**ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH**

**TRƯỜNG ĐẠI HỌC BÁCH KHOA**

**KHOA ĐIỆN – ĐIỆN TỬ**

**BỘ MÔN ĐIỆN TỬ**


**ĐỒ ÁN 2**

# ĐỀ TÀI: DESIGN OF A TWO-LEVEL CACHE SYSTEM FOR RISC-V ARCHITECTURE

GVHD : **TS TRẦN HOÀNG LINH**

SVTH : **NGUYỄN ĐẮC TÂM**

MSSV : **2114714**

TP. HỒ CHÍ MINH, THÁNG 12 NĂM 2024

# LỜI CẢM ƠN

Lời đầu tiên em xin trân trọng gửi lời cảm ơn tới thầy TS. Trần Hoàng Linh – Bộ môn Điện tử, khoa Điện – Điện tử, trường Đại học Bách Khoa – ĐHQG – HCM đã tạo điều kiện và giúp đỡ em trong quá trình thực hiện đồ án này. Em cũng muốn gửi lời cảm ơn đến TA. Cao Xuân Hải đã giúp đỡ tận tình về mặt chuyên môn trong suốt quá trình em thực hiện đồ án.

Bên cạnh đó em cũng xin gửi lời tri ân đến các thầy cô khoa Điện – Điện tử, trường Đại học Bách Khoa – ĐHQG – HCM đã cung cấp những kiến thức quý giá trong suốt thời gian học tập tại trường giúp em có kiến thức nền tảng để hoàn thành đồ án này.

# LỜI CAM ĐOAN

Em xin cam đoan đồ án này hoàn toàn là sản phẩm của quá trình nghiên cứu, thực hiện của bản thân, không sao chép bất kì tài liệu, ấn phẩm nào. Các tư liệu tham khảo được sử dụng đúng quy định và có trích dẫn rõ ràng. Các kết quả, kết luận được đưa ra hoàn toàn dựa trên số liệu khách quan, rõ ràng và trung thực.

Em xin chịu trách nhiệm hoàn toàn về lời cam đoan này.

Hồ Chí Minh, tháng 12 năm 2024

Nguyễn Đắc Tâm

# MỤC LỤC

# DANH MỤC HÌNH ẢNH

# DANH MỤC BẢNG

# 1. Giới thiệu đề tài

## 1.1. Tổng quan đề tài

Trong bối cảnh ngành công nghệ thông tin và điện tử phát triển nhanh chóng, việc thiết kế các bộ vi xử lý hiệu quả, tiết kiệm năng lượng và phù hợp với nhiều ứng dụng đang là một thách thức lớn đối với các kỹ sư. RISC-V, một kiến trúc tập lệnh mã nguồn mở, đã nổi lên như một giải pháp tối ưu nhờ tính linh hoạt, khả năng tùy chỉnh cao và cộng đồng phát triển rộng lớn.

Mục tiêu của việc thiết kế vi xử lý là đạt được hiệu quả và tốc độ cao, nhưng quá trình đọc và ghi từ vi xử lý tới bộ nhớ chiếm một lượng lớn thời gian, do đó ảnh hưởng rất lớn đến tốc độ của vi xử lý. Sử dụng bộ nhớ đệm, hay được gọi là cache, chính là giải pháp giúp tối ưu được tốc độ cho vi xử lý.

Cache là một bộ nhớ nhỏ có tốc độ truy xuất cao và được thiết kế gần vi xử lý (có thể ở trong hoặc ngoài tùy thuộc vào level của cache và yêu cầu thiết kế). Cache chứa dữ liệu thường xuyên được truy cập từ bộ nhớ chính, qua đó giúp giảm thời gian truy cập dữ liệu của vi xử lý. Cache thường được thiết kế nhiều tầng và tạo thành cấu trúc theo tầng gọi là memory hierarchy, việc thiết kế nhiều tầng cache (multilevel caches) giúp cải thiện hiệu năng của cache nhờ việc giảm thời gian dữ liệu được truyền tải từ bộ nhớ chính khi cache không chứa dữ liệu cần truy cập. Trong kiến trúc CPU hiện đại, việc sử dụng hệ thống bộ nhớ đệm hai cấp (L1 và L2) là giải pháp phổ biến để cân bằng giữa tốc độ truy cập và dung lượng lưu trữ.

Đề tài "Design of a Two-level Cache System for RISC-V Architecture" tập trung vào việc xây dựng và mô phỏng một vi xử lý RISC-V tích hợp hệ thống bộ nhớ đệm L1 và L2. Đề tài không chỉ hướng tới việc thiết kế một hệ thống đáp ứng các yêu cầu cơ bản của RISC-V mà còn tối ưu hóa hiệu năng thông qua việc sử dụng bộ nhớ đệm hai cấp.

## 1.2. Mục tiêu đề tài

Mục tiêu của đề tài là nắm vững các nguyên lý và đặc điểm của kiến trúc RISC-V, bao gồm tập lệnh, tổ chức phần cứng, và khả năng mở rộng. Triển khai một CPU RISC-V cơ bản theo tập lệnh RV32I để làm nền tảng cho thiết kế.

Sử dụng ngôn ngữ mô tả phần cứng SystemVerilog thiết kế 5-stage pipelined RISC-V processor dựa trên tập lệnh RV32I, processor được xử lý hazard bằng kỹ thuật forwarding, có bộ branch prediction 2-bit scheme. Thiết kế bộ nhớ cho processor bao gồm 2 tầng, L1 và L2 cache. Trong đó

có 2 L1 caches, bao gồm 1 data cache và 1 instruction cache, 1 L2 cache có kích thước lớn hơn nhưng chậm hơn. Cache được thiết kế theo cấu trúc set associative có sử dụng giải thuật tree-PLRU cho replacement policy. Từ thiết kế, đánh giá các thông số cho biết hiệu quả của việc thiết kế hệ thống Two-level cache cho vi xử lý Risc-V.

### 1.3. Cấu trúc của báo cáo

Bài báo cáo này bao gồm những phần chính như sau:

- Phần 1: Giới thiệu về đề tài và nêu ra mục tiêu chính cần thực hiện.
- Phần 2: Tập trung vào nền tảng lý thuyết cho những thành phần cấu thành nền thiết kế của đề tài.
- Phần 3: Tập trung chi tiết vào thiết kế, giải thích những module và chức năng của chúng. Cấu trúc của cache hay các thành phần cấu tạo nên cache cũng được giải thích trong phần này.
- Phần 4: Kiểm tra các lệnh được thiết kế cũng như hoạt động của hệ thống cache trong CPU.
- Phần 5: Phân tích, đánh giá hiệu năng của hệ thống cache trong vi xử lý.
- Phần 6: Đưa ra kết luận và hướng phát triển của đề tài.

### 1.4. Source code

Tất cả source code cho đề tài được đăng tải lên trang Github:

https://github.com/Stork1323/riscv_cache

# 2. Cơ sở lý thuyết

## 2.1. Giới thiệu về RISC-V

RISC-V là một kiến trúc tập lệnh (ISA - Instruction Set Architecture) theo chuẩn RISC (Reduced Instruction Set Computer), được thiết kế với mục tiêu đơn giản, linh hoạt và mở. Khác với nhiều ISA thương mại như x86 hay ARM, RISC-V có mã nguồn mở và được phát triển nhằm phục vụ cả giáo dục lẫn thương mại. Các đặc điểm nổi bật của RISC-V đã làm cho nó trở thành một sự lựa chọn hấp dẫn trong nhiều lĩnh vực từ thiết kế hệ thống nhúng cho đến các ứng dụng máy tính cao cấp.

RISC-V bắt đầu vào năm 2010 tại Đại học California, Berkeley, bởi một nhóm nghiên cứu do giáo sư David Patterson và các đồng nghiệp dẫn đầu. Nhóm nghiên cứu này đặt mục tiêu tạo ra một kiến trúc đơn giản, dễ học và dễ triển khai. Một trong những lý do chính để phát triển RISC-V là để giải quyết những hạn chế của các kiến trúc tập lệnh hiện có, chẳng hạn như sự phức tạp, chi phí cấp phép cao, và không thể mở rộng tự do. RISC-V được thiết kế với tư duy mở, có thể sử dụng trong cả giáo dục lẫn thương mại mà không gặp các vấn đề bản quyền.

RISC-V tuân theo các nguyên tắc chính của kiến trúc RISC, bao gồm:

- Đơn giản hóa tập lệnh: Chỉ sử dụng một tập nhỏ các lệnh cơ bản, mỗi lệnh có độ dài cố định và đơn giản. Điều này giúp tối ưu hóa quá trình xử lý, giảm thiểu số lệnh phức tạp cần phải xử lý.
- Load-Store Architecture: RISC-V là kiến trúc load-store, tức là tất cả các thao tác trên dữ liệu đều được thực hiện trong thanh ghi, còn việc truy cập bộ nhớ chỉ được thực hiện thông qua các lệnh load và store.
- Độ dài lệnh cố định: Các lệnh trong RISC-V có độ dài 32-bit, giúp đơn giản hóa bộ giải mã và tăng hiệu quả khi xử lý.
- Modular: RISC-V được thiết kế với khả năng mở rộng, cho phép người dùng thêm các phần mở rộng cho các ứng dụng đặc thù như tính toán số nguyên, số chấm động, hoặc các ứng dụng vector.

Các đặc điểm nổi bật của RISC-V:

- Mã nguồn mở: RISC-V hoàn toàn miễn phí và mã nguồn mở, giúp các nhà phát triển có thể sử dụng và tùy chỉnh mà không phải trả phí cấp phép như với các kiến trúc độc quyền khác.

- Tính mô-đun và linh hoạt: RISC-V cho phép người dùng thêm hoặc bớt các phần mở rộng theo nhu cầu cụ thể của từng hệ thống. Ví dụ, có thể thêm phần mở rộng xử lý dấu phẩy động (F), phần mở rộng vector (V), hoặc phần mở rộng bảo mật (S).
- Khả năng mở rộng: Kiến trúc RISC-V có thể được triển khai cho các hệ thống từ nhúng (32-bit) đến các siêu máy tính (64-bit hoặc 128-bit). Điều này cho phép RISC-V đáp ứng được nhiều yêu cầu và mức độ phức tạp của các ứng dụng khác nhau.
- Hiệu suất cao và tiết kiệm năng lượng: Với việc tuân thủ nguyên lý thiết kế RISC, RISC-V cho phép tối ưu hóa hiệu suất tính toán và tiêu thụ năng lượng, đặc biệt phù hợp với các ứng dụng di động và nhúng.

RISC-V được chia thành nhiều phiên bản để phù hợp với các nhu cầu và ứng dụng khác nhau:

- RV32: Kiến trúc 32-bit, phù hợp với các ứng dụng nhúng, hệ thống nhỏ gọn.
- RV64: Kiến trúc 64-bit, phù hợp với các ứng dụng yêu cầu tính toán cao hoặc lượng bộ nhớ lớn.
- RV128: Kiến trúc 128-bit, được thiết kế cho các ứng dụng tương lai với yêu cầu rất lớn về hiệu năng và không gian địa chỉ.

## 2.2. Kiến trúc tập lệnh RV32I

Kiến trúc tập lệnh RV32I là tập lệnh cơ bản của RISC-V, được thiết kế để hoạt động trên bộ vi xử lý với không gian địa chỉ 32-bit. RV32I là tập lệnh cơ bản và nhỏ gọn, hỗ trợ các thao tác xử lý đơn giản và là nền tảng cho các tập lệnh mở rộng khác của RISC-V (như RV64I cho kiến trúc 64-bit). Dưới đây là các khái niệm và đặc điểm chính của RV32I:

a. Đặc điểm cơ bản của RV32I:

- 32-bit word: RV32I sử dụng word 32-bit cho địa chỉ và dữ liệu.
- Tối ưu hóa cho đơn giản: Tập lệnh của RV32I được thiết kế đơn giản, với số lượng ít các loại lệnh cơ bản, giúp dễ dàng triển khai và tối ưu hóa.
- Tính mô-đun: RV32I là một tập lệnh cơ sở, có thể được mở rộng với các tập lệnh khác như RV32E, RV32M (nhân chia), hoặc các phần mở rộng khác như F, D (số dấu phẩy động).

b. Các nhóm lệnh chính của RV32I:

- Lệnh Load và Store: các lệnh load (tải dữ liệu từ bộ nhớ vào thanh ghi) và store (lưu dữ liệu từ thanh ghi vào bộ nhớ) là cơ bản trong mọi kiến trúc RISC. Ví dụ: LW (Load Word), SW (Store Word), LB (Load Byte), SB (Store Byte).
- Lệnh tính toán số nguyên: các lệnh này thực hiện các phép tính toán số nguyên giữa các thanh ghi. Ví dụ: ADD (cộng), SUB (trừ), AND, OR, XOR.

- Lệnh dịch và so sánh: dịch bit (shift) và so sánh là các thao tác phổ biến trong nhiều ứng dụng. Ví dụ: SLL (dịch trái), SRL (dịch phải không dấu), SLT (so sánh nhỏ hơn).
- Lệnh nhảy và phân nhánh: các lệnh này điều khiển luồng chương trình, cho phép nhảy đến các địa chỉ khác nhau trong bộ nhớ. Ví dụ: JAL (Jump and Link), BEQ (nhảy khi bằng), BNE (nhảy khi không bằng).
- Lệnh immediate: đây là các lệnh tính toán với một hằng số (immediate) được gán trực tiếp vào lệnh. Ví dụ: ADDI (cộng với immediate), ORI (OR với immediate).
- Lệnh hệ thống: những lệnh này liên quan đến tương tác với phần mềm và hệ thống, như gọi hệ thống hoặc quản lý ngoại lệ. Ví dụ: ECALL (gọi hệ thống), EBREAK (dừng chương trình).

c. Thanh ghi và các đặc điểm liên quan:

- 32 thanh ghi: RV32I có 32 thanh ghi toàn phần (x0 - x31), mỗi thanh ghi rộng 32-bit. Trong đó:
- x0 luôn có giá trị bằng 0.
- Các thanh ghi khác có thể lưu trữ dữ liệu tạm thời và giá trị địa chỉ.
- Program Counter (PC): Thanh ghi PC lưu trữ địa chỉ của lệnh hiện tại đang được thực thi.

d. Định dạng lệnh của RV32I:

Tất cả các lệnh trong RV32I đều có độ dài cố định là 32 bit. Có năm định dạng lệnh chính trong RV32I, được chia theo các trường khác nhau:

- R-type: Dùng cho các lệnh tính toán số nguyên giữa các thanh ghi.
- I-type: Dùng cho các lệnh tính toán có immediate, cũng như các lệnh load.
- S-type: Dùng cho các lệnh store.
- B-type: Dùng cho các lệnh nhảy có điều kiện.
- J-type: Dùng cho các lệnh nhảy không điều kiện.

e. Ưu điểm của RV32I:

- Đơn giản và linh hoạt: RV32I chỉ có các lệnh cơ bản, giúp việc triển khai phần cứng trở nên dễ dàng và nhanh chóng.
- Tối ưu hóa hiệu suất và điện năng: Kiến trúc đơn giản giúp tiết kiệm tài nguyên phần cứng và giảm mức tiêu thụ điện năng.

- Tính mở rộng: RV32I có thể được mở rộng với các phần mở rộng khác nhau tùy theo nhu cầu ứng dụng.

f. Ứng dụng của RV32I:

RV32I chủ yếu được sử dụng trong các hệ thống nhúng, các thiết bị có giới hạn về tài nguyên và yêu cầu tiết kiệm năng lượng, nhưng cũng có thể mở rộng để đáp ứng các yêu cầu phức tạp hơn.

## 2.3. Pipeline

Trong thiết kế bộ xử lý RISC-V, kỹ thuật pipeline là một cách tiếp cận phổ biến để tăng hiệu năng bằng cách chia nhỏ quá trình xử lý các lệnh thành các giai đoạn (stages) độc lập, giúp cho việc thực hiện nhiều lệnh đồng thời. Điều này giúp tối ưu hóa thời gian xử lý, giảm độ trễ và tăng số lượng lệnh được xử lý mỗi giây (IPC - Instructions per Cycle). Pipeline của RISC-V thường bao gồm các giai đoạn chính sau:

1. IF (Instruction Fetch): ở giai đoạn này, bộ xử lý truy cập bộ nhớ để lấy mã lệnh (instruction) từ địa chỉ trong bộ đếm lệnh (Program Counter - PC). Sau đó, bộ đếm lệnh được cập nhật để chuẩn bị cho lệnh tiếp theo.
2. ID (Instruction Decode): trong giai đoạn giải mã, bộ xử lý sẽ phân tích mã lệnh đã lấy được ở giai đoạn trước để xác định loại lệnh (như load, store, jump, arithmetic) và các toán hạng cần thiết (registers và immediate). Các toán hạng sẽ được lấy ra từ bộ register file hoặc từ lệnh.
3. EX (Execute): đây là giai đoạn xử lý chính, trong đó lệnh sẽ được thực hiện. Đối với các lệnh số học hoặc logic, ALU (Arithmetic Logic Unit) sẽ tính toán kết quả. Nếu là các lệnh load/store, giai đoạn này sẽ tính toán địa chỉ bộ nhớ.
4. MEM (Memory Acces): với các lệnh load hoặc store, bộ xử lý sẽ truy cập bộ nhớ dữ liệu ở giai đoạn này. Đối với các lệnh khác, giai đoạn này có thể bỏ qua (NOP - No Operation).
5. WB (Write Back): kết quả của lệnh sẽ được ghi lại vào register file, hoàn tất quá trình thực hiện của lệnh.

Các kỹ thuật tối ưu pipeline trong RISC-V

- Forwarding: giúp giảm độ trễ do phụ thuộc dữ liệu giữa các lệnh.
- Branch Prediction: giúp giảm thời gian chờ đợi khi có lệnh rẽ nhánh bằng cách dự đoán địa chỉ PC tiếp theo.
- Hazard Detection: giảm thiểu xung đột giữa các lệnh bằng cách tạm dừng pipeline hoặc đưa các lệnh khác vào.

Pipeline giúp RISC-V xử lý nhiều lệnh song song và tận dụng tài nguyên bộ xử lý hiệu quả hơn, tạo điều kiện để đạt hiệu năng cao và độ trễ thấp trong thực thi chương trình.

Tính toán tốc độ khi sử dụng kỹ thuật pipeline:

Cho biết:

- m: số lượng instructions
- n: số lượng tầng pipeline
- t: thời gian hoàn thành ở mỗi tầng

Thời gian tính toán:

- Không pipelining: $mnt$
- Với pipelining: $(m + n - 1)t$

$$S(n) = \frac{mnt}{(m + n - 1)t} = \frac{mn}{m + n}$$

$$\lim_{m \to \infty} S(n) = n$$

Như tính toán về mặt lý thuyết, tốc độ khi sử dụng kỹ thuật pipeline sẽ đạt được gấp n lần so với không sử dụng pipeline. Do đó, khi thiết kế processor càng nhiều tầng pipeline thì sẽ đạt được tốc độ càng cao.

## 2.4. Hazard

Hazard là một loại lỗi xảy ra khi câu lệnh tiếp theo không thể thực thi ở chu kì clock tiếp đến. Có 2 loại hazard, bao gồm pipeline hazards (chứa structural hazard, data hazard) và control hazard (hoặc branch hazard).

Structural hazard xảy ra khi có tranh chấp tài nguyên, chẳng hạn như bộ ALU hoặc bộ nhớ. Kiến trúc pipeline của RISC-V thường được thiết kế để tránh loại xung đột này bằng cách tách tài nguyên cho các giai đoạn khác nhau.

Data hazard xảy ra khi một câu lệnh cần dữ liệu từ câu lệnh trước nhưng dữ liệu chưa sẵn sàng. Bao gồm 3 trường hợp:

- Lệnh thứ 2 đọc thanh ghi được ghi ở lệnh thứ nhất. Khi này 1 trong 2 hoặc cả 2 thanh ghi đưa vào bộ ALU ở tầng execute chưa được cập nhật giá trị vì lệnh trước đó đang ở tầng memory và chưa được đưa vào tầng write back để lưu lại register file. Do đó, cần phải stall lệnh ở tầng IF, ID và flush lệnh ở tầng execute.

- Lệnh thứ 3 đọc thanh ghi được ghi ở lệnh thứ nhất. Khi này 1 trong 2 hoặc cả 2 thanh ghi đưa vào bộ ALU ở tầng execute chưa được cập nhật giá trị vì lệnh thứ nhất trước đó đang ở tầng write back và lưu lại register file nhưng lệnh thứ 3 đã lấy giá trị ở tầng ID trước đó. Do đó, cần phải stall lệnh ở tầng IF, ID và flush lệnh ở tầng EX, MEM.
- Đọc thanh ghi được ghi sau lệnh load. Khi lệnh hiện tại đang ở tầng ID để lấy dữ liệu thanh ghi thì lệnh load trước đó vẫn chưa tới tầng MEM để load giá trị và trả về register file. Do đó cần phải stall lệnh ở tầng IF, ID và flush lệnh ở tầng EX, MEM, WB.

Control hazard: liên quan đến các lệnh rẽ nhánh, khi kết quả nhánh chưa được xác định. Bộ xử lý có thể dự đoán nhánh để tiếp tục pipeline, nhưng nếu dự đoán sai cần phải flush pipeline để sửa lỗi.

## 2.5. Branch prediction

Branch prediction là kỹ thuật quan trọng trong thiết kế vi xử lý để tăng hiệu suất của pipeline. Khi bộ xử lý gặp lệnh nhánh, nó phải quyết định hướng đi nào của chương trình sẽ được tiếp tục (tức là liệu điều kiện nhánh là đúng hay sai). Nếu nhánh được dự đoán đúng, pipeline có thể tiếp tục xử lý mà không cần tạm dừng. Ngược lại, nếu dự đoán sai, pipeline sẽ phải flush (xóa bỏ các lệnh không cần thiết đã nạp vào) và nạp lại các lệnh đúng, gây ra trễ.

Trong các ứng dụng hiện đại, lệnh nhánh xuất hiện rất thường xuyên, chiếm khoảng 10-20% tổng số lệnh. Nếu bộ xử lý có thể dự đoán đúng hướng của các nhánh, nó sẽ tránh được việc phải dừng pipeline, từ đó cải thiện số lệnh thực thi mỗi chu kỳ (IPC). Các kiến trúc pipeline nhiều tầng hơn đặc biệt cần đến kỹ thuật này, vì nếu một dự đoán nhánh sai, độ trễ sẽ rất lớn.

Branch prediction thường chia thành hai loại chính: Static Prediction và Dynamic Prediction.

Dự đoán tĩnh không yêu cầu phân tích từ các câu lệnh được thực thi trước mà áp dụng các quy tắc cố định:

- Always taken: luôn dự đoán rằng lệnh nhánh sẽ được thực thi. Kỹ thuật này hiệu quả cho các nhánh vòng lặp (loop), vì thường nhánh trong vòng lặp được thực thi nhiều lần.

- Always not taken: luôn dự đoán rằng lệnh nhánh sẽ không được thực thi, nghĩa là chương trình sẽ tiếp tục thực thi tuần tự. Kỹ thuật này hiệu quả cho các nhánh điều kiện ít khi xảy ra.

- Backward taken, forward not taken: đây là một kỹ thuật cải tiến, trong đó nhánh về sau (backward branch, thường là nhánh vòng lặp) được dự đoán là "taken", còn nhánh đi tới (forward branch) được dự đoán là "not taken".

Static prediction có chi phí thấp nhưng độ chính xác không cao và không thích hợp với các chương trình phức tạp có nhánh động (branch behavior) không ổn định.

Dynamic prediction sử dụng thông tin từ các lần thực thi trước đó để dự đoán nhánh hiện tại. Đây là phương pháp phức tạp và có độ chính xác cao hơn static prediction.

- 1-Bit Predictor: lưu trữ một bit cho mỗi lệnh nhánh, cho biết lần dự đoán gần nhất là "taken" hay "not taken". Khi lệnh nhánh được thực thi, predictor sẽ cập nhật giá trị này theo kết quả thực tế. Mặc dù đơn giản, 1-bit predictor dễ sai trong các vòng lặp, khi mà lệnh nhánh thay đổi trạng thái từ “taken” sang “not taken” ở lần cuối của vòng lặp.
- 2-Bit Saturating Counter: Sử dụng 2 bit để lưu trữ trạng thái của nhánh, ví dụ các trạng thái như “strongly taken”, “weakly taken”, “weakly not taken” và “strongly not taken”. Giúp cải thiện độ chính xác, vì bộ đếm chỉ thay đổi trạng thái khi có một số lần dự đoán sai liên tiếp, giúp giảm số lần dự đoán sai trong vòng lặp. Cơ chế này phổ biến trong các bộ xử lý hiện đại vì tính đơn giản và hiệu quả.
- Branch History Table (BHT): Sử dụng một bảng nhớ nhỏ để lưu trữ các lịch sử dự đoán của các lệnh nhánh. Mỗi mục trong bảng chứa các bits để lưu trạng thái (thường dùng 2-bit predictor) của một lệnh nhánh cụ thể. Khi gặp lệnh nhánh, bộ xử lý tra bảng BHT để đưa ra dự đoán. Các mục trong bảng được gán chỉ mục dựa trên địa chỉ của lệnh nhánh hoặc một phần của nó, do đó bộ xử lý có thể theo dõi nhiều nhánh cùng lúc.
- Global History Prediction (GHP): Lưu trữ lịch sử toàn cục của tất cả các nhánh trong một thanh ghi (global history register). Thường dùng với bộ dự đoán 2-bit để dự đoán hướng nhánh dựa trên chuỗi các nhánh trước đó. Bộ xử lý có thể xác định sự phụ thuộc giữa các nhánh và từ đó dự đoán chính xác hơn. Tuy nhiên, GHP đòi hỏi thêm tài nguyên và độ phức tạp cao hơn.
- Tournament Predictor: Đây là kỹ thuật phức tạp và có hiệu suất cao, kết hợp nhiều bộ dự đoán (predictor) khác nhau. Thông thường sẽ kết hợp một predictor cho các nhánh dễ đoán như vòng lặp (static hoặc local predictor) và một predictor khác cho các nhánh phức tạp hơn (global predictor). Sử dụng một bảng chọn (selector) để xác định predictor nào sẽ được dùng, dựa trên lịch sử thành công của mỗi predictor.

