# Chuyên đề 15: Phân tích Chuyên sâu về Kiến trúc Máy tính Tập lệnh Rút gọn (RISC) và So sánh với CISC

## Phần 1: Các Triết lý Kiến trúc CPU Cốt lõi: Định hình Cuộc tranh luận

### 1.1 Giới thiệu hai Xu hướng Thiết kế Vi xử lý

Trong kiến trúc máy tính, hiệu suất của một bộ xử lý trung tâm (CPU) được xác định bởi một phương trình cơ bản. Thời gian thực thi CPU có thể được biểu diễn dưới dạng tích của ba yếu tố chính: (1) số lượng lệnh cho mỗi chương trình, (2) số chu kỳ đồng hồ trung bình cho mỗi lệnh, và (3) thời gian của mỗi chu kỳ đồng hồ.1

$$Thời gian\ CPU = (\frac{Số\ lệnh}{Chương trình}) \times (\frac{Số\ chu\ kỳ}{Lệnh}) \times (\frac{Thời\ gian}{Chu\ kỳ})$$

Trong suốt lịch sử của thiết kế vi xử lý, hai triết lý kiến trúc chủ đạo đã nổi lên, mỗi triết lý đại diện cho một chiến lược đối lập nhằm tối ưu hóa phương trình này để đạt được hiệu suất cao hơn.1 Hai xu hướng này là Máy tính Tập lệnh Phức tạp (CISC) và Máy tính Tập lệnh Rút gọn (RISC).

CISC (Complex Instruction Set Computer) được định nghĩa là một kiến trúc tập lệnh phức tạp.4 Triết lý của nó là cung cấp một bộ lệnh lớn và đa dạng, trong đó nhiều lệnh có khả năng thực thi các tác vụ phức tạp, đa diện (ví dụ: truy cập bộ nhớ, tính toán và lưu trữ kết quả) chỉ trong một bước hoặc một lệnh duy nhất.5

Ngược lại, RISC (Reduced Instruction Set Computer) được định nghĩa là một kiến trúc tập lệnh rút gọn.4 Triết lý của RISC tập trung vào sự đơn giản. Nó sử dụng một bộ lệnh nhỏ, được tối ưu hóa cao, thường có kích thước cố định, được thiết kế để thực thi các lệnh một cách nhanh chóng, lý tưởng là trong một chu kỳ đồng hồ duy nhất.1

### 1.2 Triết lý CISC: Tối ưu hóa cho Lập trình viên và Bộ nhớ

Triết lý CISC, vốn là kiến trúc thống trị ban đầu, nhắm vào yếu tố đầu tiên của phương trình hiệu suất: giảm thiểu số lượng lệnh trên mỗi chương trình (Instructions/Program).1

Chiến lược để đạt được điều này là nhấn mạnh vào phần cứng.4 Các nhà thiết kế CISC đã xây dựng các lệnh phức tạp, cấp cao, có khả năng thực hiện nhiều thao tác cấp thấp (ví dụ: tải dữ liệu từ bộ nhớ, thực hiện một phép tính số học, và lưu kết quả trở lại bộ nhớ) chỉ trong một lệnh duy nhất.1 Ví dụ, trong một hệ thống CISC, có thể có một lệnh ADD duy nhất thực hiện việc cộng hai số được lưu trữ tại hai địa chỉ bộ nhớ khác nhau và lưu kết quả vào một địa chỉ bộ nhớ thứ ba.1

Sự đánh đổi cơ bản của cách tiếp cận này là phải chấp nhận sự gia tăng đáng kể trong yếu tố thứ hai của phương trình: số chu kỳ trung bình trên mỗi lệnh (Cycles/Instruction).1 Bởi vì mỗi lệnh thực hiện rất nhiều công việc, nó đòi hỏi nhiều chu kỳ đồng hồ của CPU để hoàn thành.1

### 1.3 Triết lý RISC: Tối ưu hóa cho Trình biên dịch và Hiệu suất

Triết lý RISC, được phát triển sau để giải quyết các vấn đề của CISC, áp dụng một chiến lược hoàn toàn đối lập. Mục tiêu chính của nó là tấn công yếu tố thứ hai của phương trình hiệu suất: giảm thiểu số chu kỳ trên mỗi lệnh (Cycles/Instruction), với mục tiêu lý tưởng là đạt được một chu kỳ cho mỗi lệnh.1

Chiến lược của RISC là nhấn mạnh vào phần mềm.4 Điều này đạt được bằng cách làm cho phần cứng trở nên đơn giản nhất có thể. Thay vì các lệnh phức tạp, RISC chỉ thực hiện một tập hợp nhỏ các lệnh rất cơ bản.1 Mỗi lệnh được thiết kế để chỉ thực hiện một chức năng duy nhất và đơn giản.9

Sử dụng cùng một ví dụ cộng, để cộng hai số trong bộ nhớ trên máy RISC, lập trình viên (hoặc trình biên dịch) phải viết rõ ràng một chuỗi các lệnh đơn giản:

1. Một lệnh LOAD để tải số đầu tiên từ bộ nhớ vào một thanh ghi.
2. Một lệnh LOAD thứ hai để tải số thứ hai từ bộ nhớ vào một thanh ghi khác.
3. Một lệnh ADD chỉ hoạt động trên các thanh ghi đó.
4. Một lệnh STORE để lưu kết quả từ thanh ghi trở lại bộ nhớ.1

Sự đánh đổi của RISC là sự chấp nhận một sự gia tăng trong yếu tố đầu tiên của phương trình: số lượng lệnh trên mỗi chương trình (Instructions/Program).1

### 1.4 Tóm tắt: Sự phân chia về Nơi đặt Độ phức tạp

Cuộc tranh luận RISC-CISC không chỉ đơn thuần là về kỹ thuật; đó là một cuộc tranh luận triết học sâu sắc về việc *nơi* nên đặt sự phức tạp của một hệ thống máy tính. Độ phức tạp không bị loại bỏ, nó chỉ đơn giản là được di chuyển.

