật xấp xỉ giải thuật LRU [6], cả hai giải thuật này đều có hiệu năng gần xấp xỉ giải thuật LRU nhưng lại đơn giản và có chi phí thiết kế thấp hơn nhiều so với giải thuật LRU [7]. Còn đối với giải thuật SRRIP [8], giải thuật thường được sử dụng để khắc phục các nhược điểm của giải thuật LRU và thường được cài đặt ở cấp bộ nhớ đệm cuối cùng "Last-Level Cache (LLC)". Đây là lý do đề tài quyết định sử dụng giải thuật PLRUt cho kiến trúc Multiway Set Associative Cache.

Về giải thuật PLRUt, nó là giải thuật xấp xỉ LRU (Least Recently Used) để tăng tốc hiệu suất và giảm độ phức tạp của việc triển khai. Vì đây là giải thuật xấp xỉ, nên Cache Block ít được truy cập gần đây nhất không phải luôn luôn là vị trí cần được thay thế.

Giải thuật PLRUt sử dụng N-1 bit để theo dõi trạng thái truy cập của một Cache Set đối với bộ nhớ đệm có N-Way. Các bit này sẽ được tổ chức theo cấu trúc dữ liệu cây nhị phân (Decision Binary Tree) và từ đó sẽ có quyết định thay thế Cache Block tương ứng.



Hình 2.11: Thuật toán PLRUt với 4-Way Set Associative Cache [7]

Hình 2.11 diễn giải cách hoạt động của giải thuật PLRUt sử dụng N-1 bit (với N=4) áp dụng cho bộ nhớ được tổ chức theo kiến trúc 4-Way Set Associative.

▪ Ví dụ minh họa tư tưởng thuật toán PLRUt [7]:

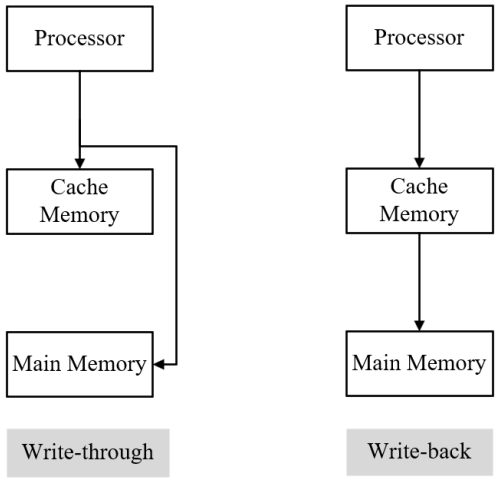
Giả sử cài đặt thuật toán PLRUt trên bộ nhớ đệm có kiến trúc 4-Way Set Associative Cache ứng với mỗi Set có 4 Way W0, W1, W2 và W3. Vậy đối với mỗi Set, ta sử dụng 3-bit {B0, B1, B2} để theo dõi trạng thái truy cập và quyết định xem sẽ thay thế Way nào. Đối với Cache Miss, khi B1 = 0 sẽ chỉ đến W0 và W1 là ít được sử dụng nhất, B1 = 1 sẽ chỉ đến W2 và W3 là ít được sử dụng nhất. B0 = 0 ứng với W0 là Way sẽ bị thay thế, B0 = 1 ứng với W1 sẽ bị thay thế. Tương tự áp dụng B2 đối với 2 Way còn lại là W2 và W3. Ngược lại, khi Cache Hit, việc cập nhật các bit cũng diễn ra tương tự như quá trình trên.

Đối với giải thuật SRRIP [8] (Static Re-Reference Interval Prediction), ý tưởng tương tự đối với giải thuật PLRUm (Pseudo Least Recently Used Policy based on MRU bits), nhưng thay vì chỉ sử dụng 1-bit để theo dõi trạng thái của Cache Block thì SRRIP tăng mức độ chi tiết dự đoán Cache Block được tham chiếu lại trong tương lai bằng cách sử dụng m-bit cho mỗi Cache Block.

2.4.2. Chính sách ghi

Trong suốt quá trình truy cập bộ nhớ đệm, dữ liệu cần được ghi xuống bộ nhớ chính để đảm bảo tính nhất quán của dữ liệu. Do đó, các chính sách ghi (hay còn gọi là Write Policy) sẽ cần được sử dụng, bao gồm Write-through và Write-back, chi tiết được thể hiện trong Hình 2.12 bên dưới:

- Write-through: dữ liệu ghi vào bộ nhớ đệm thì đồng thời cũng được ghi vào bộ nhớ chính (Main Memory).
- Write-back: dữ liệu được ghi vào bộ nhớ đệm khi Cache Block bị thay thế và đã bị chỉnh sửa, cần có bit Dirty (D) để theo dõi dữ liệu đã bị chỉnh sửa.



Hình 2.12: Minh họa Write-through và Write-back

Write-through tuy không yêu cầu thêm bit D nhưng lại yêu cầu truy cập bộ nhớ chính nhiều hơn (Write access) để ghi dữ liệu, so với Write-back, dữ liệu chỉ được ghi trở lại bộ nhớ chính khi Block là "victim block" và bit D bằng 1. Điều này giúp Write-back giảm số lần truy cập ghi không cần thiết vào bộ nhớ chính.

2.5. Tổng quan về Cache Coherence

2.5.1. Vấn đề Cache Coherence

Trong một hệ thống đa vi xử lý, sự không nhất quán dữ liệu có thể xảy ra giữa các bộ nhớ đệm của các bộ xử lý khi sử dụng chung một bộ nhớ chia sẻ. Khi một chương trình cùng được thực thi bởi nhiều luồng song song bởi các bộ xử lý độc lập, chúng có thể vô tình truy cập cùng lúc vào dữ liệu có địa chỉ nằm tại vùng nhớ chia sẻ chung. Việc này có thể dẫn đến sự khác nhau về góc nhìn dữ liệu (View of Memory) giữa các bản sao của dữ liệu đó tại các bộ nhớ đệm sở hữu chúng.

Phương pháp giải quyết vấn đề nhất quán bộ nhớ đệm (Cache Coherence) sẽ được thực hiện bằng cách thiết kế các giao thức hỗ trợ duy trì trạng thái thống nhất cho mỗi khối dữ liệu được lưu trong các bộ nhớ đệm.

2.5.2. Phương pháp giải quyết Cache Coherence

Để giải quyết vấn đề đồng nhất bộ nhớ đệm trong hệ thống đa vi xử lý (Multiprocessor System), các giao thức đảm bảo nhất quán bộ nhớ đệm (Cache-

Coherence Protocol) đã được sử dụng, phổ biến nhất là giao thức Directory-based và giao thức Snooping:

- Giao thức Directory-based: là một giao thức dựa trên việc sử dụng một thư mục trung tâm hoặc phân tán để theo dõi trạng thái và vị trí của mỗi Cache Block. Khi một bộ xử lý muốn đọc hoặc ghi một Cache Block, nó gửi một yêu cầu đến thư mục. Thư mục sau đó chấp nhận hoặc từ chối yêu cầu và phối hợp trực tiếp giữa bộ xử lý yêu cầu và các bộ xử lý khác có một bản sao của khối đó.
- Giao thức Snooping: là loại giao thức đồng nhất bộ nhớ đệm được triển khai rộng rãi đầu tiên, hỗ trợ giao dịch thời gian thấp và thiết kế đơn giản, tất cả các bộ xử lý đồng nhất đều theo dõi các yêu cầu và xử lý chúng theo cùng một thứ tự.

Điểm mạnh của các giao thức Snooping là tốc độ xử lý cao và thiết kế đơn giản. Tuy nhiên, điểm yếu của các giao thức Snooping là hạn chế ở khả năng mở rộng, khi số lượng bộ xử lý tăng lên, băng thông trên Bus cũng tăng, có thể dẫn đến sự cạnh tranh và giảm hiệu suất hệ thống. Do vậy, Snooping chỉ phù hợp với các hệ thống nhỏ với số lượng bộ xử lý nhỏ.

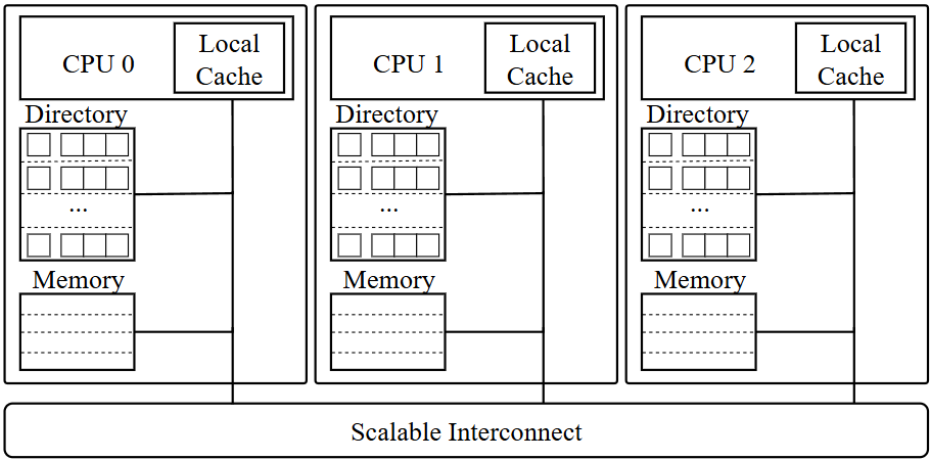
Trong khi đó, điểm mạnh của giao thức Directory-based là có thể dễ dàng mở rộng, phù hợp với các hệ thống lớn nhiều bộ xử lý. Tuy nhiên, nhược điểm của nó chính là độ phức tạp và tiêu tốn nhiều tài nguyên.

Với mục tiêu của đề tài áp dụng lên hệ thống có số lượng bộ xử lý nhỏ, phương pháp Snooping sẽ là lựa chọn phù hợp để giải quyết vấn đề Cache Coherence.

2.5.3. Giao thức Directory-based

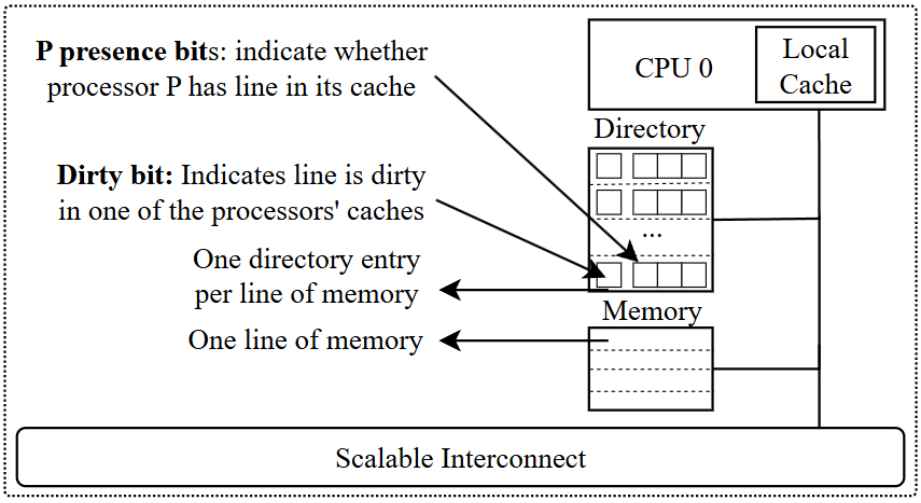
Giao thức Directory-based là một giao thức sử dụng cho hệ thống đa vi xử lý có bộ nhớ dùng chung được phân phối (Distributed Memory System), xây dựng với một bộ danh mục nhiều ngăn để lưu trữ và giám sát lượng lớn các khối dữ liệu (Cache Block) trong bộ nhớ đệm của từng bộ xử lý bao gồm cả trạng thái, dữ liệu,

bộ xử lý nào đang nắm dữ bản sao dữ liệu đó. Với giao thức này, các bộ xử lý có thể trực tiếp trao đổi yêu cầu và dữ liệu mà không thông qua việc thông báo với những bộ xử lý khác khi không cần thiết. Hình 2.13 bên dưới minh họa một hệ thống điển hình sử dụng Directory-based.



Hình 2.13: Hệ thống đa vi xử lý với bộ nhớ sử dụng Directory-based [9]

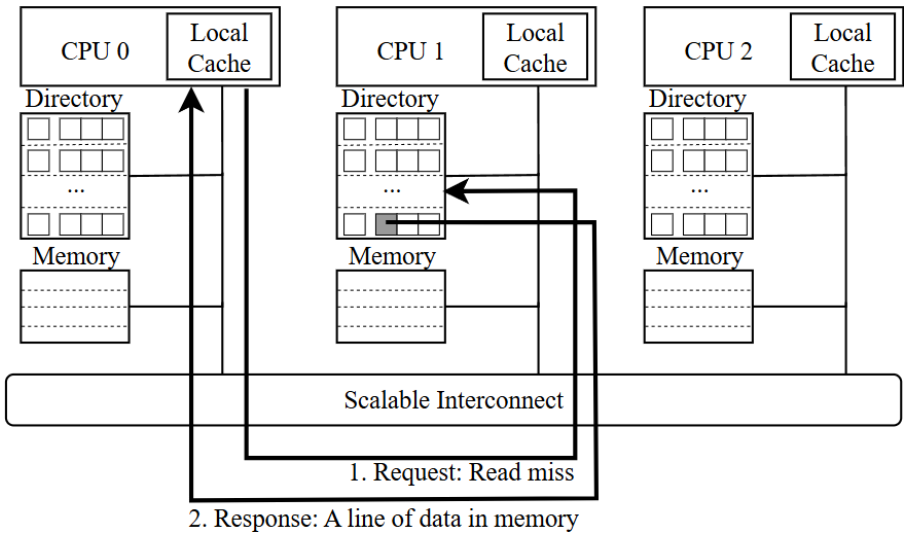
Hình 2.14 bên dưới là một bộ xử lý với bộ nhớ đệm riêng và danh mục riêng đơn giản, mỗi hàng trong danh mục đại diện cho một hàng nhớ trong bộ nhớ chính với Dirty bit là bit thông tin cho biết dữ liệu ở hàng này bị chỉnh sửa hay chưa.



Hình 2.14: Mô hình bộ xử lý với Directory đơn giản [9]

Hình 2.15 bên dưới là một ví dụ cụ thể cho các yêu cầu truy cập dữ liệu trong hệ thống đa vi xử lý sử dụng Directory-based cho bộ nhớ chung được phân phối. Bộ

xử lý CPU 0 truy cập miss một dữ liệu có địa chỉ tại vùng dữ liệu chia sẻ chung và yêu cầu dữ liệu đó từ một bộ xử lý khác (dữ liệu đó chưa bị chỉnh sửa).



Hình 2.15: Quá trình đọc dữ liệu chưa chỉnh sửa giữa CPU0 và CPU1 [9]

- 1. CPU 0 gửi yêu cầu đọc khối dữ liệu đến Directory của CPU 1.
- 2. Directory của CPU 1 kiểm tra bit Dirty của khối dữ liệu đó, nếu Dirty = 0 (chưa qua chỉnh sửa) nó sẽ phản hồi lại bằng cách gửi bản sao dữ liệu đó sang cho bộ nhớ đệm của của CPU 0.

2.5.4. Giao thức Snooping

Giao thức Snooping là giao thức mà tất cả các bộ xử lý đều phải tham gia với một hệ thống giao dịch thông qua các Bus với bộ nhớ chia sẻ chung yêu cầu các bộ xử lý phải cập nhật trạng thái và trả dữ liệu theo đúng nguyên tắc của giao thức.

❖ **Giao thức MSI:**

Giao thức MSI là giao thức đồng nhất bộ nhớ đệm cơ bản, MSI là viết tắt của Modified, Shared, Invalid. Mỗi Cache Block được duy trì một trong ba trạng thái này để duy trì tính nhất quán trong hệ thống:

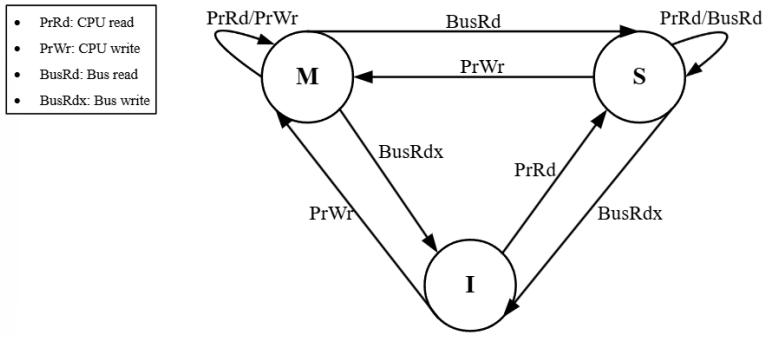
- Modified: Cache Block là duy nhất, đã được chỉnh sửa.
- Shared: Cache Block chia sẻ giữa các bộ nhớ đệm, có thể đã chỉnh sửa.
- Invalid: Cache Block không tồn tại, không hợp lệ.

Yêu cầu từ bên bộ xử lý cho bộ nhớ đệm bao gồm:

- PrRd: yêu cầu của bộ xử lý để đọc một Cache Block.
- PrWr: yêu cầu của bộ xử lý để ghi một Cache Block.

Yêu cầu từ Bus cho bộ nhớ đệm bao gồm:

- BusRd: khi có Read Miss xảy ra trong bộ nhớ đệm của một bộ xử lý, nó gửi một yêu cầu BusRd trên Bus và mong đợi nhận lại một Cache Block.
- BusRdX: khi có Write Miss xảy ra trong bộ nhớ đệm của một bộ xử lý, nó gửi một yêu cầu BusRdX trên Bus, trả về một Cache Block và vô hiệu hóa Block đó trong các bộ nhớ đệm của các bộ xử lý khác.
- BusWB: trước khi dữ liệu được đem đi để đọc, nó gửi một BusWB trên Bus để Write-back dữ liệu vào Main Memory.



Hình 2.16: Sơ đồ trạng thái của giao thức MSI [10]

Như máy trạng thái thể hiện ở Hình 2.16, dù là đọc hoặc ghi dữ liệu của khối nằm trong chính bộ nhớ đệm của nó, nó đều phải chuyển Write-back lại không quan tâm đến dữ liệu đó là mới hay cũ.

Ưu điểm của giao thức MSI:

- Có thể có nhiều bản sao của dữ liệu tại cùng một thời điểm
- Tiêu tốn ít tài nguyên, xây dựng giao thức đơn giản.

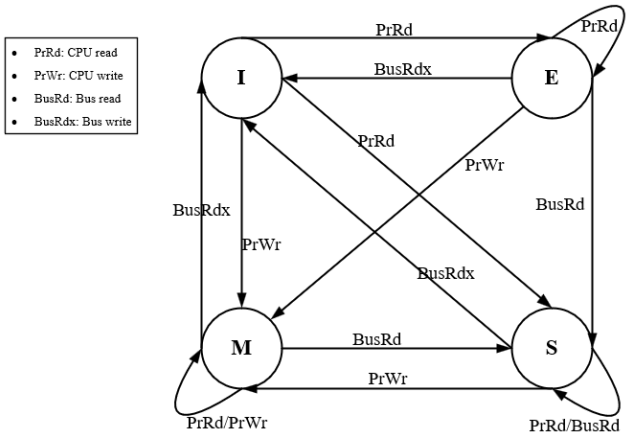
Nhược điểm của giao thức MSI:

- Khi yêu cầu đọc dữ liệu của một khối mà nó chỉ có 1 bản sao duy nhất thì vẫn phải chuyển sang trạng thái Shared để đọc, tốn nhiều thời gian truy cập.
- Khi yêu cầu ghi dữ liệu luôn cần phải thông báo với Cache khác, gây ra các yêu cầu Snoop không cần thiết.

❖ **Giao thức MESI:**

Giao thức MESI là giao thức được phát triển từ giao thức MSI. Trong đó, thêm vào trạng thái mới Exclusive (E), cung cấp khả năng tự chỉnh sửa dữ liệu độc quyền mà không cần phải thông báo và vô hiệu hóa các bản sao hay Block tương ứng ở các bộ nhớ đệm khác, giúp giải quyết vấn đề nhược điểm của MSI.

- Exclusive: khối dữ liệu chỉ tồn tại ở một bộ nhớ đệm duy nhất. Do đó nó có thể sửa đổi dữ liệu mà không cần thông báo với bộ nhớ đệm khác.



Hình 2.17: Sơ đồ trạng thái của giao thức MESI [10]

Hình 2.17 cho thấy rõ chức năng của trạng thái Exclusive khi có thể dễ dàng đánh dấu dữ liệu là độc quyền, chỉ cần Write-back mỗi khi dữ liệu bị chỉnh sửa.

Ưu điểm của giao thức MESI:

- Giảm bớt số lần thông báo vô hiệu hóa bản sao dữ liệu ở các bộ nhớ đệm khác khi 1 bộ xử lý yêu cầu ghi, chỉnh sửa dữ liệu bằng cách ghi dữ liệu từ trạng thái Exclusive chuyển sang Modified trong im lặng.

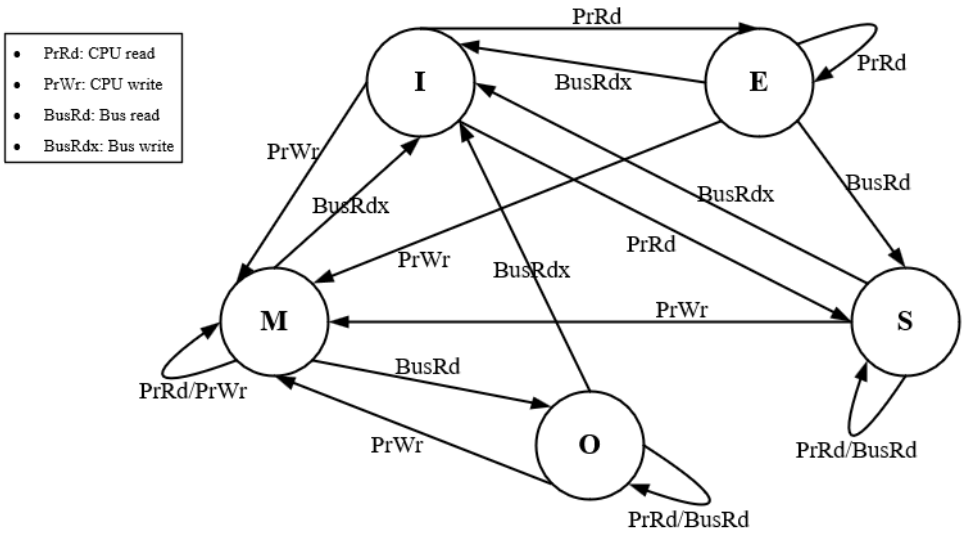
Nhược điểm của giao thức MESI:

- Độ phức tạp của phần cứng: triển khai giao thức MESI đòi hỏi phần cứng phức tạp để theo dõi, điều chỉnh trạng thái dữ liệu.
- Overhead Write-back: dữ liệu phải được Write-back về bộ nhớ chính (Main Memory) khi chuyển từ trạng thái "Modified" sang "Shared" hoặc “Invalid”, điều này tạo ra các lần Write-back không cần thiết.

❖ **Giao thức MOESI:**

Giao thức MOESI là phiên bản mở rộng của giao thức MESI, giới hạn việc ghi trở lại dữ liệu từ bộ nhớ đệm vào bộ nhớ chính (Write-back). Trạng thái mới "Owned" được phát minh để tránh việc ghi lại không cần thiết dữ liệu vào bộ nhớ chính trong quá trình chuyển từ trạng thái "Modified" sang "Shared" hoặc “Invalid”.

Owned: ở trạng thái này, dữ liệu vẫn được giữ lại trong bộ nhớ đệm của bộ xử lý hiện tại, nhưng chỉ có bộ xử lý đó có quyền sửa đổi dữ liệu, đánh dấu rằng dữ liệu này là dữ liệu đã được chỉnh sửa, chỉ Write-back khi bị loại bỏ, từ đó giúp giảm thiểu số lần Write-back không cần thiết. Hình 2.18 thể hiện rõ sự hiệu quả khi có thêm trạng thái Owned.



Hình 2.18: Sơ đồ trạng thái của giao thức MOESI [10]

2.6. Sơ lược về kiến trúc tập lệnh RISC-V RV32I

2.6.1. Tổng quát

RISC-V được định nghĩa là một kiến trúc tập lệnh số nguyên 32-bit (RV32I), là nền tảng cho các tập lệnh cơ sở. Với tập lệnh cơ sở này, bộ vi xử lý RISC-V đã tích hợp các lệnh cơ bản như điều khiển luồng, thao tác thanh ghi, truy cập bộ nhớ,... nhằm đảm bảo khả năng hiện thực đầy đủ các loại xử lý trên bộ vi xử lý. Điều này cho phép các bộ vi xử lý RISC-V đáp ứng hiệu quả các yêu cầu cơ bản của phần mềm và trình biên dịch, đồng thời phục vụ tốt cho các ứng dụng trong lĩnh vực máy tính để bàn, máy chủ cũng như các thiết bị nhúng.

Bên cạnh đó, từ nền tảng tập lệnh cơ sở, các nhà phát triển có thể linh hoạt lựa chọn và bổ sung thêm nhiều tập lệnh mở rộng theo nhu cầu sử dụng, hướng tới các mục tiêu đa dạng trong thực tiễn. Các tập lệnh mở rộng này được tiêu chuẩn hóa, đảm bảo tính nhất quán trong thiết kế phần cứng và phần mềm dựa trên nền tảng RISC-V (Bảng 2.1).

Bảng 2.1: Các tập lệnh cơ sở và mở rộng tiêu chuẩn của kiến trúc RISC-V [11]

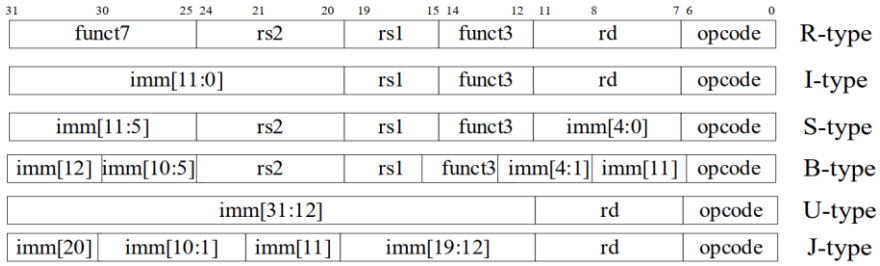
aw	Tên tập lệnh (đầy đủ)	Chú thích
RV32I	Base Integer Instruction Set, 32-bit	Các tập lệnh cơ sở
RV32E	Base Integer Instruction, (embedded)	
RV64I	Base Integer Instruction Set, 64-bit	
RV128I	Base Integer Instruction Set, 128-bit	
M	Integer Multiplication and Division	Các tập lệnh mở rộng tiêu chuẩn
F	Single-Precision Floating Point	
C	Decimal Floating Point	
...	...	

2.6.2. Kiến trúc tập lệnh cơ sở

Trong kiến trúc tập lệnh RISC-V, tất cả các lệnh đều được mã hóa theo các định dạng cố định, nhằm đơn giản hóa quá trình giải mã và thực thi lệnh. Cụ thể, RV32I định nghĩa 6 nhóm định dạng lệnh cơ bản, bao gồm:

- R-type (Register type): Sử dụng cho các phép toán số học hoặc logic giữa các thanh ghi. Định dạng này gồm hai thanh ghi nguồn (rs1, rs2), một thanh ghi đích (rd), cùng các trường chức năng (funct3, funct7) và opcode.
- I-type (Immediate type): Dùng cho các lệnh thao tác với một toán hạng là hằng số (immediate), như lệnh load từ bộ nhớ, phép toán số học với immediate, hoặc một số lệnh nhảy.
- S-type (Store type): Áp dụng cho các lệnh lưu dữ liệu từ thanh ghi vào bộ nhớ; immediate được chia thành hai phần nhỏ.
- B-type (Branch type): Dành cho các lệnh nhánh có điều kiện, với immediate để phục vụ tính toán địa chỉ nhảy.
- U-type (Upper immediate type): Được sử dụng cho các lệnh nạp giá trị immediate lớn vào phần trên của thanh ghi (ví dụ: LUI, AUIPC).
- J-type (Jump type): Sử dụng cho các lệnh nhảy không điều kiện (jump), với immediate 20-bit cho phép nhảy xa.

Các định dạng này được thể hiện trực quan trong Hình 2.19, mỗi trường trong lệnh đóng vai trò xác định rõ chức năng và cách xử lý của bộ vi xử lý RISC-V.



Hình 2.19: Định dạng lệnh của kiến trúc RISC-V [11]

Về hệ thống thanh ghi, RV32I trang bị một thanh ghi chương trình (PC) và một Register File gồm 32 thanh ghi, trong đó x0 luôn mang giá trị 0, còn x1 đến

x31 đảm nhiệm các mục đích khác nhau. Tất cả đều có độ rộng 32-bit (riêng RV64I, thanh ghi mở rộng lên 64-bit). Tuân theo mô hình Load-Store đặc trưng của kiến trúc RISC-V, chỉ các lệnh Load và Store mới truy cập bộ nhớ, còn lại mọi phép toán đều thực hiện trực tiếp trên thanh ghi.