Khi dự đoán nhánh sai, các lệnh không cần thiết trong pipeline sẽ phải flush để pipeline có thể bắt đầu lại với đường đi đúng. Việc này có thể làm mất vài chu kỳ clock, dẫn đến giảm hiệu năng. Để giảm thiểu tác động của dự đoán sai, các kỹ thuật như Delayed Branch có thể được áp dụng, trong

đó một vài lệnh sau lệnh nhánh sẽ được thực thi trước khi kiểm tra kết quả của nhánh, giúp giảm thời gian pipeline bị dừng.

Branch prediction là kỹ thuật rất quan trọng để đạt hiệu suất cao trong thiết kế vi xử lý hiện đại. Các bộ xử lý ngày nay thường dùng các kỹ thuật dự đoán động phức tạp như 2-bit predictor, BHT, và Tournament Predictor để tăng độ chính xác và hiệu năng. Dự đoán nhánh càng chính xác, pipeline càng ít bị gián đoạn, giúp cải thiện IPC và tối ưu hiệu suất tổng thể của hệ thống.

## 2.5. Cache

### *2.5.1. Khái niệm*

Bộ nhớ đệm, được gọi là cache, nó là vùng bộ nhớ có tốc độ truy xuất cao dùng để lưu trữ tạm thời những câu lệnh chương trình hoặc dữ liệu. Những câu lệnh và dữ liệu này được trích xuất từ bộ nhớ chính và thường được sử dụng lại ngay sau đó. Mục đích chính của bộ nhớ cache là tăng tốc độ truy cập lặp lại đối với cùng một địa chỉ bộ nhớ và những địa chỉ ở gần đó. Để đạt được hiệu quả, truy cập dữ liệu đối với cache phải nhanh hơn rất nhiều so với truy cập vào dữ liệu ở bộ nhớ chính.

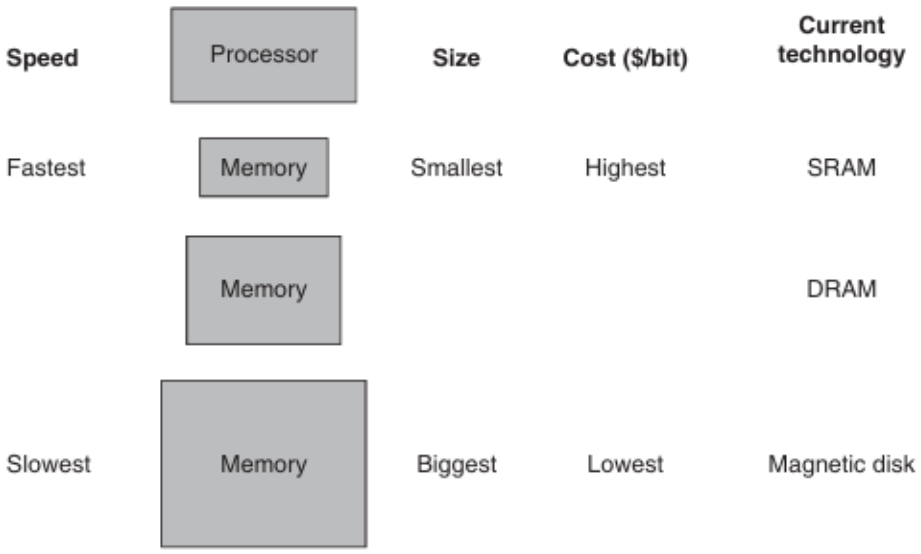
Khi cache được sử dụng, mỗi lần truy cập dữ liệu bộ nhớ đều bắt đầu tìm kiếm từ cache. Nếu dữ liệu yêu cầu có trong cache, processor sẽ trích xuất và sử dụng nó ngày lập tức, đây được gọi là cache hit. Nếu dữ liệu không tìm kiếm được trong cache, đây được gọi là cache miss, instruction hay data phải được trích xuất từ bộ nhớ. Trong quá trình lấy dữ liệu yêu cầu, nội dung được sao chép vào cache để sử dụng sau đó.

Bộ nhớ cache cải thiện hiệu suất của máy tính vì nhiều thuật toán được hệ điều hành và các ứng dụng thực thi thể hiện tính cục bộ tham chiếu. Tính cục bộ tham chiếu đề cập đến việc tái sử dụng dữ liệu đã được truy cập gần đây (điều này được gọi là tính cục bộ theo thời gian) và việc truy cập dữ liệu ở gần với dữ liệu đã được truy cập trước đó (điều này được gọi là tính cục bộ theo không gian).

Thông thường, các vùng bộ nhớ cache có kích thước nhỏ hơn so với bộ nhớ chính. Cache được thiết kế để đạt tốc độ tối đa, điều này có nghĩa là chúng phức tạp hơn và tốn kém hơn tính theo mỗi bit so với công nghệ lưu trữ dữ liệu được sử dụng trong bộ nhớ chính. Do kích thước hạn chế, cache thường nhanh chóng bị lấp đầy. Khi bộ nhớ cache không còn trống để lưu trữ dữ mới, một

dữ liệu cũ phải bị loại bỏ. Cache controller sẽ sử dụng một chính sách thay thế để chọn dữ liệu sẽ bị ghi đè bởi dữ liệu mới.

Mục tiêu của cache trong processor là tối đa hóa tỷ lệ cache hit theo thời gian, từ đó thực thi được những câu lệnh với tốc độ cao nhất có thể. Để đạt được mục tiêu này, cache phải xác định những câu lệnh và dữ liệu nào sẽ được đặt vào cache và giữ lại để sử dụng trong tương lai.




Hình 1. Cấu trúc cơ bản của hệ thống memory

Hình trên mô tả hệ thống bộ nhớ theo các tiêu chí như tốc độ, kích thước, chi phí và công nghệ. Đối với các loại bộ nhớ được thiết kế càng gần processor thì tốc độ phải càng tăng và kích thước càng nhỏ, tương ứng với đó thì chi phí cũng càng tăng.

### *2.5.2. Công nghệ bộ nhớ*

Có bốn công nghệ chính được sử dụng trong hệ thống phân cấp bộ nhớ hiện nay. Bộ nhớ chính được triển khai bằng DRAM (dynamic random access memory), trong khi các cấp gần bộ xử lý hơn (bộ nhớ cache) sử dụng SRAM (static random access memory). DRAM có chi phí trên mỗi bit thấp hơn so với SRAM, mặc dù nó chậm hơn đáng kể. Sự chênh lệch về giá xuất phát từ việc DRAM sử dụng diện tích ít hơn đáng kể trên mỗi bit bộ nhớ, do đó DRAM có dung lượng lớn hơn với cùng số lượng silicon. Công nghệ thứ ba là bộ nhớ flash. Đây là bộ nhớ không bay hơi (nonvolatile) được sử dụng làm bộ nhớ thứ cấp trong các thiết bị di động cá nhân. Công nghệ thứ tư, được sử dụng để triển khai có kích thước lớn nhất và chậm nhất trong hệ thống phân cấp bộ

nhớ trong các máy chủ là đĩa từ. Thời gian truy cập và chi phí trên mỗi bit khác nhau đáng kể giữa các công nghệ này, như bảng dưới đây cho thấy, sử dụng các giá trị của năm 2012.

| Memory technology | Typical access time | $ per GiB in 2012 |
|---|---|---|
| SRAM semiconductor memory | 0.5–2.5 ns | $500–$1000 |
| DRAM semiconductor memory | 50–70 ns | $10–$20 |
| Flash semiconductor memory | 5,000–50,000 ns | $0.75–$1.00 |
| Magnetic disk | 5,000,000–20,000,000 ns | $0.05–$0.10 |

Hình 2. Tốc độ và chi phí theo từng công nghệ bộ nhớ

### *2.5.3. Level 1 cache*

Trong kiến trúc đa bậc của bộ nhớ cache, các cấp cache được đánh số bắt đầu từ 1. Bộ nhớ cache cấp 1 (còn được gọi là L1 cache) là bộ nhớ cache đầu tiên mà processor tìm kiếm khi yêu cầu một lệnh hoặc mục dữ liệu từ bộ nhớ. Vì đây là điểm dừng đầu tiên trong quá trình tìm kiếm cache, nên L1 cache thường được xây dựng bằng công nghệ SRAM nhanh nhất có sẵn và được đặt càng gần mạch logic của processor càng tốt.

Việc tập trung vào tốc độ làm cho L1 cache đắt đỏ và tiêu tốn nhiều năng lượng, điều này có nghĩa là nó phải khá nhỏ so với bộ nhớ chính, đặc biệt là trong các ứng dụng có chi phí hạn chế. Ngay cả khi có kích thước khiêm tốn, một bộ nhớ cache cấp 1 có thể mang lại sự cải thiện đáng kể về hiệu suất so với một processor tương tự không sử dụng bộ nhớ cache.

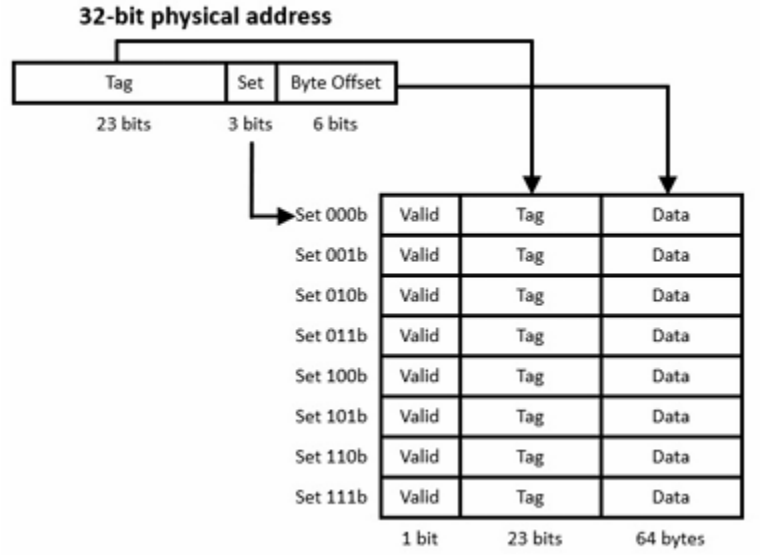
Bộ xử lý (hoặc MMU, nếu có) chuyển dữ liệu giữa DRAM và cache với các khối dữ liệu có kích thước cố định được gọi là dòng cache (cache lines). Các máy tính sử dụng module DDR DRAM thường có kích thước dòng cache là 64 bytes. Tất cả các cache ở nhiều bậc thường sử dụng chung một kích thước cache line.

Các bộ xử lý hiện đại thường chia bộ nhớ cache L1 thành hai phần có kích thước bằng nhau, một phần cho câu lệnh và một phần cho data. Cấu hình này được gọi là cache tách biệt (split cache). Trong split cache, bộ nhớ instruction cache cấp 1 được gọi là L1 I-cache và bộ nhớ data cache cấp 1 được gọi là L1 D-cache. Bộ xử lý sử dụng các bus riêng biệt để truy cập từng bộ nhớ cache, do đó triển khai một khía cạnh quan trọng của kiến trúc Harvard. Cách sắp xếp này tăng tốc độ thực thi lệnh bằng cách cho phép truy cập đồng thời cả câu lệnh và dữ liệu, giả sử rằng cả hai đều hit vào L1 cache.

### *2.5.4. Direct-mapped cache*

Direct-mapped cache là một khối bộ nhớ được tổ chức dưới dạng mảng một chiều của các tập hợp cache, trong đó mỗi địa chỉ trong bộ nhớ chính ánh xạ tới một tập hợp duy nhất trong cache. Mỗi tập hợp cache bao gồm các mục sau:

- Một cache line, chứa một khối dữ liệu được đọc từ bộ nhớ chính.
- Một giá trị tag, chỉ vị trí trong bộ nhớ chính tương ứng với dữ liệu được lưu trong cache.
- Một valid bit, cho biết liệu tập hợp cache có chứa dữ liệu hay không.



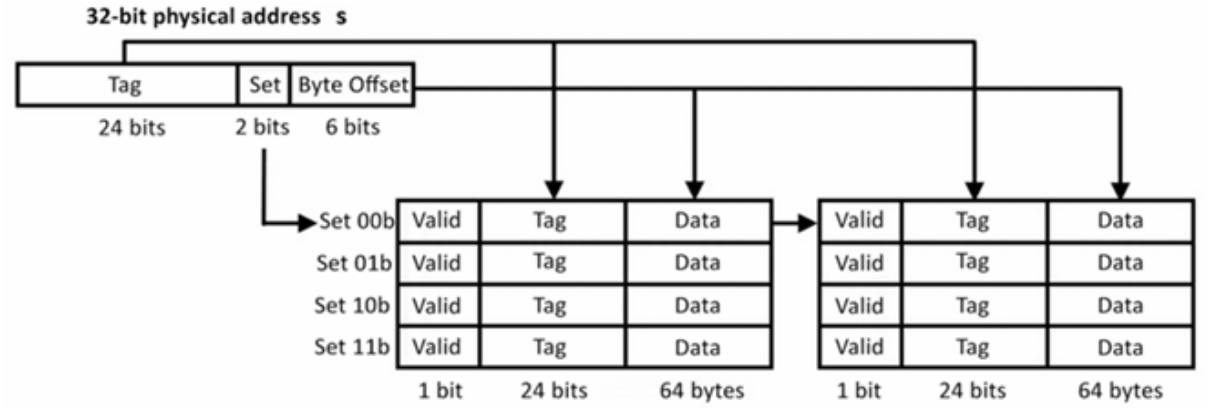

Hình 3. Mối quan hệ giữa một địa chỉ vật lý 32-bit và bộ nhớ cache

Hình ảnh này đại diện cho bộ nhớ cache lệnh, chỉ cho phép đọc. Bộ nhớ cache dữ liệu có thêm một bit nữa, được gọi là dirty bit. Nếu bit này được set, nó cho biết dữ liệu trong một dòng đã được sửa đổi và hệ thống phải ghi dữ liệu đó vào bộ nhớ trước khi cache line có thể được giải phóng để tái sử dụng.

Có những lúc bộ nhớ cache không chứa dữ liệu, chẳng hạn như ngay sau khi bật nguồn bộ xử lý. Valid bit cho mỗi tập hợp cache ban đầu được xóa để chỉ ra rằng không có dữ liệu trong tập hợp. Khi valid bit bị xóa, việc sử dụng tập hợp để tra cứu bị cản trở. Một khi dữ liệu đã được tải vào một cache set, phần cứng sẽ set valid bit.

### *2.5.5. Set associative cache*

Trong bộ nhớ two-way set associative cache, bộ nhớ được chia thành hai bộ nhớ cache có kích thước bằng nhau. Mỗi bộ trong đó có một nửa số mục so với bộ nhớ direct-mapped cache có cùng tổng kích thước. Phần cứng sẽ kiểm tra cả hai bộ nhớ cache song song trong mỗi lần truy cập bộ nhớ, và một lần hit có thể xảy ra ở bất kỳ bộ nào. Sơ đồ sau minh họa việc so sánh đồng thời một tag địa chỉ 32-bit với các tag chứa trong hai L1 I-cache:




Hình 4. Sự vận hành của set associative cache

Khi xảy ra cache miss, memory management logic phải chọn bảng cache nào trong hai bảng sẽ được sử dụng làm điểm đến cho dữ liệu đang được tải từ DRAM. Một phương pháp phổ biến là theo dõi bảng liên quan nào trong hai bảng đã không được truy cập lâu nhất và ghi đè lên mục đó. Chính sách thay thế này, gọi là Least-Recently Used (LRU), yêu cầu hỗ trợ phần cứng để theo dõi cache line nào đã không được sử dụng lâu nhất. Chính sách LRU dựa trên nguyên tắc tính cục bộ theo thời gian, cho rằng dữ liệu không được truy cập trong một khoảng thời gian dài ít có khả năng được truy cập lại sớm.

Một phương pháp khác để chọn giữa hai bảng là chỉ đơn giản là xen kẽ giữa chúng trong các lần chèn cache liên tiếp. Phần cứng để thực hiện chính sách thay thế này đơn giản hơn so với chính sách LRU, nhưng có thể ảnh hưởng đến hiệu suất do việc chọn ngẫu nhiên dòng bị vô hiệu hóa. Việc chọn chính sách thay thế cache đại diện cho một sự đánh đổi giữa độ phức tạp của phần cứng và sự gia tăng hiệu suất.

Trong bộ nhớ two-way set associative cache, hai cache line từ các vị trí vật lý khác nhau có các trường Set tương ứng có thể tồn tại trong cache đồng thời, với điều kiện cả hai cache line đều hợp

lệ. Đồng thời, so với bộ nhớ direct-mapped cache có cùng tổng kích thước, mỗi bộ two-way set associative cache chỉ bằng một nửa kích thước. Đây là một sự đánh đổi khác trong thiết kế: số lượng giá trị Set duy nhất có thể được lưu trữ trong bộ nhớ cache liên kết theo tập hợp 2 chiều bị giảm so với direct-mapped, nhưng nhiều cache line có trường Set giống nhau có thể được lưu trong cache đồng thời.

Cấu hình cache mang lại hiệu suất hệ thống tổng thể tốt nhất sẽ phụ thuộc vào các pattern truy cập bộ nhớ liên quan đến các loại hoạt động xử lý được thực hiện trên hệ thống.

Set associative cache có thể chứa nhiều hơn hai bộ nhớ cache như trong ví dụ này. Các bộ xử lý hiện đại thường có 4, 8 hoặc 16 bộ nhớ cache song song, được biết đến như 4-way, 8-way và 16-way set associative cache. Các bộ nhớ cache này được gọi là set associative vì một trường Tag địa chỉ liên kết với tất cả các cache line trong tập hợp cùng lúc.

Ưu điểm của multi-way set associative so với direct-mapped cache là nó thường có tỷ lệ cache hit cao hơn, dẫn đến hiệu suất hệ thống tốt hơn trong hầu hết các ứng dụng thực tế. Nếu multi-way set associative cache có hiệu suất tốt hơn one-way direct mapping, tại sao không tăng mức bậc associativity thậm chí nhiều hơn nữa? Nếu tăng đến giới hạn tối đa, quá trình này sẽ kết thúc với fully associative caching.

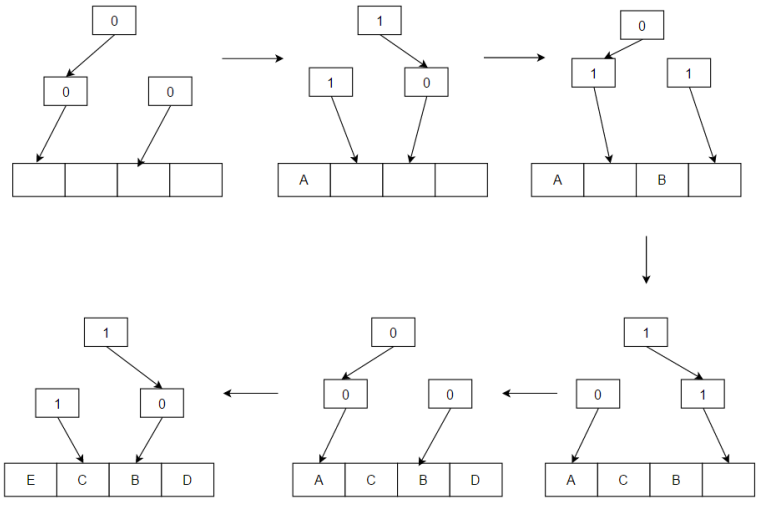
### *2.5.6. Thuật toán thay thế LRU*

Các thuật toán thay thế cache được thiết kế một cache hiệu quả để thay thế dữ liệu trong cache khi không gian đã đầy. Least Recently Used (LRU) là một trong những thuật toán đó. Như tên gọi gợi ý, khi bộ nhớ cache đầy, LRU sẽ chọn dữ liệu được sử dụng ít nhất gần đây và loại bỏ nó để tạo chỗ cho dữ liệu mới. Độ ưu tiên của dữ liệu trong cache thay đổi theo nhu cầu của dữ liệu đó, tức là nếu một số dữ liệu được lấy hoặc cập nhật gần đây thì độ ưu tiên của dữ liệu đó sẽ thay đổi và được gán mức ưu tiên cao nhất, và độ ưu tiên của dữ liệu sẽ giảm đi nếu nó vẫn không được sử dụng qua các lần vận hành.

Pseudo-LRU hay PLRU là một hộ của các thuật toán cache cải thiện hiệu năng của thuật toán Least Recently Used (LRU) bằng cách thay thế các giá trị dựa trên các thước đo xấp xỉ về tuổi thay vì duy trì chính xác tuổi của từng giá trị trong cache. Tree-PLRU là một loại của LRU và có đạt được hiệu quả cao.

Tree-PLRU là một thuật toán hiệu quả để chọn một mục có khả năng không được truy cập gần đây nhất, dựa trên một tập hợp các mục và một chuỗi sự kiện truy cập đến các mục. Kỹ thuật này được sử dụng trong bộ nhớ cache CPU của Intel 486 và trong nhiều bộ xử lý thuộc dòng PowerPC, chẳng hạn như PowerPC G4 của Freescale được Apple Computer sử dụng.

Thuật toán hoạt động như sau: hãy xem xét một cây tìm kiếm nhị phân cho các mục đang xét. Mỗi nút của cây có một cờ một bit biểu thị đi sang trái để chèn một phần tử pseudo-LRU hoặc đi sang phải để chèn một phần tử pseudo-LRU. Để tìm một phần tử pseudo-LRU, duyệt qua cây theo các giá trị của cờ. Để cập nhật cây khi có một lần truy cập đến mục N, duyệt qua cây để tìm N và trong quá trình duyệt, đặt cờ nút để biểu thị hướng ngược lại với hướng đã đi.




Hình 5. Thuật toán tree-PLRU

Thuật toán này có thể không tối ưu vì nó chỉ là một phép xấp xỉ. Ví dụ, trong sơ đồ trên với các cache line A, C, B, D, nếu mẫu truy cập là: C, B, D, A, khi xảy ra việc loại bỏ, B sẽ được chọn thay vì C. Điều này xảy ra vì cả A và C đều nằm trong cùng một nửa, và việc truy cập A sẽ hướng thuật toán sang nửa còn lại, nơi không chứa cache line C.

### *2.5.7. Multilevel cache*

Multilevel cache là một trong những kỹ thuật cải thiện hiệu suất của cache bằng cách giảm 'Miss Penalty'. Miss Penalty đề cập đến thời gian cần thiết để mang dữ liệu vào cache từ bộ nhớ chính khi xảy ra 'miss' trong cache.

Thời gian truy cập trung bình Multilevel cache (Tavg):

$$T_{avg} = H_1 * C_1 + (1 - H_1) * (H_2 * C_2 + (1 - H_2) * M)$$

Trong đó:

- H1 là hit rate trong bộ nhớ cache L1.
- H2 là hit rate trong bộ nhớ cache L2.
- C1 là thời gian truy cập thông tin trong bộ nhớ cache L1.
- C2 là thời gian chuyển thông tin từ bộ nhớ cache L2 sang bộ nhớ cache L1.
- M là Miss penalty để chuyển thông tin từ bộ nhớ chính sang bộ nhớ cache L2.

Hệ thống multilevel cache trong các bộ xử lý hiện đại mang lại cải thiện hiệu suất đáng kể so với các hệ thống tương đương nhưng không có bộ nhớ cache. Bộ nhớ đệm cho phép các processor có tốc độ cao có thể chạy với việc truy cập bộ nhớ DRAM có độ trễ lớn. Mặc dù bộ nhớ cache làm tăng đáng kể độ phức tạp của thiết kế bộ xử lý và tiêu tốn nhiều diện tích die và năng lượng, các nhà cung cấp bộ xử lý đã xác định rằng việc sử dụng kiến trúc multilevel cach hoàn toàn xứng đáng với các chi phí này.

Ưu điểm của tổ chức multilevel cache:

- Giảm thời gian truy cập: Nhờ có nhiều cấp độ cache, thời gian truy cập dữ liệu được sử dụng thường xuyên giảm đáng kể. Dữ liệu đầu tiên sẽ được tìm trong bộ nhớ cache nhỏ nhất, nhanh nhất, và nếu không tìm thấy, sẽ tìm ở cấp độ cache lớn hơn và chậm hơn.
- Cải thiện hiệu suất hệ thống: Với thời gian truy cập nhanh hơn, hiệu suất hệ thống được cải thiện do CPU ít phải chờ dữ liệu được lấy từ bộ nhớ.
- Giảm chi phí: Nhờ có nhiều cấp độ cache, tổng lượng bộ nhớ cache cần thiết có thể được tối thiểu hóa. Điều này là do các cấp độ cache nhỏ hơn và nhanh hơn thường đắt hơn trên mỗi đơn vị bộ nhớ so với các cấp lớn hơn và chậm hơn.
- Tiết kiệm năng lượng: Vì dữ liệu được tìm kiếm trước tiên ở cấp độ cache nhỏ nhất, khả năng cao là dữ liệu sẽ được tìm thấy ở đó, giúp giảm tiêu thụ năng lượng của hệ thống.

Nhược điểm của tổ chức multilevel cache:

- Độ phức tạp: Việc thêm nhiều cấp độ cache làm tăng độ phức tạp của hệ thống cache và toàn bộ hệ thống. Điều này có thể làm cho việc thiết kế, gỡ lỗi và bảo trì hệ thống trở nên khó khăn hơn.

- Độ trễ cao hơn khi cache miss: Nếu dữ liệu không được tìm thấy trong bất kỳ cấp độ cache nào, nó sẽ phải được lấy từ bộ nhớ chính, điều này có độ trễ cao hơn. Độ trễ này có thể đặc biệt rõ ràng trong các hệ thống có cấu trúc cache nhiều cấp.
- Chi phí cao hơn: Mặc dù việc có nhiều cấp độ cache có thể giảm tổng lượng cache cần thiết, nó cũng có thể làm tăng chi phí hệ thống do cần thêm phần cứng cache.
- Vấn đề đồng bộ cache (cache coherence issues): Khi số lượng cấp độ cache tăng lên, việc duy trì tính nhất quán của dữ liệu giữa các cấp độ trở nên khó khăn hơn. Điều này có thể dẫn đến việc dữ liệu không nhất quán được đọc từ cache, gây ra các vấn đề trong hoạt động tổng thể của hệ thống.