Triết lý CISC 4 cho rằng sự phức tạp nên nằm trong *phần cứng*. Bằng cách tạo ra các lệnh mạnh mẽ 3, gánh nặng được giảm bớt cho các lập trình viên hợp ngữ thời kỳ đầu, những người thấy việc lập trình rất tẻ nhạt và dễ gây lỗi.1 Nó cũng đơn giản hóa công việc cho các nhà thiết kế trình biên dịch ban đầu, vì các cấu trúc ngôn ngữ cấp cao có thể được ánh xạ trực tiếp sang các lệnh phần cứng phức tạp.3

Triết lý RISC 4 đưa ra lập luận hoàn toàn ngược lại: sự phức tạp nên được chuyển sang *phần mềm*. Bằng cách đơn giản hóa triệt để phần cứng 1, các nhà thiết kế RISC đã tạo ra một mục tiêu ổn định, nhanh chóng và có thể dự đoán được. Sau đó, trách nhiệm thực hiện công việc phức tạp được chuyển cho một thành phần mới: "trình biên dịch thông minh".13 Trình biên dịch hiện đại, thay vì phần cứng, chịu trách nhiệm phân tích một tác vụ phức tạp và chia nhỏ nó thành chuỗi tối ưu nhất của các lệnh đơn giản. Do đó, sự phức tạp đã được *chuyển* từ nhà thiết kế CPU (CISC) sang nhà thiết kế trình biên dịch (RISC). Đây là sự đánh đổi cơ bản xác định tất cả các khác biệt kỹ thuật sẽ được thảo luận sau đây.

## Phần 2: Bối cảnh Lịch sử và Nguồn gốc Phát triển

### 2.1 Sự thống trị ban đầu của CISC

Trong những ngày đầu của ngành công nghiệp máy tính, vào những năm 1970, CISC không chỉ là một lựa chọn; nó là kiến trúc tiền thân và thống trị.3 Sự thống trị này không phải là ngẫu nhiên, mà là một giải pháp kỹ thuật trực tiếp cho những hạn chế công nghệ nghiêm ngặt của thời đại đó.

Động lực chính đầu tiên là chi phí và sự khan hiếm của bộ nhớ. Vào thời điểm đó, bộ nhớ RAM (Random Access Memory) cực kỳ đắt đỏ. Ví dụ, vào năm 1977, 1MB DRAM có giá khoảng 5.000 USD.2 Do đó, việc tiết kiệm không gian bộ nhớ là một ưu tiên thiết kế hàng đầu.

Động lực chính thứ hai là trạng thái của phần mềm. Lập trình bằng hợp ngữ (assembly) là một công việc tẻ nhạt, tốn thời gian và cực kỳ dễ gây lỗi.1 Đồng thời, các trình biên dịch cho các Ngôn ngữ Cấp cao (HLLs) vẫn còn ở giai đoạn sơ khai và chưa được tối ưu hóa.3

Kiến trúc CISC đã giải quyết cả hai vấn đề này một cách khéo léo. Bằng cách đóng gói nhiều thao tác vào một lệnh phức tạp duy nhất, các chương trình CISC yêu cầu ít lệnh hơn để hoàn thành một tác vụ. Điều này dẫn đến "kích thước mã nhỏ hơn", hay còn gọi là mật độ mã cao.1 Kích thước mã nhỏ hơn trực tiếp tiết kiệm không gian RAM quý giá. Đồng thời, các lệnh CISC phức tạp này được thiết kế để hỗ trợ trực tiếp các cấu trúc ngôn ngữ cấp cao (như các vòng lặp hoặc các lời gọi hàm phức tạp) 2, làm cho việc viết mã hợp ngữ bằng tay trở nên dễ quản lý hơn và đơn giản hóa công việc cho các trình biên dịch non trẻ.1

Do đó, thiết kế của CISC không phải là một sai lầm hay một thiết kế "kém cỏi". Nó là một giải pháp tối ưu và hợp lý, được điều chỉnh một cách hoàn hảo cho các ràng buộc công nghệ của những năm 1970. Khi bộ nhớ là nút thắt cổ chai chính, việc đầu tư ngân sách bóng bán dẫn (transistor) vào phần cứng giải mã phức tạp 10 để làm cho các lệnh trở nên "đậm đặc" hơn là một quyết định kỹ thuật đúng đắn.

### 2.2 Sự ra đời của RISC: Quan sát Thực nghiệm

Nguồn gốc của triết lý RISC bắt nguồn từ một dự án nghiên cứu mang tính thay đổi cuộc chơi tại IBM vào cuối những năm 1970, được gọi là dự án IBM 801.9 Dự án này được lãnh đạo bởi John Cocke, người sau này đã nhận giải thưởng Turing cho công trình của mình.14

Nhóm của Cocke đã thực hiện một phân tích thực nghiệm chi tiết về mã được tạo ra bởi các trình biên dịch CISC. Phát hiện của họ đã gây chấn động: phần lớn các lệnh CISC phức tạp—chính là những lệnh được thiết kế công phu để hỗ trợ HLLs—lại *hầu như không bao gờ được sử dụng* bởi các trình biên dịch.14 Các trình biên dịch, vốn thấy khó khăn trong việc xác định thời điểm hoàn hảo để sử dụng một lệnh phức tạp, thay vào đó lại ưa thích sử dụng các chuỗi lệnh đơn giản hơn, dễ dự đoán hơn.