Đồng thời, RV32I quy định không gian địa chỉ là 32-bit, sử dụng phương thức lưu trữ Little-endian. Tương ứng, RV64I và RV128I mở rộng không gian địa chỉ lên lần lượt 64-bit và 128-bit, phục vụ cho những ứng dụng và kiến trúc khác nhau. Ngoài ra, phiên bản RV32E được thiết kế tối ưu cho hệ thống nhúng với chỉ 16 thanh ghi 32-bit, các thanh ghi còn lại của RV32I sẽ là tùy chọn.

Như vậy, nhờ cấu trúc định dạng lệnh rõ ràng cùng hệ thống thanh ghi linh hoạt, kiến trúc tập lệnh RISC-V đáp ứng tốt cả về tính đơn giản lẫn khả năng mở rộng cho nhiều loại ứng dụng khác nhau.

2.7. Tổng quan về giao thức AXI và ACE

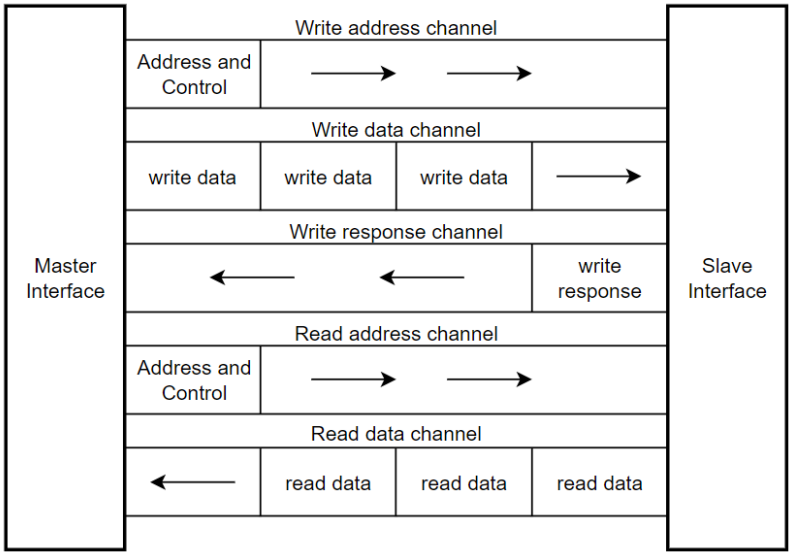
2.7.1. Tổng quan về AXI

Giao thức AXI (Advanced Extensible Interface) là một phần của chuẩn AMBA do ARM phát triển, được giới thiệu lần đầu vào năm 1996 nhằm hỗ trợ giao tiếp giữa các bộ điều khiển và vi xử lý. Giao thức AXI lần đầu tiên xuất hiện trong AMBA 3.0 (năm 2003) và được mở rộng thêm với AMBA 4.0 vào năm 2010, trong đó bao gồm bản cải tiến của AXI. Giao thức AXI sở hữu giao diện mở rộng (eXtensible) giúp loại bỏ điểm tắc nghẽn trên bus, nâng cao băng thông và giảm độ trễ. Vì vậy, AXI được dùng rộng rãi trong các hệ thống sử dụng giao diện AMBA.

Trong khóa luận này, nhóm tập trung vào loại giao thức AXI-Full [12] (viết tắt là AXI), ngoài ra còn có hai biến thể khác là AXI-Lite và AXI-Stream [12], cụ thể:

- AXI-Full: Dùng cho các ứng dụng cần băng thông và hiệu suất cao.
- AXI-Lite: Thích hợp cho các kết nối bộ nhớ hoặc thanh ghi dung lượng nhỏ, ít tín hiệu điều khiển.
- AXI-Stream: Dùng để truyền dữ liệu dạng luồng với tốc độ lớn.

Hình 2.20 chỉ ra 5 kênh giao tiếp của giao thức AXI sử dụng cho việc truyền tải dữ liệu theo cùng một chuẩn quy định.



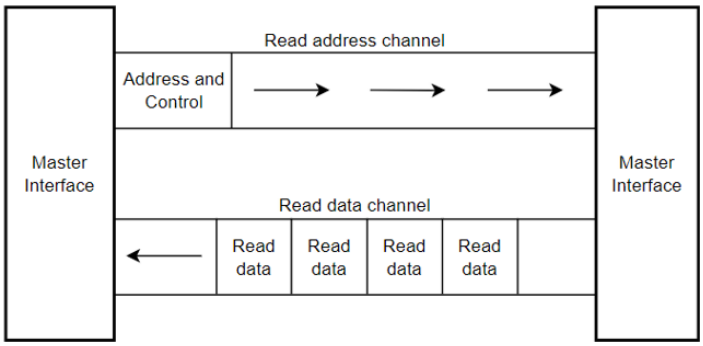
Hình 2.20: Kiến trúc chung của giao thức AXI [12]

Kiến trúc tổng thể của AXI bao gồm hai thành phần: AXI Master và AXI Slave, kết nối với nhau thông qua 5 kênh riêng biệt (gồm 2 kênh cho đọc và 3 kênh cho ghi dữ liệu), cụ thể:

- Write Address Channel: Kênh địa chỉ ghi.
- Write Data Channel: Kênh dữ liệu ghi.
- Write Response Channel: Kênh phản hồi ghi.
- Read Address Channel: Kênh địa chỉ đọc.
- Read Data Channel: Kênh dữ liệu đọc.

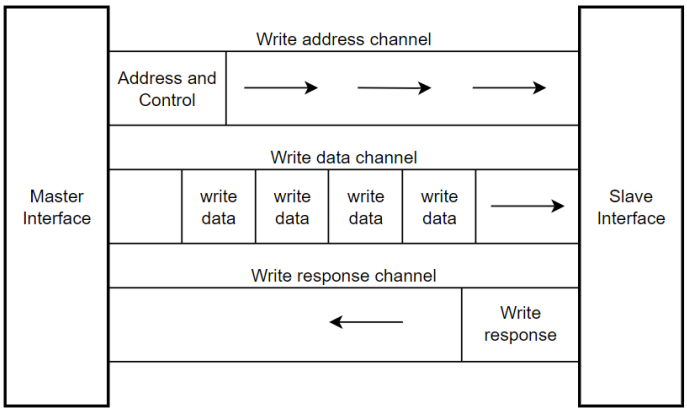
Giao thức AXI có khả năng kết nối linh hoạt giữa nhiều AXI Master và nhiều AXI Slave, nhờ vậy giúp hệ thống hoạt động hiệu quả hơn.

Quá trình đọc bắt đầu khi Master gửi tín hiệu yêu cầu đọc qua kênh địa chỉ đọc (Read Address Channel) như Hình 2.21. Sau khi nhận lệnh, Slave phản hồi bằng việc trả dữ liệu về cho Master thông qua kênh đọc dữ liệu (Read Data Channel), đồng thời gửi tín hiệu xác nhận transaction đã hoàn thành.



Hình 2.21: Một transaction đọc của giao thức AXI [12]

Quá trình ghi sử dụng kênh địa chỉ ghi, kênh dữ liệu và kênh phản hồi ghi như Hình 2.22 giao thức AXI hỗ trợ Burst với chiều dài đa dạng (từ 1 đến 16 hoặc hơn), mỗi Burst truyền nhiều dữ liệu có kích thước từ 8-bit đến 1024-bit.



Hình 2.22: Một transaction ghi của giao thức AXI [12]

2.7.2. Tổng quan về ACE

Giao thức ACE (AXI Coherency Extensions) [12] là giao thức mở rộng của giao thức AXI nhằm cung cấp cơ chế đảm bảo Cache Coherence. Ngoài các tính năng của giao thức AXI, giao thức ACE còn hỗ trợ:

- Kết nối Coherence giữa các thành phần trong hệ thống đa vi xử lý.
- Kết nối Coherence giữa các thành phần hỗ trợ các giao thức Cache Coherence khác nhau như MSI, MESI, MOESI.
- Tạo lớp đóng gói Coherence bên ngoài đối với các thành phần không hỗ trợ Coherence, cho phép chúng hoạt động hiệu quả trong hệ thống yêu cầu Coherence.

- Hỗ trợ kết nối trong hệ thống nhiều lớp bộ nhớ đệm (Multi-level Cache).
- Hỗ trợ đảm bảo Coherence với nhiều độ chi tiết khác nhau, theo Cache Line hoặc theo khối bộ nhớ lớn.
- Có thể được sử dụng như kết nối chính của hệ thống và có thể kết nối nhiều hệ thống với nhau.

Kiến trúc tổng thể của giao thức ACE gồm 5 kênh của giao thức AXI và thêm 3 kênh hỗ trợ Snoop, nhằm đảm bảo Coherence:

- Write Address Channel: Kênh địa chỉ ghi.
- Write Data Channel: Kênh dữ liệu ghi.
- Write Response Channel: Kênh phản hồi ghi.
- Read Address Channel: Kênh địa chỉ đọc.
- Read Data Channel: Kênh dữ liệu đọc.
- Snoop Address Channel: Kênh địa chỉ snoop.
- Snoop Data Channel: Kênh dữ liệu snoop.
- Snoop Response Channel: Kênh phản hồi snoop.

Giao thức ACE bổ sung thêm một số tín hiệu vào kênh địa chỉ đọc của giao thức AXI trong Bảng 2.2:

Bảng 2.2: Tín hiệu bổ sung kênh địa chỉ đọc của giao thức ACE [12]

Tín hiệu	Nguồn	Mô tả
ARSNOOP[3:0]	Master	Xác định loại giao dịch đọc chia sẻ (Shareable read transactions)
ARDOMAIN[1:0]	Master	Xác định vùng truy cập của giao dịch đọc (Shareability domain of a read transaction)
ARBAR[1:0]	Master	Xác định liệu giao dịch này có phải là một rào chắn đọc (Read barrier) hay không

Giao thức ACE bổ sung thêm một số tín hiệu vào kênh địa chỉ ghi của giao thức AXI trong Bảng 2.3:

Bảng 2.3: Tín hiệu bổ sung kênh địa chỉ ghi của giao thức ACE [12]

Tín hiệu	Nguồn	Mô tả
AWSNOOP[2:0]	Master	Xác định loại giao dịch ghi chia sẻ (Shareable write transactions)
AWDOMAIN[1:0]	Master	Xác định vùng truy cập của giao dịch ghi (Shareability domain of a write transaction)
AWBAR[1:0]	Master	Xác định liệu giao dịch này có phải là một rào chắn ghi (Write barrier) hay không
AWUNIQUE	Master	Xác định dữ liệu có được phép ở trạng thái Unique hay không, chỉ sử dụng đối với hệ thống hỗ trợ giao dịch WriteEvict

Giao thức ACE bổ sung thêm một số tín hiệu vào kênh dữ liệu đọc của giao thức AXI trong Bảng 2.4:

Bảng 2.4: Tín hiệu bổ sung kênh dữ liệu đọc của giao thức ACE [12]

Tín hiệu	Nguồn	Mô tả
RRESP[3:2]	Interconnect	Phản hồi đọc, xác định trạng thái của dữ liệu đọc

Đối với kênh địa chỉ snoop, giao thức ACE hỗ trợ các tín hiệu trong Bảng 2.5:

Bảng 2.5: Tín hiệu kênh địa chỉ snoop [12]

Tín hiệu	Nguồn	Mô tả
ACVALID	Interconnect	Báo hiệu rằng các tín hiệu địa chỉ snoop hợp lệ
ACREADY	Master	Báo hiệu rằng Master sẵn sàng nhận giao dịch snoop từ kênh địa chỉ snoop

ACADDR[ac-1:0]	Interconnect	Địa chỉ bắt đầu của giao dịch snoop (độ rộng phụ thuộc vào bus: ac)
ACSNOOP[3:0]	Interconnect	Loại giao dịch snoop
ACPROT[2:0]	Interconnect	Thuộc tính bảo vệ của giao dịch snoop

Đối với kênh phản hồi snoop, giao thức ACE hỗ trợ các tín hiệu trình bày trong Bảng 2.6:

Bảng 2.6: Tín hiệu kênh phản hồi snoop [12]

Tín hiệu	Nguồn	Mô tả
CRVALID	Master	Báo hiệu rằng các tín hiệu phản hồi snoop hiện tại là hợp lệ
CRREADY	Interconnect	Báo hiệu rằng Interconnect sẵn sàng nhận phản hồi snoop từ kênh phản hồi snoop
CRRESP[4:0]	Master	Phản hồi snoop, chỉ ra trạng thái xử lý của giao dịch snoop

Đối với kênh dữ liệu snoop, giao thức ACE hỗ trợ các tín hiệu trong Bảng 2.7:

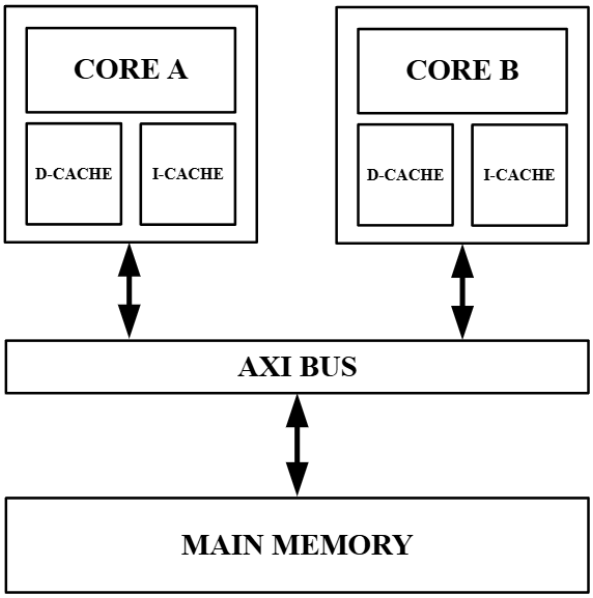
Bảng 2.7: Tín hiệu kênh dữ liệu snoop [12]

Tín hiệu	Nguồn	Mô tả
CDVALID	Master	Báo hiệu rằng dữ liệu snoop hiện tại là hợp lệ
CDREADY	Interconnect	Báo hiệu rằng Interconnect sẵn sàng nhận dữ liệu snoop trong chu kỳ hiện tại
CDDATA[cd-1:0]	Master	Dữ liệu snoop được truyền (độ rộng phụ thuộc vào bus dữ liệu snoop: cd)
CDLAST	Master	Chỉ ra rằng đây là lần truyền dữ liệu cuối cùng trong một giao dịch snoop

CHƯƠNG 3. THIẾT KẾ HỆ THỐNG

3.1. Mô tả tổng quan hệ thống

Thiết kế hệ thống đa vi xử lý gồm 2 lõi xử lý sở hữu riêng một bộ nhớ đệm gồm Instruction Cache (Bộ nhớ đệm lệnh) và Data Cache (Bộ nhớ đệm dữ liệu), giao thức MOESI được cài đặt trong các Data Cache, được kết nối với nhau thông qua một AXI Bus mở rộng ACE với nhóm kênh hỗ trợ trợ snooping, bus này cũng là thành phần giúp truyền tải dữ liệu giữa các vi xử lý và bộ nhớ chính. Hình 3.1 bên dưới thể hiện rõ cấu trúc hệ thống mà khóa luận xây dựng và thiết kế.



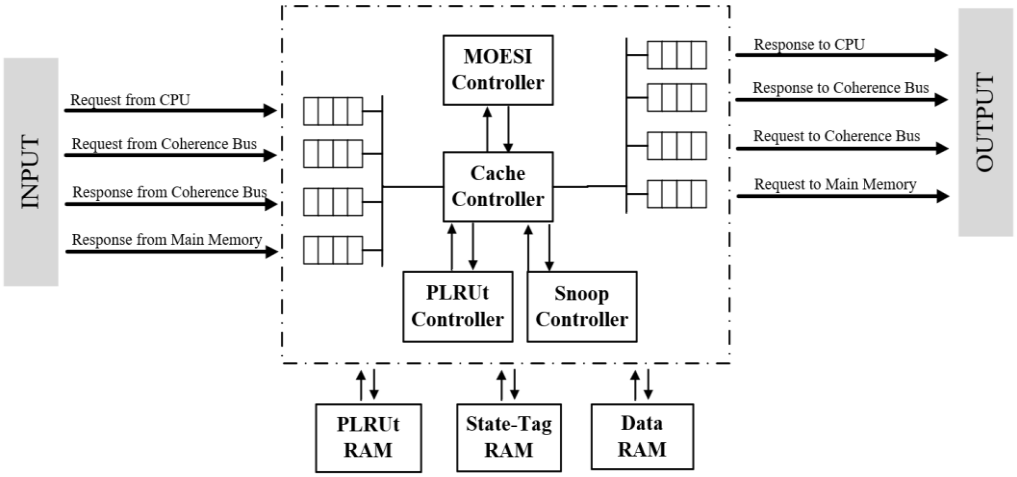
Hình 3.1: Sơ đồ hệ thống đa vi xử lý áp dụng Cache Coherence

Khi xuất hiện một truy cập, các lõi xử lý sẽ tiếp nhận truy cập và kiểm tra xem liệu dữ liệu đó có tồn tại bên trong bộ nhớ đệm của chính mình (hit hoặc miss) hay không, từ đó lõi xử lý sẽ tiến hành thao tác trên dữ liệu hoặc gửi thông báo, gửi yêu cầu chia sẻ dữ liệu để có thể đọc đến lõi còn lại thông qua giao thức Cache Coherence MOESI đã cài đặt cùng với sự hỗ trợ của AXI Bus mở rộng ACE.

3.2. Thiết kế hệ thống phần cứng

3.2.1. Thiết kế phần cứng bộ nhớ đệm

Đề tài tập trung chú trọng thiết kế phần cứng D-Cache để thực hiện đồng nhất dữ liệu giữa các vi xử lý và có đầy đủ các khối chức năng của một Multiway Set Associative Cache hoàn chỉnh, vì vậy D-Cache sẽ bao gồm các khối chức năng được thể hiện chi tiết trong Hình 3.2 như sau:



Hình 3.2: Tổng quan các khối thành phần trong D-Cache

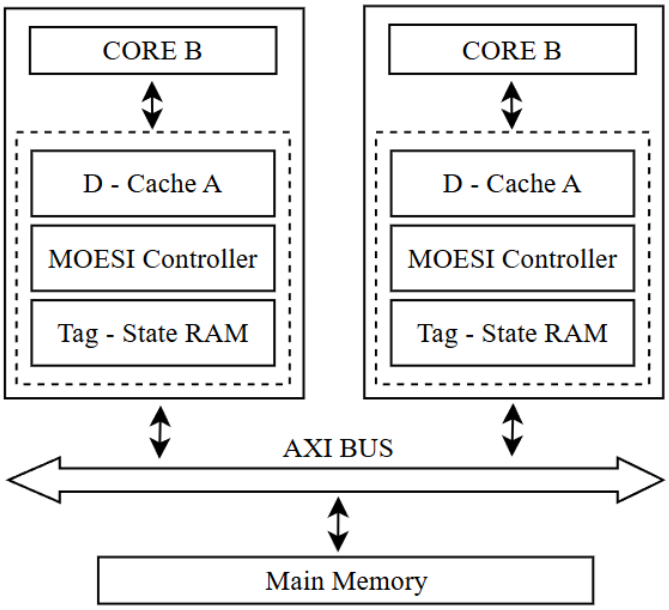
- State-Tag RAM: lưu trữ các bit của giao thức MOESI và các bit Tag của địa chỉ vật lý.
- PLRUt RAM: Lưu trữ các bit PLRUt của các Set trong Cache.
- Data RAM: Lưu dữ liệu bộ đệm Cache.
- MOESI Controller: Khối điều khiển thực hiện giao thức MOESI Snoopy.
- PLRUt Controller: Khối điều khiển thực hiện giải thuật PLRUt.
- Snoop Controller: Khối điều khiển snoop yêu cầu truy cập từ Cache khác.
- Cache Controller: Khối điều khiển của bộ nhớ đệm, thực hiện điều khiển toàn bộ hoạt động của hệ thống.

Bộ nhớ đệm của nhóm được thiết kế với kích thước 4KB gồm 4-Way và 16-Set, kích thước một Cache Line là 64-byte. I-Cache được thiết kế tương tự D-Cache nhưng không được tích hợp Cache Coherence.

3.2.2. Thiết kế trình đồng nhất dữ liệu bộ nhớ đệm Cache Coherence

Trình đồng nhất dữ liệu bộ nhớ đệm Cache Coherence được thực hiện thông qua việc áp dụng giao thức MOESI Snoopy được đề cập trong Hình 2.18, giao thức này giúp cung cấp kênh giao tiếp chung giữa các luồng thực thi cùng một chương trình trong hệ thống đa vi xử lý. Trong quá trình thực thi, toàn bộ các truy cập dữ liệu trong vùng dùng chung sẽ đều được theo dõi bởi các vi xử lý nhờ vào trạng thái của các khối dữ liệu được cung cấp bởi giao thức MOESI, bao gồm 5 trạng thái:

- Modified (Đã chỉnh sửa): Thông báo dữ liệu đã bị chỉnh sửa, không còn giống với dữ liệu gốc trong bộ nhớ chính.
- Owned (Được sở hữu): Đánh dấu dữ liệu đã bị chỉnh sửa bởi khối đang sở hữu nó, chỉ cần Write-back lại bộ nhớ duy nhất khi dữ liệu bị loại bỏ.
- Exclusive (Độc quyền): Dữ liệu này là độc quyền và chỉ có duy nhất một bản sao tại bộ nhớ đệm sở hữu nó, giống như trong bộ nhớ chính.
- Shared (Được chia sẻ): Dữ liệu được chia sẻ, các bộ xử lý khác có thể truy cập để đọc dữ liệu này.
- Invalid (Vô hiệu): Thông báo khối đang vô hiệu, không thể sử dụng.

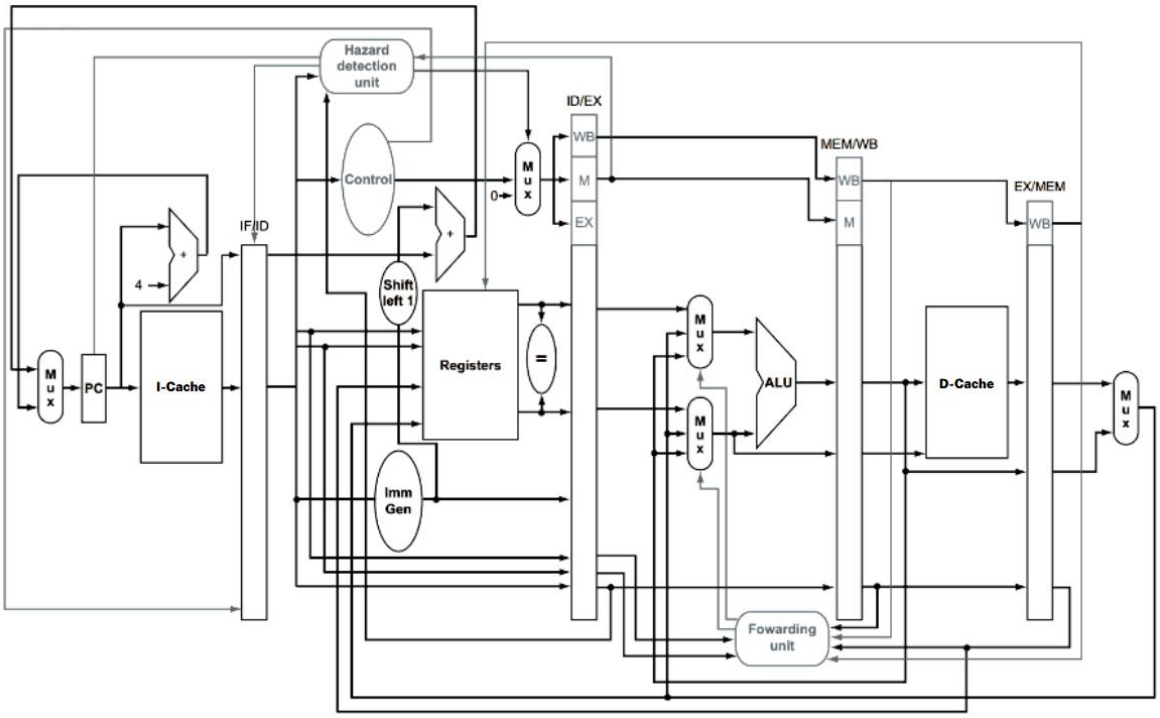


Hình 3.3: Mọi truy cập dữ liệu chung được theo dõi qua trạng thái của MOESI

Hình 3.3 cho thấy mọi truy cập dữ liệu trong hệ thống đa vi xử lý đều được cập nhật khi tiến vào bộ nhớ đệm. Khối MOESI Controller sẽ tiếp nhận mọi thông tin sau đó tiến hành kiểm tra và cập nhật trạng thái dữ liệu trong Tag–State RAM, từ đó sẽ được ra những tín hiệu điều khiển đọc/ghi, chia sẻ dữ liệu hay thông báo phù hợp với từng tình huống.

3.2.3. Thiết kế phần cứng lõi xử lý RISC-V RV32I

Lõi xử lý RISC-V RV32I được thiết kế áp dụng kiến trúc pipeline với 5 tầng xử lý, giúp cải thiện tần số hoạt động của hệ thống [19].



Hình 3.4: Kiến trúc RISC-V pipeline 5 tầng [13]

Hình 3.4 thể hiện rõ kiến trúc thiết kế được chia ra thành 5 tầng hoạt động, cụ thể như sau:

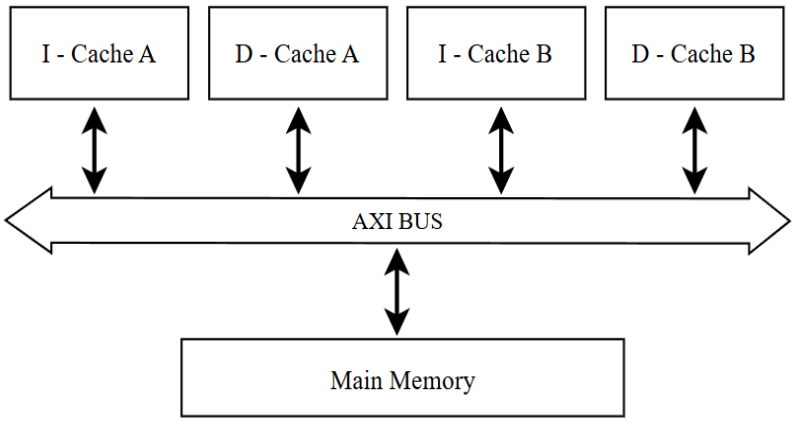
- Tầng IF (Instruction Fetch): Truy cập I-Cache để lấy ra lệnh tiếp theo cần thực hiện, dựa trên giá trị của bộ đếm chương trình (PC - Program Counter). Lệnh vừa được nạp sẽ được chuyển tiếp đến giai đoạn giải mã (ID).

- Tầng ID (Instruction Decode): Lệnh vừa nạp sẽ được giải mã để xác định loại lệnh, các toán hạng nguồn và đích. Đồng thời, các giá trị toán hạng sẽ được đọc từ bộ thanh ghi (Register File). Ngoài ra, giai đoạn này cũng thực hiện các tín hiệu điều khiển, phát hiện Data Hazard (xung đột dữ liệu) và Control Hazard (xung đột điều khiển), chuẩn bị các tín hiệu cần thiết cho các bước kế tiếp.
- Tầng EX (Execution): ALU thực hiện các phép tính toán số học, logic hoặc tính địa chỉ truy cập bộ nhớ (đối với lệnh load/store). Các phép toán nhánh (branch) cũng sẽ được tính toán ở giai đoạn này để xác định đường đi tiếp theo của chương trình.
- Tầng MEM (Memory Access): Thực hiện các thao tác truy cập bộ nhớ đệm dữ liệu D-Cache, ví dụ như nạp dữ liệu (load) từ bộ nhớ vào thanh ghi, hoặc ghi dữ liệu (store) từ thanh ghi ra bộ nhớ.
- Tầng WB (Write back): Kết quả tính toán từ ALU hoặc dữ liệu nạp từ bộ nhớ sẽ được ghi trở lại bộ thanh ghi.

Bên cạnh đó, vấn đề Data Hazard còn được xử lý thông qua khối Forwarding Unit, giúp ngay lập tức trả về dữ liệu được yêu cầu để có thể tránh xung dột dữ liệu và cải thiện thời gian thực thi của toàn bộ thiết kế.

3.2.4. Thiết kế phần cứng AXI Bus mở rộng ACE

AXI Bus mở rộng ACE được thiết kế theo giao thức AXI tích hợp thêm các kênh snoop, hoạt động như một kênh truyền dữ liệu và cũng đồng thời là một bộ phân xử các truy cập, kết nối các bộ nhớ đệm và bộ nhớ chính trong hệ thống đa vi xử lý. AXI bus mở rộng ACE hỗ trợ chức năng snooping phục vụ cho việc giải quyết Cache Coherence thông qua các kênh snoop.