### *2.5.8. Level 2 cache*

Các bộ xử lý hiện đại hiệu suất cao thường chứa một bộ nhớ cache L2 trên chip. Không giống như bộ nhớ cache L1, bộ nhớ cache L2 thường kết hợp cả instruction và data trong một vùng nhớ duy nhất, thể hiện kiến trúc Von Neumann thay vì kiến trúc Harvard đặc trưng của cấu trúc split L1 cache. L2 cache được thiết kế theo kiến trúc Von Neumann sẽ dễ dàng kiểm soát hơn và phân bổ instruction và data một cách linh hoạt.

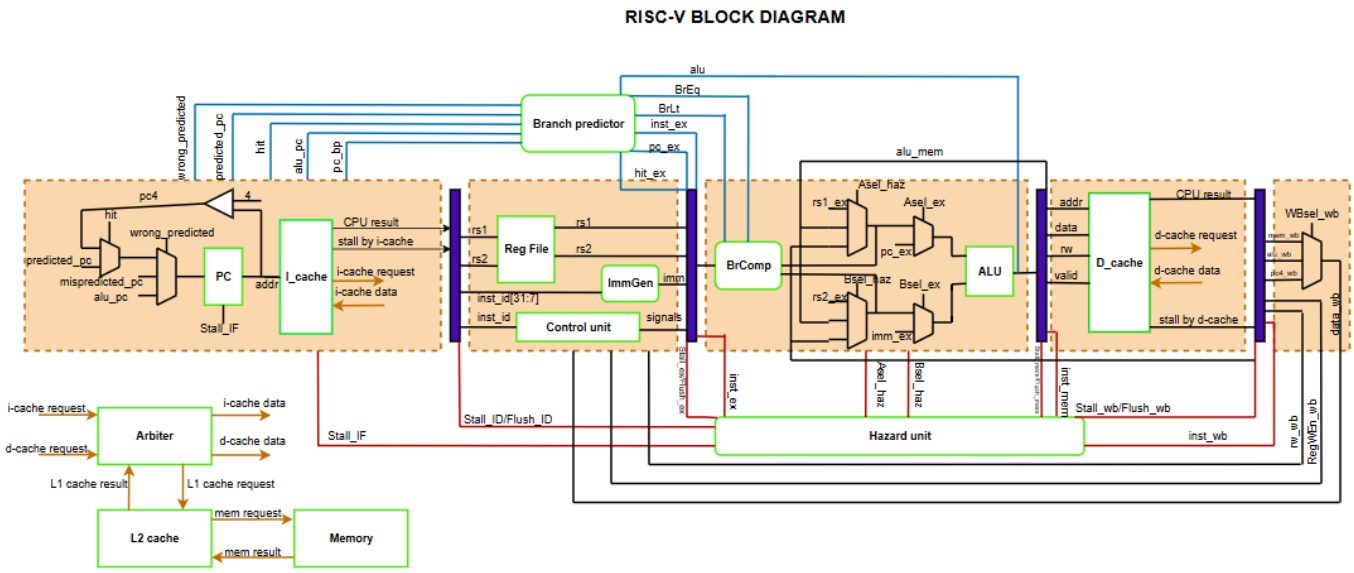
Bộ nhớ cache L2 thường chậm hơn bộ nhớ cache L1, nhưng vẫn nhanh hơn nhiều so với việc truy cập trực tiếp vào DRAM. Mặc dù bộ nhớ cache L2 sử dụng các SRAM bit cell với cấu hình mạch cơ bản giống nhau, thiết kế mạch của L2 tập trung vào việc giảm diện tích mỗi bit trên die và mức tiêu thụ năng lượng so với thiết kế của L1. Những điều chỉnh này cho phép bộ nhớ cache L2 lớn hơn nhiều so với L1, nhưng đồng thời cũng khiến nó chậm hơn đáng kể.

# 3. Thiết kế hệ thống

## 3.1. Thiết kế hệ thống tổng quan

Risc-v processor được thiết kế hỗ trợ tập tập RV32I theo kiến trúc pipelined 5 tầng:

- Fetch (IF): có chức năng đọc câu lệnh từ bộ nhớ và tăng PC, ở thiết kế này, tầng Fetch chứa Instruction cache, do đó câu lệnh sẽ được lấy từ Instruction cache. Trong trường hợp i-cache không chứa câu lệnh cần fetch, dữ liệu sẽ được đưa từ L2 cache hoặc từ Memory vào i-cache để truyền tới tầng phía sau.
- Decode (ID): tầng này đảm nhiệm nhiệm vụ chia câu lệnh thành những thành phần nhỏ hơn, tính toán những tín hiệu điều khiển, trích xuất những giá trị thanh ghi nội trong processor, lấy ra số immediate để sử dụng ở tầng sau đó.
- Execute (EX): đẩy những giá trị của thanh ghi đọc từ Register file, giá trị immediate hoặc forwarded data từ những câu lệnh khác (trong trường hợp data hazard) vô bộ ALU để thực hiện tính toán. Đồng thời tầng này còn so sánh các giá trị từ câu lệnh để xác định địa chỉ cho PC nếu gặp câu lệnh nhảy.
- Memory (MEM): đọc hoặc ghi dữ liệu vào bộ nhớ, với địa chỉ từ được decode từ tầng ID và data từ ALU hoặc từ câu lệnh. Trong tầng này, bộ nhớ được ghi hoặc đọc chính là data cache, tùy vào trường hợp mà d-cache có thể đọc hoặc ghi từ L2 cache hoặc Memory.
- Writeback (WB): ghi giá trị từ bộ ALU hoặc đọc từ d-cache, hoặc địa chỉ PC (trong trường hợp lệnh nhảy) vào trong Register file.




Hình 6. Sơ đồ khối toàn hệ thống

Trong thiết kế trên, những hazard được xử lý thông qua khối Hazard unit. Khối này sẽ so sánh những câu lệnh ở 3 tầng EX, MEM và WB để điều khiển tín hiệu giúp bộ ALU trong tầng EX nhận được những data đúng từ tầng MEM hoặc WB, bên cạnh đó khối này còn điều khiển những tín hiệu Stall (dừng hoạt động) hay Flush (xóa câu lệnh trong 1 tầng) ở các tầng thích hợp khi xảy ra hazard. Khối Branch predictor giúp dự đoán địa chỉ PC khi gặp câu lệnh nhảy giúp tăng hiệu suất cho toàn hệ thống.

Cấu trúc bộ nhớ được phân theo 3 tầng chính, tầng trên cùng là i-cache và d-cache được thiết kế theo kiến trúc Harvard có kích thước nhỏ nhất và đạt được tốc độ cao nhất, tầng giữa là L2 cache thiết kế theo kiến trúc von Neumann chứa cả instruction và data, kích thước lớn hơn i-cache và d-cache nhưng tốc độ chậm hơn. Tầng cuối là bộ nhớ RAM, bộ nhớ này thường là một bộ nhớ riêng biệt nằm bên ngoài processor, có kích thước lớn và tốc độ truy xuất chậm nhất.

Những câu lệnh được hỗ trợ trong thiết kế:

- Register-to-register operations: ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND.
- Immediate operations: ADDI, SLLI, SLTI, SLTIU, XORI, SRLI, SRAI, ORI, ANDI.
- Load: LW.
- Store: SW.

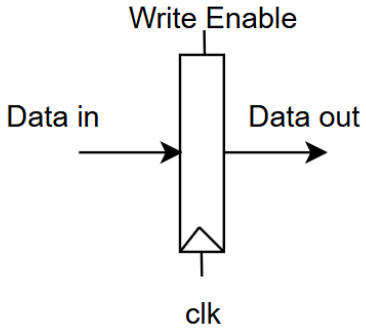
- Branch: BEQ, BNE, BLT, BGE, BLTU, BGEU.
- Jump: JAL, JALR.
- Upper immediate operations: AUIPC, LUI.

## 3.2. Thiết kế hệ thống chi tiết

### *3.2.1. Khối Program Counter, Register File, Immediate Generator, Branch Comparator, Arithmetic Logic Unit, Control Unit*

#### *3.2.1.1. Program Counter*

Program counter là bộ đếm chương trình, bản chất thì program counter là một thanh ghi 32 bit. Ở mỗi cạnh lên xung clock, giá trị thanh ghi sẽ được cập nhật.




Hình 7. Program counter

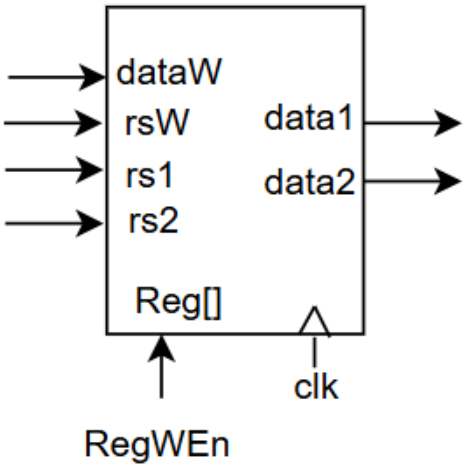
Bảng 1. Bảng input/output của khối Program counter

| Ngõ vào | Ngõ ra |
|---|---|
| Data In: Bus dữ liệu ngõ vào 32 bit.<br>Clk: Xung clock kích cạnh lên. | Data Out: bus dữ liệu ngõ ra 32 bit. |

Hoạt động của PC (Program counter): ở mỗi cạnh lên xung clock, Data Out = Data In, ở những thời điểm khác Data Out sẽ không thay đổi và giữ nguyên giá trị trước đó.

#### *3.2.1.2. Register File*

Register file có 32 thanh ghi 32 bit.




Hình 8. Khối Register file

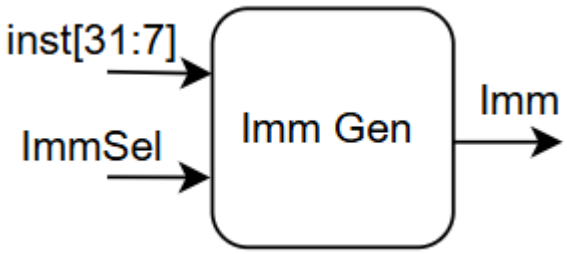
Bảng 2. Input/output của Register file

| Ngõ vào | Ngõ ra |
|---|---|
| dataW: 32 bit, chứa dữ liệu được ghi vào thanh ghi.<br>rsW, rs1, rs2: 5 bit, lần lượt là địa chỉ của thanh ghi cần ghi và 2 thanh ghi cần đọc.<br>clk: xung clock kích cạnh lên.<br>RegWEn: 1 bit, là tín hiệu cho phép ghi. | data1, data2: 32 bit, là giá trị của 2 thanh ghi được đọc. |

Hoạt động của RegFile: Những thanh ghi sẽ được truy cập thông qua 5 bit địa chỉ, data1 = R[rs1], data2 = R[rs2], R[rsW] = dataW chỉ khi tín hiệu RegWEn = 1. Hoạt động ghi chỉ được thực hiện khi có cạnh lên xung clock, còn hoạt động đọc thanh ghi diễn ra như một hệ tổ hợp.

### *3.2.1.3. Immediate Generator*

Immediate generator là bộ tạo số immediate từ instruction.




Hình 9. Khối Immediate generator

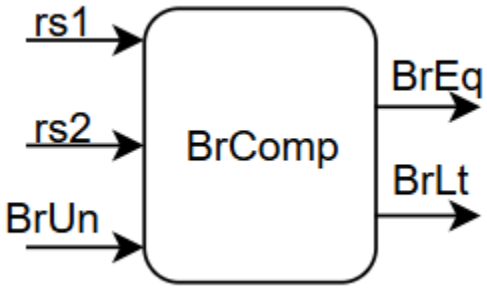
Bảng 3. Input/output của khối Immediate generator

| Ngõ vào | Ngõ ra |
|---|---|
| Inst[31:7]: 25 bit cao nhất của instruction.<br>ImmSel: 3 bit, là tín hiệu chọn loại lệnh (I, S, B, J, U). | Imm: 32 bit chứa dữ liệu số immediate. |

Hoạt động: bộ ImmGen hoạt động như bộ tổ hợp, với mỗi tín hiệu ImmSel cụ thể thì sẽ tạo được immediate number từ ngõ vào instruction.

#### *3.2.1.4. Branch Comparator*

Branch comparator là bộ so sánh phục vụ cho những lệnh rẽ nhánh.




Hình 10. Khối Branch comparator

Bảng 4. Input/output của khối Branch comparator

| Ngõ vào | Ngõ ra |
|---|---|
| rs1, rs2: 32 bit chứa dữ liệu của 2 thanh ghi được lấy từ khối register file.<br>BrUn: 1 bit, tín hiệu báo hiệu lệnh rẽ nhánh so sánh số không dấu. | BrEq: 1 bit, là tín hiệu báo hiệu 2 thanh ghi bằng nhau.<br>BrLt: 1 bit, là tín hiệu báo hiệu thanh ghi thứ nhất bé hơn thanh ghi thứ hai. |

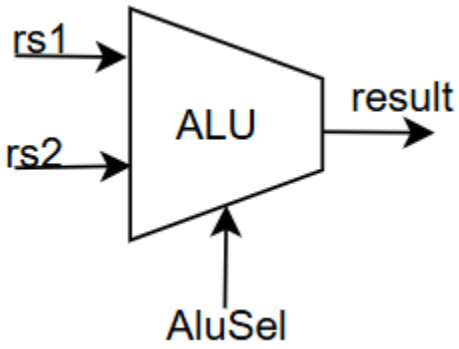
Hoạt động: Nếu BrUn = 1 thì BrComp sẽ so sánh 2 thanh ghi dưới dạng số không dấu. Bộ BrComp sẽ so sánh 2 thanh ghi, nếu rst1 = rst2 thì BrEq = 1 và BrLt = 0, nếu rst1 < rst2 thì BrEq = 0 và BrLt = 1, với các trường hợp còn lại thì BrLt và BrEq đều bằng 0.

### *3.2.1.5. Arithmetic Logic Unit*

Arithmetic logic unit là bộ tính toán các biểu thức +, -, <<, >>, >>>, ^, &, |.

| alu_op | Description (R-type) | Description (I-type) |
|---|---|---|
| ADD | $rd = rs1 + rs2$ | $rd = rs1 + imm$ |
| SUB | $rd = rs1 - rs2$ | $n/a$ |
| SLT | $rd = (rs1 < rs2)?1 : 0$ | $rd = (rs1 < imm)?1 : 0$ |
| SLTU | $rd = (rs1 < rs2)?1 : 0$ | $rd = (rs1 < imm)?1 : 0$ |
| XOR | $rd = rs1 \oplus rs2$ | $rd = rs1 \oplus imm$ |
| OR | $rd = rs1 \vee rs2$ | $rd = rs1 \vee imm$ |
| AND | $rd = rs1 \wedge rs2$ | $rd = rs1 \wedge imm$ |
| SLL | $rd = rs1 << rs2[4:0]$ | $rd = rs1 << imm[4:0]$ |
| SRL | $rd = rs1 >> rs2[4:0]$ | $rd = rs1 >> imm[4:0]$ |
| SRA | $rd = rs1 >>> rs2[4:0]$ | $rd = rs1 >>> imm[4:0]$ |

Hình 11. Các phép toán bộ ALU thực hiện




Hình 12. Khối ALU

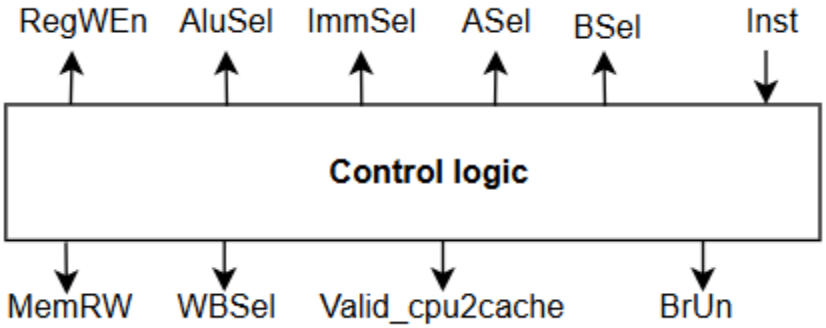
Bảng 5. Input/output của khối ALU

| Ngõ vào | Ngõ ra |
|---|---|
| rs1, rs2: 32 bit chứa dữ liệu 2 thanh ghi từ Regfile, hoặc giá trị PC hoặc giá trị từ ImmGen.<br>AluSel: 3 bit, là tín hiệu lựa chọn phép tính cần thực hiện. | result: 32 bit, chứa kết quả của phép toán được thực hiện. |

Hoạt động: Với mỗi tín hiệu AluSel, các phép toán như cộng, trừ, dịch phải, dịch trái,… sẽ được thực hiện như hình 11 và kết quả được lưu vào ngõ ra result.

#### *3.2.1.6. Control Unit*

Control Unit là khối điều khiển các tín hiệu lựa chọn cho các khối khác hoạt động.




Hình 13. Khối Control unit

| Inst[31:0] | ImmSel | BrUn | ASel | BSel | ALUSel | MemRW | RegWEn | WBSel | Valid_cpu2cache |
|---|---|---|---|---|---|---|---|---|---|
| add | * | * | Reg | Reg | Add | Read | 1 | ALU | 0 |
| sub | * | * | Reg | Reg | Sub | Read | 1 | ALU | 0 |
| (R-R Op) | * | * | Reg | Reg | (Op) | Read | 1 | ALU | 0 |
| addi | I | * | Reg | Imm | Add | Read | 1 | ALU | 0 |
| lw | I | * | Reg | Imm | Add | Read | 1 | Mem | 1 |
| sw | S | * | Reg | Imm | Add | Write | 0 | * | 1 |
| beq | B | * | PC | Imm | Add | Read | 0 | * | 0 |
| beq | B | * | PC | Imm | Add | Read | 0 | * | 0 |
| bne | B | * | PC | Imm | Add | Read | 0 | * | 0 |
| bne | B | * | PC | Imm | Add | Read | 0 | * | 0 |
| blt | B | 0 | PC | Imm | Add | Read | 0 | * | 0 |
| bltu | B | 1 | PC | Imm | Add | Read | 0 | * | 0 |
| jalr | I | * | Reg | Imm | Add | Read | 1 | PC+4 | 0 |
| jal | J | * | PC | Imm | Add | Read | 1 | PC+4 | 0 |
| auipc | U | * | PC | Imm | Add | Read | 1 | ALU | 0 |

Hình 14. Control unit truth table

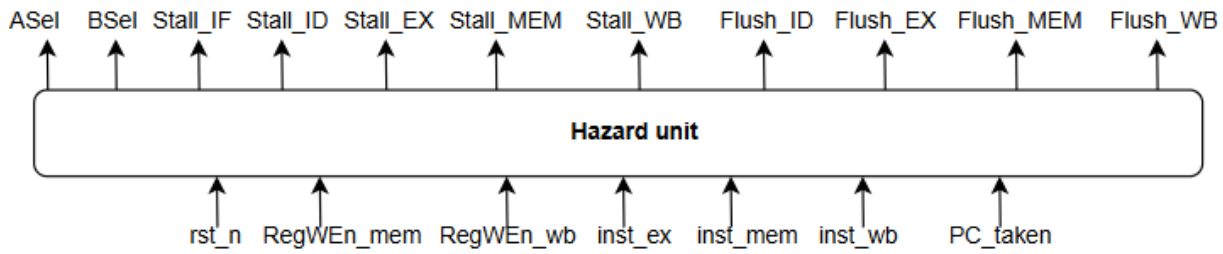
Bảng 6. Input/Output khối Control unit

| Ngõ vào | Ngõ ra |
|---|---|
| inst: 32 bit, là instruction được fetch từ i-cache, được decode giúp tạo nên các tín hiệu điều khiển cho ngõ ra. | RegWEn: 1 bit, tín hiệu cho phép ghi vào thanh ghi ở khối Register file.<br>AluSel: 4 bit, chọn phép tính cho bộ ALU.<br>ASel: 1 bit, tín hiệu điều khiển bộ mux, nếu bằng 0 thì ngõ ra là giá trị rs1 hoặc forwarding |

| | |
|---|---|
| | data, ngược lại là giá trị PC (lệnh rẽ nhánh hoặc lệnh nhảy).<br>BSel: 1 bit, tín hiệu điều khiển bộ mux, nếu bằng 0 thì ngõ ra là giá trị rs2 hoặc forwarding data, ngược lại là giá trị imm (trong trường hợp không phải lệnh loại R).<br>ImmSel: 3 bit, chọn loại câu lệnh để tạo số immediate.<br>MemRW: 1 bit, tín hiệu báo đọc hoặc ghi vào d-cache.<br>WBSel: 2 bit, chọn dữ liệu để writeback cho Register file.<br>BrUn: 1 bit, là tín hiệu báo hiệu số không dấu, bằng 1 khi ở lệnh rẽ nhánh có 2 thông số so sánh là số không dấu.<br>Valid_cpu2cache: 1 bit, tín hiệu cho phép CPU truy cập d-cache. |

#### *3.2.1.7. Hazard unit*

Hazard unit là khối xử lý các data hazard khi có dữ liệu trong câu lệnh tầng trước phụ thuộc vào kết quả từ câu lệnh ở tầng sau, do kết quả này chưa được ghi vào Register file mà đã có câu lệnh cần sử dụng kết quả này. Data hazard được xử lý bằng phương pháp forwarding, nó đưa các dữ liệu cần thiết từ tầng sau ngược lại tầng trước để tránh những hazard.



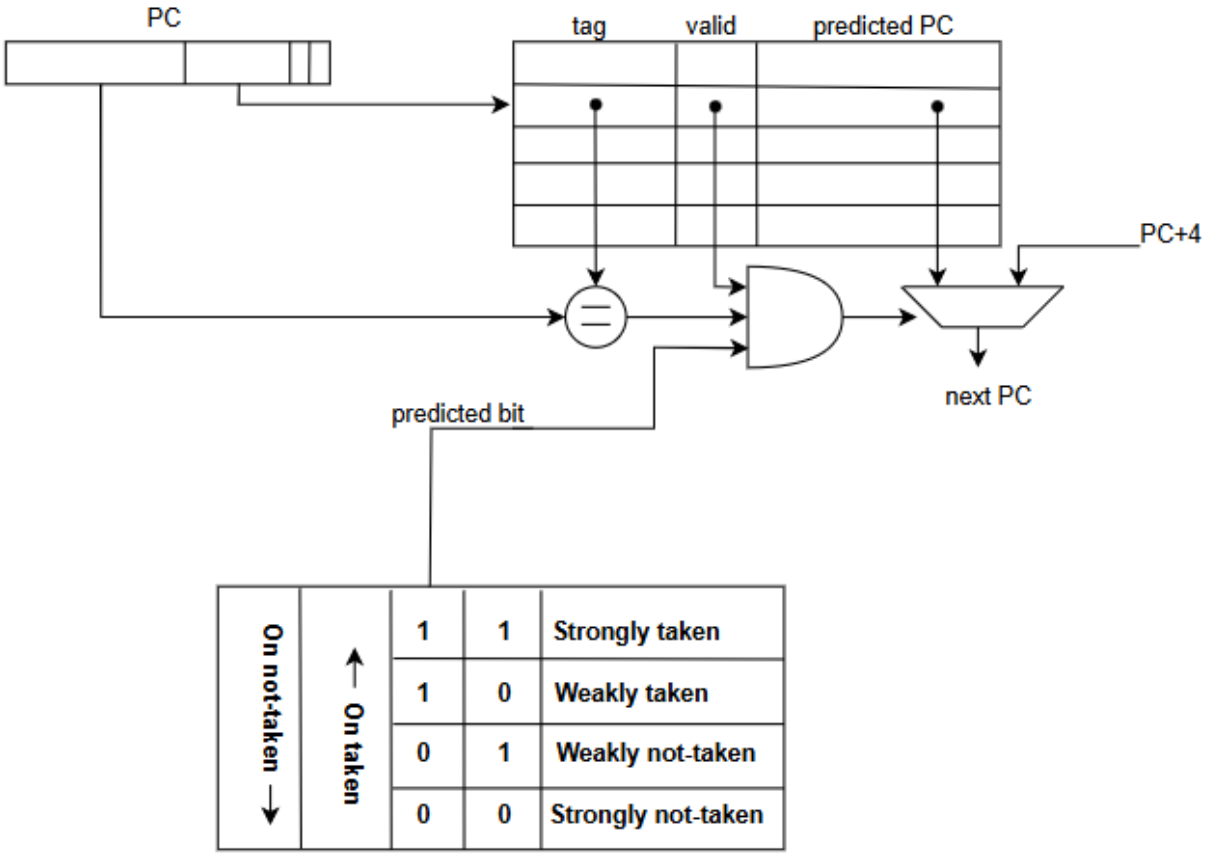

Hình 15. Khối Hazard unit

Bảng 7. Input/Output khối Hazard unit

| Ngõ vào | Ngõ ra |
|---|---|
| rst_n: 1 bit, tín hiệu reset tích cực mức thấp.<br>RegWEn_mem: 1 bit, tín hiệu register write enable ở tầng MEM.<br>RegWEn_wb: 1 bit, tín hiệu register write enable ở tầng WB.<br>inst_ex: 32 bit, instruction ở tầng EX.<br>inst_mem: 32 bit, instruction ở tầng MEM.<br>inst_wb: 32 bit, instruction ở tầng WB.<br>PC_taken: 2 bit, đây là tín hiệu lựa chọn địa chỉ ngõ vào cho bộ PC, được kết nối với tín hiệu wrong_predicted ở bộ Branch predictor. | ASel: 2 bit, lựa chọn 1 trong 3 dữ liệu rs1, forwarding data từ tầng MEM hoặc từ tầng WB.<br>BSel: 2 bit, lựa chọn 1 trong 3 dữ liệu rs2, forwarding data từ tầng MEM hoặc từ tầng WB.<br>Stall_IF, Stall_ID, Stall_EX, Stall_MEM, Stall_WB: 1 bit, lần lượt là tín hiệu stall cho các tầng IF, ID, EX, MEM, WB.<br>Flush_ID, Flush_EX, Flush_MEM, Flush_WB: 1 bit, lần lượt là tín hiệu flush cho các tầng ID, EX, MEM, WB. |

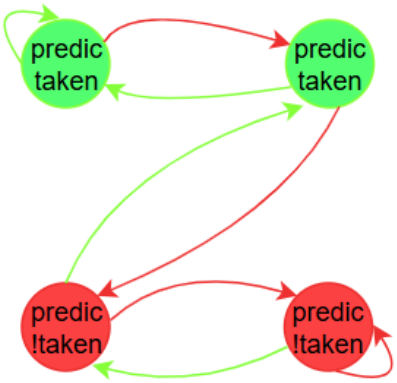
### *3.2.1.8. Branch predictor*

Branch predictor là bộ dự đoán địa chỉ nhảy cho PC khi gặp lệnh rẽ nhánh hoặc lệnh nhảy. Có nhiều loại kỹ thuật cho branch predictor, ở đây được thiết kế bằng kỹ thuật two-bit prediction.