Phát hiện này đã dẫn đến câu thần chú sáng lập của triết lý RISC, như được giải thích bởi một nhà nghiên cứu của IBM: "Đừng thêm sự phức tạp [vào phần cứng] trừ khi nó tự trả giá bằng tần suất bạn sử dụng nó".14 Nói cách khác, thà tối ưu hóa cho trường hợp phổ biến (các lệnh đơn giản) còn hơn là làm chậm toàn bộ hệ thống để hỗ trợ các trường hợp đặc biệt, hiếm khi xảy ra (các lệnh phức tạp).

Những khái niệm này nhanh chóng lan sang giới học thuật, nơi chúng được hệ thống hóa và thương mại hóa thông qua hai dự án có ảnh hưởng lớn vào đầu những năm 1980: dự án Berkeley RISC (do David Patterson lãnh đạo, dẫn đến kiến trúc SPARC) và dự án Stanford MIPS (do John Hennessy lãnh đạo, dẫn đến kiến trúc MIPS).9

### 2.3 Sự thay đổi của Thủy triều

Đến cuối những năm 1980 và đầu những năm 1990, bối cảnh công nghệ đã thay đổi hoàn toàn, đảo ngược chính những ràng buộc đã sinh ra CISC.

Đầu tiên, chi phí bộ nhớ đã sụp đổ. "Giá RAM đã giảm đáng kể".2 Đến năm 1994, 1MB DRAM chỉ có giá 6 USD (đã điều chỉnh theo lạm phát), so với 5.000 USD vào năm 1977.2 Đột nhiên, lợi thế chính của CISC—mật độ mã cao để tiết kiệm RAM—trở nên gần như không còn quan trọng.

Thứ hai, "công nghệ trình biên dịch cũng đã trở nên tinh vi hơn".2 Các trình biên dịch thông minh này đã trở nên thành thạo trong việc phân tích mã và tạo ra các chuỗi lệnh được tối ưu hóa cao. Trên thực tế, các trình biên dịch này *không muốn* các lệnh CISC phức tạp, khó đoán; chúng *thích* các lệnh RISC đơn giản, có thể dự đoán được, mà chúng có thể sắp xếp lại và tối ưu hóa một cách tự do.11

Cuối cùng, cuộc đua về tốc độ đã chuyển sang một mặt trận mới: pipelining (thực thi đường ống). Các nhà thiết kế nhận ra rằng để tăng tốc độ xung nhịp và thông lượng, CPU cần phải hoạt động giống như một dây chuyền lắp ráp (assembly line), xử lý nhiều lệnh ở các giai đoạn khác nhau cùng một lúc.15 Và kiến trúc CISC, với các lệnh có độ dài thay đổi và thời gian thực thi không nhất quán, vốn đã "khó" để thực hiện pipelining một cách hiệu quả.1

Kết quả là một sự thay đổi kiến trúc mang tính địa chấn. CISC, với tư cách là con đường dẫn đến hiệu suất cao, đã "cạn kiệt" (ran out of steam).15 Các công ty lớn đã từ bỏ các kiến trúc CISC hàng đầu của họ để chuyển sang RISC: DEC đã từ bỏ VAX (CISC) để chuyển sang Alpha (RISC) 15, và ngay cả IBM, sau dự án 801, cuối cùng cũng đã áp dụng RISC với kiến trúc POWER.9

## Phần 3: Phân tích Kỹ thuật So sánh Chuyên sâu: RISC so với CISC

Những khác biệt triết học giữa RISC và CISC biểu hiện thành những lựa chọn thiết kế kỹ thuật hoàn toàn trái ngược nhau, ảnh hưởng đến mọi thứ, từ định dạng lệnh đến cách CPU tương tác với bộ nhớ.

### 3.1 Thiết kế Tập lệnh: Đơn giản, Cố định so với Phức tạp, Thay đổi

Đây là sự khác biệt cơ bản nhất.

* **RISC:** Sử dụng một tập lệnh nhỏ, đơn giản, và quan trọng nhất là có *độ dài cố định* (fixed-length).1 Hầu hết các lệnh RISC (ví dụ, trong MIPS hoặc ARM 32-bit) đều có cùng kích thước, thường là 32-bit.16
* **CISC:** Sử dụng một tập lệnh lớn, phức tạp, và có *độ dài thay đổi* (variable-length).1 Ví dụ, một lệnh x86 (IA-32) có thể có độ dài bất kỳ từ 1 đến 12 byte.2

Độ dài lệnh cố định của RISC so với độ dài thay đổi của CISC là một trong những khác biệt kỹ thuật quan trọng nhất, vì nó có ảnh hưởng trực tiếp đến hiệu quả giải mã và pipelining. Trong thiết kế RISC, với các lệnh có độ dài cố định 9, phần cứng *luôn* biết nơi bắt đầu của lệnh tiếp theo (ví dụ: chỉ cần cộng 4 byte vào bộ đếm chương trình). Điều này làm cho logic "tìm nạp, giải mã và phát hành... đơn giản hơn đáng kể".9 Ngược lại, trong CISC, phần cứng *không thể* biết lệnh tiếp theo bắt đầu ở đâu cho đến khi nó *giải mã hoàn toàn* lệnh hiện tại để xác định độ dài của nó.2 "Độ dài lệnh thay đổi dẫn đến giải mã và lập lịch không hiệu quả" 2, tạo ra một nút thắt cổ chai cơ bản ngay tại giai đoạn đầu của pipeline mà RISC hoàn toàn tránh được.