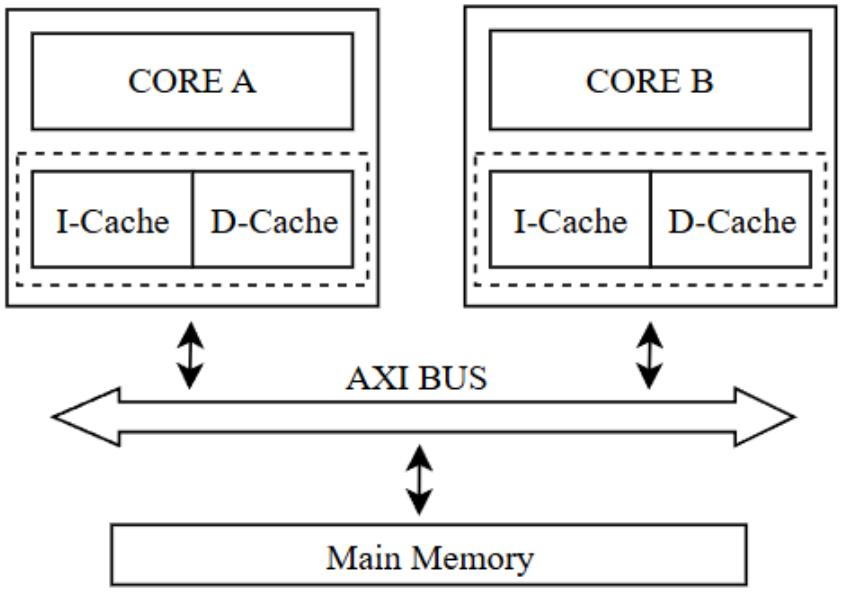


Hình 3.5: Kiến trúc của AXI Bus

Hình 3.5 bên dưới thể hiện rõ kiến trúc AXI Bus được thực hiện với 4 Master lần lượt là I-Cache và D-Cache của CPU A và CPU B, bộ nhớ chính chia sẻ chung sẽ đóng vai trò Slave tiếp nhận và xử lý các yêu cầu truy cập dữ liệu từ các bộ xử lý.

3.3. Thiết kế hệ thống mô phỏng phần mềm

Nhằm phục vụ mục đích mô phỏng và đánh giá tính chính xác và hiệu suất của thiết kế, nhóm tích hợp thiết kế phần cứng vào hệ thống phần mềm tạo thành hệ thống đa vi xử lý được mô phỏng như trong Hình 3.6.



Hình 3.6: Hệ thống đa vi xử lý mô phỏng bằng Python

3.3.1. Thiết kế phần mềm bộ nhớ đệm

Bộ nhớ đệm được thiết kế hoàn chỉnh với đầy đủ các chức năng tương tự với thiết kế phần cứng, bao gồm:

- Được thiết kế theo kiến trúc 4-Way Set Associative.
- Áp dụng các chính sách Read/Write Allocate tích hợp bộ phân xử First-Come First-Served (FCFS) kết hợp Round-Robin (RR).
- Cài đặt giải thuật thay thế khối dữ liệu PLRUt (Tree-based Pseudo Least Recently Used).
- Cài đặt giao thức đồng nhất dữ liệu Cache Coherence MOESI Snoopy.

3.3.2. Thiết kế phần mềm lõi xử lý

Lõi xử lý được thiết kế để chịu trách nhiệm thực hiện tạo các yêu cầu đọc/ghi vào bộ nhớ đệm, nhằm mô phỏng hoạt động đọc/ghi của các CPU trong thực tế. Trong đó, các lệnh đọc ghi sẽ truy cập vào các vùng địa chỉ riêng biệt của CPU, vùng địa chỉ dùng chung giữa các CPU. Đồng thời, giải thuật thay thế khối dữ liệu PLRUt (Tree-based Pseudo Least Recently Used) cũng được áp dụng.

3.3.3. Thiết kế phần mềm AXI Bus mở rộng ACE

AXI Bus được thiết kế phần mềm với khả năng hỗ trợ định tuyến và phân xử các yêu cầu truy cập bộ nhớ, hoạt động tương tự cổng mux/demux, tích hợp bộ phân xử First-Come First-Served (FCFS) kết hợp Round-Robin (RR). Giao thức AXI mở rộng ACE hỗ trợ kênh giao tiếp phục vụ snooping giữa các bộ nhớ đệm trong hệ thống đa vi xử lý [12].

3.3.4. Thiết kế phần mềm Main Memory

Bộ nhớ chính được thiết kế có thể thực hiện các chức năng cơ bản như đọc, ghi dữ liệu giống một bộ nhớ thông thường với khả năng lưu trữ đồng thời nhiều truy cập bằng FIFO (First-In, First-Out). Hỗ trợ thực hiện và phản hồi các truy cập theo giao thức AXI, mô phỏng Block RAM trong phần mềm Vivado với kích thước 64KB.

CHƯƠNG 4. THIẾT KẾ CHI TIẾT

4.1. Thiết kế chi tiết phần cứng

Thiết kế chi tiết phần cứng sẽ bao gồm thiết kế chi tiết các thành phần:

- CPU RV32I: Được thiết kế theo kiến trúc pipeline 5 tầng, có xử lý xung đột dữ liệu và xung đột điều khiển, hỗ trợ toàn bộ tập lệnh Integer 32-bit.
- I-Cache: Thiết kế theo kiến trúc 4-Way Cache Associative và 16-Set riêng biệt cho từng lõi, không hỗ trợ Coherence. Sử dụng chính sách thay thế PLRUt, chính sách cấp phát Read/Write Allocate.
- D-Cache: Thiết kế theo kiến trúc 4-Way Cache Associative và 16-Set riêng biệt cho từng lõi, hỗ trợ giao thức Coherence MOESI. Sử dụng chính sách thay thế PLRUt, chính sách cấp phát Read/Write Allocate.
- AXI Bus mở rộng ACE: Được thiết kế hỗ trợ 4 Master (2 I-Cache và 2 D-Cache) và 1 Slave là bộ nhớ chính chia sẻ chung. Sử dụng bộ phân xử theo chính sách First-Come First-Served (FCFS) kết hợp Round-Robin (RR). Có tích hợp thêm các kênh và tín hiệu mở rộng ACE để hỗ trợ Cache Coherence.

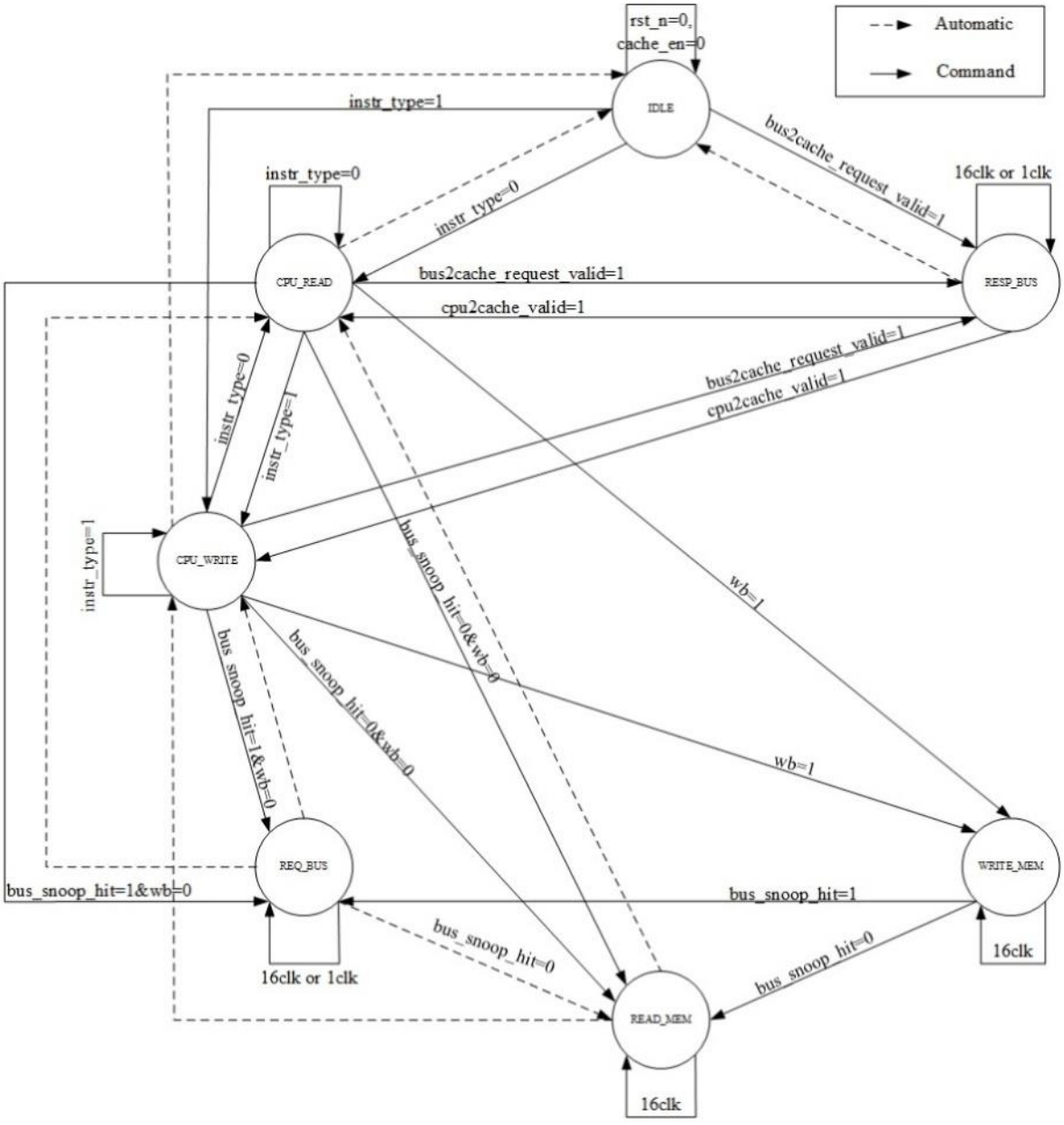
4.1.1. Thiết kế phần cứng Multiway Set Associative Cache

- **Thiết kế Máy trạng thái**

Bảng 4.1: Chức năng của các trạng thái trong Cache

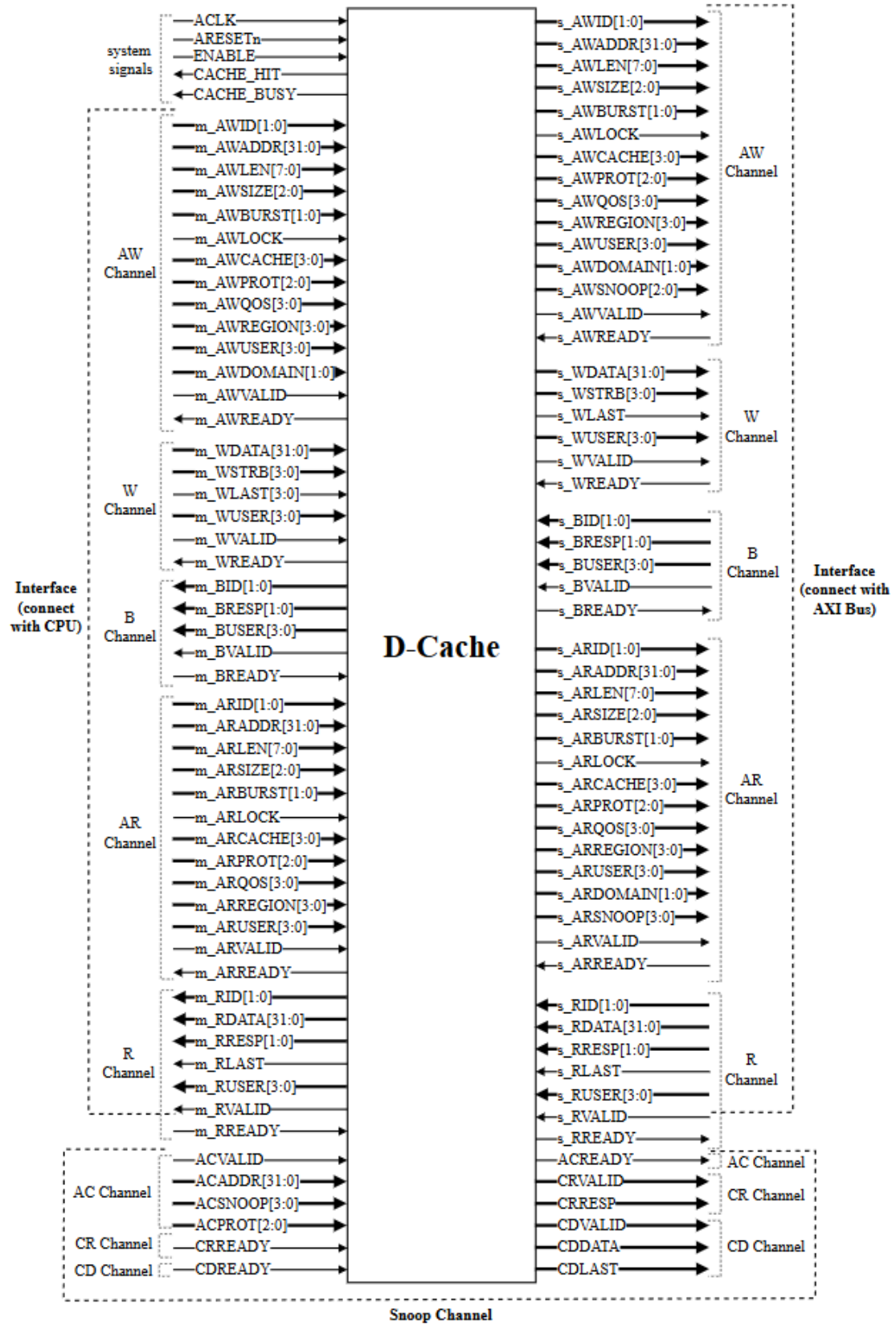
Trạng thái	Chức năng
IDLE	Trạng thái rảnh không hoạt động
CACHE_READ	Thực hiện đọc dữ liệu từ Cache
CACHE_WRITE	Thực hiện ghi dữ liệu vào Cache
READ_MEM	Thực hiện fetch Cache Block từ Main Memory
WRITE_MEM	Thực hiện Write-back Block xuống Main Memory
REQ_BUS	Yêu cầu fetch hoặc invalid Block ở Cache khác
RESP_BUS	Thực hiện fetch hoặc invalid Block cho Cache khác

Cache được thiết kế để hoạt động dựa theo sự điều khiển của Cache Controller, Bảng 4.1 bên trên và Hình 4.1 bên dưới là chi tiết sơ đồ hoạt động của máy trạng thái Cache và chức năng của từng trạng thái được sử dụng.



Hình 4.1: Sơ đồ máy trạng thái Cache

Ta có thiết kế hệ thống D-Cache bao gồm các tín hiệu thể hiện trong Hình 4.2 và Bảng 4.2.



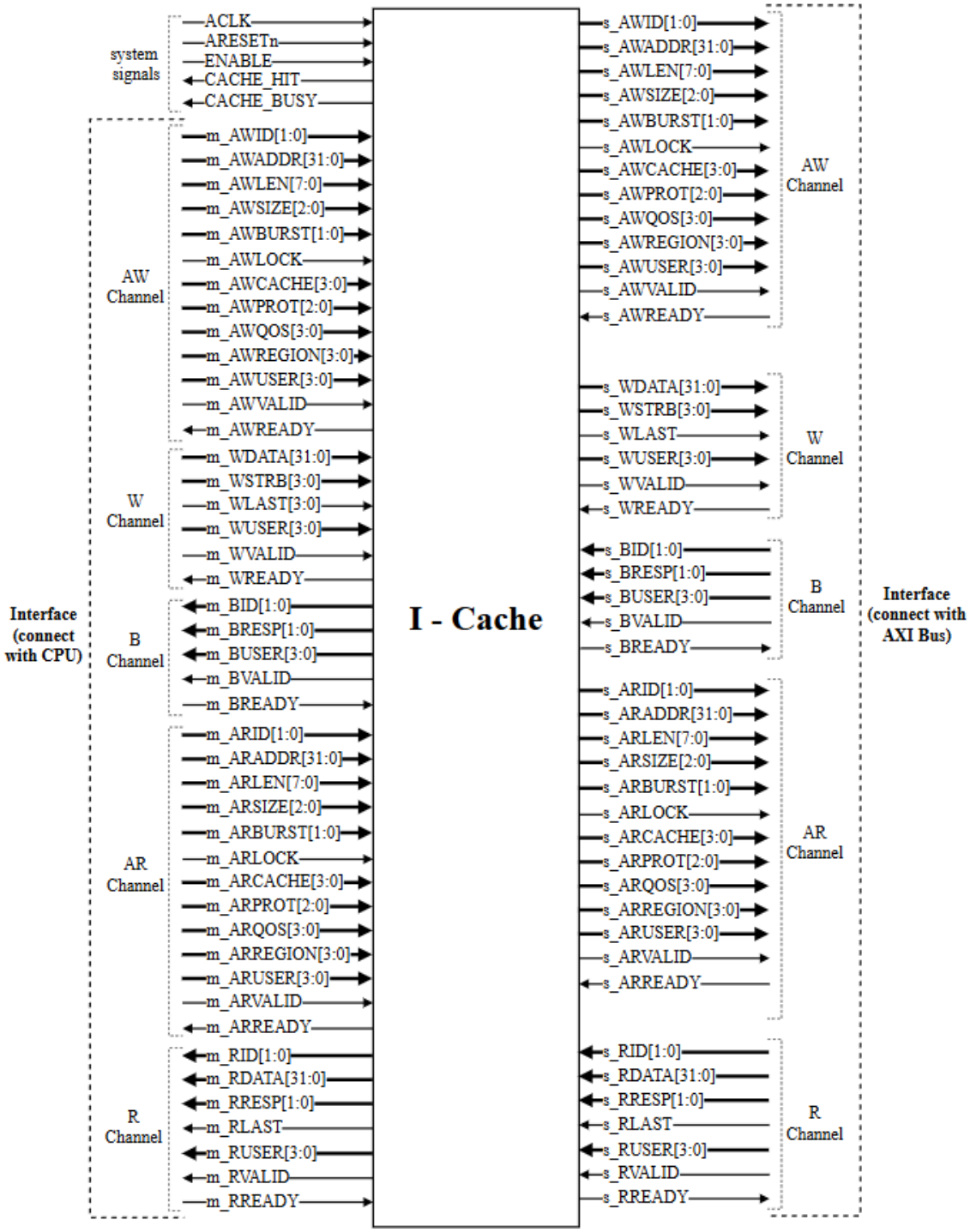
Hình 4.2: Tổng quan tín hiệu D-Cache

Bảng 4.2: Mô tả tín hiệu của D-Cache

STT	Tín hiệu	I/O	Số bit	Mô tả
Tín hiệu input hệ thống				
1	ACLK	I	1	Tín hiệu xung clock
2	ARESETn	I	1	Tín hiệu reset tích cực mức thấp
3	ENABLE	I	1	Tín hiệu kích hoạt cache
4	CACHE_HIT	O	1	Tín hiệu thông báo cache hit hay không
5	CACHE_BUSY	O	1	Tín hiệu thông báo cache đang bận
Nhóm kênh AXI thực hiện giao tiếp với CPU				
6	AW Channel	I/O		Kênh truyền địa chỉ ghi dữ liệu
7	W Channel	I/O		Kênh truyền dữ liệu để ghi
8	B Channel	I/O		Kênh phản hồi việc ghi dữ liệu
9	AR Channel	I/O		Kênh truyền địa chỉ dữ liệu được yêu cầu đọc
10	R Channel	I/O		Kênh truyền dữ liệu yêu cầu đọc được trả về
Nhóm kênh AXI thực hiện giao tiếp với AXI Bus				
11	AW Channel	I/O		Kênh truyền địa chỉ ghi dữ liệu
12	W Channel	I/O		Kênh truyền dữ liệu để ghi
13	B Channel	I/O		Kênh phản hồi việc ghi dữ liệu
14	AR Channel	I/O		Kênh truyền địa chỉ dữ liệu được yêu cầu đọc
15	R Channel	I/O		Kênh truyền dữ liệu yêu cầu đọc được trả về
Nhóm kênh ACE thực hiện hỗ trợ Snoop dữ liệu				
16	AC Channel	I/O		Kênh truyền địa chỉ để snoop yêu cầu truy cập
17	CR Channel	I/O		Kênh phản hồi trạng thái snoop
18	CD Channel	I/O		Kênh snoop dữ liệu

Hình 4.2 cho thấy các kênh giao tiếp thuộc giao thức AXI được sử dụng để khối Data Cache có thể dễ dàng truyền nhận dữ liệu với CPU và AXI Bus mở rộng ACE. Đặc biệt là các kênh ACE hỗ trợ việc Snooping [18], hỗ trợ việc snoop các yêu cầu truy cập dữ liệu, đảm bảo đồng nhất dữ liệu giữa các bộ nhớ đệm.

Hình 4.3 và Bảng 4.3 bên dưới là thiết kế tổng quan và danh sách các tín hiệu được sử dụng cho I-Cache. Các nhóm kênh giao tiếp của giao thức AXI cũng được sử dụng để kết nối với CPU và AXI Bus.



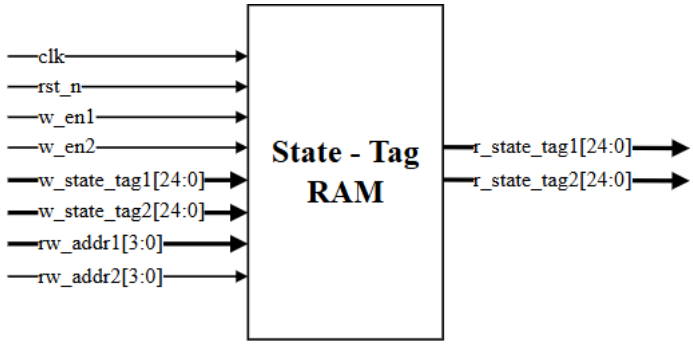
Hình 4.3: Tổng quan tín hiệu I-Cache

Bảng 4.3: Mô tả tín hiệu của I-Cache

STT	Tín hiệu	I/O	Số bit	Mô tả
Tín hiệu input hệ thống				
1	ACLK	I	1	Tín hiệu xung clock
2	ARESETn	I	1	Tín hiệu reset tích cực mức thấp
3	ENABLE	I	1	Tín hiệu kích hoạt cache
4	CACHE_HIT	O	1	Tín hiệu thông báo cache hit hay không
5	CACHE_BUSY	O	1	Tín hiệu thông báo cache đang bận
Nhóm kênh AXI thực hiện giao tiếp với CPU				
6	AW Channel	I/O		Kênh truyền địa chỉ ghi lệnh
7	W Channel	I/O		Kênh truyền lệnh để ghi
8	B Channel	I/O		Kênh phản hồi việc ghi lệnh
9	AR Channel	I/O		Kênh truyền địa chỉ lệnh được yêu cầu đọc
10	R Channel	I/O		Kênh truyền lệnh yêu cầu đọc được trả về
Nhóm kênh AXI thực hiện giao tiếp với AXI Bus				
11	AW Channel	I/O		Kênh truyền địa chỉ ghi lệnh
12	W Channel	I/O		Kênh truyền lệnh để ghi
13	B Channel	I/O		Kênh phản hồi việc ghi lệnh
14	AR Channel	I/O		Kênh truyền địa chỉ lệnh được yêu cầu đọc
15	R Channel	I/O		Kênh truyền lệnh yêu cầu đọc được trả về

- **Khối State – Tag RAM**

Khối State-Tag RAM là nơi lưu trữ các bit phục vụ cho giao thức MOESI và các bit Tag của địa chỉ vật lý. Cụ thể trong thiết kế sẽ có 3-bit [24:22] là các bit ứng với các trạng thái M (Modified), O (Owned), E (Exclusive), S (Shared), I (Invalid) và 22-bit Tag [21:0]. Chi tiết được trình bày trong Hình 4.4, Bảng 4.4.



Hình 4.4: Khối State-Tag RAM

Bảng 4.4: Mã hóa các trạng thái trong giao thức MOESI

Trạng thái	Giá trị
M (Modified)	000
O (Owned)	001
E (Exclusive)	010
S (Shared)	011
I (Invalid)	100

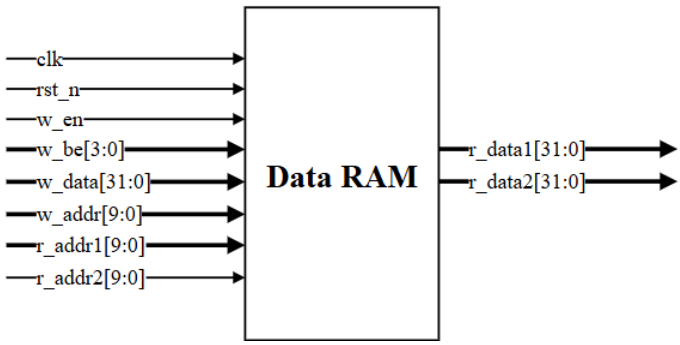
Bảng 4.5 bên dưới là danh sách các tín hiệu được sử dụng trong khối State-Tag RAM có thể liên tục cập nhật trạng thái của khối dữ liệu được truy cập với 2 port được hỗ trợ.

Bảng 4.5: Tín hiệu của khối State-Tag RAM

STT	Tín hiệu	I/O	Số bit	Mô tả
1	clk	I	1	Tín hiệu xung clock
2	rst_n	I	1	Tín hiệu reset tích cực mức thấp
3	w_en1	I	1	Tín hiệu write enable
4	w_en2	I	1	Trạng thái cần ghi
5	w_state_tag1[24:0]	I	4	Địa chỉ đọc port 1
6	w_state_tag2[24:0]	I	4	Địa chỉ đọc port 2
7	rw_addr1	O	3	Trạng thái đọc từ port 1
8	rw_addr2	O	3	Trạng thái đọc từ port 2
9	r_state_tag1[24:0]	O	24	Tag đọc từ port 1
10	r_state_tag2[24:0]	O	24	Tag đọc từ port 2

- **Khối Data RAM**

Khối Data RAM là khối có chức năng lưu trữ dữ liệu của D-Cache bao gồm 64x64 byte dữ liệu. Chi tiết các tín hiệu được trình bày trong Hình 4.5 và Bảng 4.6.



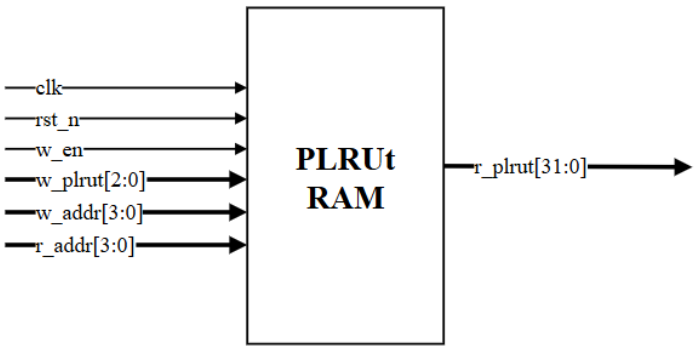
Hình 4.5: Khối Data RAM

Bảng 4.6: Tín hiệu của khối Data RAM

STT	Tín hiệu	I/O	Số bit	Mô tả
1	clk	I	1	Tín hiệu xung clock
2	rst_n	I	1	Tín hiệu reset tích cực mức thấp
3	wr_en	I	1	Tín hiệu write enable
4	wr_data	I	32	Dữ liệu ghi
5	wr_addr	I	10	Địa chỉ ghi
6	rd_addr1	I	10	Địa chỉ đọc port 1
7	rd_addr2	I	10	Địa chỉ đọc port 2
8	rd_data1	O	32	Dữ liệu đọc từ port 1
9	rd_data2	O	32	Dữ liệu đọc từ port 2

- **Khối PLRUt RAM**

Khối PLRUt RAM có chức năng lưu trữ các bit PLRUt để phục vụ giải thuật thay thế Cache Block. Khối PLRUt bao gồm 16x3 bit như Hình 4.6 và Bảng 4.7.