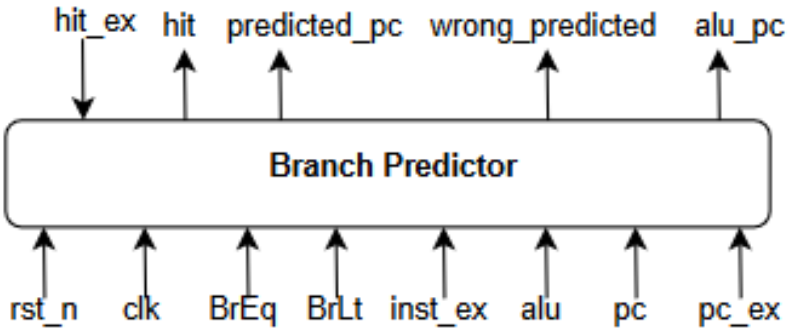



Hình 16. Cấu trúc của khối Branch predictor

Giá trị predicted PC được sử dụng hay không được biểu diễn bằng giản đồ dưới đây.




Hình 17. Sơ đồ predicted bit




Hình 18. Khối Branch Predictor

Bảng 8. Input/Output khối Branch predictor

| Ngõ vào | Ngõ ra |
|---|---|
| BrEq: 1 bit, tín hiệu so sánh bằng từ tầng EX.<br>BrLt: 1 bit, tín hiệu so sánh bé hơn từ tầng EX.<br>alu: 32 bit, địa chỉ PC được tính từ bộ ALU.<br>pc_ex: 32 bit, địa chỉ PC từ tầng EX.<br>inst_ex: 32 bit, instruction từ tầng EX.<br>hit_ex: 1 bit, tín hiệu chọn predicted PC hoặc PC+4, tín hiệu này được nối từ tầng EX.<br>pc: 32 bit, địa chỉ PC ở tầng IF.<br>rst_n: 1 bit, tín hiệu reset tích cực mức thấp. | hit: 1 bit, tín hiệu dự đoán, chọn predicted PC hoặc PC+4.<br>predicted_pc: 32 bit, địa chỉ PC dự đoán được lấy từ BTB (Branch Target Buffer).<br>wrong_predicted: 2 bit, tín hiệu cho biết dự đoán sai.<br>alu_pc: 32 bit, địa chỉ PC cần nhảy. |

| | |
|---|---|
| clk: 1 bit, tín hiệu clock tích cực cạnh lên. | |

**Tính toán IPC:**

Giả sử :

Xác suất gặp load data hazard là $P_d = 5\%$

Số lượng câu lệnh cần delay là $\Delta_d = 1$

Số lượng lệnh rẽ nhánh là $P_{br} = 20\%$

Xác suất nhảy khi gặp lệnh rẽ nhánh là $P_{mís} = 10\%$

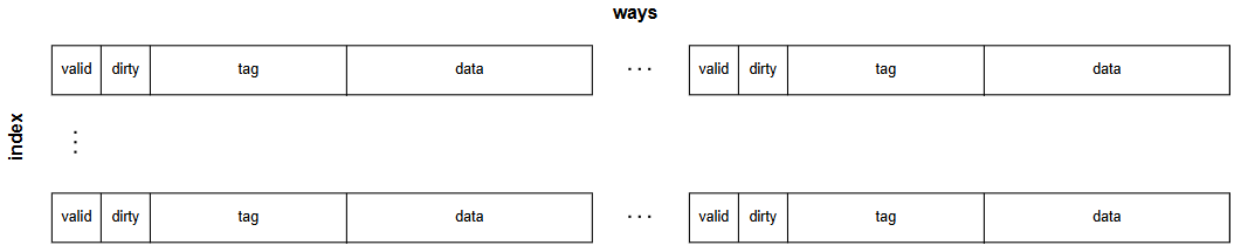
Số câu lệnh cần delay là $\Delta_{br} = 2$

Ta tính được:

$$IPC = \frac{N_{inst}}{N_{cycle}} = \frac{1}{P_d\,\Delta_d + P_{br}P_{mís}\Delta_{br} + 1} = \frac{1}{5\% \times 1 + 20\% \times 10\% \times 2 + 1} = 91.7\%$$

### *3.2.1. Cấu trúc của Cache*

Cache được thiết kế theo cấu trúc set-associative. Cấu trúc này được mô tả như hình dưới.



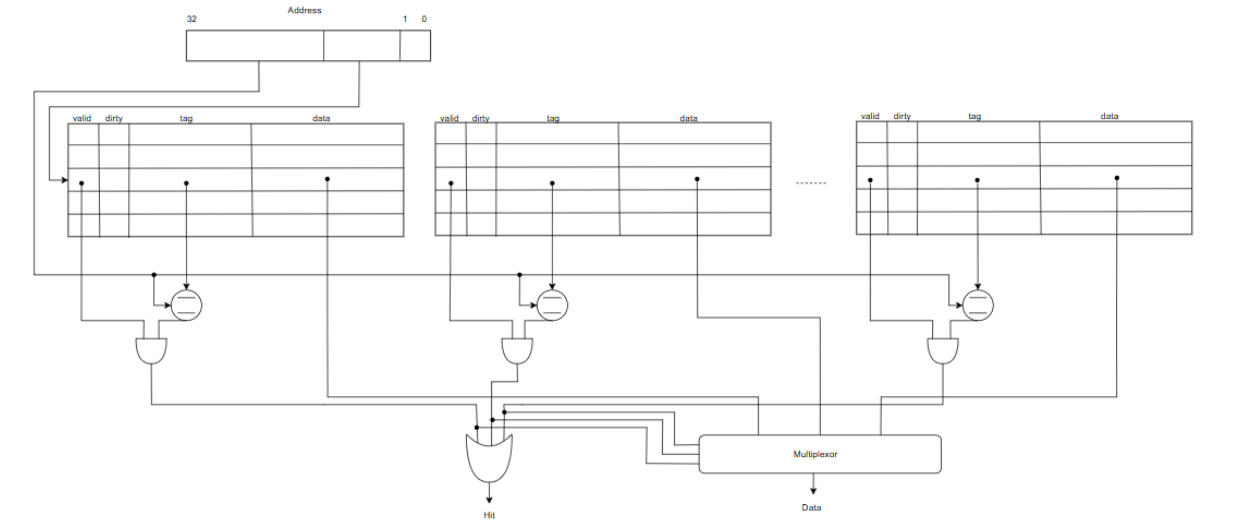

Hình 19. Cấu trúc của Cache

Cache được thiết kế theo nhiều set (xét theo chiều dọc), mỗi set chứa nhiều way (xét theo chiều ngang). Trong mỗi way có các bit sau:

- Valid: 1 bit. Đây là bit cho biết liệu block này có chứa dữ liệu hay không.
- Dirty: 1 bit. Đây là bit cho biết data trong block đã có trong cache ở tầng dưới hoặc memory hay chưa.

- Tag: Đây là các bit dùng để kiểm tra địa chỉ. Số bit của Tag được xác định bằng cách $32\ bits - 2\ offset\ bits - n\ bits - m\ bits$. Trong đó n là số bit offset của block, m là số bit index.
- Data: chứa dữ liệu được lưu trữ trong cache. Gồm 128 bits, tương đương 4 words, do đó n=2.
- Index: đây là các bit xác định địa chỉ của cache. Gồm m bits.

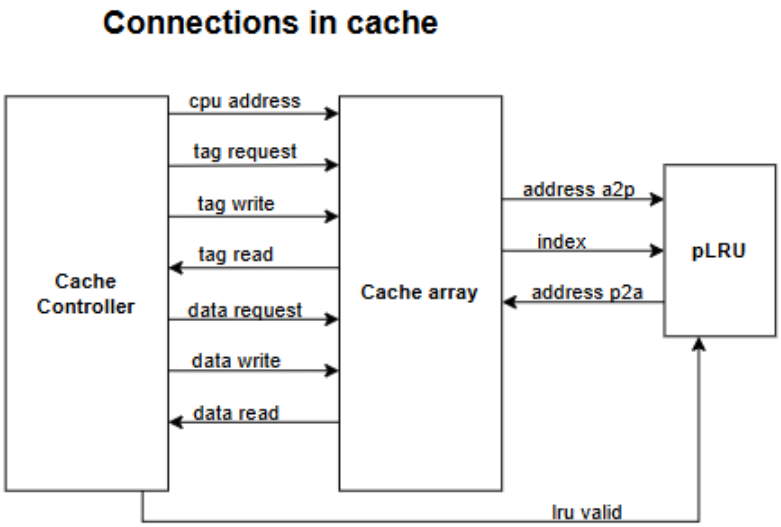
Dưới đây là cấu trúc mô tả cách truy cập cache qua các chỉ số của địa chỉ.




Hình 20. Cấu trúc truy cập Cache

### *3.2.1. Những kết nối trong Cache*

Trong bộ nhớ Cache, có 3 submodule gồm cache controller, cache array, pLRU. Trong đó cache controller đóng vai trò điều khiển hoạt động của cache, giúp truyền địa chỉ và data từ CPU tới các ô nhớ trong cache và ngược lại. Cache array là khối chứa những dữ liệu của cache, bao gồm phần valid bit, dirty bit, tag, data. Khối pLRU đóng vai trò quyết định địa chỉ thay thế cho dữ liệu trong cache, với cache được thiết kế theo cấu trúc set-associative.



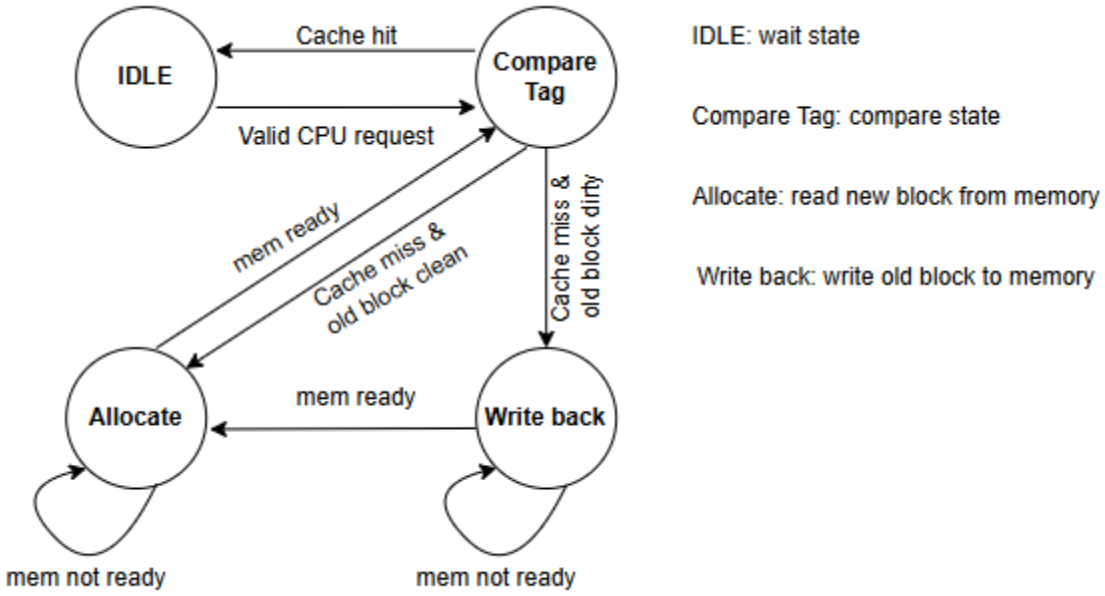

Hình 21. Những kết nối trong Cache

Các tín hiệu kết nối giữa những module được mô tả như sau:

- CPU address: 32 bit address, chứa địa chỉ từ CPU khi truy cập cache.
- Tag request, data request: chứa n bit index và 1 bit we, trong đó n là tổng số set trong cache, we là tín hiệu write enable cho phép tạo 1 tag/data mới trong cache.
- Tag write, tag read: gồm 1 bit valid, 1 bit dirty, n bit tag.
- Data write, data read: 128 bit (1 block, 4 words), chứa dữ liệu ghi/đọc từ cache.
- Address a2p, address p2a: n bit, trong đó n là số way của cache.
- Index: n bit, trong đó n là tổng số set trong cache.

### *3.2.1. Cache controller*

Để điều khiển hoạt động của cache, phương pháp đơn giản nhất là sử dụng finite-state machine (FSM).



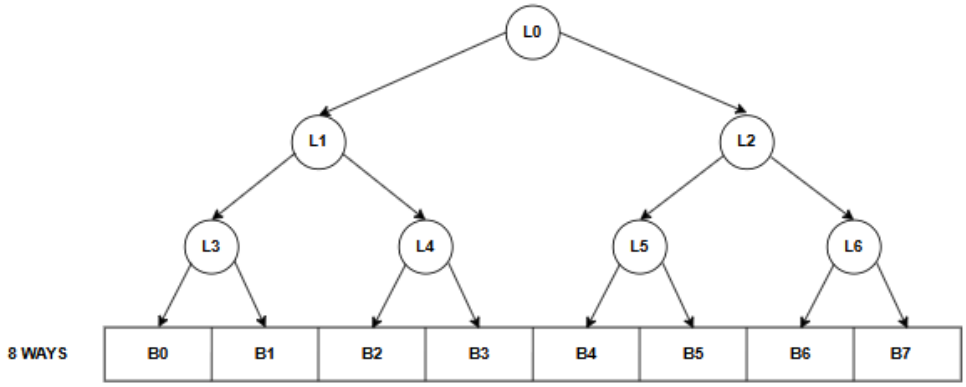

Hình 22. Cache controller FSM

Sơ đồ trên mô tả 4 trạng thái của cache controller:

- IDLE: Đây là trạng thái đợi cho tới khi tín hiệu yêu cầu ghi hoặc đọc từ bộ xử lý, trạng thái này sẽ chuyển đến trạng thái Compare Tag.
- Compare Tag: như tên gọi, trạng thái này kiểm tra xem yêu cầu đọc hoặc ghi là hit hay miss. Phần tag trong địa chỉ sẽ được dùng để so sánh với tag trong cache. Nếu dữ liệu trong khối cache được tham chiếu bởi phần chỉ số của địa chỉ là hợp lệ và phần tag của địa chỉ khớp với tag trong cache, thì đó là hit. Dữ liệu sẽ được đọc từ block đã chọn nếu đó là lênh load hoặc được ghi vào block đã chọn nếu đó là lệnh store. Sau đó, tín hiệu Cache Ready sẽ được set. Nếu đó là lệnh ghi, dirty bit sẽ được set lên 1. Lưu ý rằng một write hit cũng set valid bit và phần tag, mặc dù điều này có vẻ không cần thiết, nhưng nó được bao gồm vì tag là một bộ nhớ duy nhất, do đó để thay đổi dirty bit, chúng ta cũng cần thay đổi valid bit và tag. Nếu là hit và block là valid, FSM quay về trạng thái IDLE. Nếu miss, trước tiên cache tag được cập nhật và sau đó chuyển đến trạng thái Write-Back nếu block tại vị trí này có giá trị dirty bit là 1, hoặc chuyển đến trạng thái Allocate nếu giá trị là 0.
- Write-Back: trạng thái này ghi block 128-bit vào bộ nhớ bằng cách sử dụng địa chỉ được tạo từ tag và cache index. Trạng thái này sẽ giữ nguyên chờ tín hiệu Ready từ bộ nhớ. Khi ghi vào bộ nhớ hoàn tất, FSM sẽ chuyển sang trạng thái Allocate.
- Allocate: block mới sẽ được lấy (fetch) từ bộ nhớ. Trạng thái này sẽ giữ nguyên chờ tín hiệu Ready từ bộ nhớ. Khi đọc từ bộ nhớ hoàn tất, FSM sẽ chuyển sang trạng thái Compare Tag.

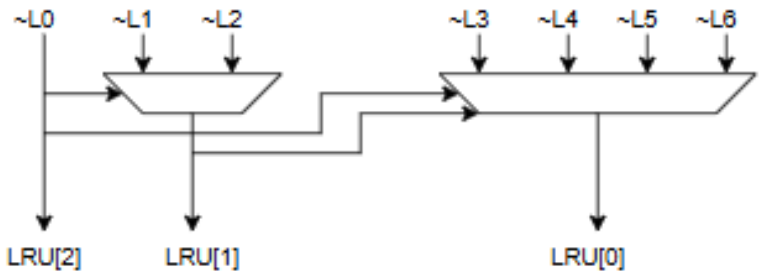
### *3.2.2. Thuật toán tree-PLRU*

Cấu trúc tree-PLRU được biểu diễn dưới dạng những node như 1 cái cây nhị phân, mỗi node chứa 2 giá trị, giá trị này quyết định sự thay đổi của những node ở tầng thấp hơn. Dưới đây là hình ảnh mô tả cấu trúc của tree-PLRU cho 8-ways set associative cache.




Hình 23. Cấu trúc tree-PLRU 8 node

Trong thiết kế tree-PLRU 8-ways set associative cache như hình trên. Tree-PLRU gồm có 3 tầng, tầng đầu tiên có L0, tầng thứ hai là L1 và L2, tầng thứ ba chứa L3, L4, L5, L6. Giả sử, mỗi way là một block, ví dụ khi truy cập block B5, giá trị nhị phân là 101, lúc này LRU tree sẽ được cập nhật lại giá trị mỗi lần truy cập, như vậy L0 sẽ bằng 1, L2 bằng 0 và L5 sẽ bằng 1. Như quy tắc trên, mỗi lần truy cập cache, LRU tree sẽ cập nhật lại giá trị và lưu trữ trạng thái ở từng node (L0, L1, L2,…).

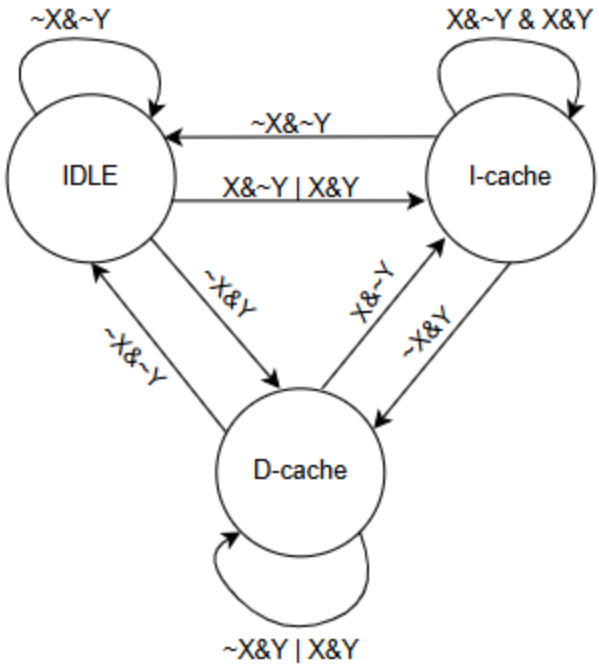



Hình 24. Mạch ngõ ra của tree-PLRU

Từ sơ đồ mạch trên, thu được các giá trị địa chỉ để xác định way nào cần thay thế. Ở 8-ways set associative cache, phải dùng 3 bit để xác định địa chỉ, theo giá trị MSB đến LSB thì LRU[2], LRU[1], LRU[0] như sơ đồ trên là 3 bit địa chỉ của khối tree-PLRU.

### *3.2.3. Cache arbiter*

Cache arbiter là một bộ phân xử giúp quyết định chọn dữ liệu và địa chỉ từ icache hay dcache chuyển tiếp tới L2 cache. Dưới đây là sơ đồ khối thể hiện các trạng thái của khối Cache arbiter.



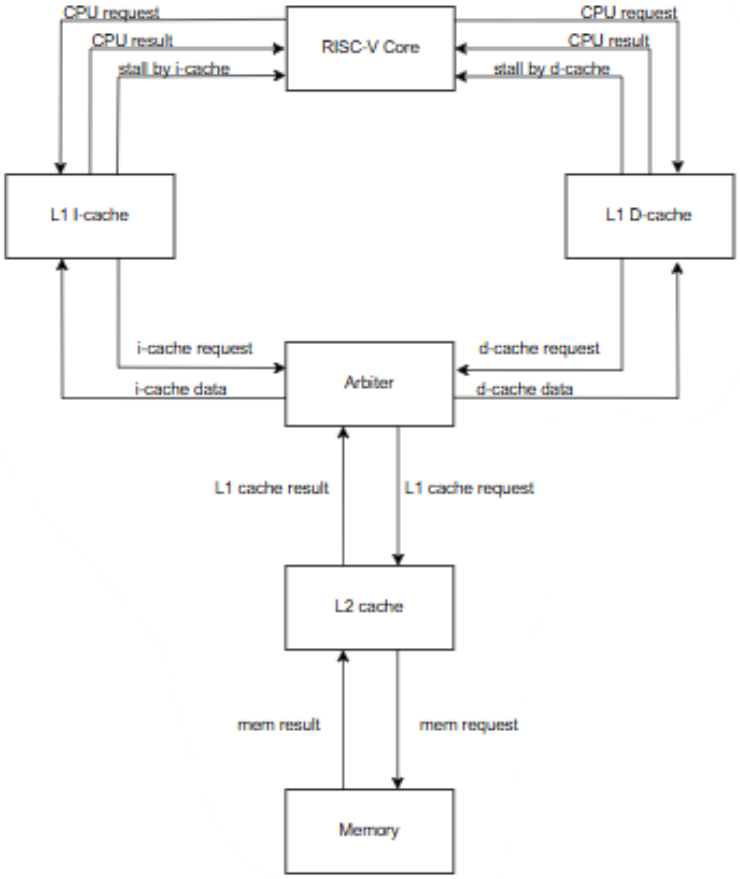

Hình 25. Sơ đồ khối Cache arbiter

Trong sơ đồ khối trên, các tín hiệu X và Y lần lượt là tín hiệu icache request valid và dcache request valid. Khối Cache arbiter sẽ ưu tiên instruction cache hơn khi cả 2 loại cache cùng truy cập đến L2 cache. Các trạng thái được mô tả như sau:

- IDLE: trạng thái khi không có L1 cache nào yêu cầu truy cập xuống L2 cache. Khi này giá trị L1 cache request sẽ bằng 0.
- I-cache: trạng thái cho phép instruction L1 cache truy cập vào L2 cache. Giá trị L1 cache request bằng i-cache request.
- D-cache: trạng thái cho phép data L1 cache truy cập vào L2 cache. Lúc này, giá trị L1 cache request bằng d-cache request.

### *3.2.3. Cấu trúc hệ thống Two-level cache*

Từ sơ đồ chi tiết toàn hệ thống, có thể mô tả hệ thống two-level cache như sơ đồ:




Hình 26. Cấu trúc hệ thống two-level cache

Sơ đồ trên mô tả những kết nối giữa processor, hai L1 cache là icache và dcache, kết nối với cache arbiter, L2 cache và giao tiếp với memory. Trong thiết kế toàn hệ thống, Instruction cache và Data cache được thiết kế có kích thước bằng nhau và được đặt gần processor, vì level 1 cache cần được truy cập với tốc độ rất nhanh do đó được đặt gần processor để tối ưu delay từ dây nối tới những phần của processor. Cả Instruction cache và Data cache đều được thiết kế theo kiến trúc Havard để đạt được tốc độ tối ưu nhất. L2 cache được thiết kế dạng unified, tức là L2 cache chứa cả Instruction và Data, thêm vào đó L2 cache còn bao gồm cả nội dung của Instruction cache và Data cache. L2 cache được thiết kế theo kiến trúc Von Neumann với ưu điểm dễ kiểm soát và tận dụng được độ linh hoạt khi phân bổ giữa instruction và data. Các tín đường bus được mô tả:

- CPU result: 32 bit data, 1 bit ready.
- CPU request: 32 bit address, 32 bit data, 1 bit read/write, 1 bit valid.

- i-cache request, d-cache request, L1 cache request, mem request: 32 bit address, 128 bit data, 1 bit read/write, 1 bit valid.
- L1 cache result, mem result, i-cache data, d-cache data: 128 bit data, 1 bit ready.

# 4. Mô phỏng và đánh giá thiết kế

## 4.1. Kiểm tra các câu lệnh được thiết kế

Có những lệnh cần được kiểm tra như sau: add, sub, xor, or, and, sll, srl, sra, slt, sltu, addi, xori, ori, andi, slli, srli, srai, slti, sltiu, lw, sw, jal, jalr, lui, auipc.

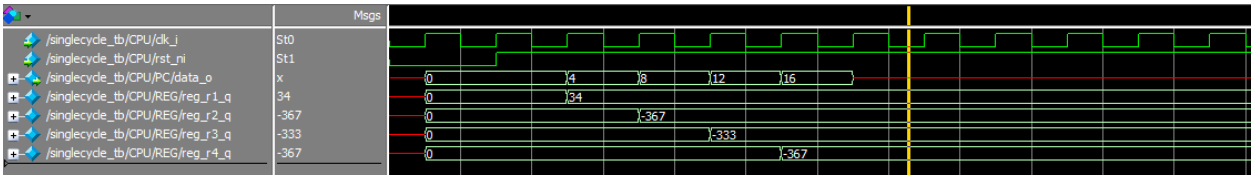
**Kiểm tra lệnh addi, and, sub**

Mã assembly:

| PC | Machine Code | Basic Code | Original Code |
|---|---|---|---|
| 0x0 | 0x02200093 | addi x1 x0 34 | addi x1, x0, 34 |
| 0x4 | 0xE9100113 | addi x2 x0 -367 | addi x2, x0, -367 |
| 0x8 | 0x002081B3 | add x3 x1 x2 | add x3, x1, x2 |
| 0xc | 0x40118233 | sub x4 x3 x1 | sub x4, x3, x1 |

Hình 27. Mã assembly lệnh addi, and, sub

Waveform:




Hình 28. Waveform lệnh addi, and, sub

Giải thích:

- Câu lệnh addi x1, x0, 34 thực hiện gán x1 = x0 + 34 =34
- Câu lệnh addi x2, x0, -367 thực hiện gán x2 =x0 - 367 = -367
- Câu lệnh addi x3, x1, x2 thực hiện gán x3 = x1 + x2 = 34 - 367 = -333
- Câu lệnh sub x4, x3, x1 thực hiện gán x4 = x3 – x1 = -333 – 34 = -367
- Từ waveform, sau mỗi lần tăng pc, mỗi câu lệnh được thực hiện lần lượt ở cạnh lên xung clock thứ 2 x1 được gán giá trị 34, ở cạnh lên xung clock thứ 3 x2 được gán giá trị - 367, ở cạnh lên thứ 4 x3 được gán giá trị -333, ở cạnh lên xung clock thứ 5 x4 được gán giá trị -367.