### 3.2 Kiến trúc Truy cập Bộ nhớ: "Load-Store" so với "Memory-to-Memory"

Một đặc điểm cốt lõi, gần như là định nghĩa, của kiến trúc RISC là kiến trúc "load-store".9

* **RISC (Load-Store Architecture):** Trong mô hình này, có một sự phân chia nghiêm ngặt. *Chỉ* các lệnh load và store chuyên dụng mới được phép truy cập bộ nhớ ngoài (RAM).9 Tất cả các lệnh tính toán khác—như ADD, SUB, OR, AND—*chỉ* được phép hoạt động trên dữ liệu đã có trong các thanh ghi nội bộ của CPU.9 Do đó, nó còn được gọi là kiến trúc "thanh ghi-thanh ghi" (register-to-register).19
* **CISC (Memory-to-Memory / Register-Memory):** Các bộ xử lý CISC không có sự phân chia này. Các lệnh CISC có thể thực hiện các thao tác kết hợp truy cập bộ nhớ và tính toán trong một bước.1 Ví dụ, một lệnh ADD của CISC có thể lấy một toán hạng từ một thanh ghi và toán hạng kia *trực tiếp từ bộ nhớ* 20, hoặc thậm chí lấy cả hai toán hạng từ bộ nhớ và ghi kết quả trở lại bộ nhớ.10

Kiến trúc load-store không phải là một lựa chọn tùy ý; nó là một hệ quả trực tiếp của triết lý RISC (một chu kỳ mỗi lệnh), được thiết kế đặc biệt để *cô lập* sự chậm chạp và không thể đoán trước của việc truy cập bộ nhớ. Mục tiêu của RISC là làm cho mọi lệnh (hoặc hầu hết) thực thi trong một chu kỳ đồng hồ duy nhất.1 Tuy nhiên, truy cập bộ nhớ (RAM) vốn dĩ *chậm* (chậm hơn nhiều so với tốc độ của CPU) và có thời gian *không thể đoán trước* (ví dụ, một "cache miss" có thể làm CPU dừng lại hàng trăm chu kỳ).9 Nếu một kiến trúc cho phép một lệnh ADD thông thường truy cập bộ nhớ (như CISC), thì thời gian thực thi của lệnh ADD đó sẽ trở nên thay đổi và kéo dài nhiều chu kỳ. Điều này *phá vỡ hoàn toàn* triết lý một chu kỳ của RISC.

Giải pháp của RISC là: "cô lập logic để đối phó với sự chậm trễ... của việc truy cập bộ nhớ... chỉ vào hai lệnh" (load và store).9 Bằng cách này, tất cả các lệnh *khác* (hàng trăm lệnh ALU, logic, dịch chuyển, v.v.) có thể được đảm bảo là đơn giản, nhanh chóng, có thể dự đoán được và thực thi trong một chu kỳ. Đây là một sự đánh đổi thiết kế cơ bản để đơn giản hóa pipeline và đạt được thông lượng cao.9

### 3.3 Quản lý Thanh ghi: Ưu thế về Số lượng của RISC

Sự khác biệt trong kiến trúc truy cập bộ nhớ dẫn trực tiếp đến sự khác biệt trong việc sử dụng thanh ghi.

* **RISC:** Có "nhiều thanh ghi mục đích chung hơn" (more general-purpose registers).1 Các kiến trúc RISC cổ điển thường có 32 thanh ghi hoặc nhiều hơn. Ngân sách bóng bán dẫn (transistor) tiết kiệm được từ việc đơn giản hóa logic điều khiển được sử dụng cho các thanh ghi bộ nhớ này.4
* **CISC:** Có "ít thanh ghi hơn".1 Các kiến trúc CISC lịch sử như x86 ban đầu chỉ có một số lượng nhỏ các thanh ghi chuyên dụng, vì các thao tác có thể được thực hiện trực tiếp trên bộ nhớ, làm giảm nhu cầu về thanh ghi.1

Một tệp thanh ghi lớn (large register file) không chỉ là một tính năng "hay-nên-có" của RISC; nó là một *yêu cầu* tuyệt đối để kiến trúc load-store có thể hoạt động hiệu quả. Kiến trúc load-store 9 *bắt buộc* rằng tất cả các toán hạng phải nằm trong thanh ghi trước khi tính toán. Nếu một CPU chỉ có một vài thanh ghi (như CISC truyền thống) 1, trình biên dịch sẽ liên tục phải thực hiện "tràn thanh ghi" (register spilling)—nghĩa là, liên tục STORE một giá trị ra bộ nhớ chậm chạp chỉ để giải phóng một thanh ghi cho một phép tính mới, và sau đó LOAD một giá trị khác vào. Điều này sẽ làm cho "kích thước mã lớn hơn" 1 của RISC (vốn đã là một nhược điểm) trở nên *tồi tệ hơn* đáng kể, bị thổi phồng bởi vô số lệnh load/store không cần thiết.