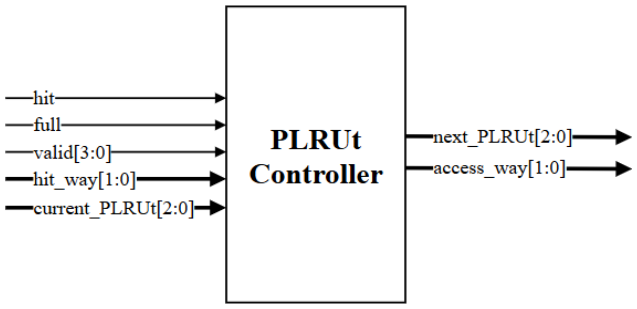


Hình 4.6: Khối PLRUt RAM

Bảng 4.7: Tín hiệu của khối PLRUt RAM

STT	Tín hiệu	I/O	Số bit	Mô tả
1	clk	I	1	Tín hiệu xung clock
2	rst_n	I	1	Tín hiệu reset tích cực mức thấp
3	w_en	I	1	Tín hiệu write enable
4	w_plrut	I	3	Các bit PLRUt cần ghi
5	w_addr	I	4	Địa chỉ ghi
6	r_addr	I	4	Địa chỉ đọc
7	r_plrut	O	3	Các bit PLRUt đọc được

- **Khối PLRUt Controller**



Hình 4.7: Khối PLRUt Controller

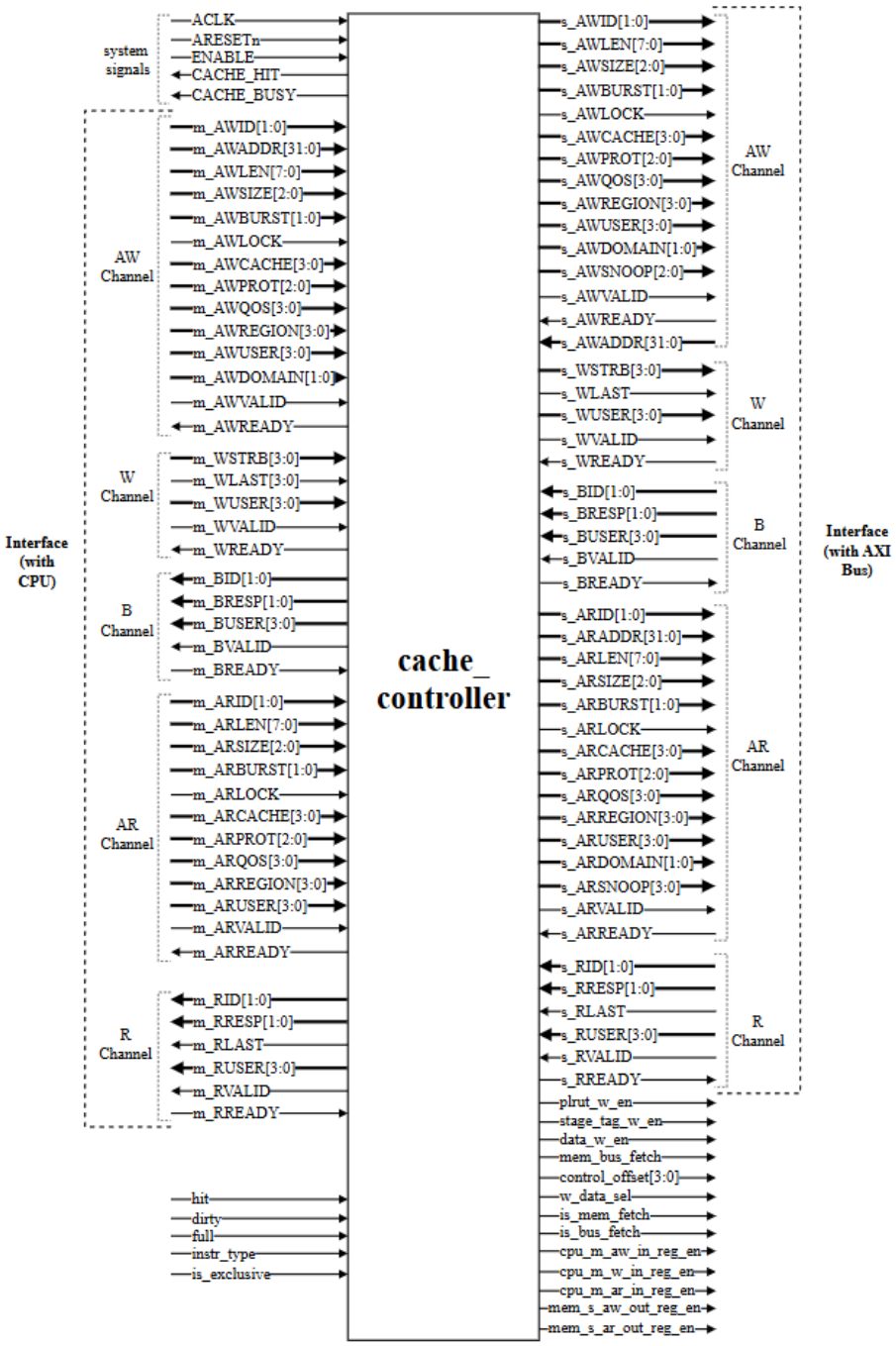
Khối PLRUt Controller điều khiển các chức năng chuyển đổi, cập nhật các bit PLRUt sau mỗi lần truy cập bộ nhớ, thực hiện tìm kiếm Block nạn nhân ("victim block") để thay thế. Chi tiết các tín hiệu trình bày trong Hình 4.7, Bảng 4.8.

Bảng 4.8: Tín hiệu của khối PLRUt Controller

STT	Tín hiệu	I/O	Số bit	Mô tả
1	hit	I	1	Tín hiệu báo truy cập hit
2	full	I	1	Tín hiệu báo Set đầy
3	valid	I	4	Tín hiệu báo Block tồn tại
4	hit_way[1:0]	I	2	Way được truy cập hit
5	Current_PLRUt	I	3	Các bit PLRUt hiện tại
6	Next_PLRUt	O	3	Các bit PLRUt sau khi truy cập
7	access_way	O	2	Way được chọn để truy cập

- **Khối Cache Controller**

Khối Cache Controller là máy trạng thái có chức năng điều khiển các truy cập, điều khiển các khối PLRUt Controller, MOESI Controller, phản hồi với CPU, giao tiếp với Cache khác thông qua Bus, giao tiếp với bộ nhớ chính điều khiển toàn bộ hoạt động của hệ thống. Chi tiết các tín hiệu trình bày trong Hình 4.8 và Bảng 4.9.



Hình 4.8: Khối Cache Controller

Bảng 4.9: Tín hiệu của khối Cache Controller

STT	Tín hiệu	I/O	Số bit	Mô tả
Tín hiệu điều khiển hệ thống				
1	ACLK	I	1	Tín hiệu xung clock
2	ARESETn	I	1	Tín hiệu reset tích cực mức thấp
3	ENABLE	I	1	Tín hiệu kích hoạt cache
4	CACHE_HIT	O	1	Tín hiệu thông báo cache hit hay không
5	CACHE_BUSY	O	1	Tín hiệu thông báo cache đang bận
Nhóm kênh AXI thực hiện giao tiếp với CPU				
6	AW Channel	I/O		Kênh truyền địa chỉ ghi dữ liệu
7	W Channel	I/O		Kênh truyền dữ liệu để ghi
8	B Channel	I/O		Kênh phản hồi việc ghi dữ liệu
9	AR Channel	I/O		Kênh truyền địa chỉ dữ liệu được yêu cầu đọc
10	R Channel	I/O		Kênh truyền dữ liệu yêu cầu đọc được trả về
Nhóm kênh AXI thực hiện giao tiếp với AXI Bus				
11	AW Channel	I/O		Kênh truyền địa chỉ ghi dữ liệu
12	W Channel	I/O		Kênh truyền dữ liệu để ghi
13	B Channel	I/O		Kênh phản hồi việc ghi dữ liệu
14	AR Channel	I/O		Kênh truyền địa chỉ dữ liệu được yêu cầu đọc
15	R Channel	I/O		Kênh truyền dữ liệu yêu cầu đọc được trả về
Các tín hiệu điều khiển khối Datapath				
16	hit	1		Tín hiệu báo truy cập hit
17	dirty	1		Tín hiệu báo dữ liệu đã bị chỉnh sửa
18	full	1		Tín hiệu báo Set đầu
19	instr_type	1		Loại truy cập (Đọc/Ghi)
20	is_exclusive	1		Dữ liệu truy cập ở trạng thái Exclusive không
21	plrut_w_en	1		Tín hiệu chi phép ghi các bit PLRUt
22	state_tag_w_en	1		Tín hiệu cho phép ghi các bit State – Tag
23	data_w_en	1		Tín hiệu cho phép ghi dữ liệu

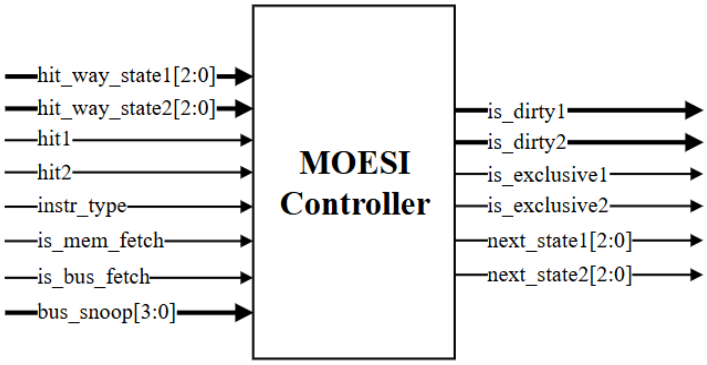
24	mem_bus_fetch	1		Tín hiệu yêu cầu fetch dữ liệu
25	control_offset[3:0]	4		Các bit offset của địa chỉ truy cập
26	w_data_sel	1		Tín hiệu lựa chọn dữ liệu ghi
27	is_mem_fetch	1		Tín hiệu yêu cầu fetch dữ liệu từ Mem
28	is_bus_fetch	1		Tín hiệu yêu cầu fetch dữ liệu từ Bus
29	cpu_m_aw_in_reg_en	1		Cho phép địa chỉ ghi kênh AW(CPU→ Cache)
30	cpu_m_w_in_reg_en	1		Cho phép dữ liệu ghi kênh W (CPU → Cache)
31	cpu_m_ar_in_reg_en	1		Cho phép địa chỉ đọc kênh AR(CPU→Cache)
32	mem_s_aw_out_reg_en	1		Cho phép địa chỉ ghi kênh AW(Mem→ Cache)
33	mem_s_ar_out_reg_en	1		Cho phép dữ liệu ghi kênh R (Mem→Cache)

4.1.2. Cài đặt giao thức MOESI

Giao thức MOESI (Modified, Owned, Exclusive, Shared, Invalid) là một giao thức Cache Coherence được thiết kế để duy trì tính nhất quán dữ liệu trong các hệ thống đa vi xử lý. Nó mở rộng giao thức MESI bằng cách thêm trạng thái "Owned", cho phép một Cache lưu trữ dữ liệu sửa đổi mà không cần ghi lại ngay vào bộ nhớ chính, từ đó tối ưu hóa hiệu suất và giảm độ trễ truy cập bộ nhớ. Giao thức MOESI sử dụng cơ chế Snooping để đảm bảo tất cả các bộ nhớ đệm trong hệ thống đều có cùng một phiên bản dữ liệu chính xác.

- **Khối MOESI Controller**

Khối MOESI Controller là khối phục vụ thực hiện giao thức đảm bảo Cache Coherence MOESI Snoopy. Khối này có chức năng chuyển đổi trạng thái của các Block sau các lần truy cập bằng cách nhận vào các tín hiệu điều khiển từ khối Cache Controller và các tín hiệu truy cập từ hệ thống, đồng thời kiểm tra thông báo cho Cache Controller xem liệu rằng Block vừa truy cập đã được chỉnh sửa hay chưa (Dirty). Chi tiết các tín hiệu được trình bày trong Hình 4.9 và Bảng 4.10.



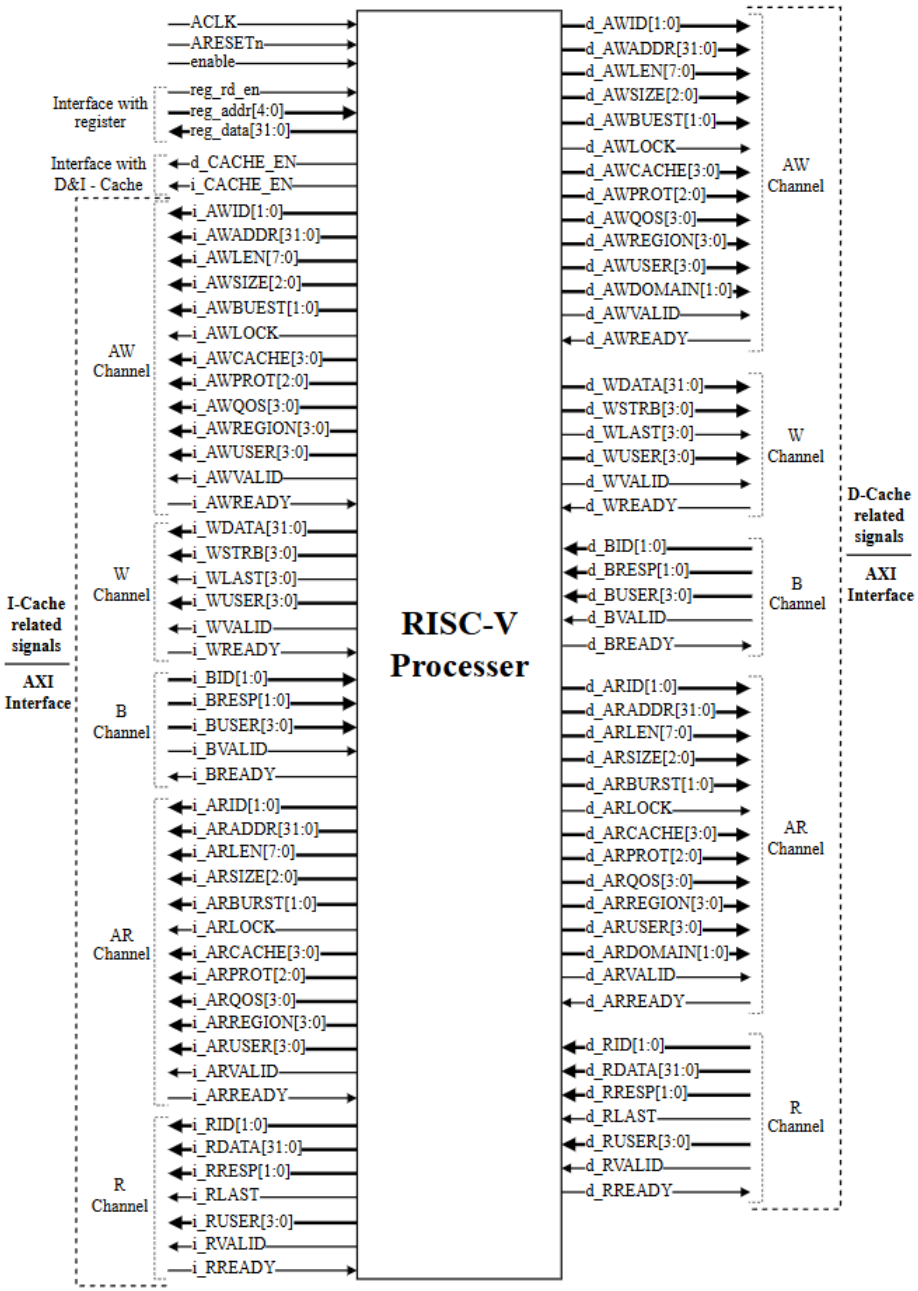
Hình 4.9: Khối MOESI Controller

Bảng 4.10: Tín hiệu của khối MOESI Controller

STT	Tín hiệu	I/O	Số bit	Mô tả
1	hit_way_state1[2:0]	I	3	Trạng thái của Way được truy cập port 1
2	hit_way_state[2:0]	I	3	Trạng thái của Way được truy cập port 2
3	hit1	I	1	Tín hiệu báo truy cập hit cho port 1
4	hit2	I	1	Tín hiệu báo truy cập hit cho port 2
5	instr_type	I	1	Loại truy cập (Đọc/Ghi)
6	is_mem_fetch	I	1	Tín hiệu yêu cầu fetch từ Mem
7	is_bus_fetch	I	1	Tín hiệu yêu cầu fetch từ Bus
8	bus_snoop[3:0]	I	4	Tín hiệu snoop từ bus
9	is_dirty1	I	1	Tín hiệu báo bit dirty cho port 1
10	is_dirty2	I	1	Tín hiệu báo bit dirty cho port 2
11	is_exclusive1	O	1	Tín hiệu báo trạng thái Exclusive cho port 1
12	is_exclusive2	O	1	Tín hiệu báo trạng thái Exclusive cho port 2
13	next_state1[2:0]	O	3	Trạng thái tiếp theo cho port 1
14	next_state2[2:0]	o	3	Trạng thái tiếp theo cho port 2

4.1.3. Thiết kế phần cứng lõi xử lý RISC-V RV32I

Lõi xử lý trong hệ thống được thiết kế theo kiến trúc RISC-V RV32I có áp dụng kiến trúc pipeline 5 tầng và tích hợp AXI wrapper để giao tiếp với I-Cache và D-Cache như trong Hình 4.10 và Bảng 4.11.



Hình 4.10: Lõi xử lý RISC-V RV32I

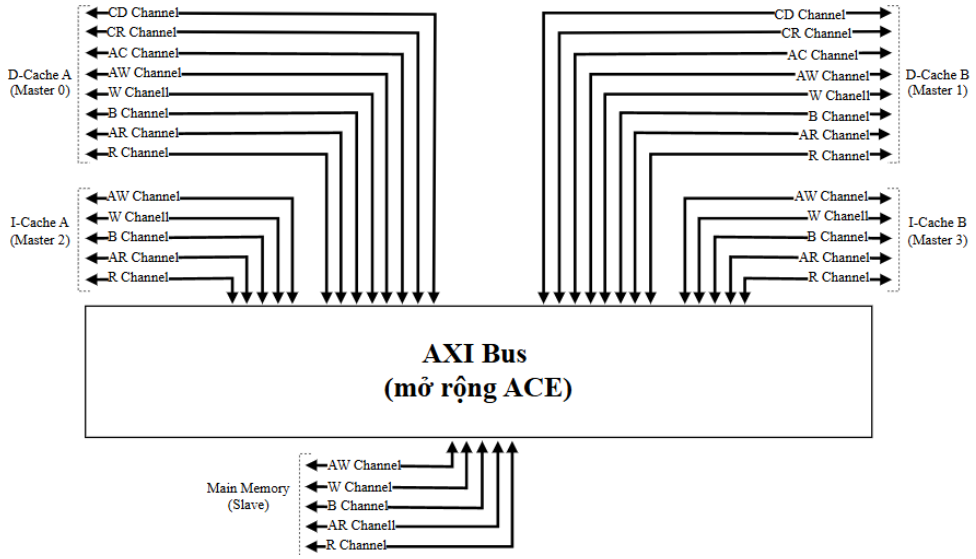
Bảng 4.11: Tín hiệu sử dụng trong khối RISC-V Processor

STT	Tín hiệu	I/O	Số bit	Mô tả
1	ACLK	I	1	Tín hiệu xung clock
2	ARESETn	I	1	Tín hiệu reset tích cực mức thấp
3	Enable	I	1	Tín hiệu kích hoạt cache
4	reg_rd_en	I	1	Tín hiệu cho phép thanh ghi đọc

5	reg_addr[4:0]	I	5	Địa chỉ đọc thanh ghi
6	reg_data[31:0]	O	32	Dữ liệu đọc từ thanh ghi
Nhóm kênh AXI thực hiện giao tiếp với I – Cache				
7	AW Channel	I/O		Kênh truyền địa chỉ ghi dữ liệu
8	W Channel	I/O		Kênh truyền dữ liệu để ghi
9	B Channel	I/O		Kênh phản hồi việc ghi dữ liệu
10	AR Channel	I/O		Kênh truyền địa chỉ dữ liệu được yêu cầu đọc
11	R Channel	I/O		Kênh truyền dữ liệu yêu cầu đọc được trả về
Nhóm kênh AXI thực hiện giao tiếp với D – Cache				
12	AW Channel	I/O		Kênh truyền địa chỉ ghi dữ liệu
13	W Channel	I/O		Kênh truyền dữ liệu để ghi
14	B Channel	I/O		Kênh phản hồi việc ghi dữ liệu
15	AR Channel	I/O		Kênh truyền địa chỉ dữ liệu được yêu cầu đọc
16	R Channel	I/O		Kênh truyền dữ liệu yêu cầu đọc được trả về

4.1.4. Thiết kế phần cứng AXI Bus mở rộng ACE

AXI Bus được thiết kế theo giao thức AXI, hỗ trợ giải quyết giao tiếp cho 4 Master gồm I-Cache, D-Cache của 2 lõi và 1 Slave là bộ nhớ chính. Đồng thời, AXI Bus còn được tích hợp thêm một số kênh giao tiếp mở rộng từ giao thức ACE nhằm hỗ trợ việc snoop yêu cầu và dữ liệu trong Cache Coherence như Hình 4.11.



Hình 4.11: Cấu trúc AXI Bus mở rộng ACE

4.2. Thiết kế chi tiết phần mềm

4.2.1. Thiết kế phần mềm Cache

Vì đây là phần trọng tâm của đề tài nên nhóm đã tập trung vào thiết kế Cache theo kiến trúc Multiway Set Associative Cache với 4-Way và 16-Set, hỗ trợ lưu trữ 4KB. Việc thiết kế Cache với dung lượng vừa phải giúp cân bằng giữa tỷ lệ hit/miss và yêu cầu về tốc độ cao của thiết kế.

Như đã trình bày ở CHƯƠNG 2, Cache được tích hợp các giải thuật và giao thức sau:

- PLRUt: Tree-based Pseudo Least Recently Used (PLRUt) là giải thuật xấp xỉ LRU (Least Recently Used) để tăng tốc hiệu suất và giảm độ phức tạp của việc triển khai.
- Read Allocate: Bộ nhớ đệm sẽ thực hiện cấp phát và lưu trữ lại Block khi xuất hiện truy cập Read vào Block đó (xảy ra lần đầu truy cập Block đó và miss).
- Write Allocate: Bộ nhớ đệm sẽ thực hiện cấp phát và lưu trữ lại Block khi xuất hiện truy cập Write vào Block đó (xảy ra lần đầu truy cập Block đó và miss).
- Write-back: dữ liệu được ghi vào bộ nhớ đệm khi Cache Block bị thay thế và đã bị chỉnh sửa, cần có bit Dirty (D) để theo dõi dữ liệu đã bị chỉnh sửa.
- MOESI: MOESI là giao thức duy trì tính nhất quán bộ nhớ đệm trong các hệ thống đa vi xử lý, mở rộng từ MESI bằng cách thêm trạng thái Owned (O). Trạng thái này cho phép một bộ nhớ đệm sở hữu dữ liệu hợp lệ và chia sẻ trực tiếp với các bộ nhớ đệm khác mà không cần truy cập bộ nhớ chính. Điều này giúp giảm số lần truy cập bộ nhớ, tăng hiệu suất và tối ưu hóa thời gian xử lý. MOESI hỗ trợ chia sẻ dữ liệu đã chỉnh sửa (Dirty data) mà bộ nhớ chính chưa cập nhật.

Cache được thiết kế phần mềm để hoạt động dựa theo sự điều khiển của Cache Controller, sơ đồ máy trạng thái và chức năng cụ thể của từng trạng thái tương tự như thiết kế phần cứng đã được trình bày tại Hình 4.1 và Bảng 4.1.

Các hàm chức năng được sử dụng trong lớp Cache được trình bày trong Bảng 4.12.

Bảng 4.12: Chức năng các hàm trong lớp Cache

Hàm	**Chức năng**
get_word_offset()	Lấy 4-bit Word Offset từ giao dịch
get_set()	Lấy 4-bit Set từ giao dịch
get_tag()	Lấy 22-bit tag từ giao dịch
is_other_cache_hit()	Kiểm tra xem block có trong Other Cache
fcfs_arbiter()	Bộ phân xử Arbiter theo giải thuật FCFS kết hợp RR
receive_master()	Nhận Read/Write Request từ Master
read_handler()	Thực hiện Read từ Cache
write_handler()	Thực hiện Write từ Cache
send_master()	Gửi Read/Write Response cho Master
send_slave()	Gửi Read/Write Request cho Slave
receive_slave()	Nhận Read/Write Response từ Slave
send_other_cache()	Gửi Snoop Request/Response cho Other Cache
receive_other_cache()	Nhận Snoop Request/Response từ Other Cache
deadlock_solve()	Phát hiện và giải quyết Deadlock
print_cache_plrut_mem()	In ra các bit PLRUt của các block
print_cache_state()	In ra trạng thái các block trong Cache
print_cache_tag()	In ra tag của các block
print_cache_data()	In dữ liệu được lưu trữ trong Cache
update()	Cập nhật trạng thái tiếp theo của Cache
run()	Hàm gọi có hàm trên để thực hiện chạy Cache

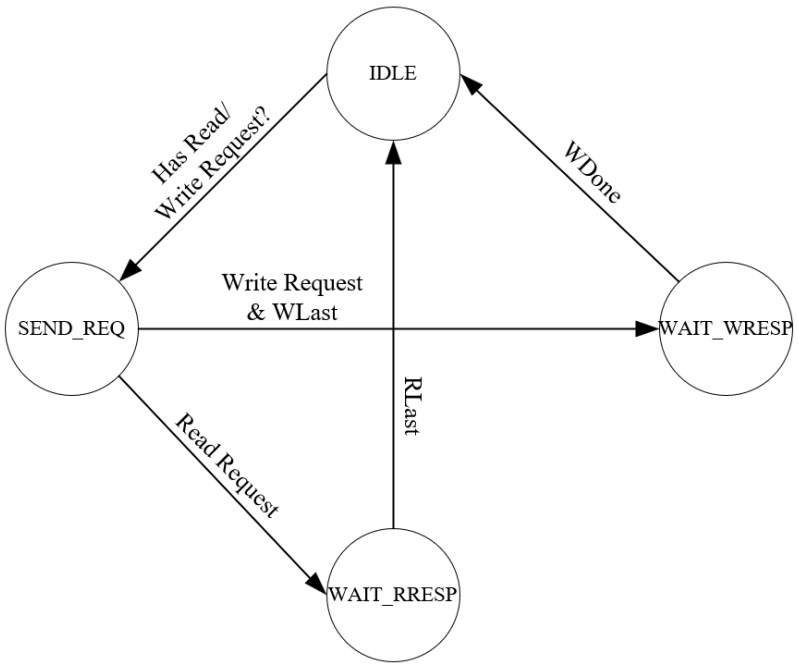
Lớp Cache được thiết kế phân cấp gồm nhiều lớp (Class) con, chi tiết trong Bảng 4.13.

Bảng 4.13: Định nghĩa các lớp con thuộc lớp Cache

Lớp	Định nghĩa
STATE_TAG_MEM	Lưu trữ và xử lý state và tag của các block
DATA_MEM	Lưu trữ dữ liệu
PLRUT_MEM	Lưu trữ các PLRUt-bit cho Cache 4-Way
PLRUT_CONTROLLER	Thực hiện giải thuật PLRUt cho Cache 4-Way
MOESI_CONTROLLER	Thực hiện MOESI Protocol

4.2.2. Thiết kế phần mềm CPU

Đối với CPU, để đơn giản thiết kế và phục vụ tốt cho quá trình mô phỏng đánh giá hiệu suất, nhóm thiết kế CPU thực hiện các lệnh LoadWord (Read), StoreWord (Write) nhằm tăng tối đa số truy cập bộ nhớ . Hình 4.12 thể hiện máy trạng thái của thiết kế phần mềm CPU.



Hình 4.12: Sơ đồ máy trạng thái CPU

Ở từng trạng thái, CPU thực hiện các nhiệm vụ cụ thể được trình bày trong Bảng 4.14.

Bảng 4.14: Chức năng của các trạng thái trong CPU

Trạng thái	Chức năng
IDLE	Trạng thái rảnh không hoạt động
SEND_REQ	Gửi Read/Write Request cho Slave (D-Cache)
WAIT_RRESP	Đợi Read Response từ Slave (D-Cache)
WAIT_WRESP	Đợi Write Response từ Slave (D-Cache)

Trong đó, CPU sẽ thực hiện đọc tệp tin đầu vào để load lệnh lên bộ nhớ lệnh, sau đó sẽ tiến hành gửi lệnh xuống cho Cache. Các lệnh được CPU gửi xuống Cache thực hiện Read/Write 1-word, sau khi đã gửi Read/Write Request, CPU phải đợi Response thì mới được xem là hoàn thành một lệnh. Chức năng các hàm trong lớp Processor được trình bày trong Bảng 4.15.

Bảng 4.15: Chức năng các hàm sử dụng trong lớp Processor

Hàm	Chức năng
init()	Khởi tạo CPU, thực hiện load lệnh
convert2local_addr()	Chuyển đổi địa chỉ global thành địa chỉ local
send_slave()	Gửi Request cho Slave (Cache)
recv_slave()	Nhận Response từ Slave (Cache)
update()	Cập nhật trạng thái tiếp theo của CPU
run()	Gọi các hàm trên để tiến hành chạy CPU

4.2.3. Thiết kế phần mềm AXI Bus mở rộng ACE

Interconnect là kiến trúc mạng phức tạp liên kết các thành phần trong Hệ thống trên Chip (System on Chip) như CPU, GPU, bộ nhớ và thiết bị ngoại vi để trao đổi dữ liệu hiệu quả. Qua thời gian, kiến trúc này đã phát triển từ bus đơn giản đến mạng liên kết phức tạp như Crossbars và NoC (Network on Chip) để đáp ứng

nhu cầu băng thông và hiệu suất cao. Interconnect đóng vai trò quan trọng trong tối ưu hóa hiệu suất và tiết kiệm năng lượng của hệ thống chip.