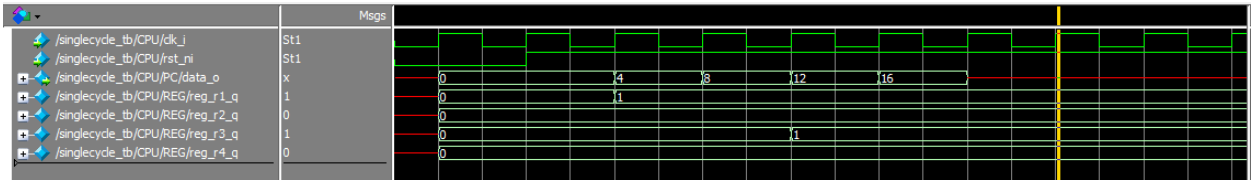
**Kiểm tra lệnh xor:**

Mã assembly:

| PC | Machine Code | Basic Code | Original Code |
|---|---|---|---|
| 0x0 | 0x00100093 | addi x1 x0 1 | addi x1, x0, 1 |
| 0x4 | 0x00000113 | addi x2 x0 0 | addi x2, x0, 0 |
| 0x8 | 0x0020C1B3 | xor x3 x1 x2 | xor x3, x1, x2 |
| 0xc | 0x0011C233 | xor x4 x3 x1 | xor x4, x3, x1 |

Hình 29. Mã assembly lệnh xor

Waveform:




Hình 30. Waveform lệnh xor

Giải thích:

- Câu lệnh addi x1, x0, 1 thực hiện gán x1 = x0 + 1 = 1
- Câu lệnh addi x2, x0, 0 thực hiện gán x2 = x0 + 0 = 0
- Câu lệnh xor x3, x1, x2 thực hiện gán x3 = x2 ^ x1 = 0 ^ 1 = 1
- Câu lệnh xor x4, x3, x1 thực hiện gán x4 = x3 ^ x1 = 1 ^ 1 = 0
- Từ waveform, sau mỗi lần tăng pc, mỗi câu lệnh được thực hiện lần lượt ở cạnh lên xung clock thứ 2 x1 được gán giá trị 1, ở cạnh lên xung clock thứ 3 x2 được gán giá trị 0, ở cạnh lên thứ 4 x3 được gán giá trị 1, ở cạnh lên xung clock thứ 5 x4 được gán giá trị 0.
-

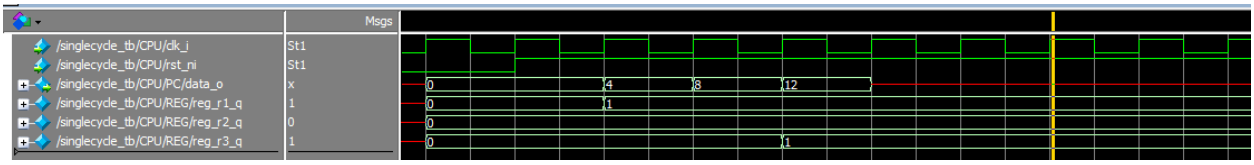
**Kiểm tra lệnh or:**

Mã assembly:

| PC | Machine Code | Basic Code | Original Code |
|---|---|---|---|
| 0x0 | 0x00100093 | addi x1 x0 1 | addi x1, x0, 1 |
| 0x4 | 0x00000113 | addi x2 x0 0 | addi x2, x0, 0 |
| 0x8 | 0x0020E1B3 | or x3 x1 x2 | or x3, x1, x2 |

Hình 31. Mã assembly lệnh or

Waveform:




Hình 32. Waveform lệnh or

Giải thích:

- Câu lệnh addi x1, x0, 1 thực hiện gán x1 = x0 + 1 = 1
- Câu lệnh addi x2, x0, 0 thực hiện gán x2 = x0 + 0 = 0
- Câu lệnh or x3, x1, x2 thực hiện gán x3 = x1 | x2
- Từ waveform, sau mỗi lần tăng pc, mỗi câu lệnh được thực hiện lần lượt ở cạnh lên xung clock thứ 2 x1 được gán giá trị 1, ở cạnh lên xung clock thứ 3 x2 được gán giá trị 0, ở cạnh lên thứ 4 x3 được gán giá trị 1.

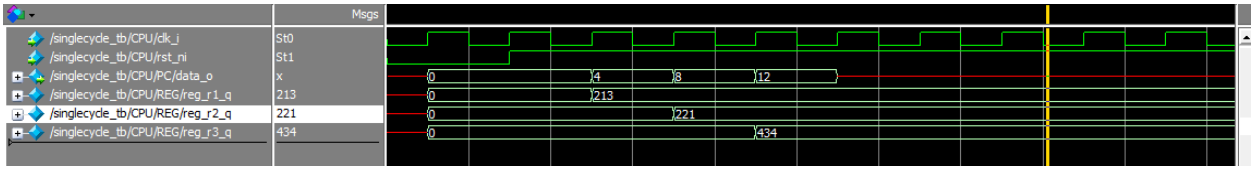
**Kiểm tra lệnh add:**

Mã assembly:

| PC | Machine Code | Basic Code | Original Code |
|---|---|---|---|
| 0x0 | 0x0D500093 | addi x1 x0 213 | addi x1, x0, 213 |
| 0x4 | 0x0DD00113 | addi x2 x0 221 | addi x2, x0, 221 |
| 0x8 | 0x002081B3 | add x3 x1 x2 | add x3, x1, x2 |

Hình 33. Mã assembly lệnh add

Waveform:




Hình 34. Waveform lệnh add

Giải thích:

- Câu lệnh addi x1, x0, 213 gán giá trị x1 = x0 + 213
- Câu lệnh addi x2, x0, 211 gán giá trị x2= x0 + 211
- Câu lệnh add x3, x2, x1 gán giá trị x3 = x2 + x1 = 434
- Từ waveform, sau mỗi lần tăng pc, mỗi câu lệnh được thực hiện lần lượt ở cạnh lên xung clock thứ 2 x1 được gán giá trị 213, ở cạnh lên xung clock thứ 3 x2 được gán giá trị 221, ở cạnh lên thứ 4 x3 được gán giá trị 434.

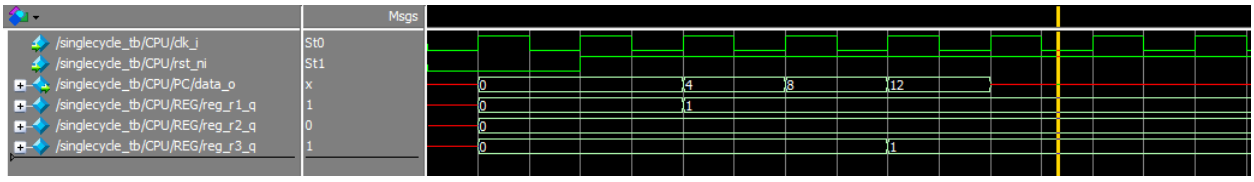
**Kiểm tra lệnh xori, ori, andi**

Mã assembly

| PC | Machine Code | Basic Code | Original Code |
|---|---|---|---|
| 0x0 | 0x00104093 | xori x1 x0 1 | xori x1,x0,1 |
| 0x4 | 0x00006113 | ori x2 x0 0 | ori x2, x0,0 |
| 0x8 | 0x0010F193 | andi x3 x1 1 | andi x3, x1,1 |

Hình 35. Mã assembly lệnh xori, ori, andi

Waveform:




Hình 36. Waveform lệnh xori, ori, andi

Giải thích

- Câu lệnh addi x1, x0, 213 gán giá trị x1 = 0 ^ 1 = 1
- Câu lệnh addi x2, x0, 211 gán giá trị x2= x0 | 0 = 0

- Câu lệnh add x3, x2, x1 gán giá trị x3 = x1 & 1 = 1
- Từ waveform, sau mỗi lần tăng pc, mỗi câu lệnh được thực hiện lần lượt ở cạnh lên xung clock thứ 2 x1 được gán giá trị 1, ở cạnh lên xung clock thứ 3 x2 được gán giá trị 0, ở cạnh lên thứ 4 x3 được gán giá trị 1.

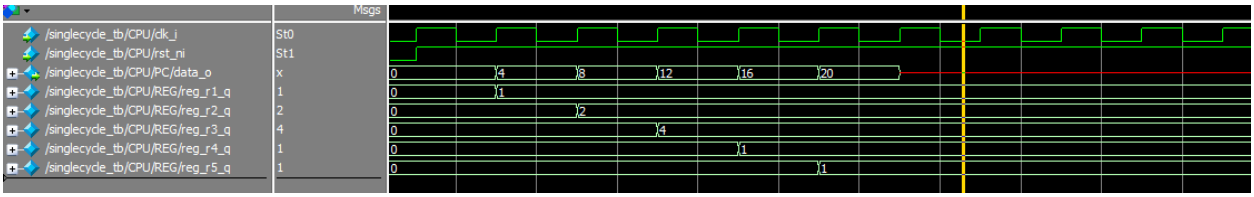
**Kiểm tra lệnh shift left logical, shight right logical, shift right arith:**

Mã assembly:

| PC | Machine Code | Basic Code | Original Code |
|---|---|---|---|
| 0x0 | 0x00100093 | addi x1 x0 1 | addi x1, x0, 1 |
| 0x4 | 0x00200113 | addi x2 x0 2 | addi x2, x0, 2 |
| 0x8 | 0x002091B3 | sll x3 x1 x2 | sll x3, x1, x2 |
| 0xc | 0x001151B3 | srl x3 x2 x1 | srl x3, x2, x1 |
| 0x10 | 0x40115233 | sra x4 x2 x1 | sra x4, x2 , x1 |

Hình 37. Mã assembly lệnh sll, srl, sra

Waveform:




Hình 38. Waveform lệnh sll, srl, sra

Giải thích:

- Câu lệnh addi x1, x0, 1 gán giá trị x1 = x0 + 1 = 1
- Câu lệnh addi x2, x0, 2 gán giá trị x2 = x0 + 2 = 2
- Câu lệnh sll x3, x1, x2 gán giá trị x3 = x1 << x2 -> 1 dịch sang trái 2 thành 100bin = 4 dec, x3 = 4
- Câu lệnh srl x4, x2, x1 gán giá trị x4 = x2 >> x1 -> 2 dịch sang phải 1 thành 1bin = 1dec, x4 = 1
- Câu lệnh sra x5, x2, x1 gán giá trị x5 = x2 >> x1 -> 2 dịch sang phải thành 1bin = 1 dec, x5 = 1
- Từ waveform, sau mỗi lần tăng pc, mỗi câu lệnh được thực hiện lần lượt ở cạnh lên xung clock thứ 2 x1 được gán giá trị 1, ở cạnh lên xung clock thứ 3 x2 được gán giá trị 2, ở cạnh

lên thứ 4 x3 được gán giá trị 4, ở cạnh lên xung clock thứ 5 x4 được gán giá trị 1, ở cạnh lên thứ 6 x5 được gán giá trị 1.

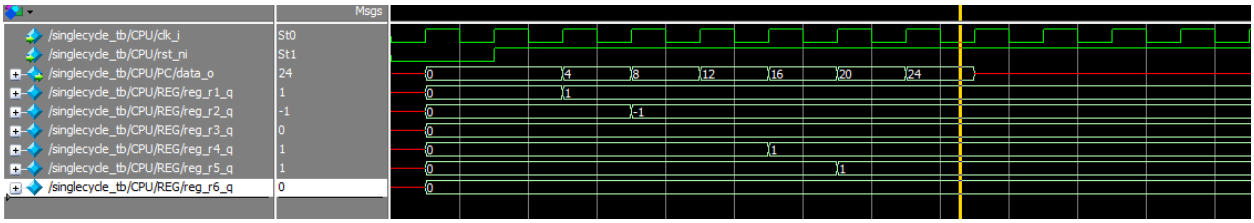
**Kiểm tra câu lệnh set less than, set less than (u), set less than imm, set less than imm (u):**

Mã assembly:

| PC | Machine Code | Basic Code | Original Code |
|---|---|---|---|
| 0x0 | 0x00100093 | addi x1 x0 1 | addi x1, x0, 1 |
| 0x4 | 0xFFF00113 | addi x2 x0 -1 | addi x2, x0, -1 |
| 0x8 | 0x0020A1B3 | slt x3 x1 x2 | slt x3, x1, x2 |
| 0xc | 0x0020B233 | sltu x4 x1 x2 | sltu x4, x1, x2 |
| 0x10 | 0x00112293 | slti x5 x2 1 | slti x5, x2 , 1 |
| 0x14 | 0x00113313 | sltiu x6 x2 1 | sltiu x6, x2, 1 |

Hình 39. Mã assembly lệnh slt, sltu, slti, sltiu

Waveform:




Hình 40. Waveform lệnh slt, sltu, slti, sltiu

Giải thích:

- Câu lệnh addi x1, x0, 1 gán giá trị x1 = x0 + 1 = 1
- Câu lệnh addi x2, x0, -1 gán giá trị x2 = x0 – 1 = -1
- Câu lệnh slt x3, x1, x2 gán giá trị x3 = (x1 < x2)?1:0 (so sánh giá trị có dấu trong thanh ghi r1 và r2, lưu lại kết quả vô r3, 1 nếu r1 < r2, 0 nếu r1 > r2)
- Câu lệnh sltu x4, x1, x2 gán giá trị x4 = (x1 < x2)?1:0 (so sánh giá trị không dấu trong thanh ghi r1 và r2, lưu lại kết quả vô r4, 1 nếu r1 < r2, 0 nếu r1 > r2)
- Câu lệnh slti x5, x2, 1 gán giá trị x5 = (x2 < 1)?1:0 (so sánh giá trị có dấu trong thanh ghi r2 và 1, lưu lại kết quả vô r5, 1 nếu r2 < 1, 0 nếu r2 > 1)
- Câu lệnh sltiu x5, x2, 1 gán giá trị x5 = (x2 < 1)?1:0 (so sánh giá trụ không dấu trong thanh ghi r2 và 1 lưu lại kết quả vô r5, 1 nếu r2 < 1, 0 nếu r2 > 1)

- Từ waveform, sau mỗi lần tăng pc, mỗi câu lệnh được thực hiện lần lượt ở cạnh lên xung clock thứ 2 x1 được gán giá trị 1, ở cạnh lên xung clock thứ 3 x2 được gán giá trị -1, ở cạnh lên thứ 4 x3 được gán giá trị 0, ở cạnh lên xung clock thứ 5 x4 được gán giá trị 1, ở cạnh lên thứ 6 x5 được gán giá trị 1, ở cạnh lên thứ 7 x6 được gán giá trị 0.

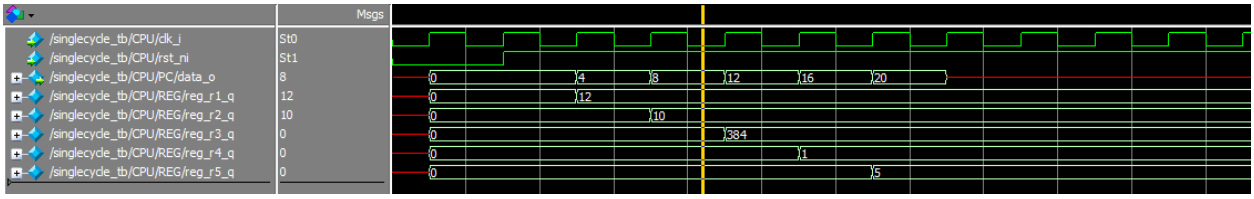
**Kiểm tra câu lệnh shift left logical imm, shift right logical imm, shift right arith imm:**

Mã assembly:

| PC | Machine Code | Basic Code | Original Code |
|---|---|---|---|
| 0x0 | 0x00C00093 | addi x1 x0 12 | addi x1, x0, 12 |
| 0x4 | 0x00A00113 | addi x2 x0 10 | addi x2, x0, 10 |
| 0x8 | 0x00509193 | slli x3 x1 5 | slli x3, x1, 5 |
| 0xc | 0x0030D213 | srli x4 x1 3 | srli x4, x1, 3 |
| 0x10 | 0x40115293 | srai x5 x2 1 | srai x5, x2 , 1 |

Hình 41. Mã assembly lệnh slli, srli, srai

Waveform:




Hình 42. Waveform lệnh slli, srli, srai

Giải thích:

- Câu lệnh addi x1, x0, 1 gán giá trị x1 = x0 + 12 = 12
- Câu lệnh addi x2, x0, 2 gán giá trị x2 = x0 + 10 = 10
- Câu lệnh slli x3, x1, 5 gán giá trị x3 = x1 << 5 -> 1100 dịch sang trái 5 thành 11000000bin = 384 dec, x3 = 384
- Câu lệnh srli x4, x1, 3 gán giá trị x4 = x1 >> 3 -> 1100 dịch sang phải 3 thành 1bin = 1dec, x4 = 1
- Câu lệnh srai x5, x2, 1 gán giá trị x5 = x2 >> 1 -> 1100 dịch sang phải thành 110bin = 5dec, x5 = 5
- Từ waveform, sau mỗi lần tăng pc, mỗi câu lệnh được thực hiện lần lượt ở cạnh lên xung clock thứ 2 x1 được gán giá trị 12, ở cạnh lên xung clock thứ 3 x2 được gán giá trị 10, ở

cạnh lên thứ 4 x3 được gán giá trị 384, ở cạnh lên xung clock thứ 5 x4 được gán giá trị 1, ở cạnh lên thứ 6 x5 được gán giá trị 5.

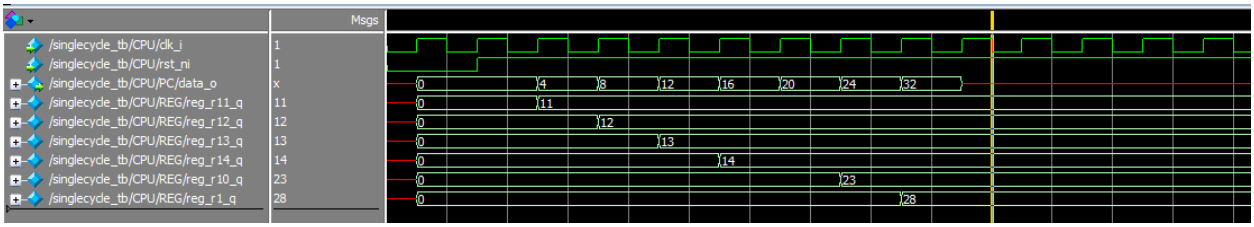
**Kiểm tra lệnh branch equal, jump and link:**

Mã assembly:

| PC | Machine Code | Basic Code | Original Code |
|---|---|---|---|
| 0x0 | 0x00B00593 | addi x11 x0 11 | addi x11, x0, 11 |
| 0x4 | 0x00C00613 | addi x12 x0 12 | addi x12, x0, 12 |
| 0x8 | 0x00D00693 | addi x13 x0 13 | addi x13, x0, 13 |
| 0xc | 0x00E00713 | addi x14 x0 14 | addi x14, x0, 14 |
| 0x10 | 0x00E68663 | beq x13 x14 12 | beq x13,x14,Else |
| 0x14 | 0x00C58533 | add x10 x11 x12 | add x10,x11,x12 |
| 0x18 | 0x008000EF | jal x1 8 | jal x1, Exit |
| 0x1c | 0x40C58533 | sub x10 x11 x12 | Else:sub x10,x11,x12 |

Hình 43. Mã assembly lệnh beq, jal

Waveform:




Hình 44. Waveform lệnh beq, jal

Giải thích:

- Câu lệnh addi x11, x0, gán giá trị x11 = x0 + 11 = 11
- Câu lệnh addi x12, x0, gán giá trị x12 = x0 + 12 = 12
- Câu lệnh addi x11, x0, gán giá trị x13 = x0 + 13 = 13
- Câu lệnh addi x11, x0, gán giá trị x14 = x0 + 14 = 14
- Câu lệnh beq x13, x14, Else so sánh giá trị x13 và x14, nếu x13 != x14 thì thực hiên gán x10 = x11 + x12 rồi thanh ghi x1 = PC + 4 và nhảy tới nhãn Exit; nếu x13 = x14 thì nhảy tới nhãn Else thực hiện gán x10 = x11 – x12. Ở đây x13 != x14 nên x10 = x11 + x12 = 11 + 12 = 23 sau đó thanh ghi x1 = PC = PC + 4 = 24 + 4 = 28 rồi nhảy tới nhãn Exit.

- Từ waveform, sau mỗi lần tăng pc, mỗi câu lệnh được thực hiện lần lượt ở cạnh lên xung clock thứ 2 x11 được gán giá trị 11, ở cạnh lên xung clock thứ 3 x12 được gán giá trị 12, ở cạnh lên thứ 4 x13 được gán giá trị 13, ở cạnh lên xung clock thứ 5 x14 được gán giá trị 14, ở cạnh lên thứ 6 thực hiện so sánh, ở cạnh lên thứ 7 x10 được gán giá trị 23, ở cạnh lên thứ 8, x1 được gán giá trị 28.

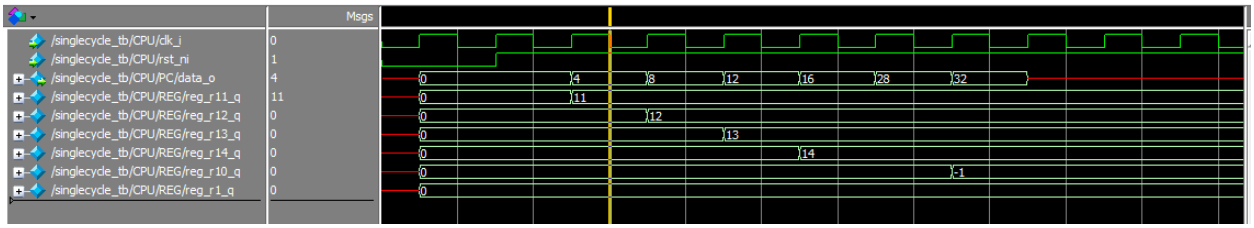
**Kiểm tra lệnh branch not equal:**

Mã assembly:

| PC | Machine Code | Basic Code | Original Code |
|---|---|---|---|
| 0x0 | 0x00B00593 | addi x11 x0 11 | addi x11, x0, 11 |
| 0x4 | 0x00C00613 | addi x12 x0 12 | addi x12, x0, 12 |
| 0x8 | 0x00D00693 | addi x13 x0 13 | addi x13, x0, 13 |
| 0xc | 0x00E00713 | addi x14 x0 14 | addi x14, x0, 14 |
| 0x10 | 0x00E69663 | bne x13 x14 12 | bne x13,x14,Else |
| 0x14 | 0x00C58533 | add x10 x11 x12 | add x10,x11,x12 |
| 0x18 | 0x008000EF | jal x1 8 | jal x1, Exit |
| 0x1c | 0x40C58533 | sub x10 x11 x12 | Else:sub x10,x11,x12 |

Hình 45. Mã assembly lệnh bne

Waveform:




Hình 46. Waveform lệnh beq

Giải thích:

- Câu lệnh addi x11, x0, gán giá trị x11 = x0 + 11 = 11
- Câu lệnh addi x12, x0, gán giá trị x12 = x0 + 12 = 12
- Câu lệnh addi x11, x0, gán giá trị x13 = x0 + 13 = 13
- Câu lệnh addi x11, x0, gán giá trị x14 = x0 + 14 = 14
- Câu lệnh bne x13, x14, Else so sánh giá trị x13 và x14, nếu x13 != x14 thì thực hiện gán x10 = x11 + x12 rồi thanh ghi x1 = PC + 4 và nhảy tới nhãn Exit; nếu x13 = x14 thì nhảy

tới nhãn Else thực hiện gán x10 = x11 – x12. Ở đây x13 != x14 nên x10 = x11 - x12 = 11 - 12 = -1.

- Từ waveform, sau mỗi lần tăng pc, mỗi câu lệnh được thực hiện lần lượt ở cạnh lên xung clock thứ 2 x11 được gán giá trị 11, ở cạnh lên xung clock thứ 3 x12 được gán giá trị 12, ở cạnh lên thứ 4 x13 được gán giá trị 13, ở cạnh lên xung clock thứ 5 x14 được gán giá trị 14, ở cạnh lên thứ 6 thực hiện so sánh, ở cạnh lên thứ 7 x10 được gán giá trị -1.