Do đó, việc "có nhiều thanh ghi hơn" 1 là *giải pháp* trực tiếp cho "vấn đề" do kiến trúc load-store tạo ra. Nó cung cấp cho trình biên dịch một "không gian làm việc" lớn, tốc độ cao, cho phép nó giữ các giá trị tạm thời và các biến thường dùng trong các thanh ghi cực nhanh mà không cần phải truy cập bộ nhớ chậm chạp.13

### 3.4 Thiết kế Đơn vị Điều khiển: Hardwired (Nối dây cứng) so với Microprogrammed (Vi chương trình/Microcode)

Đơn vị điều khiển (Control Unit) là "bộ não" của CPU, tạo ra các tín hiệu điều khiển nội bộ để thực thi một lệnh. Cách thức mà đơn vị này được thiết kế là một khác biệt quan trọng khác.

* **RISC:** Thường chỉ hoạt động trên Đơn vị Điều khiển Nối dây cứng (Hardwired Control Unit).4 Một đơn vị nối dây cứng sử dụng logic tổ hợp cố định (các cổng logic, flip-flop) để tạo ra các tín hiệu điều khiển trực tiếp từ mã lệnh (opcode).22 Thiết kế này "nhanh hơn" đáng kể nhưng "khó sửa đổi" và trở nên cực kỳ phức tạp nếu tập lệnh trở nên phức tạp.22
* **CISC:** Theo truyền thống, CISC sử dụng Đơn vị Điều khiển Vi chương trình (Microprogrammed Control Unit).4
* **Microcode (Vi mã):** Là một khái niệm thanh lịch của CISC. Thay vì xây dựng một khối logic khổng lồ để giải mã mọi lệnh phức tạp, các nhà thiết kế CISC đã lưu trữ một tập hợp các "vi lệnh" (microinstructions) trong một bộ nhớ trong (Control Memory hoặc ROM).22 Đơn vị điều khiển vi chương trình hoạt động giống như một *trình thông dịch*: nó đọc lệnh CISC phức tạp (ví dụ: CISC\_ADD\_MEM\_TO\_MEM), sau đó thực thi một *chương trình* vi mã nhỏ (ví dụ: vi-lệnh 1: nạp địa chỉ, vi-lệnh 2: đọc bộ nhớ, vi-lệnh 3: đọc bộ nhớ 2, vi-lệnh 4: cộng, vi-lệnh 5: ghi bộ nhớ) để tạo ra các tín hiệu điều khiển cần thiết theo từng chu kỳ.25 Thiết kế này "chậm hơn" (vì nó phải tìm nạp các vi lệnh từ ROM) nhưng "dễ sửa đổi" và dễ dàng xử lý các lệnh cực kỳ phức tạp.22

Sự lựa chọn đơn vị điều khiển là một hệ quả trực tiếp của độ phức tạp tập lệnh. Các lệnh RISC rất đơn giản 1, hầu hết tương ứng với một hành động phần cứng duy nhất. Do đó, việc tạo ra một bộ giải mã logic (hardwired) nhanh, hiệu quả để xử lý chúng là khả thi.22 Ngược lại, các lệnh CISC thì *phức tạp*.1 Cố gắng xây dựng một khối logic nối dây cứng duy nhất để xử lý *tất cả* các lệnh x86 2 sẽ là một "cơn ác mộng" về thiết kế và gỡ lỗi.22 Giải pháp "dễ dàng hơn" của CISC là tạo ra một CPU *bên trong* CPU. Lệnh CISC phức tạp (macro-instruction) được *thông dịch* bởi một chương trình (microcode) chạy trên một bộ vi tuần tự (microsequencer) đơn giản. Đây là một sự đánh đổi cổ điển giữa tốc độ (hardwired) và tính linh hoạt/chi phí thiết kế (microprogrammed).

### 3.5 Hiệu quả của Pipelining: Lý do RISC vượt trội trong Thực thi Đường ống

Pipelining (thực thi đường ống) là một kỹ thuật trong đó CPU hoạt động giống như một dây chuyền lắp ráp công nghiệp, với các lệnh khác nhau ở các giai đoạn thực thi khác nhau (ví dụ: Tìm nạp, Giải mã, Thực thi, Truy cập Bộ nhớ, Ghi lại) cùng một lúc. Đây là chìa khóa để đạt được thông lượng (throughput) cao.

* **RISC:** Kiến trúc RISC "phụ thuộc nhiều vào" và "tối ưu hóa cho" pipelining.1 Việc thực hiện pipelining trên RISC "dễ dàng hơn" đáng kể.1
* *Lý do:* Các đặc điểm cốt lõi của RISC—lệnh đơn giản, có độ dài cố định 9, và thời gian thực thi thống nhất (thường là 1 chu kỳ) 9—cho phép các giai đoạn của pipeline hoạt động như một dây chuyền lắp ráp nhịp nhàng, có thể dự đoán được.18 Mỗi giai đoạn mất một lượng thời gian như nhau, vì vậy không có lệnh nào làm tắc nghẽn đường ống.
* **CISC:** Làm cho pipelining "khó khăn hơn" nhiều.1
* *Lý do:* Các đặc điểm của CISC là kẻ thù của một pipeline hiệu quả: (1) Độ dài lệnh thay đổi 2 làm phức tạp và không thể dự đoán được giai đoạn Tìm nạp/Giải mã. (2) Các chế độ địa chỉ phức tạp 2 làm cho giai đoạn Thực thi/Bộ nhớ trở nên không thể đoán trước. (3) Quan trọng nhất, các lệnh mất số chu kỳ thay đổi (một lệnh có thể mất 2 chu kỳ, lệnh tiếp theo mất 50 chu kỳ) 1 gây ra "bong bóng" (bubbles) và làm đình trệ (stall) toàn bộ pipeline, vì các lệnh nhanh hơn phải chờ lệnh phức tạp, chậm chạp hoàn thành.27