Tuy nhiên, do giới hạn thời gian thực hiện và trọng tâm đề tài tập trung đi sâu vào nghiên cứu và phát triển Multiway Set Associative Cache hỗ trợ MOESI Coherency Protocol nên nhóm không đi sâu vào thiết kế phần mềm AXI Bus. Thay vào đó, AXI Bus được thiết kế dưới dạng thiết kế Mux-Demux và tích hợp bộ phân xử FCFS kết hợp RR Arbiter. Các hàm trong lớp AXI Bus được trình bày trong Bảng 4.16.

Bảng 4.16: Chức năng các hàm trong lớp AXI Bus

Hàm	**Chức năng**
fcfs_arbiter()	Bộ phân xử Arbiter theo giải thuật FCFS kết hợp RR
receive_master()	Nhận Read/Write Request từ Master
send_master()	Gửi Read/Write Response cho Master
send_slave()	Gửi Read/Write Request cho Slave
receive_slave()	Nhận Read/Write Response từ Slave
update()	Cập nhật trạng thái các Port
run()	Hàm gọi có hàm trên để chạy Interconnect

4.2.4. Thiết kế phần mềm Main Memory

Bộ nhớ chính được thiết kế phần mềm với các chức năng mô phỏng hoạt động của Block RAM trên phần mềm Vivado, có thể giao tiếp với các thành phần khác trong hệ thống thông qua giao thức AXI. Bảng 4.17 là mô tả chức năng các hàm được sử dụng khi thực hiện thiết kế phần mềm bộ nhớ chính.

Bảng 4.17: Chức năng các hàm trong lớp Main Memory

Hàm	**Chức năng**
run()	Thực hiện chạy mô phỏng thiết kế.
update()	Cập nhật lại các giá trị, trạng thái bên trong **Controller** và update Port nhận/gửi của chính nó.

process()	Tiến hành chạy thực thi các hàm xử lý yêu cầu
print_mem()	In ra màn hình dữ liệu trong Bộ nhớ
add_request()	Tiếp nhận và thêm các giao dịch vào trong S0/S1 source, Write queue và Read queue.
state_process()	Xử lý và chuyển đổi trạng thái của máy trạng thái. Trong mỗi trạng thái sẽ gửi các tín hiệu phù hợp điều khiển hoạt động quản lý và truy cập dữ liệu của Bộ nhớ.
update_state()	Cập nhật trạng thái của máy trạng thái
update()	Cập nhật lại các Port gửi sau khi gửi đi các giao dịch ghi và giao dịch đọc.
process()	Thực thi quản lý và truy cập thao tác dữ liệu.
read_write()	Thực hiện xử lý với lệnh Đọc/Ghi
activate()	Thực hiện xử lý với lệnh Activate
precharge()	Thực hiện xử lý với lệnh Precharge

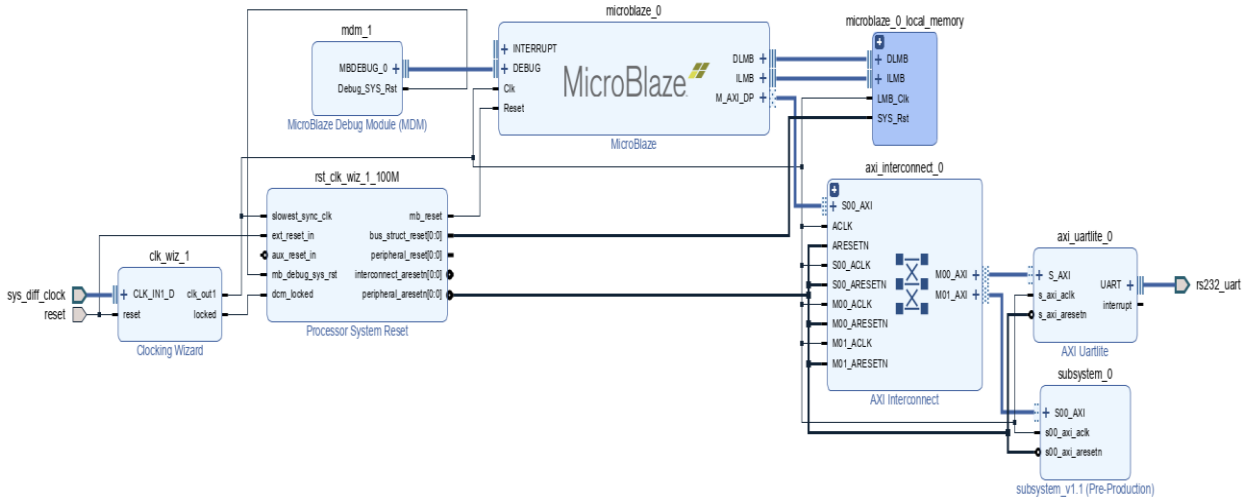
4.3. Thiết kế Block Design trên Vivado với giao thức AXI

4.3.1. Kiến trúc thiết kế Block Design giao tiếp qua AXI

Việc thiết kế và hiện thực trên FPGA được tiến hành trên phần mềm Vivado 2020.2 và phần mềm Vitis 2022.2 cùng với kit Xilinx Virtex-7 VC707. Thiết kế trong Hình 4.13 được thực hiện gồm những khối sau:

- Khối IP Clocking Wizard: Có chức năng tính toán và tạo ra tần số cần thiết cho thiết kế, cụ thể là 100MHz.
- Khối IP Processor System Reset: Cung cấp cơ chế reset.
- Khối IP MicroBlaze: Có chức năng nạp lệnh vào Main Memory.
- Khối IP Local Memory: Bao gồm DLMB chứa dữ liệu và ILMB chứa lệnh phục vụ cho khối MicroBlaze.

- Khối IP Debug Module: Hỗ trợ debug.
- Khối IP AXI Interconnect: Dùng để kết nối các thành phần trong Block Design.
- Khối IP AXI Uartlite: Dùng để nhập/xuất giá trị ra máy tính thông qua giao tiếp với IP Subsystem và khối MicroBlaze.
- Khối IP Subsystem: Đây là thiết kế của đề tài, được đóng gói lại ở dạng một AXI Slave.



Hình 4.13: Thiết kế Block Design trên Vivado

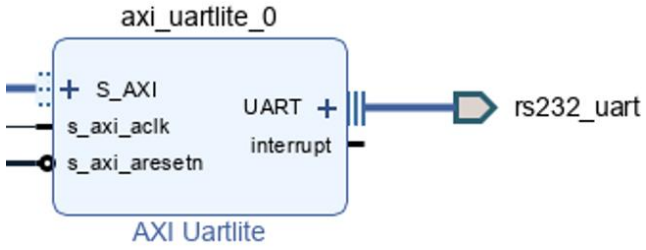
4.3.2. Các IP chính của Xilinx trong thiết kế Block Design

Với IP Microblaze: Phần mềm Vivado hỗ trợ IP Microblaze để khởi tạo giao diện có nhiệm vụ giao tiếp với IP Local Memory và IP Debug, hỗ trợ người dùng lập trình và cấu hình thiết kế thông qua code C.

Hình 4.14: IP Microblaze giao tiếp với IP Local Memory và IP Debug

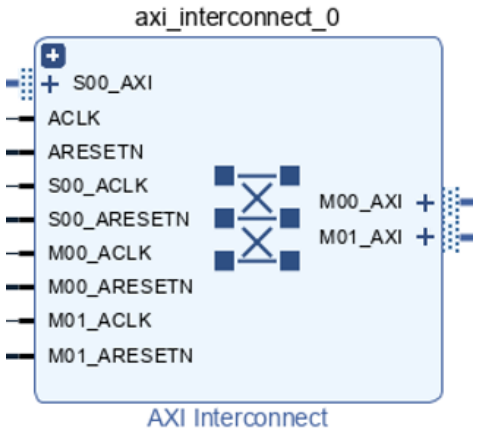
Ở Hình 4.14, khối IP Microblaze giao tiếp với IP Local Memory là một bộ nhớ BRAM được giao tiếp bởi bus LMB (Local Memory Bus) bao gồm 2 interface chính là ILMB (Instruction interface, Local Memory Bus) và DLMB (Data interface, Local Memory Bus).

Với IP AXI Uartlite: Cung cấp giao diện truyền nhận nối tiếp không đồng bộ (UART) đơn giản, được thiết kế để tích hợp với giao thức AXI4-Lite. IP này cho phép truyền dữ liệu nối tiếp giữa các thành phần thông qua bus AXI.



Hình 4.15: Khối IP AXI Uartlite

Với IP AXI Interconnect: Có nhiệm vụ kết nối các khối IP Microblaze, khối IP AXI Uartlite và khối IP Subsystem như Hình 4.13, đảm bảo phân xử và truyền nhận dữ liệu hiệu quả, chính xác.



Hình 4.16: Khối IP AXI Interconnect

CHƯƠNG 5. MÔ PHỎNG VÀ ĐÁNH GIÁ THIẾT KẾ

5.1. Kế hoạch kiểm thử

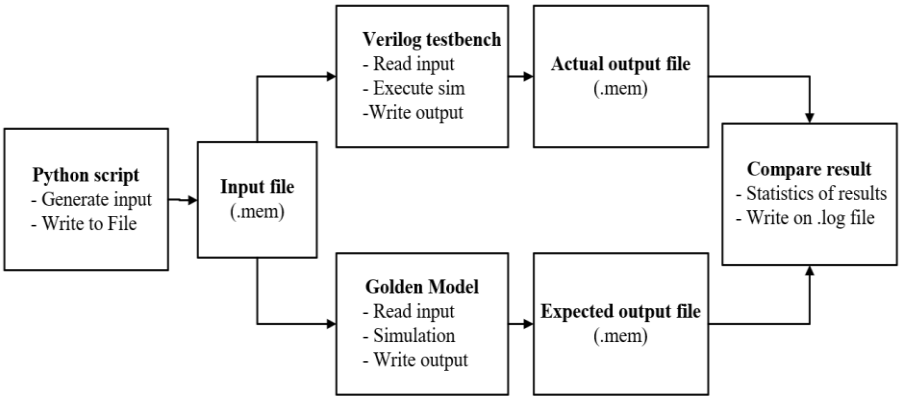
Nhằm xác minh tính đúng đắng thiết kế một cách toàn diện và khách quan, khóa luận này đã lập ra kế hoạch kiểm thử chi tiết bao gồm các nội dung như sau:

1. Kế hoạch kiểm thử lõi xử lý RISC-V RV32I.
    - Kiểm tra tính chính xác của thiết kế khi hoạt động với các nhóm lệnh.
    - Kiểm tra hoạt động của các khối chức năng thành phần bên trong.
    - Kiểm tra hoạt động của kiến trúc pipeline, các tầng được pipeline.
    - Kiểm tra việc giải quyết vấn đề Hazard với khối Hazard Unit.
2. Kiểm tra tính chính xác của AXI Bus mở rộng ACE.
    - Kiểm tra giải quyết yêu cầu giao tiếp đơn lẻ cho từng Master.
    - Kiểm tra giải quyết yêu cầu giao tiếp liên tiếp cho từng Master.
    - Kiểm tra hỗ trợ phân xử cho các Master cùng lúc yêu cầu giao tiếp.
3. Kiểm tra tính chính xác các chức năng bộ nhớ đệm (Non-Cohenrence).
    - Kiểm tra các hoạt động đọc/ghi của bộ nhớ đệm.
    - Kiểm tra hoạt động tìm kiếm và thay thế khối với giải thuật PLRUt.
    - Kiểm tra hoạt động Write-back và chính sách Read/Write Allocate.
4. Kiểm tra tính đúng đắn của giao thức MOESI Snoopy với hệ thống đa vi xử lý hỗ trợ giải quyết Cache Coherence.
    - Kiểm tra hoạt động fetch dữ liệu từ Mem/Cache và gửi/nhận snoop.
    - Kiểm tra hoạt động cập nhật trạng thái với khối MOESI Controller
    - Kiểm tra giải quyết các yêu cầu đọc/ghi đơn lẻ và liên tiếp từ CPU vào dữ liệu có địa chỉ nằm ở vùng nhớ riêng và vùng nhớ chia sẻ chung
    - Kiểm tra giải quyết các yêu cầu truy cập cùng lúc từ cả hai CPU cho nhau với địa chỉ nằm trong vùng nhớ chia sẻ chung nhưng khác nhau.
    - Kiểm tra giải quyết các yêu cầu truy cập cùng lúc từ cả hai CPU cho nhau với cùng một địa chỉ nằm trong vùng nhớ chia sẻ chung.

5.1.1. Kịch bản kiểm thử với phần mềm

Sơ đồ mô tả hệ thống kiểm thử được trình bày trong Hình 5.1. Chi tiết cách thức hoạt động của mô hình kịch bản kiểm thử như sau:

- Phần mã Python (Python script) sẽ tạo kịch bản kiểm tra và sinh ra các tệp tin đầu vào (Input file) cho các testcase đã được định sẵn trước.
- Phần mã testbench phần cứng (Verilog testbench) sẽ thực hiện đọc các tệp tin đầu vào (Input file) này, nạp vào thiết kế phần cứng, tiến hành chạy mô phỏng và ghi lại kết quả vào các tệp tin kết quả thực tế (Actual output file).
- Thiết kế phần mềm (Golden Model) được thiết kế bằng ngôn ngữ Python, thực hiện đọc các tệp tin đầu vào (Input file) sau đó tiến hành chạy mô phỏng và ghi kết quả vào các tệp tin kết quả dự kiến (Expected output file).
- Phần mã so sánh kết quả (Compare result) chịu trách nhiệm so sánh kết quả chạy giữa thiết kế phần cứng và thiết kế phần mềm, sau đó thực hiện thống kê kết quả ghi vào tệp tin result.log.



Hình 5.1: Mô hình kịch bản kiểm thử

Cấu trúc tệp tin đầu vào mẫu như Hình 5.2.

```
read    0x00071000
write   0x0007a02c      0x0c11a22e
write   0x000da81c      0xa8d48058
read    0x000ec004
```

Hình 5.2: Nội dung tệp tin đầu vào mẫu

5.1.2. Kịch bản kiểm thử với phần cứng

Do giới hạn về số trang có thể trình bày trong báo cáo, trong báo cáo khóa luận này chỉ trình bày kế hoạch kiểm thử với phần cứng các nhóm testcase điển hình, bao gồm nhiều nhất có thể các nội dung cần kiểm tra để đảm bảo thể hiện rõ tính chính xác của thiết kế với kế hoạch kiểm thử đã đề ra. Các kịch bản kiểm thử phần cứng trên phần mềm Vivado và kit Virtex-7 VC707 được chia làm 3 nhóm, các Testcase được liệt kê cụ thể trong Bảng 5.1.

Bảng 5.1: Tất cả các trường hợp Testcase thực hiện kiểm thử

Nhóm	STT	Testcase
Nhóm Testcase kiểm tra các chức năng Non-Coherence	1	Kiểm tra hoạt động Read (hit/miss), Write (hit/miss), Write-back, PLRUt, Fetch MEM, Read-Write Allocate
Nhóm Testcase kiểm tra các chức năng Coherence: Snoop Request & Response	2	Kiểm tra Rd Request - Rd Response
	3	Kiểm tra RdX Request - RdX Response
	4	Kiểm tra Invalid Request - Invalid Response
Nhóm Testcase kiểm tra chuyển trạng thái MOESI	5	Kiểm tra chuyển trạng thái: I → E, I → S, I → M
	6	Kiểm tra chuyển trạng thái: E → I, E → S, E → M
	7	Kiểm tra chuyển trạng thái: M → I, M → O
	8	Kiểm tra chuyển trạng thái: O → I, O → M
	9	Kiểm tra chuyển trạng thái: S → I, S → M

Sau khi chạy mô phỏng tất cả các Testcase đã lập ra, nhóm sẽ tiến hành so sánh kết quả thu được từ phần cứng với kết quả mong đợi để đảm bảo tính chính xác của thiết kế.

5.2. Kết quả kiểm thử thiết kế

Vì số lượng Testcase trong mỗi nhóm Testcase là khá lớn, giới hạn số lượng trang trong tài liệu này không cho phép nhóm trình bày toàn bộ tất cả các Testcase đã thực hiện kiểm thử, vì vậy nhóm xin được trình bày các Testcase tiêu biểu, điển hình và "khó" trong từng nhóm Testcase.

5.2.1. Kết quả mô phỏng phần mềm

Thực hiện chạy so sánh kết quả mô phỏng từ thiết kế phần cứng (Actual results) với kết quả của phần mềm (Expected results), sau đó thực hiện thống kế lại kết quả kiểm thử, ta có kết quả tệp tin result.log như Hình 5.3.

```
[MATCHED]    testcase_1\data_ram_A.mem
[MATCHED]    testcase_1\data_ram_B.mem
[MATCHED]    testcase_1\main_memory_result.mem
[MATCHED]    testcase_1\plrut_ram_A.mem
[MATCHED]    testcase_1\plrut_ram_B.mem
[MATCHED]    testcase_1\read_data_A.mem
[MATCHED]    testcase_1\read_data_B.mem
[MATCHED]    testcase_1\state_tag_A.mem
[MATCHED]    testcase_1\state_tag_B.mem

        ......................
        ......................

[MATCHED]    testcase_119\data_ram_A.mem
[MATCHED]    testcase_119\data_ram_B.mem
[MATCHED]    testcase_119\main_memory_result.mem
[MATCHED]    testcase_119\plrut_ram_A.mem
[MATCHED]    testcase_119\plrut_ram_B.mem
[MATCHED]    testcase_119\read_data_A.mem
[MATCHED]    testcase_119\read_data_B.mem
[MATCHED]    testcase_119\state_tag_A.mem
[MATCHED]    testcase_119\state_tag_B.mem

******************** SUMMARY ********************
 - Number of Testcases:                    119
 - Number of PASSED Testcases:             119
 - Number of FAILED Testcases:               0
 - Number of MISSED Testcases:               0
******************** END ************************
```

Hình 5.3: Kết quả mô phỏng kiểm tra bằng phần mềm

5.2.2. Kết quả mô phỏng phần cứng

- Testcase 1: Kiểm tra hoạt động Read (hit/miss), Write (hit/miss), Write-back, PLRUt, Fetch MEM, Read-Write Allocate

Bảng 5.2: Chương trình Testcase 1

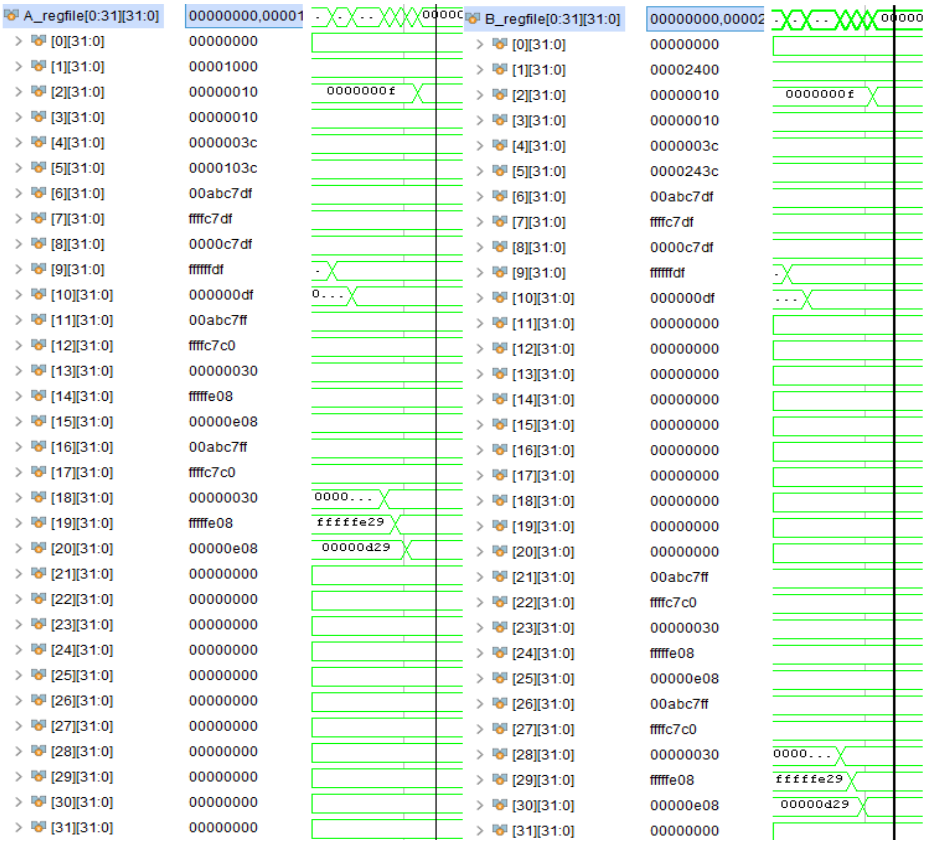
Dòng	CPU A	CPU B
1	.text	.text
2	base_addr_1:	base_addr_1:
3	addi x1, x0, 0x10	addi x1, x0, 0x24
4	slli x1, x1, 8	slli x1, x1, 8
5	addi x2, x0, 0	addi x2, x0, 0
6	addi x3, x0, 16	addi x3, x0, 16
7	store_loop1:	store_loop1:
8	beq x2, x3, base_addr_2	beq x2, x3, base_addr_2
9	lui x6, 0x00ABC	lui x6, 0x00ABC
10	ori x6, x6, 0x7EF	ori x6, x6, 0x7EF

```
11      xor     x5, x5, x6
12      andi    x5, x5, -16
13      or      x4, x5, x2
14      slli    x5, x2, 2
15      add     x6, x1, x5
16      sw      x4, 0(x6)
17      addi    x2, x2, 1
18      jal     x0, store_loop1
19  base_addr_2:
20      addi    x2, x0, 0
21      addi    x3, x0, 16
22      addi    x12, x0, -1
23  load_loop_2:
24      beq     x2, x3, base_addr_3
25      slli    x4, x2, 2
26      add     x5, x1, x4
27      lw      x6,  0(x5)
28      lh      x7,  0(x5)
29      lhu     x8,  0(x5)
30      lb      x9,  0(x5)
31      lbu     x10, 0(x5)
32      or      x11, x11, x6
33      and     x12, x12, x7
34      xor     x13, x13, x8
35      add     x14, x14, x9
36      add     x15, x15, x10
37      addi    x2, x2, 1
38      jal     x0, load_loop_2
39  base_addr_3:
40      addi    x1, x0, 0x14
41      slli    x1, x1, 8
42      lui     x5, 0x12345
43      ori     x5, x5, 0x678
44      sw      x5, 0(x1)
45  base_addr_4:
46      addi    x1, x0, 0x18
47      slli    x1, x1, 8
48      lui     x5, 0xCAFEB
49      ori     x5, x5, 0x0BE
50      sw      x5, 0(x1)
51  base_addr_5:
52      addi    x1, x0, 0x1C
53      slli    x1, x1, 8
54      lui     x5, 0x13579
55      ori     x5, x5, 0x0E0
56      sw      x5, 0(x1)
57  base_addr_6:
58      addi    x1, x0, 0x20
59      slli    x1, x1, 8
60      lui     x5, 0x2468A
61      ori     x5, x5, 0x0CE
62      sw      x5, 0(x1)
```

```
    xor     x5, x5, x6
    andi    x5, x5, -16
    or      x4, x5, x2
    slli    x5, x2, 2
    add     x6, x1, x5
    sw      x4, 0(x6)
    addi    x2, x2, 1
    jal     x0, store_loop1
base_addr_2:
    addi    x2, x0, 0
    addi    x3, x0, 16
    addi    x22, x0, -1
load_loop_2:
    beq     x2, x3, base_addr_3
    slli    x4, x2, 2
    add     x5, x1, x4
    lw      x6,  0(x5)
    lh      x7,  0(x5)
    lhu     x8,  0(x5)
    lb      x9,  0(x5)
    lbu     x10, 0(x5)
    or      x21, x21, x6
    and     x22, x22, x7
    xor     x23, x23, x8
    add     x24, x24, x9
    add     x25, x25, x10
    addi    x2, x2, 1
    jal     x0, load_loop_2
base_addr_3:
    addi    x1, x0, 0x28
    slli    x1, x1, 8
    lui     x5, 0x12345
    ori     x5, x5, 0x678
    sw      x5, 0(x1)
base_addr_4:
    addi    x1, x0, 0x2c
    slli    x1, x1, 8
    lui     x5, 0xCAFEB
    ori     x5, x5, 0x0BE
    sw      x5, 0(x1)
base_addr_5:
    addi    x1, x0, 0xc
    slli    x1, x1, 8
    lui     x5, 0x13579
    ori     x5, x5, 0x0E0
    sw      x5, 0(x1)
base_addr_6:
    addi    x1, x0, 0xc4
    slli    x1, x1, 4
    lui     x5, 0x2468A
    ori     x5, x5, 0x0CE
    sw      x5, 0(x1)
```

63	base_addr_7:	base_addr_7:
64	addi x1, x0, 0x10	addi x1, x0, 0x24
65	slli x1, x1, 8	slli x1, x1, 8
66	addi x2, x0, 0	addi x2, x0, 0
67	addi x3, x0, 16	addi x3, x0, 16
68	addi x17, x0, -1	addi x27, x0, -1
69	load_loop_7:	load_loop_7:
70	beq x2, x3, quick_check	beq x2, x3, quick_check
71	slli x4, x2, 2	slli x4, x2, 2
72	add x5, x1, x4	add x5, x1, x4
73	lw x6, 0(x5)	lw x6, 0(x5)
74	lh x7, 0(x5)	lh x7, 0(x5)
75	lhu x8, 0(x5)	lhu x8, 0(x5)
76	lb x9, 0(x5)	lb x9, 0(x5)
77	lbu x10, 0(x5)	lbu x10, 0(x5)
78	or x16, x16, x6	or x26, x26, x6
79	and x17, x17, x7	and x27, x27, x7
80	xor x18, x18, x8	xor x28, x28, x8
81	add x19, x19, x9	add x29, x29, x9
82	add x20, x20, x10	add x30, x30, x10
83	addi x2, x2, 1	addi x2, x2, 1
84	jal x0, load_loop_7	jal x0, load_loop_7
85	quick_check:	quick_check:
86	xor x21, x11, x16	xor x11, x21, x26
87	xor x22, x12, x17	xor x12, x22, x27
88	xor x23, x13, x18	xor x13, x23, x28
89	xor x24, x14, x19	xor x14, x24, x29
90	xor x25, x15, x20	xor x15, x25, x30
91	end_program:	end_program:

Theo chương trình trong Bảng 5.2, từ dòng số 2 đến 38 cả hai CPU thực hiện ghi 16 giá trị tạo ngẫu nhiên từ các lệnh lui, ori, xor, andi, or vào D-Cache với địa chỉ bắt đầu lần lượt 0x1000 và 0x2400 để kiểm tra hoạt động Write (hit/miss) và Write Allocate, sau đó thực hiện đọc lại 16 giá trị này bằng các lệnh lw, lh, lhu, lb, lbu để kiểm tra hoạt động Read hit. Tiếp đến, từ dòng 39 tới 57, CPU A ghi vào 0x1400, 0x1800, 0x1c00 và CPU B ghi vào 0x2800, 0x2c00, 0x0c00 để làm đầy 4 Way của Set 0. Sau đó 2 CPU lần lượt đọc từ địa chỉ 0x1000 và 0x2400 để kiểm tra hoạt động Read miss, Write-back, PLRUt, Fetch MEM, Read Allocate. Kết quả mô phỏng phần cứng trùng khớp với kết quả mong đợi như Bảng 5.3 và Hình 5.4.