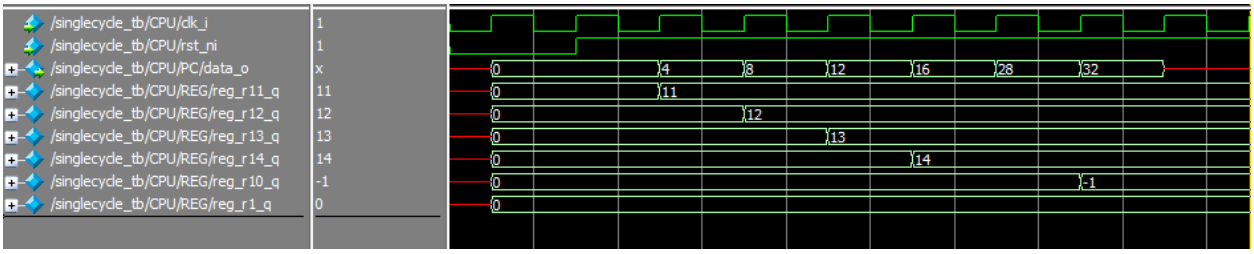
**Kiểm tra lệnh branch less than:**

Mã assembly:

| PC | Machine Code | Basic Code | Original Code |
|---|---|---|---|
| 0x0 | 0x00B00593 | addi x11 x0 11 | addi x11, x0, 11 |
| 0x4 | 0x00C00613 | addi x12 x0 12 | addi x12, x0, 12 |
| 0x8 | 0x00D00693 | addi x13 x0 13 | addi x13, x0, 13 |
| 0xc | 0x00E00713 | addi x14 x0 14 | addi x14, x0, 14 |
| 0x10 | 0x00E6C663 | blt x13 x14 12 | blt x13,x14,Else |
| 0x14 | 0x00C58533 | add x10 x11 x12 | add x10,x11,x12 |
| 0x18 | 0x008000EF | jal x1 8 | jal x1, Exit |
| 0x1c | 0x40C58533 | sub x10 x11 x12 | Else:sub x10,x11,x12 |

Hình 47. Mã assembly lệnh blt

Waveform:




Hình 48. Waveform lệnh blt

Giải thích:

- Câu lệnh addi x11, x0, gán giá trị x11 = x0 + 11 = 11
- Câu lệnh addi x12, x0, gán giá trị x12 = x0 + 12 = 12
- Câu lệnh addi x11, x0, gán giá trị x13 = x0 + 13 = 13
- Câu lệnh addi x11, x0, gán giá trị x14 = x0 + 14 = 14

- Câu lệnh bne x13, x14, Else so sánh giá trị x13 và x14, nếu x13 > x14 thì thực hiện gán x10 = x11 + x12 rồi thanh ghi x1 = PC + 4 và nhảy tới nhãn Exit; nếu x13 < x14 thì nhảy tới nhãn Else thực hiện gán x10 = x11 – x12. Ở đây x13 < x14 nên x10 = x11 - x12 = 11 - 12 = -1.
- Từ waveform, sau mỗi lần tăng pc, mỗi câu lệnh được thực hiện lần lượt ở cạnh lên xung clock thứ 2 x11 được gán giá trị 11, ở cạnh lên xung clock thứ 3 x12 được gán giá trị 12, ở cạnh lên thứ 4 x13 được gán giá trị 13, ở cạnh lên xung clock thứ 5 x14 được gán giá trị 14, ở cạnh lên thứ 6 thực hiện so sánh, ở cạnh lên thứ 7 x10 được gán giá trị -1.

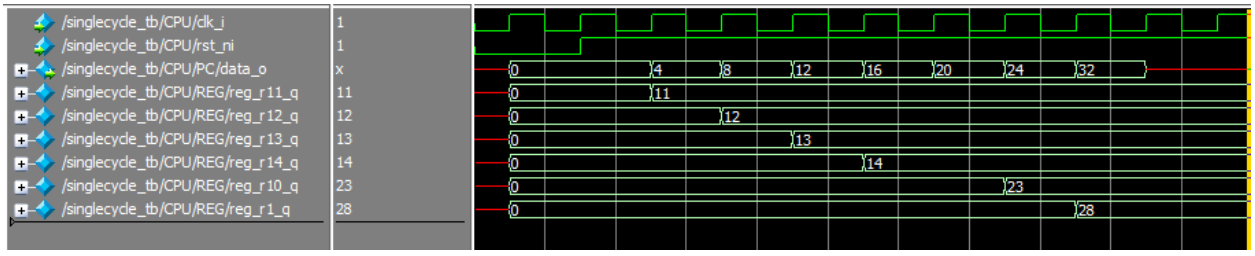
**Kiểm tra lệnh branch greater than:**

Mã assembly:

| PC | Machine Code | Basic Code | Original Code |
|---|---|---|---|
| 0x0 | 0x00B00593 | addi x11 x0 11 | addi x11, x0, 11 |
| 0x4 | 0x00C00613 | addi x12 x0 12 | addi x12, x0, 12 |
| 0x8 | 0x00D00693 | addi x13 x0 13 | addi x13, x0, 13 |
| 0xc | 0x00E00713 | addi x14 x0 14 | addi x14, x0, 14 |
| 0x10 | 0x00E6D663 | bge x13 x14 12 | bge x13,x14,Else |
| 0x14 | 0x00C58533 | add x10 x11 x12 | add x10,x11,x12 |
| 0x18 | 0x008000EF | jal x1 8 | jal x1, Exit |
| 0x1c | 0x40C58533 | sub x10 x11 x12 | Else:sub x10,x11,x12 |

Hình 49. Mã assembly lệnh bge

Waveform:




Hình 50. Waveform lệnh bge

Giải thích:

- Câu lệnh addi x11, x0, gán giá trị x11 = x0 + 11 = 11
- Câu lệnh addi x12, x0, gán giá trị x12 = x0 + 12 = 12
- Câu lệnh addi x11, x0, gán giá trị x13 = x0 + 13 = 13

- Câu lệnh addi x11, x0, gán giá trị x14 = x0 + 14 = 14
- Câu lệnh bne x13, x14, Else so sánh giá trị x13 và x14, nếu x13 < x14 thì thực hiện gán x10 = x11 + x12 rồi thanh ghi x1 = PC + 4 và nhảy tới nhãn Exit; nếu x13 >= x14 thì nhảy tới nhãn Else thực hiện gán x10 = x11 – x12. Ở đây x13 < x14 nên x10 = x11 + x12 = 11 + 12 = 23.
- Từ waveform, sau mỗi lần tăng pc, mỗi câu lệnh được thực hiện lần lượt ở cạnh lên xung clock thứ 2 x11 được gán giá trị 11, ở cạnh lên xung clock thứ 3 x12 được gán giá trị 12, ở cạnh lên thứ 4 x13 được gán giá trị 13, ở cạnh lên xung clock thứ 5 x14 được gán giá trị 14, ở cạnh lên thứ 6 thực hiện so sánh, ở cạnh lên thứ 7 x10 được gán giá trị 23, ở cạnh lên thứ 8 x1 được gán giá trị 28.

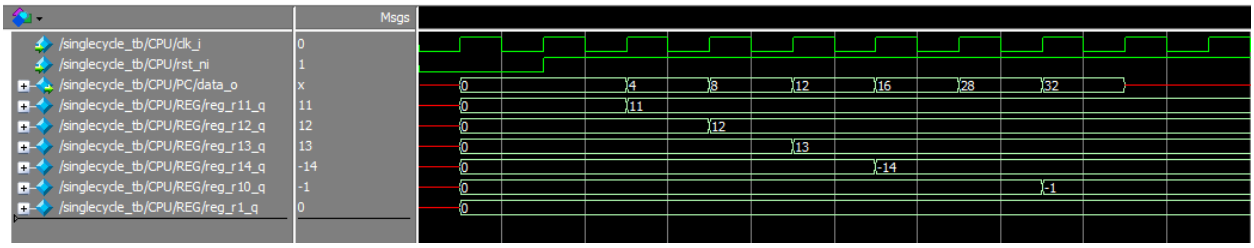
**Kiểm tra lệnh branch less than usigned:**

Mã assembly:

| PC | Machine Code | Basic Code | Original Code |
|---|---|---|---|
| 0x0 | 0x00B00593 | addi x11 x0 11 | addi x11, x0, 11 |
| 0x4 | 0x00C00613 | addi x12 x0 12 | addi x12, x0, 12 |
| 0x8 | 0x00D00693 | addi x13 x0 13 | addi x13, x0, 13 |
| 0xc | 0xFF200713 | addi x14 x0 -14 | addi x14, x0, -14 |
| 0x10 | 0x00E6E663 | bltu x13 x14 12 | bltu x13,x14,Else |
| 0x14 | 0x00C58533 | add x10 x11 x12 | add x10,x11,x12 |
| 0x18 | 0x008000EF | jal x1 8 | jal x1, Exit |
| 0x1c | 0x40C58533 | sub x10 x11 x12 | Else:sub x10,x11,x12 |

Hình 51. Mã assembly lệnh bltu

Waveform:




Hình 52. Waveform lệnh bltu

Giải thích:

- Câu lệnh addi x11, x0, gán giá trị x11 = x0 + 11 = 11

- Câu lệnh addi x12, x0, gán giá trị x12 = x0 + 12 = 12
- Câu lệnh addi x11, x0, gán giá trị x13 = x0 + 13 = 13
- Câu lệnh addi x11, x0, gán giá trị x14 = x0 + 14 = -14
- Câu lệnh bne x13, x14, Else so sánh giá trị không dấu x13 và x14, nếu x13 > x14 thì thực hiện gán x10 = x11 + x12 rồi thanh ghi x1 = PC + 4 và nhảy tới nhãn Exit; nếu x13 < x14 thì nhảy tới nhãn Else thực hiện gán x10 = x11 – x12. Ở đây x13 < x14 nên x10 = x11 - x12 = 11 - 12 = -1.
- Từ waveform, sau mỗi lần tăng pc, mỗi câu lệnh được thực hiện lần lượt ở cạnh lên xung clock thứ 2 x11 được gán giá trị 11, ở cạnh lên xung clock thứ 3 x12 được gán giá trị 12, ở cạnh lên thứ 4 x13 được gán giá trị 13, ở cạnh lên xung clock thứ 5 x14 được gán giá trị 14, ở cạnh lên thứ 6 thực hiện so sánh, ở cạnh lên thứ 7 x10 được gán giá trị -1.

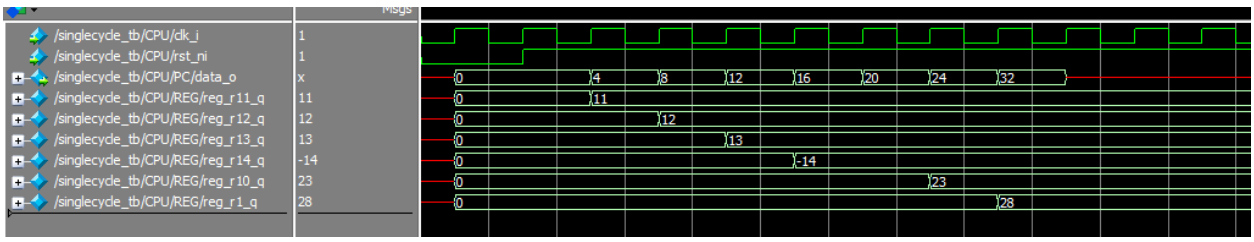
**Kiểm tra lệnh branch greater than unsigned:**

Mã assembly:

| PC | Machine Code | Basic Code | Original Code |
|---|---|---|---|
| 0x0 | 0x00B00593 | addi x11 x0 11 | addi x11, x0, 11 |
| 0x4 | 0x00C00613 | addi x12 x0 12 | addi x12, x0, 12 |
| 0x8 | 0x00D00693 | addi x13 x0 13 | addi x13, x0, 13 |
| 0xc | 0xFF200713 | addi x14 x0 -14 | addi x14, x0, -14 |
| 0x10 | 0x00E6F663 | bgeu x13 x14 12 | bgeu x13,x14,Else |
| 0x14 | 0x00C58533 | add x10 x11 x12 | add x10,x11,x12 |
| 0x18 | 0x008000EF | jal x1 8 | jal x1, Exit |
| 0x1c | 0x40C58533 | sub x10 x11 x12 | Else:sub x10,x11,x12 |

Hình 53. Mã assembly lệnh bgeu

Waveform:




Hình 54. Waveform lệnh bgeu

Giải thích:

- Câu lệnh addi x11, x0, gán giá trị x11 = x0 + 11 = 11
- Câu lệnh addi x12, x0, gán giá trị x12 = x0 + 12 = 12
- Câu lệnh addi x11, x0, gán giá trị x13 = x0 + 13 = 13
- Câu lệnh addi x11, x0, gán giá trị x14 = x0 + 14 = -14
- Câu lệnh bne x13, x14, Else so sánh giá trị không dấu x13 và x14, nếu x13 < x14 thì thực hiện gán x10 = x11 + x12 rồi thanh ghi x1 = PC + 4 và nhảy tới nhãn Exit; nếu x13 >= x14 thì nhảy tới nhãn Else thực hiện gán x10 = x11 – x12. Ở đây x13 < x14 nên x10 = x11 + x12 = 11 + 12 = 23.
- Từ waveform, sau mỗi lần tăng pc, mỗi câu lệnh được thực hiện lần lượt ở cạnh lên xung clock thứ 2 x11 được gán giá trị 11, ở cạnh lên xung clock thứ 3 x12 được gán giá trị 12, ở cạnh lên thứ 4 x13 được gán giá trị 13, ở cạnh lên xung clock thứ 5 x14 được gán giá trị 14, ở cạnh lên thứ 6 thực hiện so sánh, ở cạnh lên thứ 7 x10 được gán giá trị 23, ở cạnh lên thứ 8 x1 được gán giá trị 28.

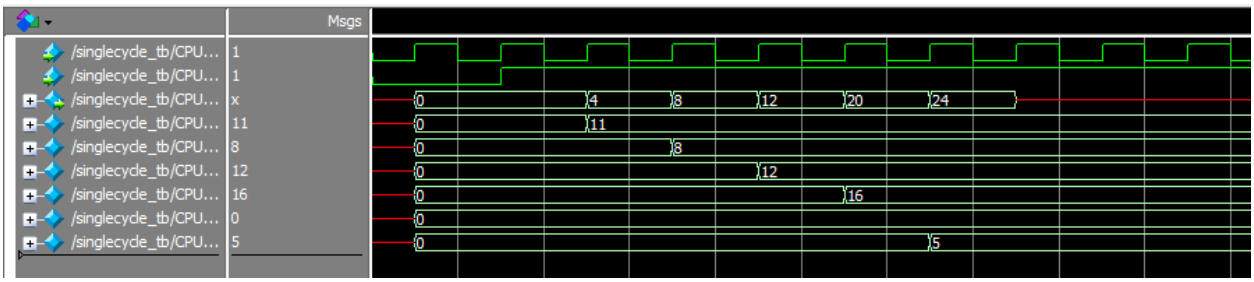
**Kiểm tra lệnh jump and link register:**

Mã assembly:

| PC | Machine Code | Basic Code | Original Code |
|---|---|---|---|
| 0x0 | 0x00B00593 | addi x11 x0 11 | addi x11, x0, 11 |
| 0x4 | 0x00800613 | addi x12 x0 8 | addi x12, x0, 8 |
| 0x8 | 0x00C00113 | addi x2 x0 12 | addi x2, x0, 12 |
| 0xc | 0x008100E7 | jalr x1 x2 8 | jalr x1, x2, 8 |
| 0x10 | 0xFF200713 | addi x14 x0 -14 | addi x14, x0, -14 |
| 0x14 | 0x00500193 | addi x3 x0 5 | addi x3, x0, 5 |

Hình 55. Mã assembly lệnh jalr

Waveform:




Hình 56. Waveform lệnh jalr

Giải thích:

- Câu lệnh addi x11, x0, 11 gán x11 = x0 + 11 = 11
- Câu lệnh addi x12, x0, 8 gán x12 = x0 + 8 = 8
- Câu lệnh addi x2, x0, 12 gán x2 = x0 + 12 = 12
- Câu lệnh jalr x1, x2, 8 thực hiện gán x1 = PC + 4 = 14 + 4 = 16 và nhảy đến địa chỉ PC = x2 + 8 = 12 + 12 = 20 và sau đó thực hiện câu lệnh ở PC = 20 là addi x3, x0, 5 gán x3 = 5 và bỏ qua câu lệnh addi x14, x0, -14
- Từ waveform, sau mỗi lần tăng pc, mỗi câu lệnh được thực hiện lần lượt ở cạnh lên xung clock thứ 2 x11 được gán giá trị 11, ở cạnh lên xung clock thứ 3 x12 được gán giá trị 8, ở cạnh lên thứ 4 x2 được gán giá trị 12, ở cạnh lên xung clock thứ 5 x1 được gán giá trị 16, ở cạnh lên thứ 6 x3 được gán giá trị 5.

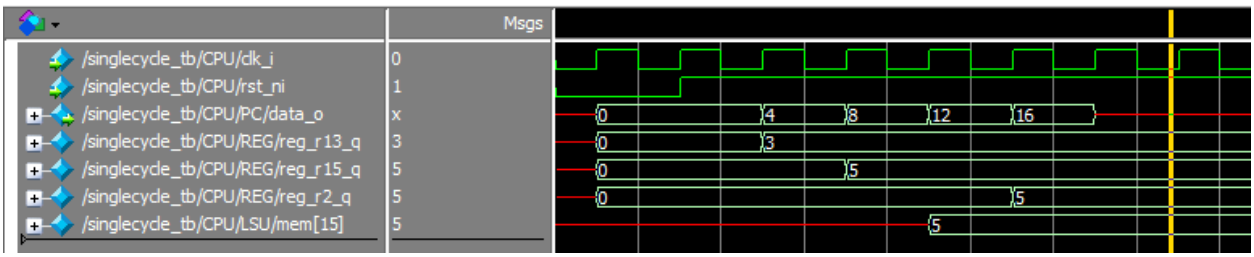
**Kiểm tra lệnh load word, store word:**

Mã assembly:

```
addi x13, x0, 3
addi x15, x0, 5
sw x15, 12(x13)
lw x2, 12(x13)
```

Hình 57. Mã assembly lệnh lw, sw

Waveform:




Hình 58. Waveform lệnh lw, sw

Giải thích:

- Câu lệnh addi x13, x0, 3 gán x10 = x0 + 3 = 3
- Câu lệnh addi x15, x0, 5 gán x15 = x0 + 5
- Câu lệnh sw x15, 12(x13) lưu giá trị địa chỉ 12 + x13 = 12 + 3 = 15 vào x15 = 5
- Câu lệnh lw x2, 12(x13) lấy giá trị từ địa chỉ 12 + x13 = 12 + 3 = 15 vào x2 = 5
- Từ waveform, sau mỗi lần tăng pc, mỗi câu lệnh được thực hiện lần lượt ở cạnh lên xung clock thứ 2 x13 được gán giá trị 3, ở cạnh lên xung clock thứ 3, x15 được gán giá trị 5, ở cạnh lên thứ 4 địa chỉ 15 được gán giá trị 5, ở cạnh lên xung clock thứ 5, x2 được gán giá trị 5 từ địa chỉ 15.

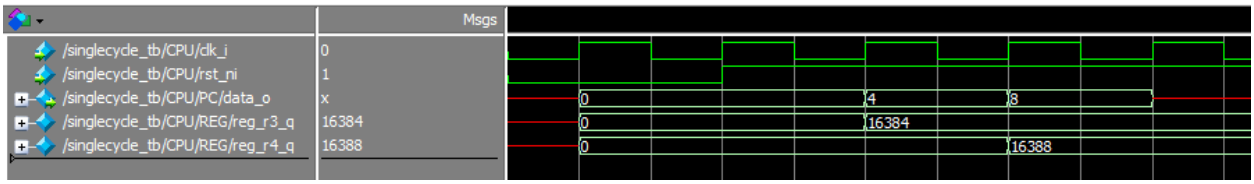
**Kiểm tra lệnh add upper imm to PC, load upper imm to PC:**

Mã assembly:

| PC | Machine Code | Basic Code | Original Code |
|---|---|---|---|
| 0x0 | 0x000041B7 | lui x3 4 | lui x3, 4 |
| 0x4 | 0x00004217 | auipc x4 4 | auipc x4, 4 |

Hình 59. Mã assembly lệnh lui, auipc

Waveform:



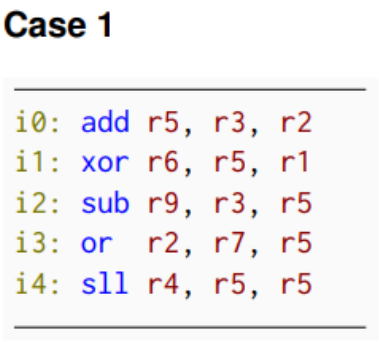

Hình 60. Waveform lệnh lui, auipc

Giải thích:

- Câu lệnh lui x3, 4 thực hiện gán giá trị x3 = 10bin << 12= 16384
- Câu lệnh auipc x4, 4 thực hiện gán giá trị x4 = PC + 10bin << 12 = 4dec + 16384dec = 16388dec.

### 4.2. Kiểm tra các trường hợp Hazard

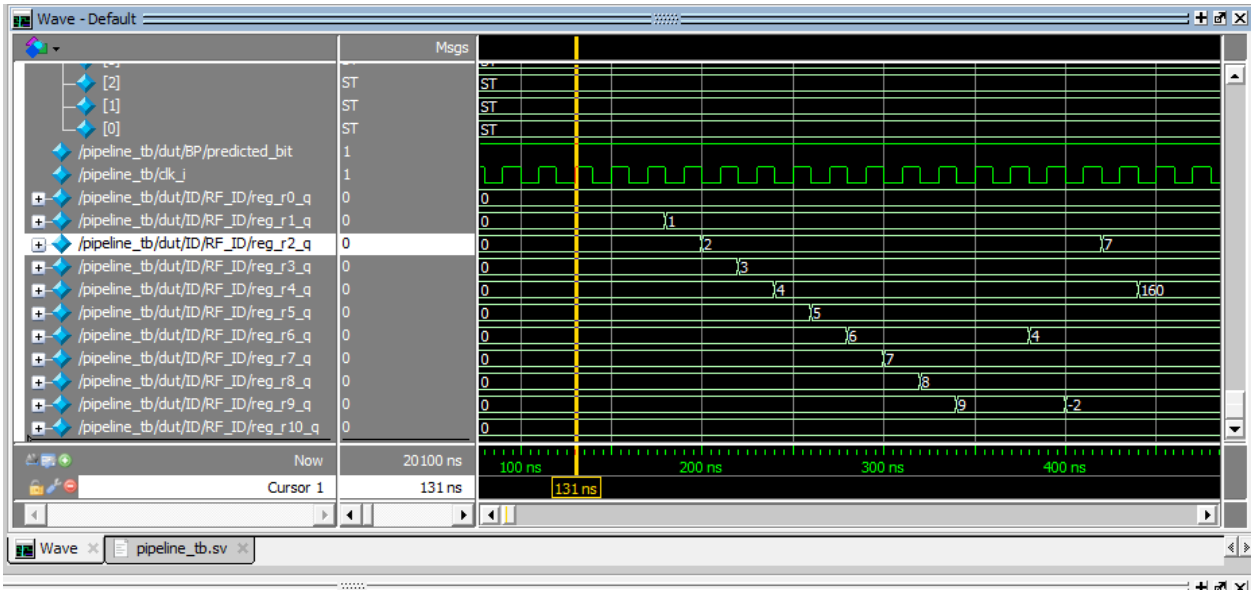
**Trường hợp 1:**




Chương trình assembly được sử dụng như sau:

- addi x1, x0, 1
- addi x2, x0, 2
- addi x3, x0, 3
- addi x4, x0, 4
- addi x5, x0, 5
- addi x6, x0, 6
- addi x7, x0, 7
- addi x8, x0, 8
- addi x9, x0, 9
- 
- add x5, x3, x2
- xor x6, x5, x1
- sub x9, x3, x5
- or x2, x7, x5
- sll x4, x5, x5

Kết quả mô phỏng thu được:




Hình 61. Waveform của trường hợp hazard 1

Các câu lệnh đọc thanh ghi x5 được thực hiện sau khi lệnh add ghi vào thanh ghi này.

**Trường hợp 2:**

**Case 2**

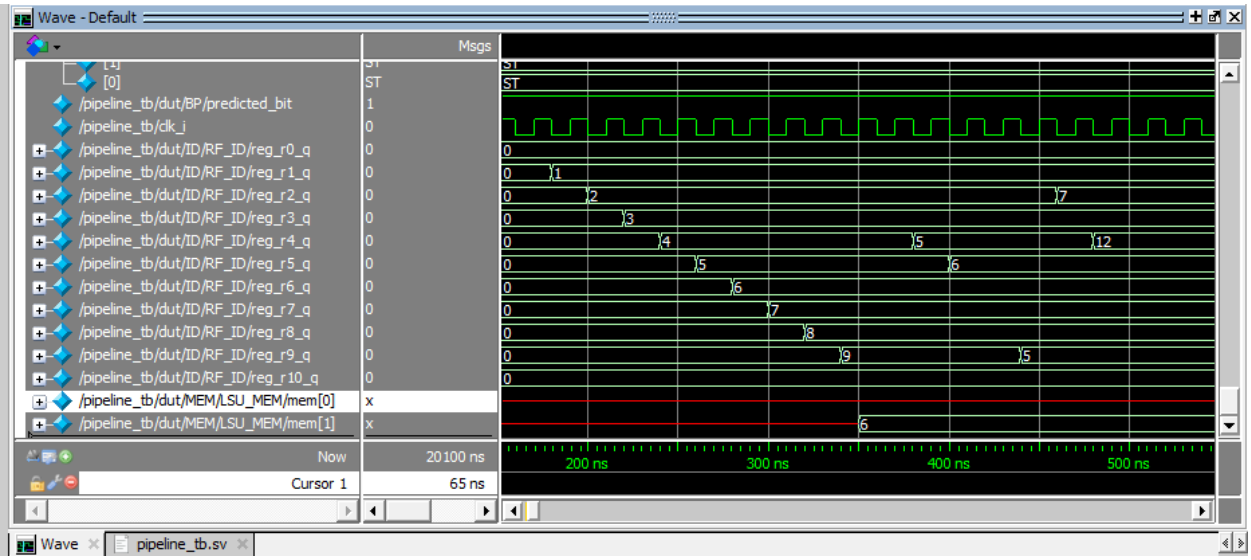
```
i0: add r4, r3, r2
i1: lw  r5, 0x40(r1)
i2: sub r9, r5, r1
i3: or  r2, r7, r5
i4: sll r4, r5, r1
```

Chương trình assembly được sử dụng:

- addi x1, x0, 1
- addi x2, x0, 2
- addi x3, x0, 3
- addi x4, x0, 4
- addi x5, x0, 5
- addi x6, x0, 6
- addi x7, x0, 7
- addi x8, x0, 8
- addi x9, x0, 9

- sw x6, 0(x1)
- 
- add x4, x3, x2
- lw x5, 0(x1)
- sub x9, x5, x1
- or x2, x7, x5
  sll x4, x5, x1

Kết quả mô phỏng thu được:




Hình 62. Waveform của trường hợp hazard 2

Thanh ghi x5 được load giá trị từ địa chỉ nhớ 1, sau đó các câu lệnh sau đọc giá trị thanh ghi x5 để tính toán.