Khả năng pipelining hiệu quả là *lý do chính* khiến "nhược điểm" của RISC (cần nhiều lệnh hơn) không phải là một điểm yếu chí mạng về hiệu suất. Hãy xem xét lại ví dụ ADD 1:

* CISC thực hiện 1 lệnh, nhưng lệnh đó có thể mất 10 chu kỳ.1 Trong 10 chu kỳ đó, pipeline bị đình trệ.27
* RISC thực hiện 4 lệnh (Load, Load, Add, Store), mỗi lệnh 1 chu kỳ.1

Nếu thực thi tuần tự (không có pipeline), RISC thắng (4 chu kỳ so với 10). Nhưng quan trọng hơn là *thông lượng* của pipeline. Trong một pipeline 5 giai đoạn, sau khi pipeline được nạp đầy, CPU RISC có thể *hoàn thành* một lệnh *mỗi chu kỳ*.9 Bốn lệnh này có thể được hoàn thành trong một khoảng thời gian ngắn (ví dụ: 8 chu kỳ để hoàn thành cả 4), nhưng quan trọng là chúng không làm đình trệ các lệnh theo sau chúng. "Dây chuyền lắp ráp" 14 của RISC duy trì một thông lượng cao, ổn định, trong khi "xưởng thủ công" của CISC thực hiện các tác vụ phức tạp nhưng không nhất quán, làm gãy nhịp của pipeline.

### Bảng 1: So sánh Kỹ thuật Toàn diện giữa Kiến trúc RISC và CISC

Bảng dưới đây tóm tắt các khác biệt kỹ thuật cơ bản được thảo luận trong phần này, dựa trên dữ liệu từ nhiều nguồn.1

| **Tiêu chí (Characteristic)** | **RISC (Reduced Instruction Set Computing)** | **CISC (Complex Instruction Set Computing)** |
| --- | --- | --- |
| **Triết lý (Philosophy)** | Nhấn mạnh vào Phần mềm (Emphasis on Software) 4 | Nhấn mạnh vào Phần cứng (Emphasis on Hardware) 4 |
| **Tập lệnh (Instruction Set)** | Nhỏ, đơn giản (Small, simple) 1 | Lớn, phức tạp (Large, complex) 1 |
| **Độ dài lệnh (Instruction Length)** | Cố định (Fixed-length) [1, 9, 16] | Thay đổi (Variable-length) 1 |
| **Truy cập Bộ nhớ (Memory Access)** | Kiến trúc "Load-Store" (Load-Store Architecture) 9 | "Memory-to-Memory" / "Register-Memory" [10, 18, 20] |
| **Đơn vị Điều khiển (Control Unit)** | Nối dây cứng (Hardwired) [4, 12, 22] | Vi chương trình (Microprogrammed) (thường) [4, 22] |
| **Số lượng Thanh ghi (Registers)** | Nhiều thanh ghi mục đích chung (Many GPRs) 1 | Ít thanh ghi hơn (Fewer registers) 1 |
| **Chế độ Địa chỉ (Addressing Modes)** | Đơn giản, ít (Simple, few) 1 | Phức tạp, nhiều (Complex, many) 1 |
| **Chu kỳ/Lệnh (CPI)** | Thấp (Thường 1) (Low (Often 1)) 1 | Cao (Nhiều chu kỳ) (High (Multi-cycle)) 1 |
| **Hiệu quả Pipelining (Pipelining)** | Dễ dàng, hiệu quả cao (Easy, highly efficient) [1, 9, 16] | Khó khăn, phức tạp (Hard, complex) 1 |

## Phần 4: Vai trò then chốt của Trình biên dịch

Sự phân chia triết lý về nơi đặt độ phức tạp (phần cứng so với phần mềm) có nghĩa là vai trò và tầm quan trọng của trình biên dịch là hoàn toàn khác nhau giữa hai kiến trúc.

### 4.1 CISC: Chuyển sự phức tạp vào Phần cứng

Như đã đề cập, mục tiêu ban đầu của CISC là đơn giản hóa công việc của cả lập trình viên hợp ngữ và trình biên dịch.3 Trong mô hình này, phần cứng (hoặc vi mã của nó) đảm nhận gánh nặng của sự phức tạp.

Một câu lệnh Ngôn ngữ Cấp cao (HLL), ví dụ như a = a \* b; trong C, có thể được dịch gần như một-đối-một sang một lệnh hợp ngữ CISC phức tạp duy nhất. Ví dụ, một lệnh MULT 2:3, 5:2 giả định có thể đọc giá trị từ vị trí bộ nhớ 2:3, đọc giá trị từ 5:2, nhân chúng lại, và lưu kết quả trở lại 2:3.3

Trong kịch bản này, trình biên dịch "có rất ít việc phải làm".3 Nó chỉ đơn giản là nhận ra mẫu HLL và phát ra lệnh phần cứng tương ứng. Phần cứng tự nó xử lý tất cả các bước phụ (tải, tính toán, lưu).