Hình 5.4: Kết quả chạy mô phỏng phần cứng Testcase 1

Bảng 5.3: Kết quả mong đợi Testcase 1

CPU A		CPU B	
x0 = 00000000	x16 = 00abc7ff	x0 = 00000000	x16 = 00000000
x1 = 00001000	x17 = ffffc7c0	x1 = 00002400	x17 = 00000000
x2 = 00000010	x18 = 00000030	x2 = 00000010	x18 = 00000000
x3 = 00000010	x19 = fffffe08	x3 = 00000010	x19 = 00000000
x4 = 0000003c	x20 = 00000e08	x4 = 0000003c	x20 = 00000000
x5 = 0000103c	x21 = 00000000	x5 = 0000243c	x21 = 00abc7ff
x6 = 00abc7df	x22 = 00000000	x6 = 00abc7df	x22 = ffffc7c0
x7 = ffffc7df	x23 = 00000000	x7 = ffffc7df	x23 = 00000030
x8 = 0000c7df	x24 = 00000000	x8 = 0000c7df	x24 = fffffe08
x9 = ffffffdf	x25 = 00000000	x9 = ffffffdf	x25 = 00000e08
x10 = 000000df	x26 = 00000000	x10 = 000000df	x26 = 00abc7ff
x11 = 00abc7ff	x27 = 00000000	x11 = 00000000	x27 = ffffc7c0
x12 = ffffc7c0	x28 = 00000000	x12 = 00000000	x28 = 00000030
x13 = 00000030	x29 = 00000000	x13 = 00000000	x29 = fffffe08
x14 = fffffe08	x30 = 00000000	x14 = 00000000	x30 = 00000e08
x15 = 00000e08	x31 = 00000000	x15 = 00000000	x31 = 00000000

- Testcase 2: Kiểm tra Rd Request - Rd Response

Bảng 5.4: Chương trình Testcase 2

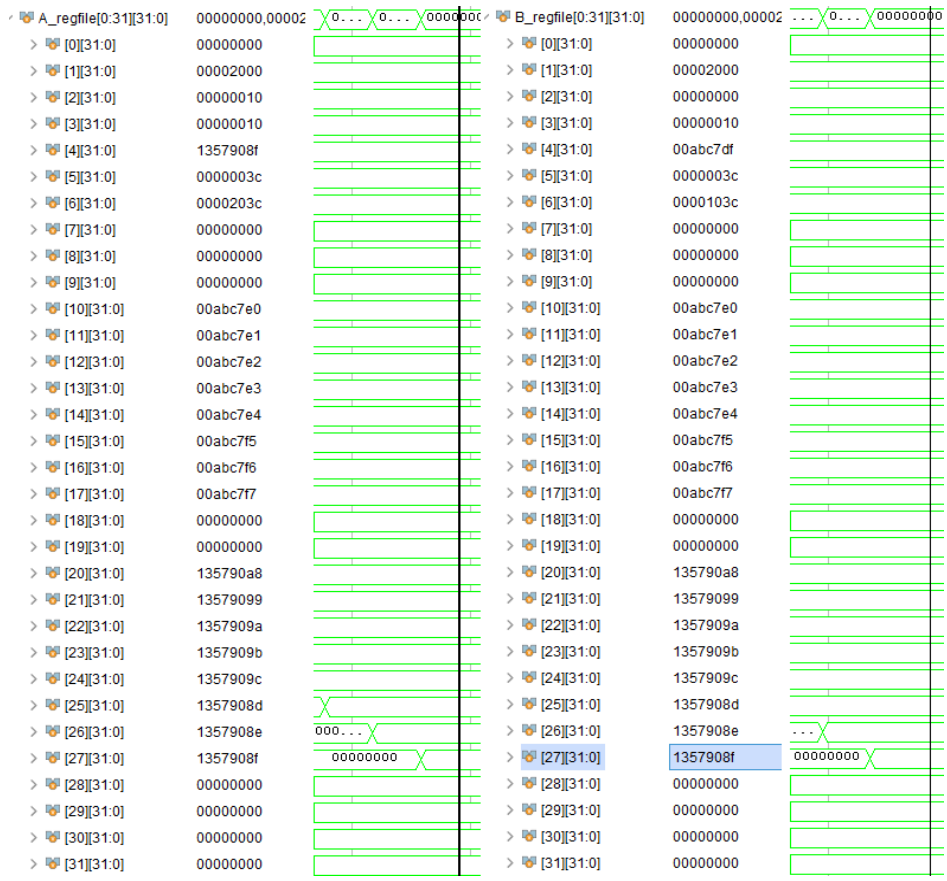
Dòng	CPU A	CPU B
1	.text	.text
2	nop_synchronize:	base_addr_1:
3	addi x1, x0, 50	addi x1, x0, 0x10

4	nop_loop:	slli x1, x1, 8
5	add x0, x0, x0	addi x2, x0, 0
6	addi x1, x1, -1	addi x3, x0, 16
7	bne x1, x0, nop_loop	store_loop1:
8	base_addr_1:	beq x2, x3, quick_check_1
9	addi x1, x0, 0x10	lui x6, 0x00ABC
10		
11	slli x1, x1, 8	ori x6, x6, 0x7EF
12	quick_check_1:	xor x5, x5, x6
13	lw x10, 0(x1)	andi x5, x5, -16
14	lw x11, 4(x1)	or x4, x5, x2
15	lw x12, 8(x1)	slli x5, x2, 2
16	lw x13, 12(x1)	add x6, x1, x5
17	lw x14, 16(x1)	sw x4, 0(x6)
18	lw x15, 20(x1)	addi x2, x2, 1
19	lw x16, 24(x1)	jal x0, store_loop1
20	lw x17, 28(x1)	quick_check_1:
21	base_addr_2:	lw x10, 0(x1)
22	addi x1, x0, 0x20	lw x11, 4(x1)
23	slli x1, x1, 8	lw x12, 8(x1)
24	addi x2, x0, 0	lw x13, 12(x1)
25	addi x3, x0, 16	lw x14, 16(x1)
26	store_loop1:	lw x15, 20(x1)
27	beq x2, x3, quick_check_2	lw x16, 24(x1)
28	lui x6, 0x13579	lw x17, 28(x1)
29	ori x6, x6, 0x0BD	nop_synchronize:
30	xor x5, x5, x6	addi x2, x0, 50
31	andi x5, x5, -16	nop_loop:
32	or x4, x5, x2	add x0, x0, x0
33	slli x5, x2, 2	addi x2, x2, -1
34	add x6, x1, x5	bne x2, x0, nop_loop
35	sw x4, 0(x6)	base_addr_2:
36	addi x2, x2, 1	addi x1, x0, 0x20
37	jal x0, store_loop1	slli x1, x1, 8
38	quick_check_2:	quick_check_2:
39	lw x20, 32(x1)	lw x20, 32(x1)
40	lw x21, 36(x1)	lw x21, 36(x1)
41	lw x22, 40(x1)	lw x22, 40(x1)
42	lw x23, 44(x1)	lw x23, 44(x1)
43	lw x24, 48(x1)	lw x24, 48(x1)
44	lw x25, 52(x1)	lw x25, 52(x1)
45	lw x26, 56(x1)	lw x26, 56(x1)
46	lw x27, 60(x1)	lw x27, 60(x1)
47	end_program:	end_program:

Theo Bảng 5.4, đối với CPU A từ dòng 2 tới 7 dùng để đồng bộ khi hai luồng CPU A và CPU B chạy song song. Dòng 2 tới 27 chương trình CPU B thực hiện lưu 16 giá trị liên tiếp vào địa chỉ bắt đầu 0x1000, sau đó CPU B tiến hành đọc lên lại để kiểm tra kết quả đã ghi. Trong lúc đó, dòng 8 tới 19 chương trình CPU A thực

hiện đọc 8 giá trị liên tiếp từ địa chỉ bắt đầu 0x1000, các giá trị này phải đảm bảo đọc được các giá đã được ghi bởi CPU B, điều này chứng minh Rd Request và Rd Response hoạt động đúng, Cache Coherence hoạt động đúng đối với thao tác đọc.

Tương tự đối với CPU B, dòng 28 tới dòng 33 dùng để đồng bộ hai luồng CPU A và CPU B. Tiếp đến dòng 20 tới dòng 45, CPU A thực hiện ghi 16 giá trị ngẫu nhiên liên tiếp với địa chỉ bắt đầu 0x2000 vào bộ nhớ đệm, sau đó tiến hành đọc lại để kiểm tra giá trị đã ghi. Từ dòng 34 tới dòng 45, CPU B thực hiện đọc 8 giá trị liên tiếp từ bộ nhớ với địa chỉ bắt đầu là 0x2020, địa chỉ này cùng Block với địa chỉ 0x2000 mà CPU A đã ghi trước đó, nên trường hợp này đảm bảo CPU B phải đọc đúng giá trị CPU A đã ghi. Bảng 5.5 và Hình 5.5 cho thấy thanh ghi x10 – x15 và thanh ghi x20 – x27 của CPU A có giá trị giống với thanh ghi x10 – x15 và thanh ghi x20 – x27 của CPU B. Các thanh ghi còn lại đều có giá trị đúng với giá trị mong đợi. Điều này chứng minh thiết kế hoạt động đúng đối với Testcase 2.



Hình 5.5: Kết quả chạy mô phỏng phần cứng Testcase 2

Bảng 5.5: Kết quả mong đợi Testcase 2

CPU A		CPU B	
x0 = 00000000	x16 = 00abc7f6	x0 = 00000000	x16 = 00abc7f6
x1 = 00002000	x17 = 00abc7f7	x1 = 00002000	x17 = 00abc7f7
x2 = 00000010	x18 = 00000000	x2 = 00000000	x18 = 00000000
x3 = 00000010	x19 = 00000000	x3 = 00000010	x19 = 00000000
x4 = 1357908f	x20 = 135790a8	x4 = 00abc7df	x20 = 135790a8
x5 = 0000003c	x21 = 13579099	x5 = 0000003c	x21 = 13579099
x6 = 0000203c	x22 = 1357909a	x6 = 0000103c	x22 = 1357909a
x7 = 00000000	x23 = 1357909b	x7 = 00000000	x23 = 1357909b
x8 = 00000000	x24 = 1357909c	x8 = 00000000	x24 = 1357909c
x9 = 00000000	x25 = 1357908d	x9 = 00000000	x25 = 1357908d
x10 = 00abc7e0	x26 = 1357908e	x10 = 00abc7e0	x26 = 1357908e
x11 = 00abc7e1	x27 = 1357908f	x11 = 00abc7e1	x27 = 1357908f
x12 = 00abc7e2	x28 = 00000000	x12 = 00abc7e2	x28 = 00000000
x13 = 00abc7e3	x29 = 00000000	x13 = 00abc7e3	x29 = 00000000
x14 = 00abc7e4	x30 = 00000000	x14 = 00abc7e4	x30 = 00000000
x15 = 00abc7f5	x31 = 00000000	x15 = 00abc7f5	x31 = 00000000

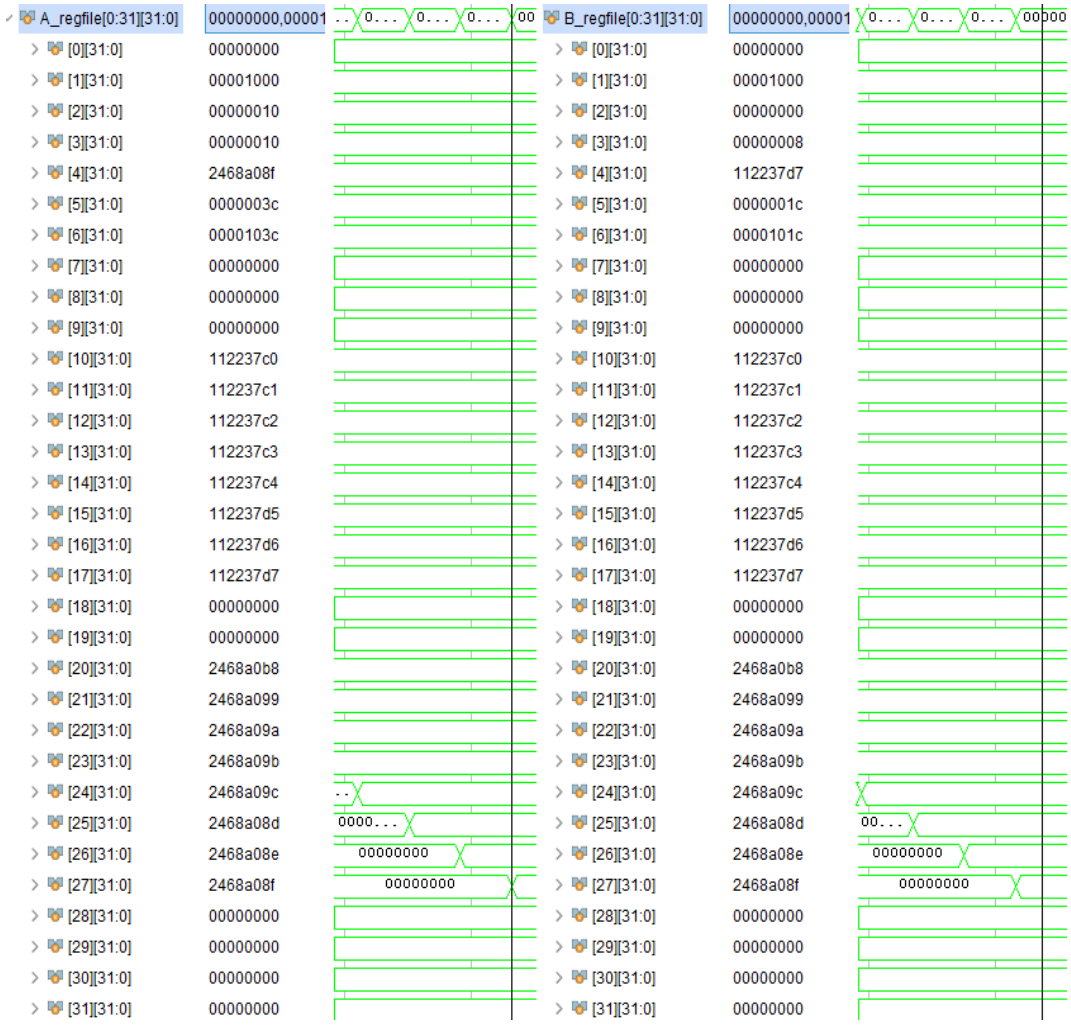
- Testcase 3: Kiểm tra RdX Request - RdX Response

Bảng 5.6: Chương trình Testcase 3

Dòng	CPU A	CPU B
1	.text	.text
2	nop_synchronize:	base_addr_1:
3	addi x1, x0, 50	addi x1, x0, 0x10
4	nop_loop:	slli x1, x1, 8
5	add x0, x0, x0	addi x2, x0, 0
6	addi x1, x1, -1	addi x3, x0, 8
7	bne x1, x0, nop_loop	store_loop1:
8	base_addr_1:	beq x2, x3, nop_synchronize
9	addi x1, x0, 0x10	lui x6, 0x11223
10	slli x1, x1, 8	ori x6, x6, 0x7C0
11	addi x2, x0, 8	xor x5, x5, x6
12	addi x3, x0, 16	andi x5, x5, -16
13	store_loop1:	or x4, x5, x2
14	beq x2, x3, quick_check_1	slli x5, x2, 2
15	lui x6, 0x2468A	add x6, x1, x5
16	ori x6, x6, 0x0BD	sw x4, 0(x6)
17	xor x5, x5, x6	addi x2, x2, 1
18	andi x5, x5, -16	jal x0, store_loop1
19	or x4, x5, x2	nop_synchronize:
20	slli x5, x2, 2	addi x2, x0, 50
21	add x6, x1, x5	nop_loop:
22	sw x4, 0(x6)	add x0, x0, x0
23	addi x2, x2, 1	addi x2, x2, -1
24	jal x0, store_loop1	bne x2, x0, nop_loop
25	quick_check_1:	quick_check_1:
26	lw x10, 0(x1)	lw x10, 0(x1)
27	lw x11, 4(x1)	lw x11, 4(x1)
28	lw x12, 8(x1)	lw x12, 8(x1)

29	lw x13, 12(x1)	lw x13, 12(x1)
30	lw x14, 16(x1)	lw x14, 16(x1)
31	lw x15, 20(x1)	lw x15, 20(x1)
32	lw x16, 24(x1)	lw x16, 24(x1)
33	lw x17, 28(x1)	lw x17, 28(x1)
34	quick_check_2:	quick_check_2:
35	lw x20, 32(x1)	lw x20, 32(x1)
36	lw x21, 36(x1)	lw x21, 36(x1)
37	lw x22, 40(x1)	lw x22, 40(x1)
38	lw x23, 44(x1)	lw x23, 44(x1)
39	lw x24, 48(x1)	lw x24, 48(x1)
40	lw x25, 52(x1)	lw x25, 52(x1)
41	lw x26, 56(x1)	lw x26, 56(x1)
42	lw x27, 60(x1)	lw x27, 60(x1)
43	end_program:	end_program:

Theo Bảng 5.6, từ dòng 2 tới dòng 7 của CPU A dùng để đồng bộ khi hai luồng CPU A và CPU B chạy song song, trong khi đó dòng 2 tới dòng 18 của CPU B thực hiện ghi 8 giá trị ngẫu nhiên liên tiếp vào bộ nhớ đệm với địa chỉ bắt đầu 0x1000. Sau khi CPU B thực hiện ghi xong, từ dòng 8 tới dòng 42, CPU A thực hiện yêu cầu ghi 8 giá trị liên tiếp với địa chỉ bắt đầu là 0x2020, tức thuộc cùng Block với địa chỉ bắt đầu 0x2000 mà CPU B ghi trước đó. Sau đó CPU A tiến hành đọc lại 16 giá trị liên tiếp với địa chỉ bắt đầu là 0x2000 lưu vào x10 – x27. Trong khi đó, dòng 19 tới 24 của chương trình CPU B được sử dụng để đồng bộ hóa, đợi CPU A thực hiện ghi xong, sau đó CPU B cũng tiến hành đọc 16 giá trị liên tiếp từ bộ nhớ đệm x10 tới x27. Theo đó, để đảm bảo Cache Coherence hoạt động đúng với thao tác ghi (RdX Request - RdX Response) thì giá trị thanh ghi x10 – x17 của CPU A phải giống với giá trị của thanh ghi x10 – x17 của CPU B, đồng thời giá trị thanh ghi x20 – x27 của CPU A phải giống với giá trị của thanh ghi x20 – x27 của CPU B. Các thanh ghi còn lại đều phải đúng với giá trị mong đợi. Hình 5.6 và Bảng 5.7 thể hiện kết quả mô phỏng phần cứng đúng với mong đợi, chứng minh thiết kế chạy đúng với Testcase 3.



Hình 5.6: Kết quả chạy mô phỏng phần cứng Testcase 3

Bảng 5.7: Kết quả mong đợi Testcase 3

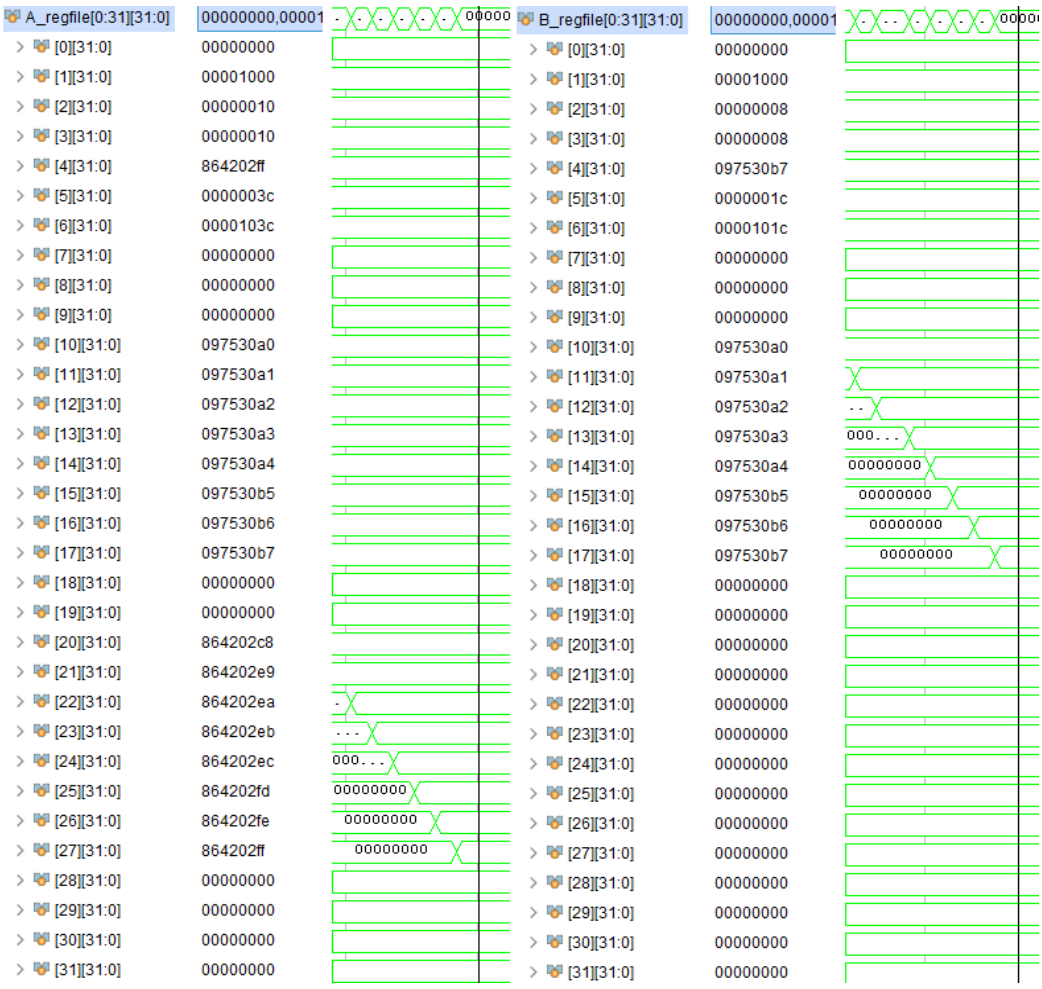
CPU A		CPU B	
x0 = 00000000	x16 = 112237d6	x0 = 00000000	x16 = 112237d6
x1 = 00001000	x17 = 112237d7	x1 = 00001000	x17 = 112237d7
x2 = 00000010	x18 = 00000000	x2 = 00000000	x18 = 00000000
x3 = 00000010	x19 = 00000000	x3 = 00000008	x19 = 00000000
x4 = 2468a08f	x20 = 2468a0b8	x4 = 112237d7	x20 = 2468a0b8
x5 = 0000003c	x21 = 2468a099	x5 = 0000001c	x21 = 2468a099
x6 = 0000103c	x22 = 2468a09a	x6 = 0000101c	x22 = 2468a09a
x7 = 00000000	x23 = 2468a09b	x7 = 00000000	x23 = 2468a09b
x8 = 00000000	x24 = 2468a09c	x8 = 00000000	x24 = 2468a09c
x9 = 00000000	x25 = 2468a08d	x9 = 00000000	x25 = 2468a08d
x10 = 112237c0	x26 = 2468a08e	x10 = 112237c0	x26 = 2468a08e
x11 = 112237c1	x27 = 2468a08f	x11 = 112237c1	x27 = 2468a08f
x12 = 112237c2	x28 = 00000000	x12 = 112237c2	x28 = 00000000
x13 = 112237c3	x29 = 00000000	x13 = 112237c3	x29 = 00000000
x14 = 112237c4	x30 = 00000000	x14 = 112237c4	x30 = 00000000
x15 = 112237d5	x31 = 00000000	x15 = 112237d5	x31 = 00000000

- Testcase 4: Kiểm tra Invalid Request - Invalid Response

Bảng 5.8: Chương trình Testcase 4

Dòng	CPU A	CPU B
1	.text	.text
2	nop_synchronize:	base_addr_1:
3	addi x1, x0, 50	addi x1, x0, 0x10
4	nop_loop:	slli x1, x1, 8
5	add x0, x0, x0	addi x2, x0, 0
6	addi x1, x1, -1	addi x3, x0, 8
7	bne x1, x0, nop_loop	store_loop1:
8	base_addr_1:	beq x2, x3, quick_check_1
9	addi x1, x0, 0x10	lui x6, 0x09753
10	slli x1, x1, 8	ori x6, x6, 0x0A0
11	quick_check_1:	xor x5, x5, x6
12	lw x10, 0(x1)	andi x5, x5, -16
13	lw x11, 4(x1)	or x4, x5, x2
14	lw x12, 8(x1)	slli x5, x2, 2
15	lw x13, 12(x1)	add x6, x1, x5
16	lw x14, 16(x1)	sw x4, 0(x6)
17	lw x15, 20(x1)	addi x2, x2, 1
18	lw x16, 24(x1)	jal x0, store_loop1
19	lw x17, 28(x1)	quick_check_1:
20	base_addr_2:	lw x10, 0(x1)
21	addi x2, x0, 8	lw x11, 4(x1)
22	addi x3, x0, 16	lw x12, 8(x1)
23	store_loop2:	lw x13, 12(x1)
24	beq x2, x3, quick_check_2	lw x14, 16(x1)
25	lui x6, 0x86420	lw x15, 20(x1)
26	ori x6, x6, 0x2C0	lw x16, 24(x1)
27	xor x5, x5, x6	lw x17, 28(x1)
28	andi x5, x5, -16	end_program:
29	or x4, x5, x2	
30	slli x5, x2, 2	
31	add x6, x1, x5	
32	sw x4, 0(x6)	
33	addi x2, x2, 1	
34	jal x0, store_loop2	
35	quick_check_2:	
36	lw x20, 32(x1)	
37	lw x21, 36(x1)	
38	lw x22, 40(x1)	
39	lw x23, 44(x1)	
40	lw x24, 48(x1)	
41	lw x25, 52(x1)	
42	lw x26, 56(x1)	
43	lw x27, 60(x1)	
44	end_program:	

Theo Bảng 5.8, dòng từ dòng 2 tới dòng 7 của CPU A dùng để đồng bộ khi hai luồng CPU A và CPU B chạy song song. Dòng 2 tới dòng 27 của CPU B thực hiện lưu 8 giá trị ngẫu nhiên liên tiếp vào bộ nhớ đệm với địa chỉ bắt đầu 0x1000, sau đó tiến hành đọc lên lại đã kiểm tra giá trị đã ghi. Trong khi đó, dòng 8 tới dòng 43 của CPU A được thực hiện sau khi CPU B thực hiện ghi xong, CPU A tiến hành đọc 8 giá trị liên tiếp với địa chỉ bắt đầu 0x1000, sau đó tiến hành ghi 8 giá trị ngẫu nhiên tiếp theo vào địa chỉ bắt đầu 0x1020, sau đó tiến hành đọc lên lại để kiểm tra giá trị đã ghi. Theo đúng mong đợi, giá trị thanh ghi x10 – x17 của CPU A đọc được phải trùng khớp với giá trị của x10 – x17 của CPU B và các thanh ghi còn lại phải đúng với giá trị mong đợi. Thêm vào đó, CPU B phải loại bỏ Block với địa chỉ bắt đầu 0x1000. Hình 5.7 và Bảng 5.9 thể hiện kết quả mô phỏng thiết kế hoạt động đúng với Testcase 4.