**Trường hợp 3:**

**Case 3**

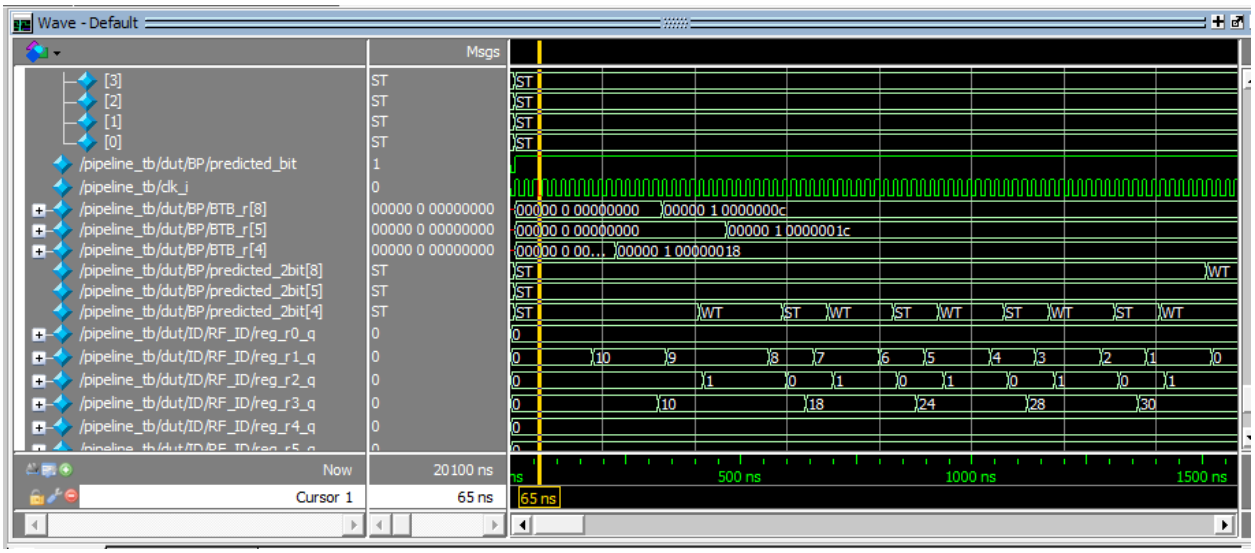
```
i0: add r4, r3, r2
i1: beq r5, r6, _L0
i2: sub r9, r5, r1
...
i8: _L0
    sll r4, r5, r1
i9: xor r6, r8, r2
```

Chương trình assembly được sử dụng:

```
addi x1, x0, 10
addi x8, x0, 1
add x3, x0, x0
_COMPARE:
and x2, x1, x8
beq x2, x0, _ADD_EVEN
jal x10, _DECREASE
_ADD_EVEN:
add x3, x1, x3
_DECREASE:
sub x1, x1, x8
bne x1, x0, _COMPARE
_EXIT:
```

Chương trình assembly trên sẽ tính tổng các số chẵn từ 1 đến 10 và lưu vào thanh ghi x3.

Kết quả mô phỏng thu được:




Hình 63. Waveform trường hợp hazard 3

Trong waveform, BTB đã được cập nhật giá trị tag và predicted pc ở index cụ thể, tương tự như BTB, trạng thái của two-bit prediction cũng được cập nhật khi nhảy sai với dự đoán.

Two-bit prediction có 4 trạng thái:

- ST (strongly taken)
- WT (weakly taken)
- SNT (strongly not taken)
- WNT (weakly not taken)

Chỉ khi lệnh nhảy được dự đoán sai 2 lần liên tiếp thì two-bit prediction mới thay đổi từ taken sang not taken hoặc ngược lại. Trong waveform cho thấy lệnh nhảy beq x2, x0, _ADD_EVEN sẽ taken và not taken liên tiếp nhau nên two-bit prediction chỉ thay đổi từ ST sang WT và ngược lại.

### 4.3. Psuedo LRU tree-based

| Node/Step | Step 1 | Step 2 | Step 3 | Step 4 | Step 5 | Step 6 | Step 7 | Step 8 |
|---|---|---|---|---|---|---|---|---|
| L0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| L1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| L2 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| L3 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| L4 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| L5 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| L6 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| | | | | | | | | |
| LRU | 111 | 011 | 101 | 001 | 110 | 010 | 100 | 000 |

Hình 64. Giá trị PLRU ở mỗi lần truy cập

Ở hình trên, giá trị của mỗi node được lưu lại qua từng step (hoặc gọi là lần truy cập), ở step 1 giá trị của các node đều bằng 0, do đó giá trị LRU sẽ có giá trị là 111 như sơ đồ 1.2. Tiếp đến step 2, giá trị LRU ở step 1 sẽ được sử dụng để cập nhật lại LRU tree, ở bước này, giá trị LRU trước đó là 111, do đó L0 được cập nhật bằng 1, vì L0 bằng 1 nên L2 sẽ được cập nhật bằng LRU[1] và bằng 1, và vì L2 bằng 1 nên L6 sẽ được cập nhật bằng LRU[0] và bằng 1. Tương tự với các step sau đó.

Đoạn code assembly như hình dưới sẽ được dùng để kiểm tra hoạt động của bộ 8-way set associative cache.

```
1  addi x1, x0, 1
2  addi x2, x0, 2
3  addi x3, x0, 333
4  addi x4, x0, 444
5  addi x5, x0, 555
6  addi x6, x0, 666
7  addi x7, x0, 777
8  addi x8, x0, 888
9
10 sw x1, 0(x0) #0/0
11 sw x2, 16(x0) #0/1
12 sw x3, 32(x0) #0/2
13 sw x8, 4(x0) #1/0
14 lw x10, 4(x0)
15 sw x4, 48(x0) #0/3
16 sw x5, 64(x0) #0/4
17 sw x8, 64(x0) #0/4
18 sw x6, 80(x0) #0/5
19 sw x7, 96(x0) #0/6
20 sw x8, 112(x0) #0/7
21 sw x1, 32(x0) #0/2
22 addi x3, x0, 50
23 loop: addi x3, x3, -1
24 add x9, x9, x8
25 bne x3, x0, loop
26 sw x9, 20(x0) #1/1
27 sw x2, 16(x0) #0/1
28 sw x5, 144(x0) #0/9
29 sw x4, 128(x0) #0/8
```

Ban đầu, các giá trị thanh ghi từ x1 đến x8 được khởi tạo để thuận tiện cho quá trình kiểm tra kết quả.

Ở dòng 10, giá trị thanh ghi x1 sẽ được lưu vào set thứ nhất (mỗi set chứa 8 ways). Tiếp đến x2, x3 cũng được lưu vào set thứ nhất, vì mỗi set chứa 8 block nên lúc này 3 block chứa giá trị của x1, x2, x3 và còn 5 block chưa có giá trị.

Ở dòng 13, giá trị thanh ghi x8 được lưu và set thứ 2. Dòng 14, giá trị ở set thứ 2 sẽ được load vào x10.

Tương ứng các dòng sau đó, x4, x5, x8 (vì x8 được lưu vào địa chỉ 64 đã lưu trước đó nên block có địa chỉ 64 sẽ thay thế x5 bằng x8), x6, x7, x8 lưu vào set thứ nhất. Lúc này set thứ nhất đã lưu đủ 8 giá trị (set thứ nhất đầy).

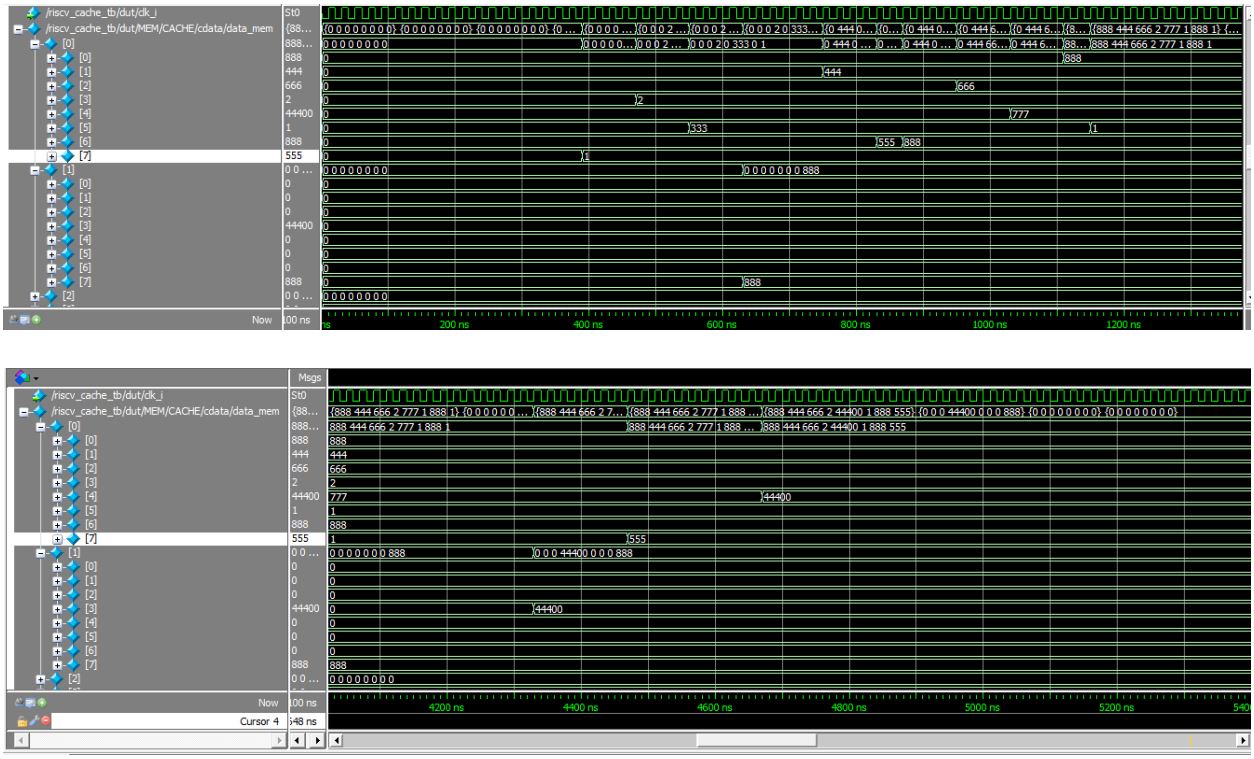
Do đó, lệnh ở dòng 21, ô có địa chỉ 32 chứa giá trị 333 (thanh ghi x3) sẽ được thay thế bằng 1 (thanh ghi x1).

Từ dòng 23 đến dòng 25, giá trị x9 sẽ được tính toán và có giá trị sau cùng là 888 nhân 50 và bằng 44400. Sau đó x9 được lưu vào set thứ 2.

Dòng 27, ô có địa chỉ 16 sẽ cập nhật giá trị của x2.

Ở dòng 28, vì set thứ nhất đã đầy và không có block nào chứa địa chỉ khớp với địa chỉ 144, do đó 1 block nào đó (được quyết định bởi khối LRU) sẽ được lưu lại giá trị vào memory và cập nhật giá trị mới. Tương tự với dòng lệnh 29.

Dưới đây là waveform mô tả đoạn code assembly phía trên.

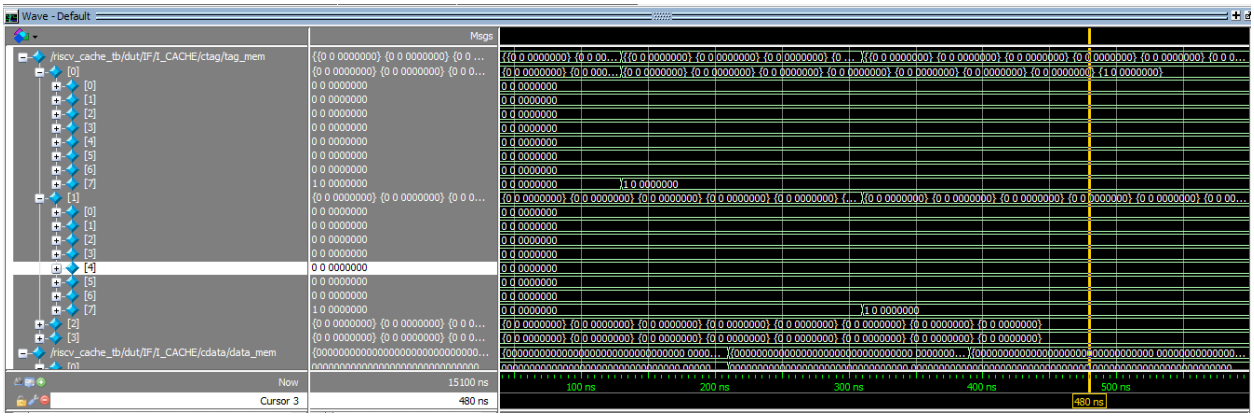



Hình 65. Waveform của pLRU tree-based

Như dữ liệu đã tính toán từ hình 65, giá trị ban đầu sẽ được lưu vào block 7 tiếp đến lần lượt là block 3, 5, 1, 6, 2, 4, 0. Và mỗi lần truy cập sau đó, LRU tree sẽ được cập nhật và đưa ra địa chỉ của block tiếp theo cần được ghi ở từng set.

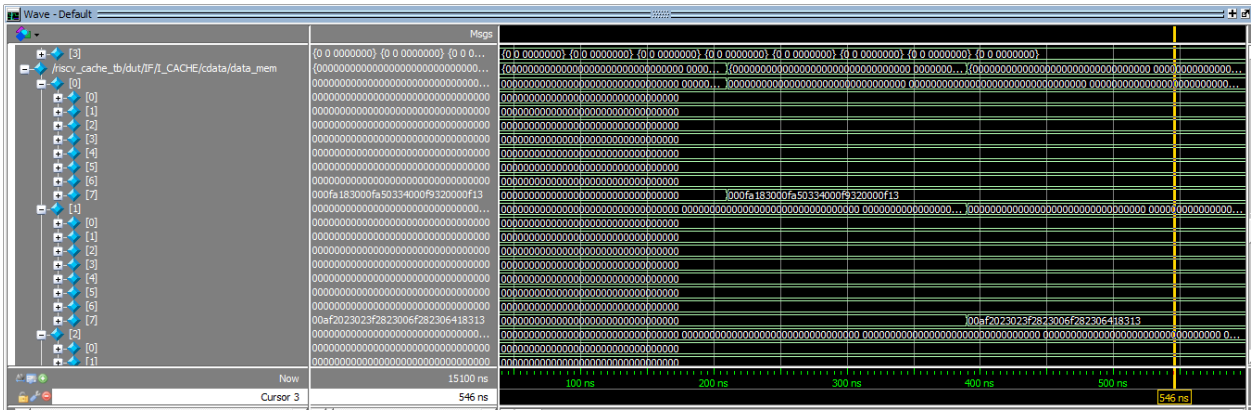
## 4.4. Kiểm tra hoạt động của Instruction cache

Instruction cache được thiết kế theo cấu trúc 8-way set associative, 1 block gồm 4 word (128 bits). Khi CPU hoạt động, CPU sẽ fetch instruction từ i-cache, nếu i-cache không chứa instruction cần đọc từ CPU phần tag của i-cache sẽ được khởi tạo. Ngược lại, instruction từ i-cache sẽ được gửi tới CPU để hoạt động.



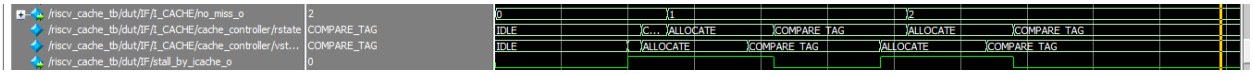

Hình 66. Waveform khởi tạo tag cho i-cache

Ở hình trên, giá trị tag được khởi tạo ở i-cache, vì i-cache được thiết kế gồm 8 way, do đó giá trị ban đầu được lưu vào way cuối cùng, mỗi phần tử là 1 mega bit bao gồm 1 bit valid, 1 bit dirty và phần tag của địa chỉ.




Hình 67. Waveform khởi tạo giá trị cho i-cache

Tương ứng với giá trị tag được khởi tạo, các giá trị instruction cũng được lưu vào i-cache. Vì i-cache được thiết kế với mỗi block gồm 4 word, do đó mỗi phần tử ở i-cache sẽ lưu được 4 instruction tương ứng với 128 bit. Như vậy, mỗi lần CPU đọc giá trị từ i-cache mà không chứa dữ liệu cần đọc (cache miss), i-cache sẽ fetch 4 instruction từ L2-cache hoặc từ Memory, trường hợp giá trị đã lưu trong i-cache (cache hit) thì i-cache không cần fetch dữ liệu từ L2-cache hoặc Memory mà sử dụng sẵn những instruction đã chứa trong i-cache.

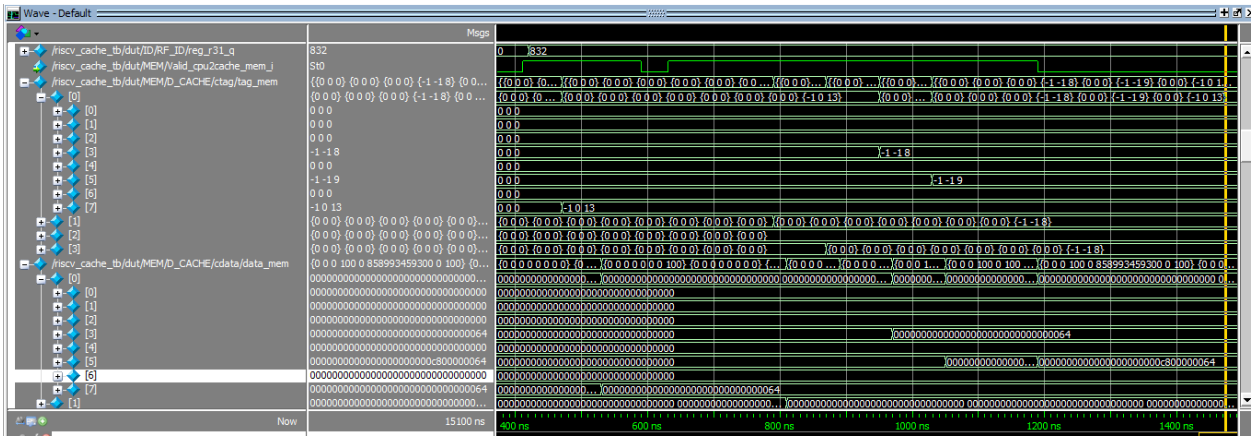



Hình 68. Waveform tín hiệu trạng thái của i-cache controller

Hoạt động của i-cache sẽ gồm 2 trạng thái chính là Compare tag và Allocate. Tương ứng với mỗi lần khởi tạo tag và lưu giá trị instruction vào i-cache, ban đầu i-cache sẽ ở trạng thái Compare tag để so sánh phần địa chỉ và dữ liệu trong i-cache, nếu i-cache không chứa instruction cần đọc từ CPU, i-cache sẽ vào trạng Allocate và fetch những instruction từ Memory ở tầng dưới sau đó trở lại trạng thái Compare tag, ngược loại nếu i-cache chứa instruction cần đọc, i-cache vẫn sẽ ở trạng thái Compare tag.

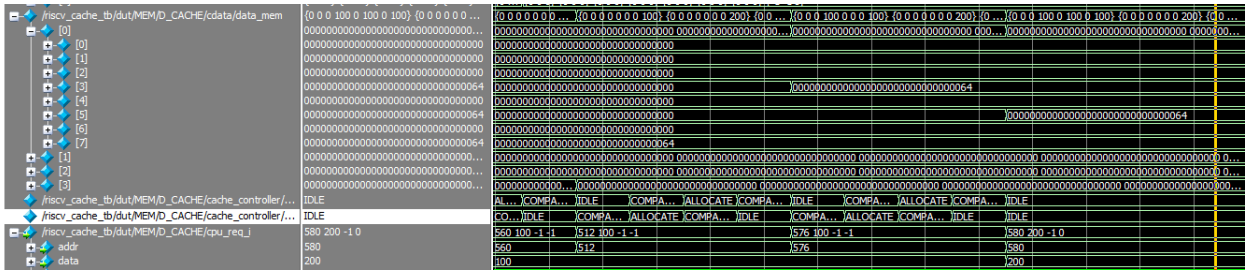
## 4.5. Kiểm tra hoạt động của Data cache

Data cache được thiết kế với cấu trúc 8-way set associative, với mỗi block chứa 4 word (128 bits). Khi CPU thực hiện lệnh lw hoặc sw, d-cache sẽ được truy cập. D-cache được thiết kế gồm 4 trạng thái, ban đầu d-cache luôn ở trạng thái IDLE. Nếu CPU thực hiện 1 trong 2 lệnh lw hoặc sw, d-cache sẽ chuyển sang trạng thái Compare tag để so sánh địa chỉ với dữ liệu trong d-cache. Trường hợp nếu d-cache không chứa dữ liệu mà CPU truy cập, d-cache sẽ chuyển sang trạng thái Allocate (nếu gặp lệnh đọc hoặc ghi nhưng dữ liệu này chưa được ghi ở các tầng dưới) hoặc Writeback (nếu gặp lệnh ghi mà dữ liệu này đã được lưu trong d-cache và chưa lưu trong các tầng dưới) để đọc hoặc ghi dữ liệu vào Memory ở các tầng dưới, ở trường hợp đọc, giá trị từ L2-cache hoặc Memory sẽ được lưu vào d-cache. Sau khi đọc hoặc ghi từ L2-cache hoặc Memory, d-cache sẽ chuyển sang trạng thái Compare tag để đảm bảo giá trị từ L2-cache hoặc Memory được lưu vào d-cache và trả lại giá trị cho CPU nếu là lệnh đọc, tiếp đến d-cache sẽ trở về trạng thái IDLE sau khi CPU đã truy cập xong d-cache. Trong trường hợp d-cache chứa dữ liệu mà CPU truy cập, d-cache chuyển từ trạng thái IDLE sang Compare tag, sau đó ghi dữ liệu vào d-cache nếu là lệnh ghi, khi gặp lệnh đọc d-cache sẽ trả lại giá trị đọc được cho CPU và sau đó trở về trạng thái IDLE.




Hình 69. Waveform lưu giá trị ở d-cache

Như hình trên, các giá trị valid, dirty và tag được khởi tạo vào d-cache khi CPU thực hiện lệnh sw. Tương ứng với các giá trị tag, các dữ liệu cũng được lưu vào phần data array của d-cache.



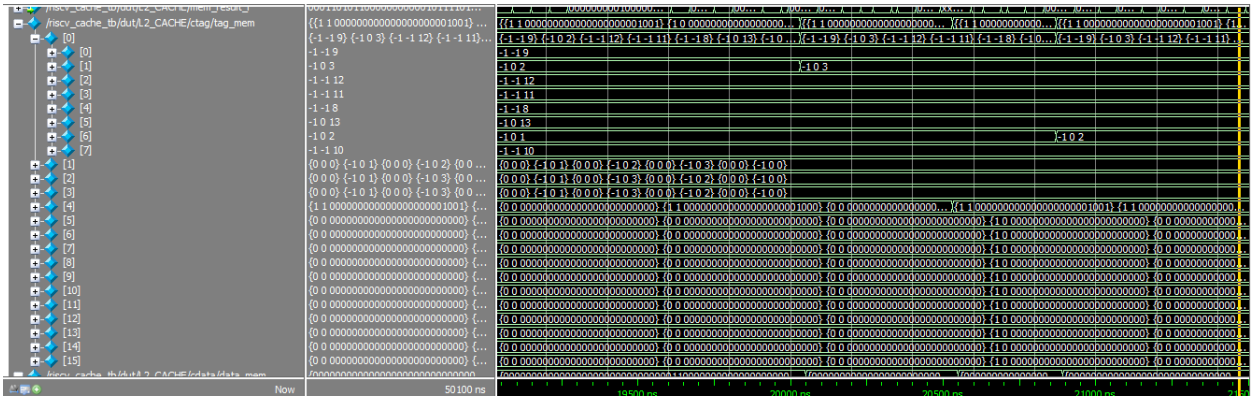

Hình 70. Waveform tín hiệu hoạt động của d-cache controller

Tương ứng với các lệnh sw từ CPU, trạng thái của d-cache controller được thay đổi ghi có lệnh lw hoặc sw. Khi d-cache không chứa giá trị CPU cần truy cập, controller chuyển từ trạng thái Compare tag qua trạng thái Allocate để lưu giá trị vào d-cache.

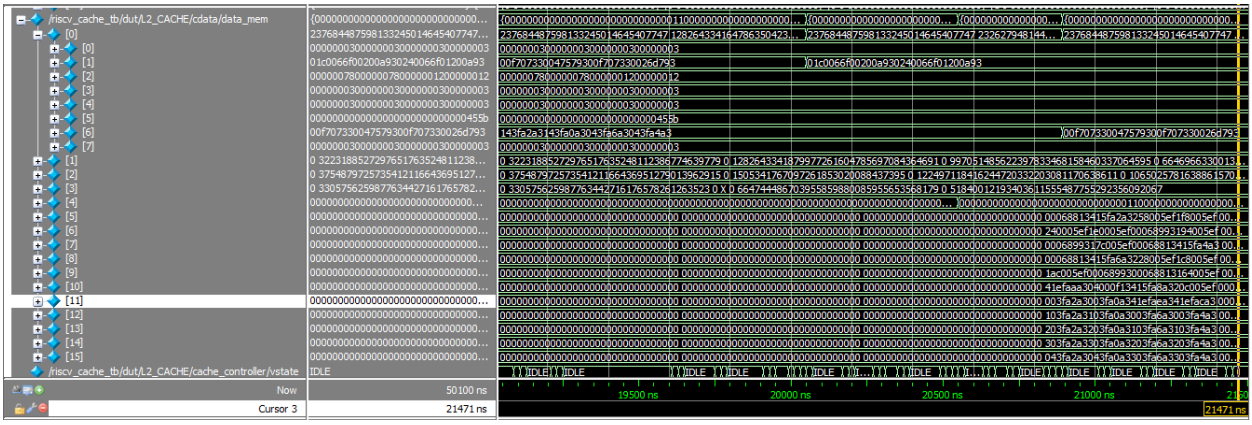
## 4.6. Kiểm tra hoạt động của L2 cache

L2 cache được thiết kế với cấu trúc 8-way set associative, với mỗi block chứa 128 bits (4 words). L2 cache chứa cả instruction và data, thiết kế theo kiểu unified và bao gồm tất cả thông tin của Instruction cache và Data cache. Do đó kích thước của L2 cache được thiết kế lớn hơn gấp nhiều lần i-cache và d-cache, ở phần kiểm tra này, L2 cache có lớn hơn i-cache hoặc d-cache (i-cache bằng kích thước d-cache) 4 lần. L2 cache sẽ được truy cập khi Instruction cache hoặc Data cache bị miss dữ liệu, tức là dữ liệu CPU cần không chứa trong i-cache hoặc d-cache. Cũng như hoạt động ở d-cache, khi L2 cache được truy cập và chứa dữ liệu mà L1 cache (i-cache hoặc d-cache)

yêu cầu thì dữ liệu sẽ được truyền từ L2 cache tới L1 cache. Ở trường hợp ngược lại, Khi L2 cache không chứa dữ liệu mà L1 cache yêu cầu, L2 cache sẽ gửi yêu cầu tới Memory để lấy dữ liệu về sau đó chuyển tiếp tới L1 cache. Hoạt động của L2 cache cũng được thiết kế tương tự với d-cache bao gồm IDLE, COMPARE_TAG, ALLOCATE, WRITEBACK. Những hình ảnh dưới đây mô phỏng dữ liệu được lưu vào L2 cache.