### 4.2 RISC: Chuyển sự phức tạp sang Phần mềm

Triết lý RISC, nhấn mạnh vào phần mềm 4, đã tạo ra một backronym (từ viết tắt ngược) nổi tiếng cho RISC: "Relegate Interesting Stuff to the Compiler" (Ủy thác những thứ thú vị cho Trình biên dịch).9

Trong mô hình RISC, trình biên dịch không thể chỉ đơn giản là phát ra một lệnh. Nó phải thực hiện công việc *biên dịch* câu lệnh HLL (a = a \* b) thành một *chuỗi* các lệnh RISC đơn giản, rõ ràng 1, như đã thấy trong ví dụ Load-Load-Add-Store.

Điều này đòi hỏi một loại trình biên dịch hoàn toàn khác: một "Trình biên dịch thông minh" (Smart Compiler).13 Hiệu suất của một hệ thống RISC không chỉ phụ thuộc vào phần cứng; nó phụ thuộc rất nhiều vào khả năng của trình biên dịch trong việc tạo ra mã tối ưu từ các khối xây dựng đơn giản mà nó được cung cấp.

### 4.3 Nhiệm vụ của "Trình biên dịch Thông minh" RISC

"Trình biên dịch thông minh" này phải thực hiện các tác vụ tối ưu hóa phức tạp mà phần cứng CISC (hoặc vi mã của nó) từng tự động xử lý. Hai trong số các nhiệm vụ quan trọng nhất là lập lịch lệnh và phân bổ thanh ghi.

* **Nhiệm vụ 1: Lập lịch lệnh (Instruction Scheduling):** Bởi vì các lệnh RISC rất đơn giản và được thiết kế cho pipeline, trình biên dịch (không phải phần cứng) phải chịu trách nhiệm sắp xếp lại các lệnh để tối ưu hóa dòng chảy của pipeline và tránh các "hazards" (xung đột) làm đình trệ. Ví dụ, thay vì thực hiện hai phép cộng liên tiếp (có thể làm quá tải đơn vị cộng), một trình biên dịch thông minh sẽ xen kẽ một lệnh add rồi đến một lệnh shift (sử dụng đơn vị dịch chuyển) để sử dụng tối đa tất cả các đơn vị thực thi của CPU, giữ cho chúng luôn bận rộn.13
* **Nhiệm vụ 2: Phân bổ Thanh ghi (Register Allocation):** Với 2-4 lần số lượng thanh ghi so với CISC 13 và một kiến trúc load-store yêu cầu mọi thứ phải ở trong thanh ghi, trình biên dịch RISC phải thực hiện một nhiệm vụ quản lý tài nguyên phức tạp. Nó phải phân tích mã để quyết định biến nào nên được giữ trong thanh ghi và trong bao lâu, với mục tiêu cuối cùng là "giảm thiểu việc truy cập bộ nhớ" chậm chạp.13

Kết quả là mã được tạo ra bởi một trình biên dịch RISC được tối ưu hóa tốt có hiệu suất cực kỳ cao, nhưng đối với mắt người, nó "hầu như không thể giải mã".13 Nó ưu tiên hiệu quả của máy hơn là khả năng đọc của con người, hoàn toàn trái ngược với mục tiêu ban đầu của CISC là làm cho hợp ngữ dễ đọc hơn.11

Sự trỗi dậy của RISC không chỉ là một chiến thắng về thiết kế phần cứng; đó là một *sự đặt cược* vào sự trưởng thành của công nghệ trình biên dịch. Các nhà thiết kế RISC ban đầu 14 đã nhận ra rằng các trình biên dịch *hiện có* đã *phớt lờ* các lệnh CISC phức tạp. Họ đưa ra giả thuyết rằng nếu họ cung cấp cho trình biên dịch một mục tiêu đơn giản hơn, dễ dự đoán hơn (ISA RISC), các trình biên dịch có thể được *dạy* để trở nên "thông minh" 13 và tạo ra mã *tốt hơn* so với bất kỳ lệnh CISC phức tạp, được nối cứng nào. Lịch sử đã chứng minh sự đặt cược này là đúng. Các tài liệu 2 và 3 xác nhận rằng khi "Công nghệ trình biên dịch... đã trở nên tinh vi hơn", nó đã làm cho "sự nhấn mạnh vào phần mềm trở nên lý tưởng." Điều này cho thấy thành công của RISC là một câu chuyện về sự *đồng tiến hóa* của thiết kế phần cứng và tối ưu hóa phần mềm.

## Phần 5: Phân tích Hiệu suất và các Yếu tố Cân nhắc

Sự khác biệt về triết lý và kỹ thuật giữa RISC và CISC dẫn đến các cấu hình hiệu suất và các yếu tố cân nhắc thiết kế hoàn toàn khác nhau.

### 5.1 Phương trình Hiệu suất CPU: Chu kỳ, Lệnh và Thời gian

Như đã thiết lập trong Phần 1, cả hai kiến trúc đều cố gắng giảm Thời gian tổng thể.1

* **CISC** tập trung vào việc giảm Số lệnh/Chương trình, chấp nhận Số chu kỳ/Lệnh cao hơn.1
* **RISC** tập trung vào việc giảm Số chu kỳ/Lệnh, chấp nhận Số lệnh/Chương trình cao hơn.1

Trong lịch sử ban đầu, khi các lệnh CISC được vi mã hóa, chúng cực kỳ chậm (CPI cao), khiến cho chiến thắng về pipeline của RISC (CPI thấp) trở nên rõ ràng.