Hình 5.7: Kết quả chạy mô phỏng phần cứng Testcase 4

Bảng 5.9: Kết quả mong đợi Testcase 4

CPU A		CPU B	
x0 = 00000000	x16 = 097530b6	x0 = 00000000	x16 = 097530b6
x1 = 00001000	x17 = 097530b7	x1 = 00001000	x17 = 097530b7
x2 = 00000010	x18 = 00000000	x2 = 00000008	x18 = 00000000
x3 = 00000010	x19 = 00000000	x3 = 00000008	x19 = 00000000
x4 = 864202ff	x20 = 864202c8	x4 = 097530b7	x20 = 00000000
x5 = 0000003c	x21 = 864202e9	x5 = 0000001c	x21 = 00000000
x6 = 0000103c	x22 = 864202ea	x6 = 0000101c	x22 = 00000000
x7 = 00000000	x23 = 864202eb	x7 = 00000000	x23 = 00000000
x8 = 00000000	x24 = 864202ec	x8 = 00000000	x24 = 00000000
x9 = 00000000	x25 = 864202fd	x9 = 00000000	x25 = 00000000
x10 = 097530a0	x26 = 864202fe	x10 = 097530a0	x26 = 00000000
x11 = 097530a1	x27 = 864202ff	x11 = 097530a1	x27 = 00000000
x12 = 097530a2	x28 = 00000000	x12 = 097530a2	x28 = 00000000
x13 = 097530a3	x29 = 00000000	x13 = 097530a3	x29 = 00000000
x14 = 097530a4	x30 = 00000000	x14 = 097530a4	x30 = 00000000
x15 = 097530b5	x31 = 00000000	x15 = 097530b5	x31 = 00000000

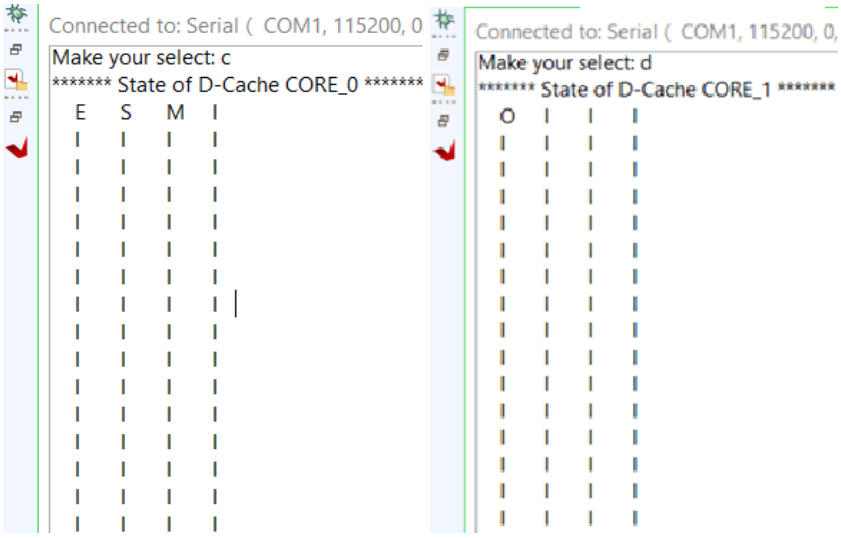
- Testcase 5: Kiểm tra chuyển trạng thái: I → E, I → S, I → M

Bảng 5.10: Chương trình Testcase 5

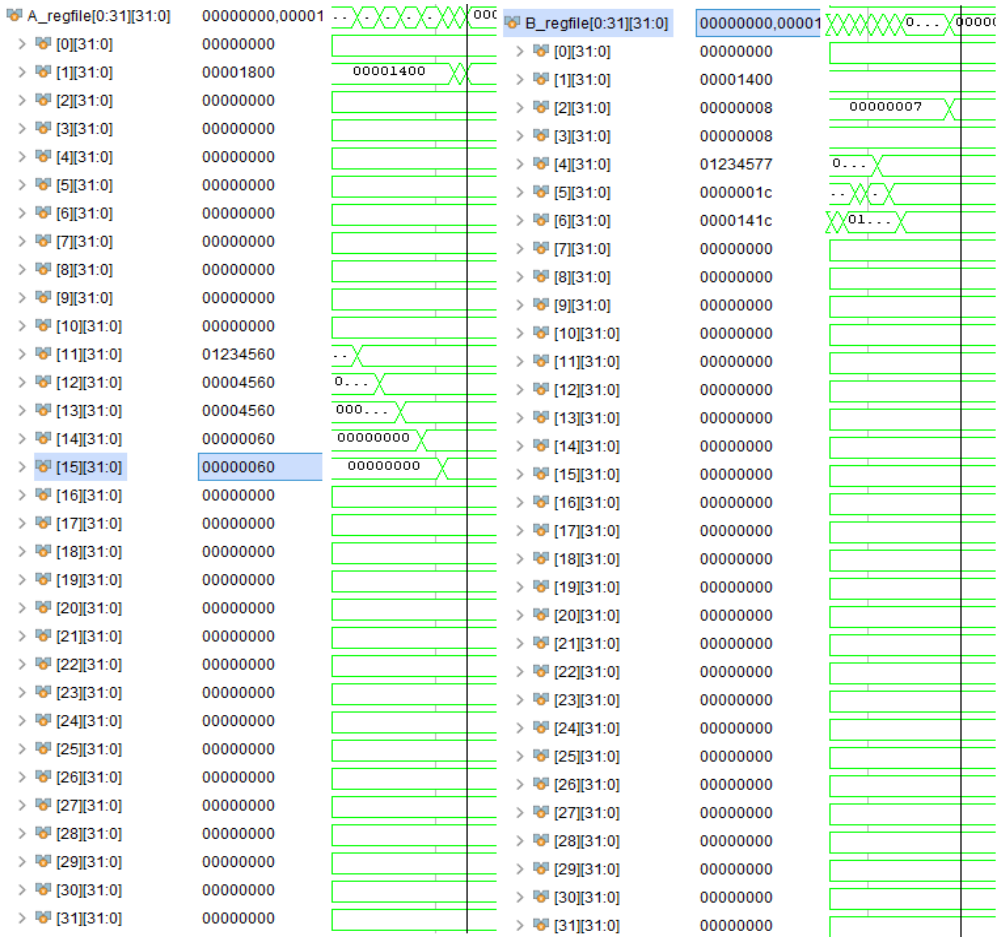
Dòng	CPU A	CPU B
1	.text	.text
2	base_addr_1:	base_addr_1:
3	addi x1, x0, 0x10	addi x1, x0, 0x14
4	slli x1, x1, 8	slli x1, x1, 8
5	load_1:	addi x2, x0, 0
6	lw x6, 0(x1)	addi x3, x0, 8
7	lh x7, 0(x1)	store_loop1:
8	lhu x8, 0(x1)	beq x2, x3, end_program
9	lb x9, 0(x1)	lui x6, 0x1234
10	lbu x10, 0(x1)	ori x6, x6, 0x567
11	nop_synchronize:	xor x5, x5, x6
12	addi x1, x0, 50	andi x5, x5, -16
13	nop_loop:	or x4, x5, x2
14	add x0, x0, x0	slli x5, x2, 2
15	addi x1, x1, -1	add x6, x1, x5
16	bne x1, x0, nop_loop	sw x4, 0(x6)
17	base_addr_2:	addi x2, x2, 1
18	addi x1, x0, 0x14	jal x0, store_loop1
19	slli x1, x1, 8	end_program:
20	load_2:	
21	lw x11, 0(x1)	
22	lh x12, 0(x1)	
23	lhu x13, 0(x1)	
24	lb x14, 0(x1)	
25	lbu x15, 0(x1)	
26	base_addr_3:	
27	addi x1, x0, 0x18	
28	slli x1, x1, 8	

29	store_3:	
30	sw x11, 0(x1)	
31	sh x12, 4(x1)	
32	sb x13, 8(x1)	
33	end_program:	

Theo Bảng 5.10, từ dòng 2 tới dòng 10, CPU A sẽ đọc từ địa chỉ 0x1000 từ dưới bộ nhớ chính lên, điều này tạo cho Block với địa chỉ bắt đầu 0x1000 chuyển từ trạng thái I → E và lưu trong D-CacheA. Đồng thời lúc đó, dòng 2 tới dòng 18 CPU B tiến hành ghi vào Block với địa chỉ bắt đầu là 0x1400, tạo thành trạng thái M trong D-CacheB. Từ dòng 11 tới dòng 16, CPU A đợi CPU B ghi xong. Sau đó từ dòng 17 tới dòng 25, CPU A tiến hành đọc Block với địa chỉ bắt đầu 0x1400 từ D-CacheB của CPU B, do đó trạng thái của Block 0x1400 trong D-CacheA chuyển từ I → S, trạng thái Block 0x1400 trong D-CacheB chuyển từ M → O. Từ dòng 26 tới dòng 32, CPU A thực hiện ghi vào Block 0x1800, tạo ra sự chuyển trạng thái đối với Block này từ I → M. Kết quả chuyển trạng thái thể hiện trong Hình 5.8.



Hình 5.8: Kết quả chuyển trạng thái MOESI Testcase 5



Hình 5.9: Kết quả chạy mô phỏng phần cứng Testcase 5

Giá trị các thanh ghi mô phỏng phần cứng trong Hình 5.9 đúng với kết quả mong đợi trong Bảng 5.11, chứng minh thiết kế chạy đúng đối với Testcase 5.

Bảng 5.11: Kết quả mong đợi Testcase 5

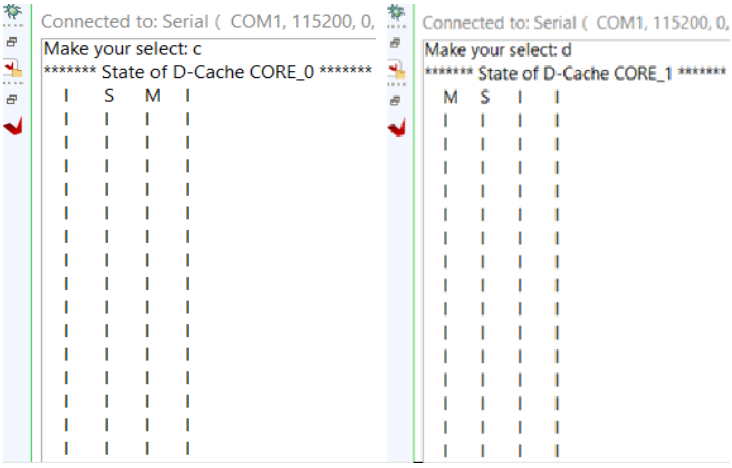
CPU A		CPU B	
x0 = 00000000	x16 = 00000000	x0 = 00000000	x16 = 00000000
x1 = 00001800	x17 = 00000000	x1 = 00001400	x17 = 00000000
x2 = 00000000	x18 = 00000000	x2 = 00000008	x18 = 00000000
x3 = 00000000	x19 = 00000000	x3 = 00000008	x19 = 00000000
x4 = 00000000	x20 = 00000000	x4 = 01234577	x20 = 00000000
x5 = 00000000	x21 = 00000000	x5 = 0000001c	x21 = 00000000
x6 = 00000000	x22 = 00000000	x6 = 0000141c	x22 = 00000000
x7 = 00000000	x23 = 00000000	x7 = 00000000	x23 = 00000000
x8 = 00000000	x24 = 00000000	x8 = 00000000	x24 = 00000000
x9 = 00000000	x25 = 00000000	x9 = 00000000	x25 = 00000000
x10 = 00000000	x26 = 00000000	x10 = 00000000	x26 = 00000000
x11 = 01234560	x27 = 00000000	x11 = 00000000	x27 = 00000000
x12 = 00004560	x28 = 00000000	x12 = 00000000	x28 = 00000000
x13 = 00004560	x29 = 00000000	x13 = 00000000	x29 = 00000000
x14 = 00000060	x30 = 00000000	x14 = 00000000	x30 = 00000000
x15 = 00000060	x31 = 00000000	x15 = 00000000	x31 = 00000000

- Testcase 6: Kiểm tra chuyển trạng thái: E → I, E → S, E → M

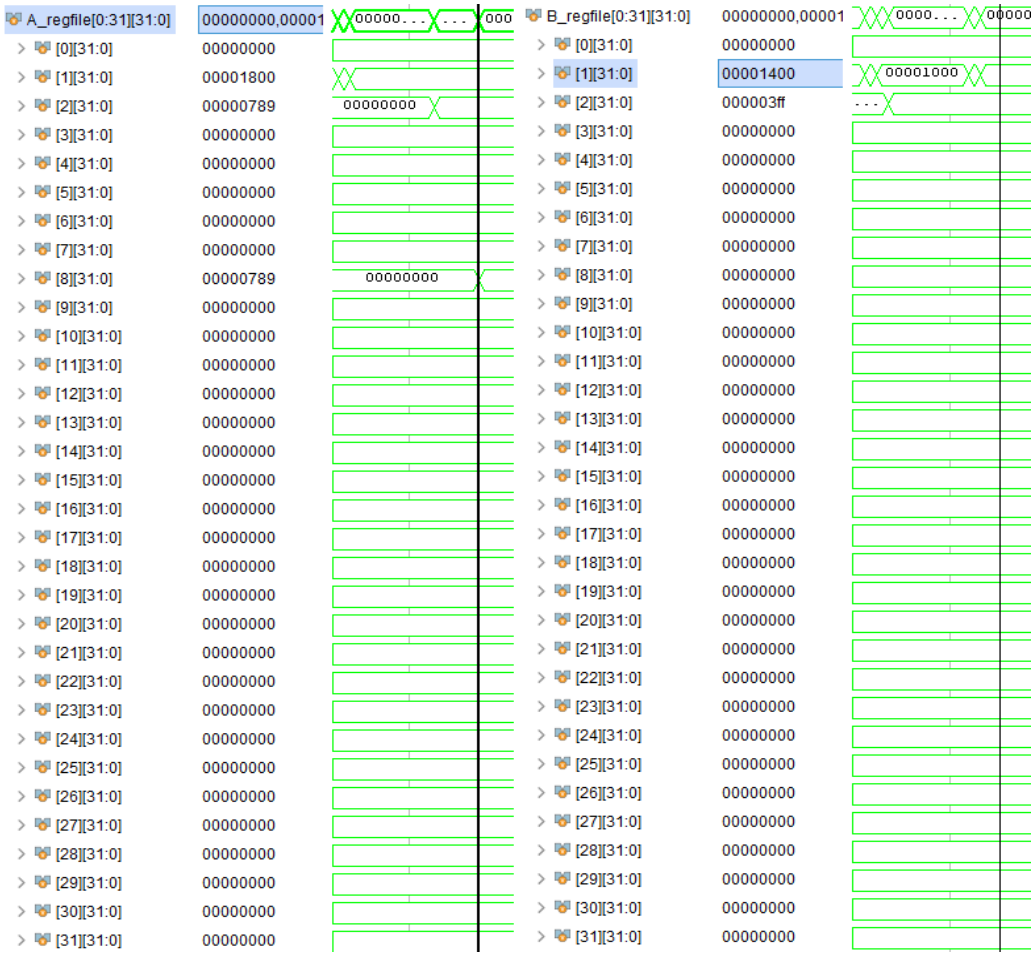
Bảng 5.12: Chương trình Testcase 6

Dòng	CPU A	CPU B
1	.text	.text
2	base_addr_1:	nop_synchronize:
3	addi x1, x0, 0x10	addi x1, x0, 150
4	slli x1, x1, 8	nop_loop:
5	load_1:	add x0, x0, x0
6	lw x6, 0(x1)	addi x1, x1, -1
7	base_addr_2:	bne x1, x0, nop_loop
8	addi x1, x0, 0x14	base_addr_1:
9	slli x1, x1, 8	addi x1, x0, 0x10
10	load_2:	slli x1, x1, 8
11	lw x7, 0(x1)	store_1:
12	base_addr_3:	addi x2, x0, 0x3FF
13	addi x1, x0, 0x18	sw x2, 0(x1)
14	slli x1, x1, 8	base_addr_2:
15	load_3:	addi x1, x0, 0x14
16	lw x7, 0(x1)	slli x1, x1, 8
17	store_3:	store_2:
18	addi x2, x0, 0x789	lw x6, 0(x1)
19	sw x2, 0(x1)	end_program:
20	quick_check:	
21	lw x8, 0(x1)	
22	end_program:	

Theo chương trình trong Bảng 5.12, dòng từ dòng 2 tới dòng 7 của CPU B dùng để đợi CPU A thực hiện xong các lệnh từ dòng 2 tới dòng 21. Sau khi CPU A thực hiện xong, trong D-CacheA lúc này chứa 3 Block: 0x1000 và 0x1400 ở trạng thái E do được đọc từ bộ nhớ chính lên, 0x1800 ở trạng thái M do được CPU ghi vào. Việc CPU A ghi vào sau khi đọc Block 0x1800 tạo ra sự chuyển trạng thái E → M. Đối với CPU B, sau khi đợi CPU A thực thi xong, CPU B thực hiện dòng 8 tới dòng 13, tức thực hiện ghi vào Block 0x1000, sẽ yêu cầu D-CacheA vô hiệu Block 0x1000, tạo nên sự chuyển trạng thái từ E → I. Tiếp theo, từ dòng 14 tới dòng 18, CPU B thực hiện đọc vào Block 0x1400, yêu cầu CPU A chia sẻ Block 0x1400, tạo nên sự chuyển trạng thái E → S. Kết quả chuyển trạng thái thể hiện trong Hình 5.10.



Hình 5.10: Kết quả chuyển trạng thái MOESI Testcase 6



Hình 5.11: Kết quả chạy mô phỏng phần cứng Testcase 6

Giá trị các thanh ghi mô phỏng phần cứng trong Hình 5.11 đúng với kết quả mong đợi trong Bảng 5.13, chứng minh thiết kế chạy đúng đối với Testcase 6.

Bảng 5.13: Kết quả mong đợi Testcase 6

CPU A		CPU B	
x0 = 00000000	x16 = 00000000	x0 = 00000000	x16 = 00000000
x1 = 00001800	x17 = 00000000	x1 = 00001400	x17 = 00000000
x2 = 00000789	x18 = 00000000	x2 = 0000003ff	x18 = 00000000
x3 = 00000000	x19 = 00000000	x3 = 00000000	x19 = 00000000
x4 = 00000000	x20 = 00000000	x4 = 00000000	x20 = 00000000
x5 = 00000000	x21 = 00000000	x5 = 00000000	x21 = 00000000
x6 = 00000000	x22 = 00000000	x6 = 00000000	x22 = 00000000
x7 = 00000000	x23 = 00000000	x7 = 00000000	x23 = 00000000
x8 = 00000789	x24 = 00000000	x8 = 00000000	x24 = 00000000
x9 = 00000000	x25 = 00000000	x9 = 00000000	x25 = 00000000
x10 = 00000000	x26 = 00000000	x10 = 00000000	x26 = 00000000
x11 = 00000000	x27 = 00000000	x11 = 00000000	x27 = 00000000
x12 = 00000000	x28 = 00000000	x12 = 00000000	x28 = 00000000
x13 = 00000000	x29 = 00000000	x13 = 00000000	x29 = 00000000
x14 = 00000000	x30 = 00000000	x14 = 00000000	x30 = 00000000
x15 = 00000000	x31 = 00000000	x15 = 00000000	x31 = 00000000

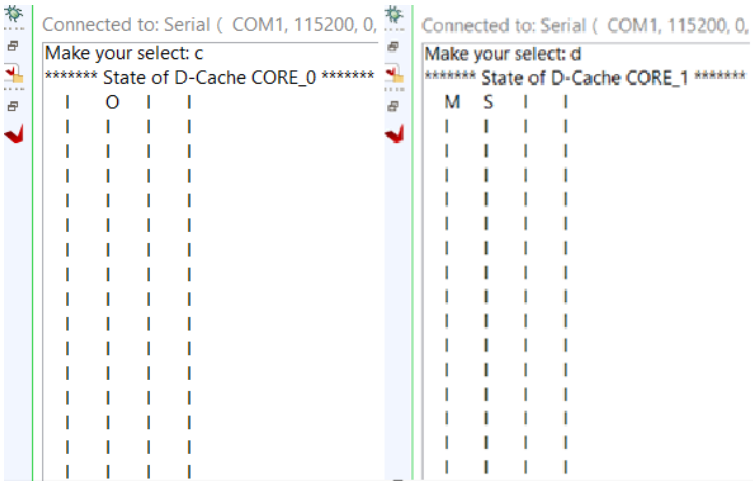
- Testcase 7: Kiểm tra chuyển trạng thái: M → I, M → O

Bảng 5.14: Chương trình Testcase 7

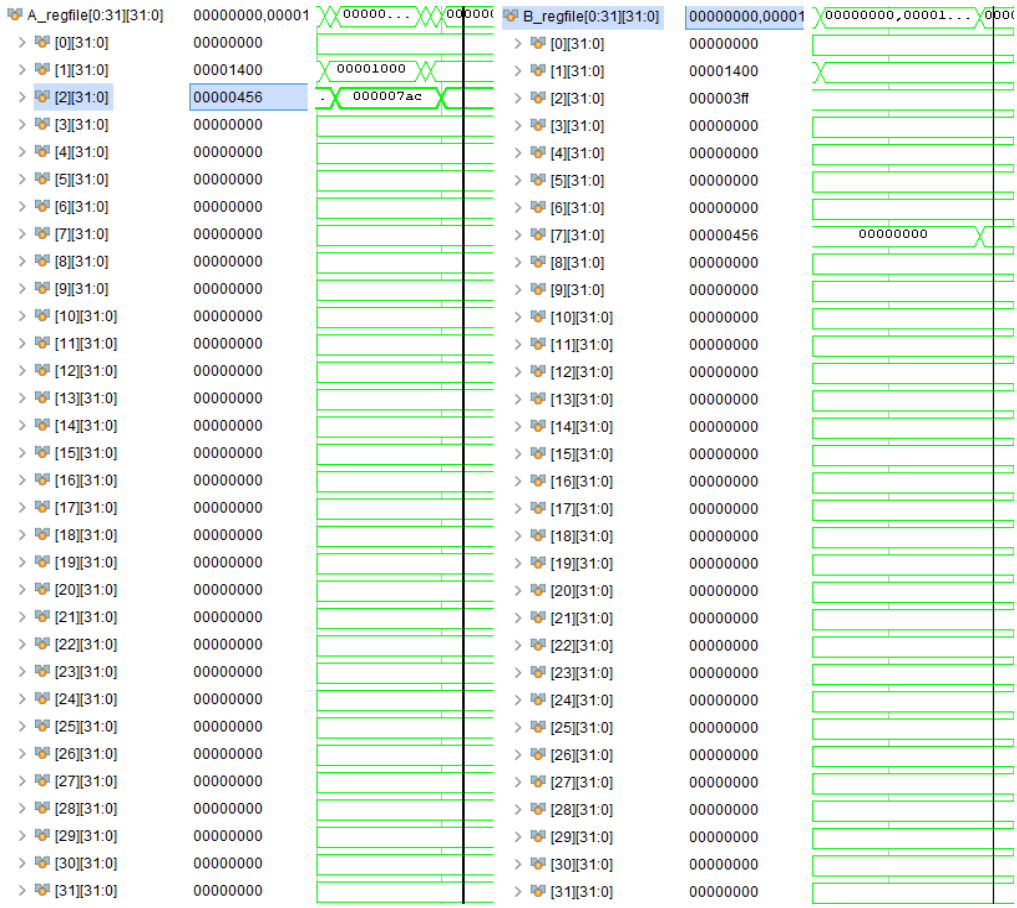
Dòng	CPU A	CPU B
1	.text	.text
2	base_addr_1:	nop_synchronize:
3	addi x1, x0, 0x10	addi x1, x0, 100
4	slli x1, x1, 8	nop_loop:
5	store_1:	add x0, x0, x0
6	addi x2, x0, 0x7AC	addi x1, x1, -1
7	sw x2, 0(x1)	bne x1, x0, nop_loop
8	base_addr_2:	base_addr_1:
9	addi x1, x0, 0x14	addi x1, x0, 0x10
10	slli x1, x1, 8	slli x1, x1, 8
11	store_2:	store_1:
12	addi x2, x0, 0x456	addi x2, x0, 0x3FF
13	sw x2, 0(x1)	sw x2, 0(x1)
14	end_program:	base_addr_2:
15		addi x1, x0, 0x14
16		slli x1, x1, 8
17		load_2:
18		lw x7, 0(x1)
19		end_program:

Theo chương trình trong Bảng 5.14, dòng từ dòng 2 tới dòng 7 của CPU B dùng để đợi CPU A thực hiện xong các lệnh từ dòng 2 tới dòng 13. Sau khi CPU A thực hiện xong, D-CacheA có 2 Block 0x1000 và 0x1400 đều ở trạng thái M. Từ dòng 8 tới dòng 13, CPU B yêu cầu ghi vào Block 0x1000, bắt buộc D-CacheA phải chuyển trạng thái Block 0x1000 từ M → I. Từ dòng 14 tới dòng 18, CPU B

yêu cầu đọc vào Block 0x1400, dẫn đến D-CacheA chuyển Block 0x1400 chuyển từ trạng thái M → O, để chia sẻ Block này với CPU B. Kết quả chuyển trạng thái thể hiện trong Hình 5.12.



Hình 5.12: Kết quả chuyển trạng thái MOESI Testcase 7



Hình 5.13: Kết quả chạy mô phỏng phần cứng Testcase 7

Giá trị các thanh ghi mô phỏng phần cứng trong Hình 5.13 đúng với kết quả mong đợi trong Bảng 5.15 chứng mình thiết kế chạy đúng đối với Testcase 7.

Bảng 5.15: Kết quả mong đợi Testcase 7

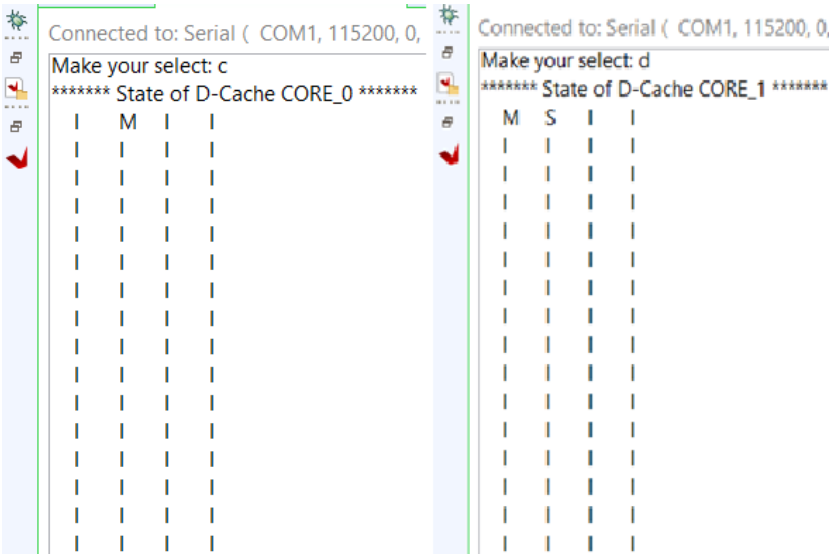
CPU A		CPU B	
x0 = 00000000	x16 = 00000000	x0 = 00000000	x16 = 00000000
x1 = 00001400	x17 = 00000000	x1 = 00001400	x17 = 00000000
x2 = 00000456	x18 = 00000000	x2 = 0000003ff	x18 = 00000000
x3 = 00000000	x19 = 00000000	x3 = 00000000	x19 = 00000000
x4 = 00000000	x20 = 00000000	x4 = 00000000	x20 = 00000000
x5 = 00000000	x21 = 00000000	x5 = 00000000	x21 = 00000000
x6 = 00000000	x22 = 00000000	x6 = 00000000	x22 = 00000000
x7 = 00000000	x23 = 00000000	x7 = 00000456	x23 = 00000000
x8 = 00000000	x24 = 00000000	x8 = 00000000	x24 = 00000000
x9 = 00000000	x25 = 00000000	x9 = 00000000	x25 = 00000000
x10 = 00000000	x26 = 00000000	x10 = 00000000	x26 = 00000000
x11 = 00000000	x27 = 00000000	x11 = 00000000	x27 = 00000000
x12 = 00000000	x28 = 00000000	x12 = 00000000	x28 = 00000000
x13 = 00000000	x29 = 00000000	x13 = 00000000	x29 = 00000000
x14 = 00000000	x30 = 00000000	x14 = 00000000	x30 = 00000000
x15 = 00000000	x31 = 00000000	x15 = 00000000	x31 = 00000000

- Testcase 8: Kiểm tra chuyển trạng thái: O → I, O → M

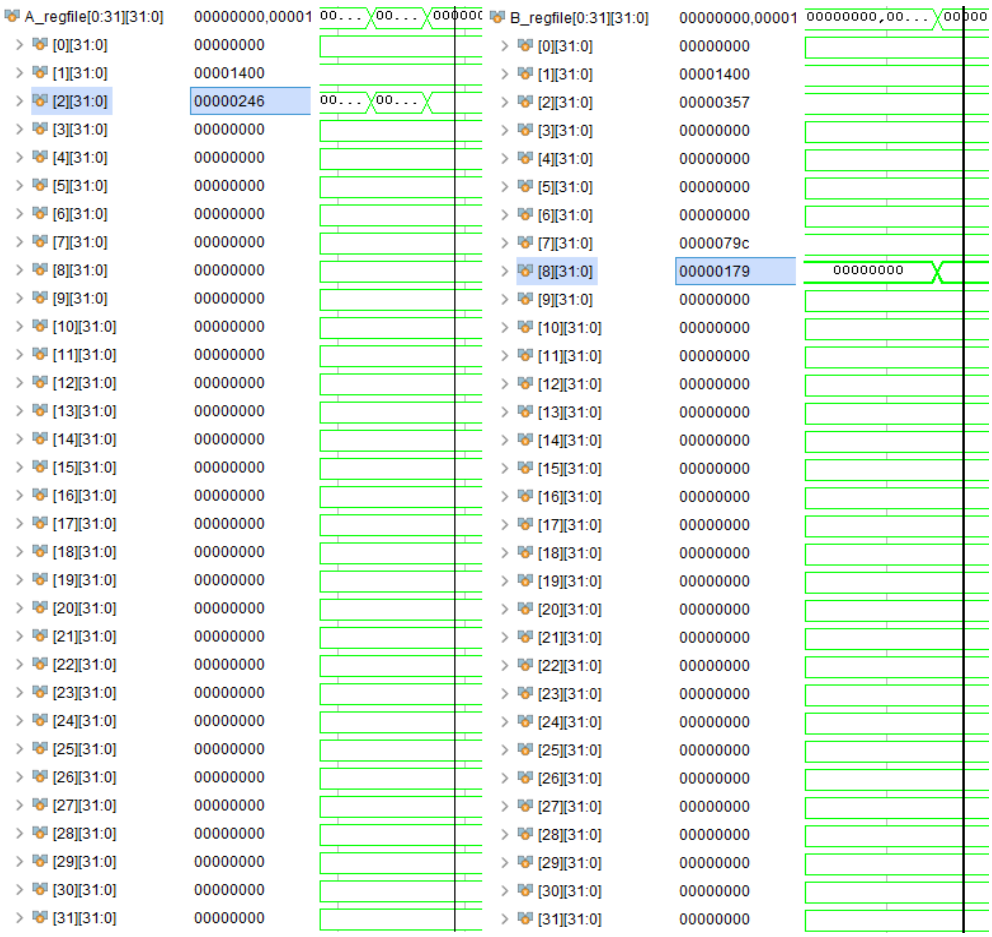
Bảng 5.16: Chương trình Testcase 8

Dòng	CPU A	CPU B
1	.text	.text
2	base_addr_1:	nop_synchronize:
3	addi x1, x0, 0x10	addi x1, x0, 100
4	slli x1, x1, 8	nop_loop:
5	store_1:	add x0, x0, x0
6	addi x2, x0, 0x79C	addi x1, x1, -1
7	sw x2, 0(x1)	bne x1, x0, nop_loop
8	base_addr_2:	base_addr_1:
9	addi x1, x0, 0x14	addi x1, x0, 0x10
10	slli x1, x1, 8	slli x1, x1, 8
11	store_2:	load_1:
12	addi x2, x0, 0x179	lw x7, 0(x1)
13	sw x2, 0(x1)	store_1:
14	nop_synchronize:	addi x2, x0, 0x357
15	addi x2, x0, 100	sw x2, 0(x1)
16	nop_loop:	base_addr_2:
17	add x0, x0, x0	addi x1, x0, 0x14
18	addi x2, x2, -1	slli x1, x1, 8
19	bne x2, x0, nop_loop	load_2:
20	store_3:	lw x8, 0(x1)
21	addi x2, x0, 0x246	end_program:
22	sw x2, 0(x1)	
23	end_program:	

Theo chương trình trong Bảng 5.16, từ dòng 2 tới dòng 7 của CPU B dùng để đợi CPU A thực hiện xong các lệnh từ dòng 2 tới dòng 13. Sau khi CPU A thực hiện xong, D-CacheA có 2 Block 0x1000 và 0x1400 đều ở trạng thái M. Sau đó dòng 14 tới dòng 19 của CPU A được dùng để đợi CPU B thực hiện xong dòng 8 tới dòng 12, Block 0x1000 trong D-CacheA chuyển từ M → O. Sau đó CPU B thực hiện dòng 13 tới dòng 15 yêu cầu ghi độc quyền vào Block 0x1000, làm cho Block 0x1000 bên D-CacheA chuyển từ trạng thái O → I. Tiếp đến CPU B thực hiện dòng 16 tới dòng 20 yêu cầu đọc vào Block 0x1400 từ D-CacheA, làm Block 0x1400 chuyển từ trạng thái M → O. Sau cùng, CPU A thực hiện dòng 20 tới dòng 22 làm Block này chuyển từ O → M. Kết quả chuyển trạng thái thể hiện trong Hình 5.14.