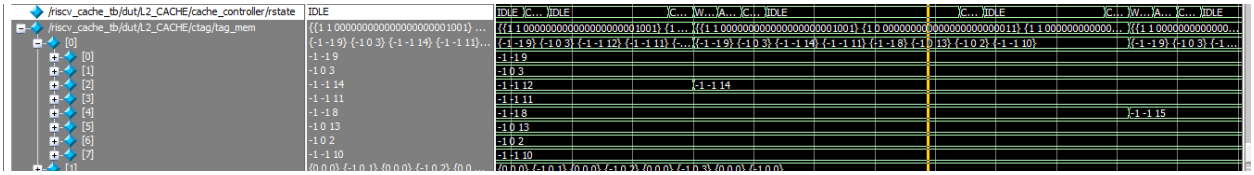
Hình 71. Waveform cho tag memory ở L2 cache



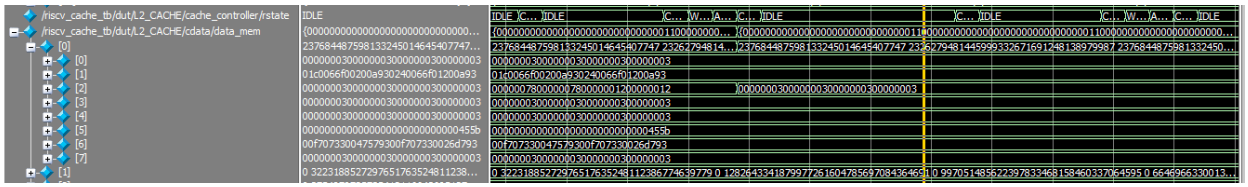
Hình 72. Waveform cho data memory ở L2 cache

Ở hình trên, L2 cache chứa cả những dữ liệu như Instruction được fetch từ Memory hay Data được lưu từ d-cache hoặc fetch từ Memory.

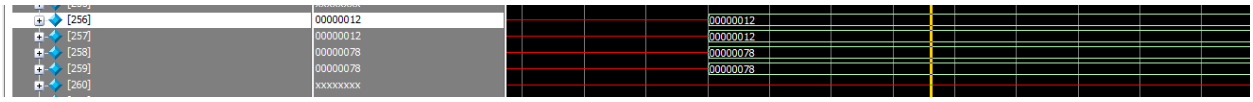
Khi 1 set ở L2 cache được lưu trữ đầy thì dữ liệu sẽ được ghi vào Memory. Tương ứng trạng thái của L2 cache sẽ chuyển từ COMPARE TAG sang WRITEBACK để lưu dữ liệu vào Memory. Waveform dưới mô tả những tín hiệu hoạt động cũng như dữ liệu của L2 cache ghi có set bị đầy.



Hình 73. Waveform cho tag memory ở L2 cache khi có set bị đầy.



Hình 74. Waveform cho data memory ở L2 cache khi set bị đầy.



Hình 75. Waveform mô tả dữ liệu được lưu và Memory khi L2 cache có set bị đầy.

Ở những hình ảnh waveform trên, dữ liệu được lưu từ L2 cache vào Memory có tag bằng 12 và data ở 4 word tương ứng là 0x12, 0x12, 0x78, 0x78. Dữ liệu bị thay thế khi 1 set ở L2 cache bị đầy và có dữ liệu mới cần lưu vào (tag là 14). Khi L2 cache ở trạng thái WRITEBACK, dữ liệu chứa 4 word sẽ được truyền tới Memory và lưu vào các ô nhớ tương ứng. Ở ví dụ này, vì L2 cache được thiết kế chứa 16 set, 4 word, 8 way nên phần tag sẽ gồm từ bit thứ 9 tới bit 32 (8 bit đầu gồm 4 bit cho 16 set, 2 bit cho word, 2 bit cho offset byte), từ đó tag có giá trị 12 sẽ tương ứng với địa chỉ 12*(2^8) là 3072, địa chỉ này sẽ lưu về Memory (ở ví dụ này, Memory có dung lượng 6KB, tương ứng là 2KB cho instruction và 4KB cho data, mà 2KB của instruction sẽ gồm 512 ô nhớ chứa 32 bits), giá trị 3072 này sẽ được chia 4 khi loại bỏ 2 bit offset byte trở thành 768, sau đó trừ đi 512 (vì phần instruction được lưu trước phần data, và để thuận tiện cho mô phỏng thì 2 mảng lưu instruction và data được tách ra riêng biệt) bằng 256. Vì vậy, giá trị ở các ô nhớ 256, 257, 258, 259 sẽ lần lượt lưu 4 words data chuyển từ L2 cache.

## 5. Đánh giá hệ thống

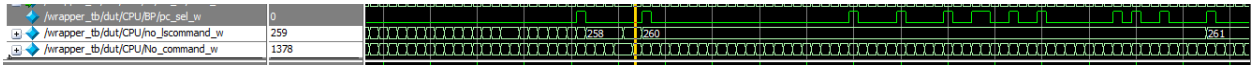
Đánh giá hiệu suất của hệ thống thông qua 2 giá trị AMAT (average memory access time) và CPI (cycles per instruction).

### ❖ Chương trình biến đổi số nhị phân thành thập phân

Hệ thống được kiểm tra bằng chương trình hợp ngữ có chức năng đọc giá trị nhị phân và biến đổi thành giá trị thập phân (có lồng các câu lệnh load và store để kiểm tra cache). Kích thước của cache được đặt với giá trị 512 bytes cho D-cache và I-cache, 2 KB cho L2-cache (dung lượng của cache được đặt khá nhỏ để dễ dàng kiểm chứng qua waveform).

Sau khi mô phỏng 3 lần đọc giá trị và xử lý, thu được kết quả như sau:

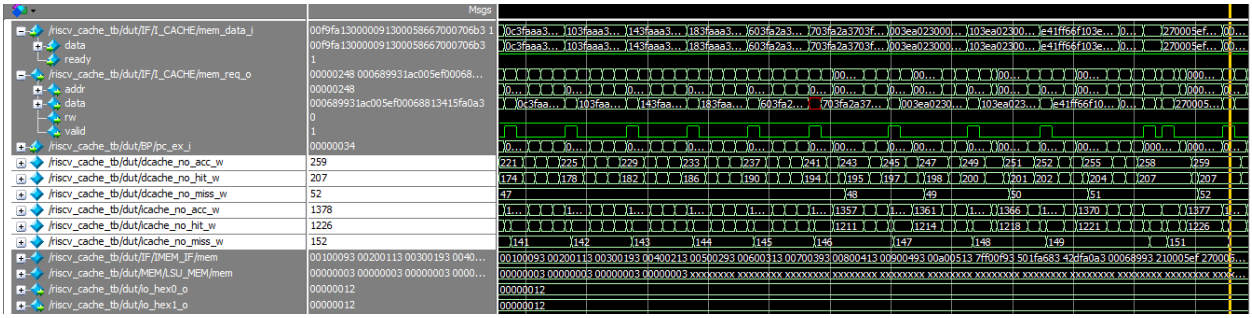
- Hệ thống không chứa cache:




Số câu lệnh thực hiện: 1378

Số câu lệnh load/store: 259

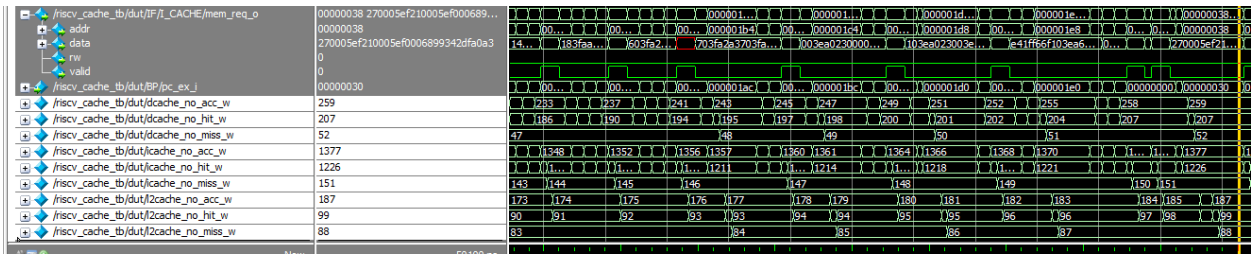
- Hệ thống One level cache:




Ở D-cache: số lần access là 259, số lần hit là 207, số lần miss là 52.

Ở I-cache: số lần access là 1377, số lần hit là 1226, số lần miss là 151.

- Hệ thống Two level cache:




Ở L1 D-cache: số lần access là 259, số lần hit là 207, số lần miss là 52.

Ở L1 I-cache: số lần access là 1377, số lần hit là 1226, số lần miss: 151.

Ở L2 cache: số lần access là 187, số lần hit là 99, số lần miss là 88.

Tính toán các thông số AMAT và CPI: giả sử thời gian truy cập L1 cache là 1ns, L2 cache là 10ns và thời gian truy cập bộ nhớ chính là 100ns. Chu kì mô phỏng là 2ns.

- Hệ thống không chứa cache:

  $AMAT = 100\ ns$

  $CPI$
  $= (\frac{time\ execute}{period} + number\ of\ miss * cycles\ penalty)/number\ of\ instructions$
  $= (\frac{2773}{2} + (259 + 1377) * \frac{100}{2})/1377 = 60.4114$

- Hệ thống One level cache:

  $AMAT\ of\ data: H1 * T1 + (1 - H1) * Tmem = \frac{207}{259} * 1 + \left(1 - \frac{207}{259}\right) * 100$
  $= 20.8764\ ns$

  $AMAT\ of\ instruction: H1 * T1 + (1 - H1) * Tmem = \frac{1226}{1377} * 1 + \left(1 - \frac{1226}{1377}\right) * 100$
  $= 11.8562\ ns$

  $AMAT = 20.8764 + 11.8562 = 32.7362\ ns$

  $CPI$
  $= (\frac{time\ execute}{period} + number\ of\ miss * cycles\ penalty)/number\ of\ instructions$
  $= (\frac{4171}{2} + (52 + 151) * \frac{100}{2})/1377 = 8.8856$

- Hệ thống Two level cache:

  $AMAT\ of\ data: H1 * T1 + (1 - H1) * (H2 * T2 + (1 - H2) * Tmem)$
  $= \frac{207}{259} * 1 + \left(1 - \frac{207}{259}\right) * \left(\frac{99}{187} * 10 + \left(1 - \frac{99}{187}\right) * 100\right) = 11.3102ns$

  $AMAT\ of\ instruction: H1 * T1 + (1 - H1) * (H2 * T2 + (1 - H2) * Tmem)$
  $= \frac{1226}{1377} * 1 + \left(1 - \frac{1226}{1337}\right) * \left(\frac{99}{187} * 10 + \left(1 - \frac{99}{187}\right) * 100\right) = 6.6313\ ns$

  $AMAT = 11.3102 + 6.6313 = 17.9415\ ns$

$$CPI = \left(\frac{time\ execute}{period} + number\ of\ miss\ L1 * cycles\ penalty\ L2 + number\ of\ miss\ L2 * cycles\ penalty\ memory\right) \Big/ number\ of\ instructions$$

$$= \left(\frac{4925}{2} + (52 + 151) * \frac{10}{2} + 88 * \frac{100}{2}\right)/1377 = 5.7208$$

Bảng dưới đây sẽ tóm tắt lại các thông số so sánh từng loại hệ thống:

Bảng 9. So sánh các thông số đối với hệ thống không chứa cache, one level cache, two level cache

| | Miss rate D-cache (%) | Miss rate I-cache (%) | Miss rate L2 cache (%) | AMAT (ns) | CPI |
|---|---|---|---|---|---|
| No cache | 100 | 100 | 100 | 100 | 60.4114 |
| One level cache | 20.08 | 10.97 | 100 | 32.7362 | 8.8856 |
| Two level cache | 20.08 | 10.97 | 47.06 | 17.9415 | 5.7208 |

Với chương trình kiểm tra assembly đơn giản, thu được các dữ liệu như hit rate, miss rate, số cycle mô phỏng được,... qua đó tính toán được các thông số như AMAT và CPI. Với hệ thống chứa cache, AMAT giảm đáng kể và đối với hệ thống two-level cache, hệ số hit rate của L2 cache giúp thông số AMAT và CPI giảm thấp hơn one-level cache. Hệ số AMAT ở hệ thống one-level cache giảm khoảng 67% so với hệ thống processor không chứa cache, CPI giảm khoảng 85%. Hệ số AMAT ở hệ thống two-level cache giảm khoảng 83% so với hệ thống processor không chứa cache, CPI giảm khoảng 90%, đối với hệ thống one-level cache giảm khoảng 45% ở hệ số AMAT và giảm 35% ở hệ số CPI. Từ đó nhận thấy khi thiết kế hệ thống cache cho processor giúp tăng hiệu suất đáng kể, khi processor được thiết kế thêm hệ thống cache sẽ làm giảm đáng kể thời gian trung bình truy cập bộ nhớ và từ đó dẫn tới hệ số CPI của toàn hệ thống cũng được giảm đáng kể, cho thấy hiệu suất của processor được cải thiện rõ rệt. Ở đây, hệ thống được đánh giá thông qua 1 chương trình đơn giản, do đó các hệ số đánh giá này có thể chênh lệch đáng kể. Với hệ thống CPU two-level cache, hệ số AMAT và CPI đều tốt hơn so với hệ thống one-level cache, từ đó kết luận rằng khi hệ thống two-level cache cho CPU đã được thiết kế để tăng được hiệu suất cho toàn hệ thống processor.

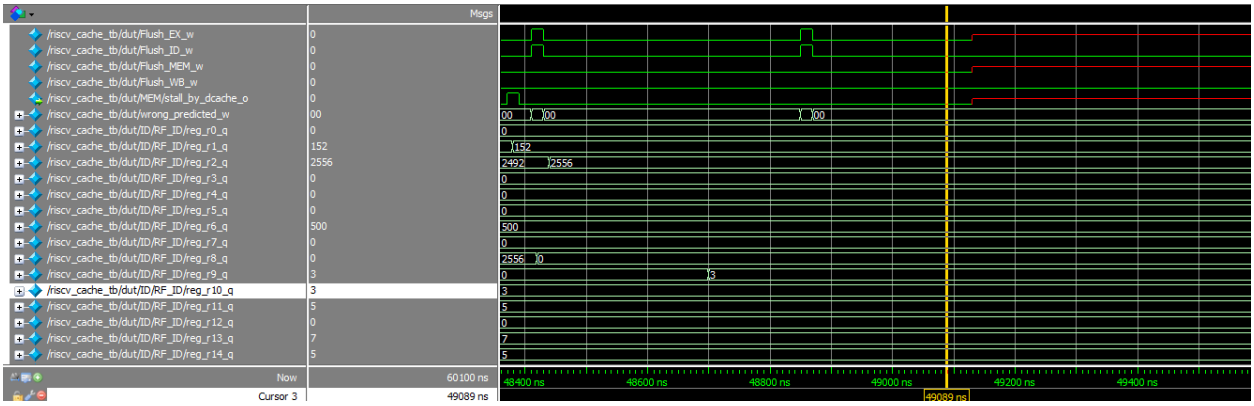
❖ **Benchmark: binary search (ASCS Lab/ECE/BU)**

Đây là chương trình thực hiện thuật toán tìm kiếm nhị phân. Source code C và assembly được đăng tải trên link sau: *https://github.com/Stork1323/riscv_cache/tree/main/benchmark/code*

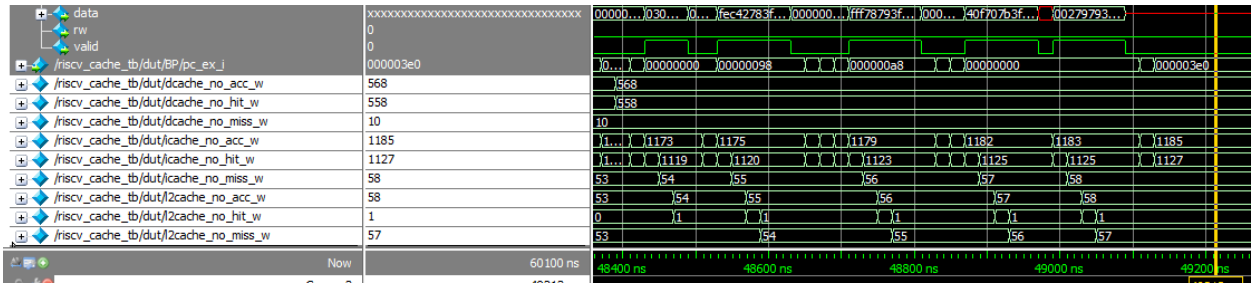
Giải thích chương trình test: ban đầu khởi tạo 1 mảng gồm 8 phần tử mang giá trị lần lượt là 0, 6, 8, 4, 3, 9, 7, 5. Hàm sort sẽ sắp xếp lại mảng này từ bé đến lớn để thực hiện tìm kiếm nhị phân. Giá trị được đưa vào hàm binary search là 5, mà sau khi sắp xếp thì mảng n chứa các giá trị lần

lượt là 0, 3, 4, 5, 6, 7, 8, 9, do đó kết quả trả về sau khi tìm kiếm là 3 (vị trí của giá trị 5 được chứa trong mảng n).

Theo như chương trình test, kết quả mang giá trị 3 sẽ được lưu vào thành ghi x10. Dưới đây là các hình ảnh minh họa mô phỏng kết quả chạy test.




Hình 76. Waveform cho benchmark binary search (1)




Hình 77. Waveform cho benchmark binary search (2)

Sau khi mô phỏng, thu được kết quả như sau:

Bảng 10. Dữ liệu thu được khi mô phỏng chương trình binary search

| Số lần access | | | Số lần miss | | | Miss rate (%) | | |
|---|---|---|---|---|---|---|---|---|
| **I-cache** | **D-cache** | **L2 cache** | **I-cache** | **D-cache** | **L2 cache** | **I-cache** | **D-cache** | **L2 cache** |
| 1185 | 568 | 58 | 58 | 10 | 57 | 4.89 | 1.76 | 98.28 |

Tính toán các thông số AMAT và CPI: giả sử thời gian truy cập L1 cache là 1ns, L2 cache là 10ns và thời gian truy cập bộ nhớ chính là 100ns. Chu kì mô phỏng là 2ns.

- Hệ thống không chứa cache:

$$AMAT = 100\ ns$$

$$CPI = (\frac{time\ execute}{period} + number\ of\ miss * cycles\ penalty)/number\ of\ instructions$$
$$= \left(\frac{3155}{2} + (568 + 1185) * \frac{100}{2}\right)/1185 = 75.2975$$

- Hệ thống One level cache:

$$AMAT\ of\ data: H1 * T1 + (1 - H1) * Tmem = \frac{558}{568} * 1 + \left(1 - \frac{558}{568}\right) * 100$$
$$= 2.743\ ns$$

$$AMAT\ of\ instruction: H1 * T1 + (1 - H1) * Tmem = \frac{1127}{1185} * 1 + \left(1 - \frac{1127}{1185}\right) * 100$$
$$= 5.8456\ ns$$

$$AMAT = 2.743 + 5.8456 = 8.5886\ ns$$

$$CPI = (\frac{time\ execute}{period} + number\ of\ miss * cycles\ penalty)/number\ of\ instructions$$
$$= \left(\frac{4561}{2} + (10 + 58) * \frac{100}{2}\right)/1185 = 4.7937$$

- Hệ thống Two level cache:

$$AMAT\ of\ data: H1 * T1 + (1 - H1) * (H2 * T2 + (1 - H2) * Tmem)$$
$$= \frac{558}{568} * 1 + \left(1 - \frac{558}{568}\right) * \left(\frac{1}{58} * 10 + \left(1 - \frac{1}{58}\right) * 100\right) = 2.7156ns$$

$$AMAT\ of\ instruction: H1 * T1 + (1 - H1) * (H2 * T2 + (1 - H2) * Tmem)$$
$$= \frac{1127}{1185} * 1 + \left(1 - \frac{1127}{1185}\right) * \left(\frac{1}{58} * 10 + \left(1 - \frac{1}{58}\right) * 100\right) = 5.7696\ ns$$

$$AMAT = 2.7156 + 5.7696 = 8.4852\ ns$$

$$CPI = (\frac{time\ execute}{period} + number\ of\ miss\ L1 * cycles\ penalty\ L2 + number\ of\ miss\ L2 * cycles\ penalty\ memory)/number\ of\ instructions$$
$$= \left(\frac{4899}{2} + (10 + 58) * \frac{10}{2} + 57 * \frac{100}{2}\right)/1185 = 4.7591$$

Bảng dưới đây sẽ tóm tắt lại các thông số so sánh từng loại hệ thống:

Bảng 11. So sánh các thông số đối với hệ thống không chứa cache, one level cache, two level cache cho benchmark binary search

| | Miss rate D-cache (%) | Miss rate I-cache (%) | Miss rate L2 cache (%) | AMAT (ns) | CPI |
|---|---|---|---|---|---|
| No cache | 100 | 100 | 100 | 100 | 75.2975 |
| One level cache | 1.76 | 4.89 | 100 | 8.5886 | 4.7937 |
| Two level cache | 1.76 | 4.89 | 98.28 | 8.4852 | 4.7591 |

## 6. Kết luận và hướng phát triển

Đề tài "Design of a Two-level Cache System for RISC-V Architrecture" đã hoàn thành các mục tiêu đặt ra, bao gồm:

- Xây dựng và triển khai thành công CPU RISC-V với tập lệnh RV32I, xử lý hazard và xây dựng bộ Branch prediction two-bit scheme.
- Thiết kế hệ thống bộ nhớ đệm hai cấp (L1 và L2 Cache), tối ưu hóa hiệu năng truy cập dữ liệu.
- Đánh giá hiệu năng thông qua các thông số như tỷ lệ cache hit/miss, AMAT, CPI.

Qua quá trình nghiên cứu và thực hiện, tôi đã hiểu rõ hơn về kiến trúc RISC-V, cơ chế hoạt động của bộ nhớ đệm, và các yếu tố ảnh hưởng đến hiệu năng của CPU. Kết quả đạt được cho thấy hệ thống CPU RISC-V với bộ nhớ đệm hai cấp có thể cải thiện đáng kể tốc độ xử lý và hiệu suất tổng thể, đặc biệt trong các ứng dụng yêu cầu truy cập dữ liệu nhanh.

Tuy nhiên, hệ thống vẫn còn những hạn chế như: tối ưu hóa bộ nhớ đệm chưa đạt mức hiệu quả cao cho các ứng dụng cụ thể, thiết kế và hiệu suất của hệ thống cache có thể chưa được đánh giá toàn diện với lượng testcase đủ lớn để đạt chỉ số corverage mong muốn.

Hướng phát triển:

- Cải tiến cache thành nonblocking cache để tăng hiệu năng cho cache. Khi có yêu cầu truy cập cache mà bị miss, cache ở tầng trên cần có thời gian gửi yêu cầu xuống cache tầng thấp hơn và đợi dữ liệu trả ngược lại, lúc này hệ thống cache sẽ phải tạm dừng (stall) processor cho tới khi hoàn thành việc đọc hay ghi dữ liệu ở cache. Phương pháp giải quyết vấn đề này chính là nonblocking cache hay còn gọi là lockup-free cache, nó giúp processor tiếp tục thực thi câu lệnh tiếp theo trong khi đợi cache xử lý yêu cầu từ câu lệnh trước.
- Thiết kế thêm Victim cache, đây là một kĩ thuật giúp giảm miss rate. Khi cache bị đầy, sẽ có những dữ liệu bị loại bỏ khỏi cache, nhưng những dữ liệu này có thể được sử dụng trong tương lai gần, Victim cache sẽ được sử dụng để lưu tạm thời những dữ liệu này nhằm giảm thời gian trễ và cải thiện hiệu năng.

# TÀI LIỆU THAM KHẢO


[1] Praveena Chauan, Gagandeep Singh, Gurmohan Singh, "Cache Controller for 4-way Set-Associative Cache," *International Journal of Computer Applications (0975 – 8887),* Vols. Volume 129 – No.1, November2015, 2015.

[2] brockboe, JerryAZR, jasonB221, "GitHub," 2021. [Online]. Available: https://github.com/brockboe/RISCV-Processor.

[3] "EduRev," [Online]. Available: https://edurev.in/t/248542/Multilevel-Cache-Organisation.

[4] "geeksforgeeks," 2023. [Online]. Available: https://www.geeksforgeeks.org/multilevel-cache-organisation/.

[5] Eze Val Hyginus Udoka, Eze Martin Chinweokwu, Edozie Enerst, Eze Chidinma Esther, "Design and Development of Effective Multi-Level Cache Memory Model," *International Journal of Recent Technology and Applied Science* , no. p-ISSN: 2721-2017, e-ISSN: 2721-7280, 2023.

[6] Safaa S. Omran, Ibrahim A. Amory, "Implementation of LRU Replacement Policy for Reconfigurable Cache Memory," *International Conference on Advanced Science and Engineering (ICOASE),* 2018.

[7] David A.Patterson, John L.Hennessy, Computer Organization and Design RISC-V Edition: The Hardware Software Interface.

[8] J. Ledin, Modern Computer Architecture, Packt Publishing Ltd, May 2022.