Hình 5.14: Kết quả chuyển trạng thái MOESI Testcase 8



Hình 5.15: Kết quả chạy mô phỏng phần cứng Testcase 8

Giá trị các thanh ghi mô phỏng phần cứng trong Hình 5.15 đúng với kết quả mong đợi trong Bảng 5.17 chứng minh thiết kế chạy đúng đối với Testcase 8.

Bảng 5.17: Kết quả mong đợi Testcase 8

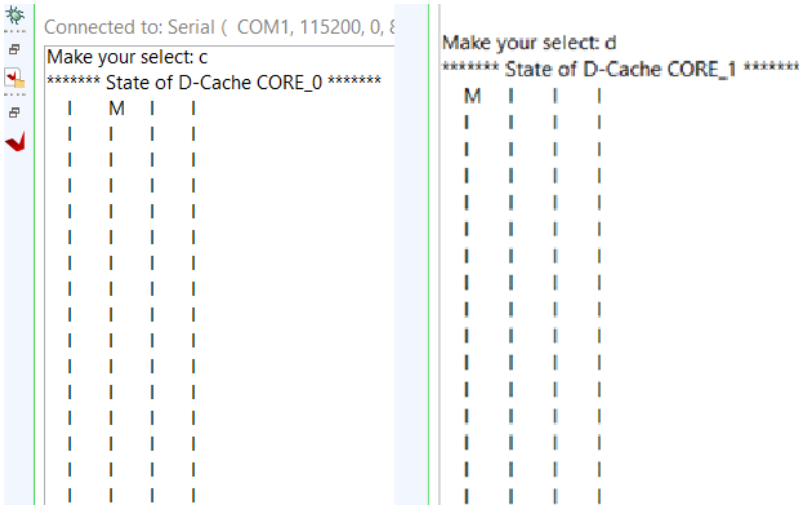
CPU A		CPU B	
x0 = 00000000	x16 = 00000000	x0 = 00000000	x16 = 00000000
x1 = 00001400	x17 = 00000000	x1 = 00001400	x17 = 00000000
x2 = 00000246	x18 = 00000000	x2 = 00000357	x18 = 00000000
x3 = 00000000	x19 = 00000000	x3 = 00000000	x19 = 00000000
x4 = 00000000	x20 = 00000000	x4 = 00000000	x20 = 00000000
x5 = 00000000	x21 = 00000000	x5 = 00000000	x21 = 00000000
x6 = 00000000	x22 = 00000000	x6 = 00000000	x22 = 00000000
x7 = 00000000	x23 = 00000000	x7 = 0000079c	x23 = 00000000
x8 = 00000000	x24 = 00000000	x8 = 00000179	x24 = 00000000
x9 = 00000000	x25 = 00000000	x9 = 00000000	x25 = 00000000
x10 = 00000000	x26 = 00000000	x10 = 00000000	x26 = 00000000
x11 = 00000000	x27 = 00000000	x11 = 00000000	x27 = 00000000
x12 = 00000000	x28 = 00000000	x12 = 00000000	x28 = 00000000
x13 = 00000000	x29 = 00000000	x13 = 00000000	x29 = 00000000
x14 = 00000000	x30 = 00000000	x14 = 00000000	x30 = 00000000
x15 = 00000000	x31 = 00000000	x15 = 00000000	x31 = 00000000

- Testcase 9: Kiểm tra chuyển trạng thái: S → I, S → M

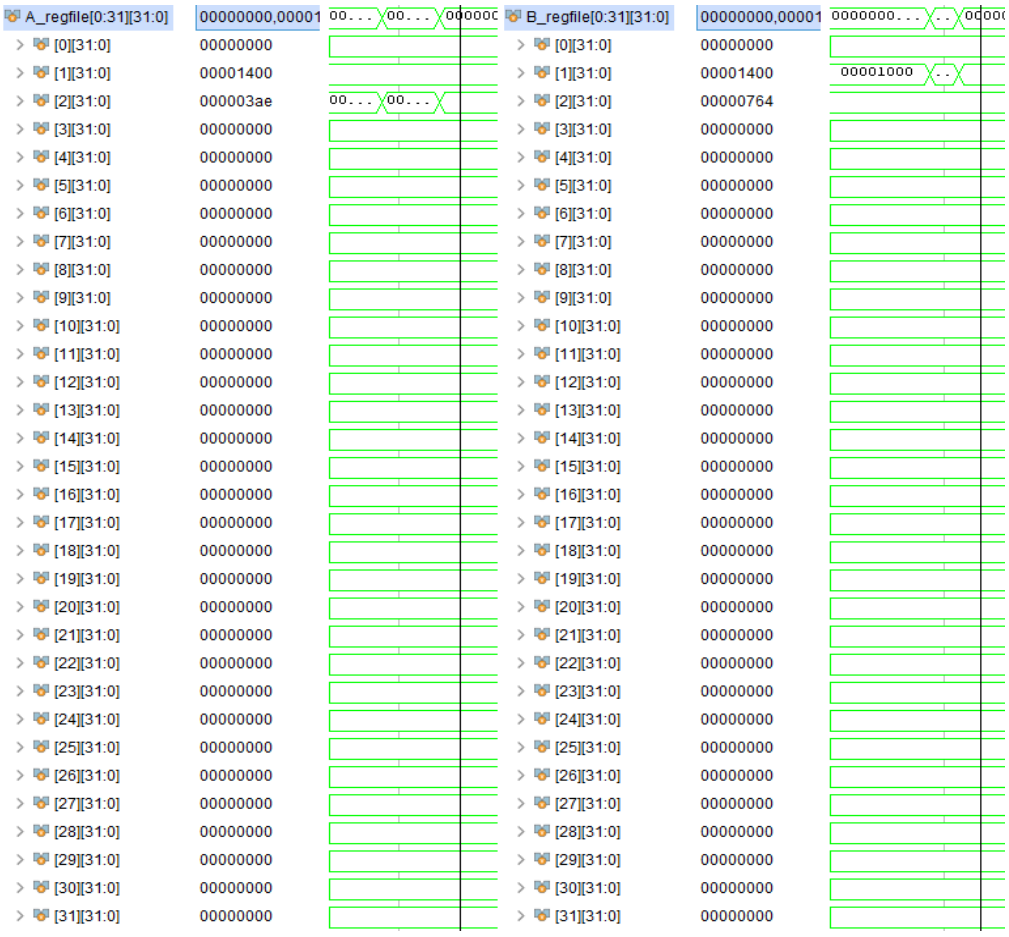
Bảng 5.18: Chương trình Testcase 9

Dòng	CPU A	CPU B
1	.text	.text
2	base_addr_1:	nop_synchronize:
3	addi x1, x0, 0x10	addi x1, x0, 100
4	slli x1, x1, 8	nop_loop:
5	store_1:	add x0, x0, x0
6	lw x6, 0(x1)	addi x1, x1, -1
7	base_addr_2:	bne x1, x0, nop_loop
8	addi x1, x0, 0x14	base_addr_1:
9	slli x1, x1, 8	addi x1, x0, 0x10
10	store_2:	slli x1, x1, 8
11	lw x7, 0(x1)	load_1:
12	nop_synchronize:	lw x7, 0(x1)
13	addi x2, x0, 150	store_1:
14	nop_loop:	addi x2, x0, 0x764
15	add x0, x0, x0	sw x2, 0(x1)
16	addi x2, x2, -1	base_addr_2:
17	bne x2, x0, nop_loop	addi x1, x0, 0x14
18	store_3:	slli x1, x1, 8
19	addi x2, x0, 0x3AE	load_2:
20	sw x2, 0(x1)	lw x8, 0(x1)
21	end_program:	end_program:

Theo chương trình trong Bảng 5.18, từ dòng 2 tới dòng 7 của CPU B dùng để đợi CPU A thực hiện xong các lệnh từ dòng 2 tới dòng 11. Sau khi CPU A thực hiện xong, D-CacheA có 2 Block 0x1000 và 0x1400 đều ở trạng thái E do được đọc từ bộ nhớ chính. Sau đó dòng 12 tới dòng 17 của CPU A được dùng để đợi CPU B thực hiện xong dòng 8 tới dòng 12, Block 0x1000 trong D-CacheA chuyển từ E → S. Sau đó CPU B thực hiện dòng 13 tới dòng 15 yêu cầu ghi độc quyền vào Block 0x1000, làm cho Block 0x1000 bên D-CacheA chuyển từ trạng thái S → I. Tiếp đến CPU B thực hiện dòng 16 tới dòng 20 yêu cầu đọc vào Block 0x1400 từ D-CacheA, làm Block 0x1400 chuyển từ trạng thái E → S. Sau cùng, CPU A thực hiện dòng 18 tới dòng 20 làm Block này chuyển từ S → M. Kết quả chuyển trạng thái thể hiện trong Hình 5.16.



Hình 5.16: Kết quả chuyển trạng thái MOESI Testcase 9



Hình 5.17: Kết quả chạy mô phỏng phần cứng Testcase 9

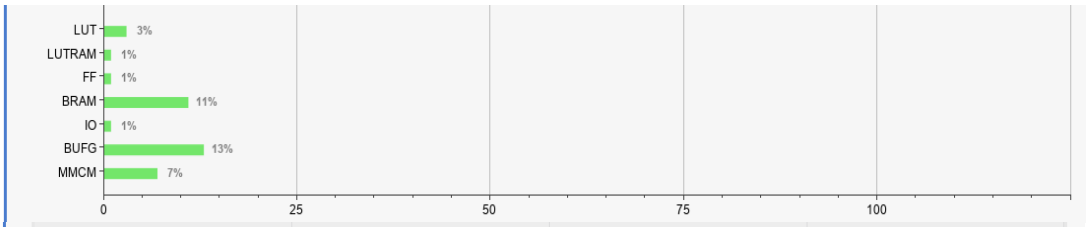
Giá trị các thanh ghi mô phỏng phần cứng trong Hình 5.17 đúng với kết quả mong đợi trong Bảng 5.19 chứng mình thiết kế chạy đúng đối với Testcase 9.

Bảng 5.19: Kết quả mong đợi Testcase 9

CPU A		CPU B	
x0 = 00000000	x16 = 00000000	x0 = 00000000	x16 = 00000000
x1 = 00001400	x17 = 00000000	x1 = 00001400	x17 = 00000000
x2 = 000003ae	x18 = 00000000	x2 = 00000764	x18 = 00000000
x3 = 00000000	x19 = 00000000	x3 = 00000000	x19 = 00000000
x4 = 00000000	x20 = 00000000	x4 = 00000000	x20 = 00000000
x5 = 00000000	x21 = 00000000	x5 = 00000000	x21 = 00000000
x6 = 00000000	x22 = 00000000	x6 = 00000000	x22 = 00000000
x7 = 00000000	x23 = 00000000	x7 = 00000000	x23 = 00000000
x8 = 00000000	x24 = 00000000	x8 = 00000000	x24 = 00000000
x9 = 00000000	x25 = 00000000	x9 = 00000000	x25 = 00000000
x10 = 00000000	x26 = 00000000	x10 = 00000000	x26 = 00000000
x11 = 00000000	x27 = 00000000	x11 = 00000000	x27 = 00000000
x12 = 00000000	x28 = 00000000	x12 = 00000000	x28 = 00000000
x13 = 00000000	x29 = 00000000	x13 = 00000000	x29 = 00000000
x14 = 00000000	x30 = 00000000	x14 = 00000000	x30 = 00000000
x15 = 00000000	x31 = 00000000	x15 = 00000000	x31 = 00000000

5.3. Kết quả tổng hợp, thực thi và đánh giá thiết kế

5.3.1. Kết quả tổng hợp, thực thi thiết kế



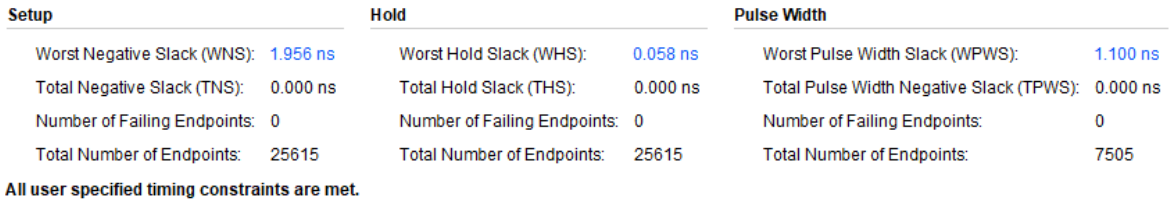
Resource	Utilization	Available	Utilization %
LUT	8598	303600	2.83
LUTRAM	390	130800	0.30
FF	6732	607200	1.11
BRAM	108.50	1030	10.53
IO	5	700	0.71
BUFG	4	32	12.50
MMCM	1	14	7.14

Hình 5.18: Tài nguyên sử dụng của hệ thống

Thiết kế sử dụng số lượng LUT là 8598, số lượng FF là 6732 và số lượng BRAM là 108.5. Lượng tài nguyên BRAM được sử dụng khá nhiều trên FPGA do việc tích hợp các vùng RAM để lưu trữ bộ nhớ làm tiêu tốn nhiều tài nguyên.

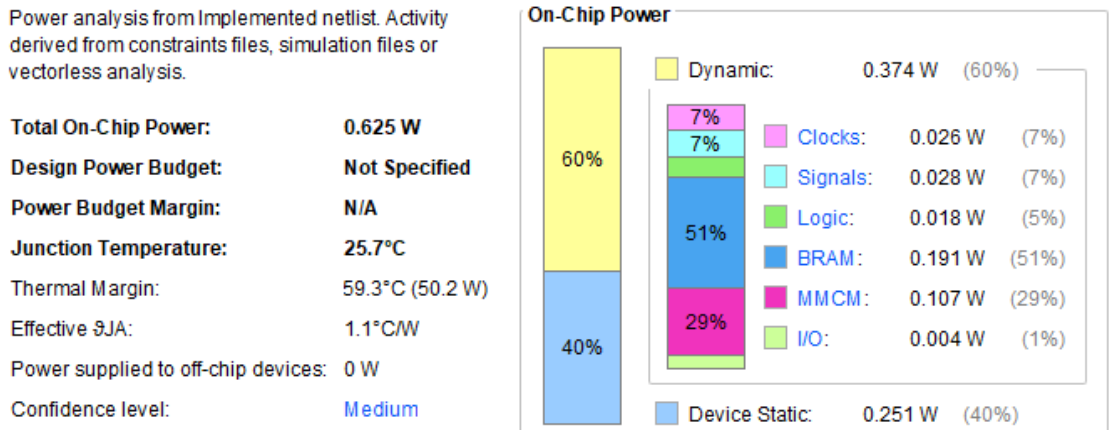
```
1 create_clock -period 10.000 -name ACLK -waveform {0.000 5.000} [get_ports ACLK]
```

Hình 5.19: Thiết lập tần số cho hệ thống



Hình 5.20: Kết quả tổng hợp timing của hệ thống

Theo kết quả tổng hợp timing của thiết kế trong Hình 5.20, tần số tối đa toàn hệ thống có thể hoạt động là 124MHz.



Hình 5.21: Báo cáo năng lượng tiêu thụ của hệ thống

Thiết kế tiêu thụ tổng cộng 0.625 W, trong đó BRAM và MMCM chiếm phần lớn năng lượng tiêu thụ - lần lượt là 51% và 29%.

5.3.2. Đánh giá thiết kế

Sau khi chạy tất cả Testcase, nhóm tính được các thông số trong Bảng 5.20:

Bảng 5.20: Kết quả thống kê hiệu năng

Thành phần	Thông số	Đơn vị	Giá trị
I-Cache	Hit Rate	(%)	99.4
	Miss Time	(ns)	205.8
	Hit Time	(ns)	22.05
D-Cache	Hit Rate	(%)	99.7
	Miss Time Clean	(ns)	235.2
	Miss Time Dirty	(ns)	396.9
	Hit Time	(ns)	22.05

Mô tả các thông số đề cập:

- Hit Rate: Tỉ lệ truy cập hit.
- Miss Time: Thời gian hoàn thành truy cập miss đối với I-Cache.
- Miss Time Clean: Thời gian hoàn thành truy cập miss nhưng không cần Write-back.
- Miss Time Dirty: Thời gian hoàn thành truy cập miss khi Set đầy, cần phải thay thế dữ liệu và Write-back dữ liệu do dữ liệu đã bị chỉnh sửa (Dirty).

Sau khi hoàn thành quá trình chạy thực thi và kiểm thử thiết kế, kết cuối cùng cho ra các thông số khá khả quan. Cụ thể, tỉ lệ Hit Rate đạt rất cao nhờ vào việc tận dụng tính Temporal Locality hay còn có thể giải thích đơn giản là xu hướng dữ liệu sẽ được dùng lại trong tương lai gần sẽ ở gần nhau, do đó khi thực hiện nhiều truy cập vào các ô nhớ liền kề thì chỉ cần tải lên dữ liệu của 1 Block gồm 16 word. Điều này chỉ gây miss truy cập đầu, toàn bộ 16 word dữ liệu sẽ được đưa lên và những truy cập kế tiếp vào ô dữ liệu đó sẽ ngay lập tức hit. Thời gian Hit Time nhờ vậy cũng được giảm xuống đáng kể khi chỉ mất từ 2 đến 3 chu kỳ để hoàn thành truy cập dữ liệu. Thời gian Miss Time thì lại mất nhiều chu kỳ hơn để thực hiện nhưng những con số này vẫn trong mức chấp nhận được, hoàn toàn có thể cải thiện được trong tương lai. Bên cạnh đó, 2 thông số còn lại là Miss Time Dirty và Miss Time Clean cũng khá cao khi hệ thống cần phải tốn khá nhiều chu kỳ cho nhiều công việc như tìm kiếm, thay thế khối và Write-back toàn bộ các ô dữ liệu (gồm 16 word dữ liệu) khi dữ liệu đó đã bị chỉnh sửa (Dirty).

Tuy thiết kế chưa được tối ưu toàn diện nhưng đã hoạt động tốt, đảm bảo mọi chức năng mong muốn được thực thi đúng. Ngoài thời gian truy cập miss cần cải thiện ra thì các chỉ số quan trọng như tần số của thiết kế hay tỉ lệ truy cập hit đều đạt mức cao, danh sách các Testcase được kiểm thử cũng đầy đủ chi tiết, bao quát được cả những trường hợp điển hình và hiếm gặp.

5.3.3. So sánh kết quả thiết kế với các thiết kế hiện có

Bảng 5.21 dưới đây so sánh kết quả của nhóm đạt được với các công trình và các đề tài liên quan.

Bảng 5.21: So sánh kết quả đạt được với các công trình khác

Các yếu tố so sánh	**Khóa luận này**	**Trong nước**		**Quốc tế**	
		Tác giả	Tác giả	Tác giả	Tác giả
		Nguyễn Thế Đạt [1]	C. T. Bình, D. P. Hùng [3]	D. P. Kaur, Sulochana [10]	H. Sh. Mahmood [14]
Kiến trúc	4-Way Set Associative	4-Way Set Associative	4-Way Set Associative	2-Way Set Associative	Direct Mapping
Chính sách thay thế	PLRUt	FIFO	MPLRU	Không đề cập	Không
Đồng bộ hóa	Có	Không	Không	Có	Không
Số lượng vi xử lý	2-Core	Không	Không	2-Core	Không
Chính sách ghi	Write-through Write-back	Write-back	Write-around Write-back	Write-back	Write-back
Tần số	136MHz	90.12 MHz	100MHz	Không đề cập	100MHz
Mô hình kiểm thử	Có	Không	Có	Không	Không
Board sử dụng	Virtex-7 VC707	Không	Virtex-7 VC707	Không	Spartan-3AN

Khóa luận này so các công trình nghiên cứu trong Bảng 5.21 có đã thiết kế thành công và sử dụng trúc bộ nhớ đệm 4-Way Set Associative phù hợp với kích thước hệ thống vừa và nhỏ. Về giải thuật thay thế, khóa luận lựa chọn cài đặt giải thuật PLRUt với khả năng thực hiện tìm kiếm khối sẽ bị thay thế mà chỉ cần dùng đến N-1 bit với kiến trúc có N-Way, tiết kiệm tài nguyên hơn khi sử dụng ít bit hơn so với giải thuật FIFO hay MPLRU thuộc đề tài của tác giả Nguyễn Thế Đạt và Cao Thanh Bình, Đặng Phi Hùng. Đồng thời, các chính sách ghi cần thiết và phổ biến trong bộ nhớ đệm cũng được áp dụng trong khóa luận này gồm chính sách Write-through và Write-back.

So về số lượng vi xử lý thì có khóa luận này và nghiên cứu của đồng tác giả D. P. Kaur, Sulochana thiết kế hệ thống đa vi xử lý với 2 lõi xử lý, đều thiết kế và sử dụng thành công bộ điều khiển Cache Coherence với giao thức MOESI Snoopy. Các đề tài khác chỉ sử dụng kiến trúc đơn lõi do đó không hỗ trợ giải quyết Cache Coherence.

Các công trình nghiên cứu được đề cập trong Bảng 5.21 và khóa luận này đều thực hiện kiểm thử thiết kế thành công, riêng đối với khóa luận này còn xây dựng thêm mô hình kiểm thử với phần mềm với Golden Model có thể cho ra kết quả tính toán mong đợi, phục vụ cho việc so sánh, đối chiếu một cách khách quan hơn. Tần số hoạt động của thiết kế được đề cập đến là tần số hoạt động của toàn hệ thống có tích hợp cả bộ nhớ đệm, trong đó tần số hoạt động của bộ nhớ đệm mà khóa luận này thực hiện có thể đạt tối đa lên đến 136MHz, cao hơn đáng kể so các đề tài còn lại được nhắc đến.

Về phần hiện thực thiết kế lên FPGA, khóa luận này (kit Virtex-7 VC707) và 2 đề tài thuộc của tác giả Cao Thanh Bình (kit Virtex-7 VC707) và đồng tác giả D. P. Kaur, Sulochana (kit Spartan-3AN) là những đề tài thực hiện thành công.

CHƯƠNG 6. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

6.1. Kết luận

Sau khi hoàn thành đề tài, nhóm đã đạt được những kết quả như sau:

- Thiết kế đúng về mặt chức năng của từng thành phần chi tiết đến toàn bộ hệ thống.
- Thiết kế được kiến trúc 4-Way Set Associative Cache.
- Hiện thực được giải thuật thay thế Cache PLRUt, chính sách Write-back và Read/Write Allocate cho kiến trúc 4-Way Set Associative Cache.
- Thiết kế và tích hợp thành công giao thức đảm bảo Cache Coherence MOESI Snoopy cho kiến trúc 4-Way Set Associative Cache cho hệ thống gồm 2 vi xử lý.
- Thiết kế hoàn thiện CPU RISC-V RV32I theo kiến trúc pipeline 5 tầng.
- Thiết kế được chức năng cơ bản của giao thức AXI mở rộng giao thức ACE nhằm hỗ trợ Cache Coherence.
- Xây dựng tích hợp được phần cứng và phần mềm mô hình hệ thống đa vi xử lý gồm: 2 CPU RISC-V RV32I, 2 I-Cache, 2 D-Cache, AXI Bus mở rộng ACE, Main Memory.
- Tần số của lõi xử lý RISC-V RV32I đạt 183MHz, bộ nhớ đệm 4-Way Set Associative Cache đạt 136MHz và của toàn hệ thống đạt 124MHz.
- Xây dựng được hệ thống đồng kiểm tra giữa phần mềm và phần cứng.

6.2. Ưu điểm và hạn chế của thiết kế

6.2.1. Ưu điểm

Ưu điểm của thiết kế:

- Hệ thống được tích hợp đầy đủ các chức năng chính của Cache trong hệ thống đa vi xử lý như giao thức đồng bộ hóa MOESI Snoopy cũng như

các chức năng điển hình của Cache như giải thuật thay thế PLRUt, chính sách Write-back và Read/Write Allocate.

- Các thiết kế thành phần cũng như toàn hệ thống hoạt động ở tần số cao.
- Hiện thực thiết kế trên FPGA thành công đúng với mục tiêu đặt ra.

6.2.2. Hạn chế

Nhược điểm của thiết kế:

- Hệ thống hiện đang hiện thực đồng nhất dữ liệu cho 2 vi xử lý, cần phải tinh chỉnh để hỗ trợ cho hệ thống N vi xử lý (N > 2).
- Đối với CPU RISC-V RV32I, lõi xử lý hiện tại không hỗ trợ các lệnh liên quan đến Control and Status Register Instructions và Environment Call and Breakpoints.

6.3. Hướng phát triển

Trong tương lai, để cải tiến chất lượng, cũng như đối với sự phát triển thêm của đề tài và để cải thiện kỹ năng, kiến thức của bản thân, nhóm đề xuất các ý sau:

- Tinh chỉnh thiết kế để hỗ trợ cho hệ thống N vi xử lý (N > 2).
- Tích hợp module MMU, DMA và các I/O tạo thành hệ thống hoàn chỉnh.

TÀI LIỆU THAM KHẢO

[1] Nguyễn Thế Đạt, “Thiết kế bộ điều khiển Cache 2 mức,” Trường Đại học Công nghệ Thông tin, Đại học Quốc gia TP.HCM, 2021.

[2] Trần Quốc Trưởng, Lê Phước Nhật Nam, “Thiết kế và hiện thực bộ vi xử lý RISC-V 32 BIT sử dụng kiến trúc superscalar hỗ trợ bộ điều khiển cache associative 4-way và đơn vị quản lý bộ nhớ trên FPGA,” Trường Đại học Công nghệ Thông tin, Đại học Quốc gia TP.HCM, 2022.

[3] Cao Thanh Bình, Đặng Phi Hùng, “Thiết kế và hiện thực lõi vi xử lý RISC-V RV32IF hỗ trợ 4-way set associative cache và bộ tạo số ngẫu nhiên thực trên FPGA,” Trường Đại học Công nghệ Thông tin, Đại học Quốc gia TP.HCM, 2023.

[4] Trần Tuấn Khánh, Phạm Tuấn Lâm, “Thiết kế và hiện thực lõi vi xử lý RISC-V RV64IM theo kiến trúc superscalar, hỗ trợ 4-Way Set Associative Cache và Branch Prediction trên FPGA,” Trường Đại học Công nghệ Thông tin, Đại học Quốc gia TP.HCM, 2023.

[5] David Money Harris, Sarah L. Harris, “Digital Design and Computer Architecture,” Morgan Kaufmann, 2nd edition, 2012.

[6] Safaa Omran, Ibrahim A. Amory, “Implementation of LRU Replacement Policy for Reconfigurable Cache Memory Using FPGA,” 2018 International Conference on Advanced Science and Engineering (ICOASE), Iraq, 2018.

[7] Hussein Al-Zoubi, Aleksandar Milenkovic, Milena Milenkovic, “Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite,” ACMSE’04, USA, 2004.

[8] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., Joel S. Emer, “High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP),” ACM SIGARCH Computer Architecture News, 2010.

[9] Anoop Tiwari, “Performance Comparison of Cache Coherence Protocol on Multi-Core Architecture,” B.Tech Thesis, National Institute of Technology, India, 2014.

[10] Daman Preet Kaur, V. Sulochana, “Design and Implementation of Cache Coherence Protocol for High-Speed Multiprocessor System,” 2nd IEEE International Conference on Power Electronics, Intelligent Control and Energy Systems (ICPEICES), India, 2018.

[11] Andrew Waterman, Krste Asanović, “The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture,” RISC-V International, 2024.

[12] Arm Ltd., “AMBA AXI and ACE Protocol Specification,” ARM IHI 0022H.c, January 2021.

[13] David A. Patterson, John L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface, RISC-V Edition,” Morgan Kaufmann, 2018.

[14] Hadeel Mahmood, Safaa Omran, “Pipelined MIPS processor with cache controller using VHDL implementation for educational purposes,” Proceedings of the 2013 International Conference on Electronics, Computer and Computation (ICECCPCE), 2014.