### 5.2 Tiêu thụ Năng lượng: Lợi thế Hiệu quả Rõ rệt của RISC

Một trong những yếu tố cân nhắc quan trọng nhất trong điện toán hiện đại, đặc biệt là trong kỷ nguyên di động, là tiêu thụ năng lượng. Về mặt này, RISC có một lợi thế rõ ràng và mang tính cấu trúc.

* **RISC:** "Tiêu thụ ít năng lượng hơn".4 Đây được coi là "một trong những lợi thế quan trọng" của RISC so với CISC.4 Hiệu quả năng lượng này làm cho RISC trở thành lựa chọn lý tưởng cho các thiết bị di động, chạy bằng pin.4
* **CISC:** "Tiêu thụ nhiều năng lượng hơn".4 Sự tiêu thụ năng lượng cao hơn này được liên kết trực tiếp với "độ phức tạp của tập lệnh phức tạp" của nó.4

Hiệu quả năng lượng vượt trội của RISC là một hệ quả vật lý trực tiếp của triết lý "phần cứng đơn giản". Phần cứng CISC, với bộ giải mã phức tạp, logic điều khiển vi mã, và các đường dẫn dữ liệu cần thiết để xử lý các lệnh nhiều thao tác, đòi hỏi nhiều bóng bán dẫn (transistor) hơn.4 Nhiều bóng bán dẫn hoạt động hơn trong mỗi lệnh phức tạp có nghĩa là tiêu thụ nhiều năng lượng hơn. Ngược lại, "phần cứng đơn giản" của RISC 1 và điều khiển nối dây cứng 4 có nghĩa là ít bóng bán dẫn hơn được kích hoạt cho mỗi lệnh đơn giản.

Ban đầu, hiệu quả năng lượng này có thể chỉ là một *tác dụng phụ* của việc theo đuổi tốc độ (thông qua phần cứng đơn giản và pipelining). Tuy nhiên, nó đã trở thành *lợi thế thị trường chính* của RISC, thúc đẩy sự thống trị hoàn toàn của nó trong thế giới ưu tiên di động, nơi tuổi thọ pin là tối quan trọng.4

### 5.3 Mật độ Mã (Code Density) so với Chi phí Bộ nhớ (Memory Cost): Một yếu tố Cân nhắc đang thay đổi

* **CISC:** Có lợi thế rõ ràng về mật độ mã, tạo ra "kích thước mã nhỏ hơn".1 Như đã thảo luận, đây là một lợi thế quan trọng trong kỷ nguyên mà RAM đắt đỏ.2
* **RISC:** Tạo ra "kích thước mã lớn hơn" 1 vì cần nhiều lệnh đơn giản hơn để thực hiện cùng một tác vụ.1

Sự sụp đổ của "giá RAM" 2 đã làm cho lợi thế này của CISC phần lớn bị vô hiệu hóa trong các máy tính để bàn và máy chủ. Tuy nhiên, cuộc tranh luận về mật độ mã không hoàn toàn biến mất. Kích thước mã lớn của RISC vẫn là một nhược điểm thực sự trong hai lĩnh vực: (1) các hệ thống nhúng (embedded systems) bị giới hạn bộ nhớ, nơi mỗi byte của ROM đều có giá trị, và (2) hiệu suất của bộ đệm lệnh (instruction cache / I-cache). Một chương trình RISC lớn hơn sẽ gây áp lực lớn hơn lên I-cache, dẫn đến nhiều cache miss hơn.

Điều này đã dẫn đến một sự hội tụ thú vị. Nhận thức được điều này, các kiến trúc RISC hiện đại đã bắt đầu áp dụng các kỹ thuật để cải thiện mật độ mã. Ví dụ, đặc tả RISC-V bao gồm một phần mở rộng "C" (Compressed) tiêu chuẩn 29, cung cấp các lệnh 16-bit (2 byte) bên cạnh các lệnh 32-bit (4 byte) tiêu chuẩn. Điều này, một cách trớ trêu, đã tái giới thiệu một dạng lệnh có độ dài thay đổi—một trong những đặc điểm của CISC—trở lại RISC, tất cả đều nhân danh mật độ mã.11

### 5.4 Độ phức tạp của Thiết kế Phần cứng, Rủi ro và Thời gian ra thị trường

* **CISC:** "Độ phức tạp trong thiết kế" là một nhược điểm đáng kể.4 Việc thiết kế, triển khai và gỡ lỗi một bộ xử lý CISC phức tạp có "rủi ro cao" và "mất nhiều thời gian hơn để thiết kế".10
* **RISC:** Ngược lại, "kiến trúc đơn giản và dễ thực hiện" của RISC 10 mang lại lợi thế thương mại rõ ràng. Nó dẫn đến "thời gian ra thị trường ngắn" và "rủi ro thấp hơn".10 Sự đơn giản này cũng cho phép các nhóm thiết kế nhỏ hơn tạo ra các lõi CPU, một yếu tố đã góp phần vào sự phổ biến của các kiến trúc như ARM và sự trỗi dậy của RISC-V.

## Phần 6: Các Họ Vi xử lý Tiêu biểu và Lĩnh vực Ứng dụng

Sự phân chia triết lý và kỹ thuật đã dẫn đến một sự phân chia thị trường rõ rệt, mặc dù sự phân chia này hiện đang bị thách thức.

### 6.1 Di sản của CISC: Intel x86, AMD64 và Sự thống trị trong PC và Máy chủ

* **Ví dụ:** Các ví dụ hiện đại và thống trị nhất của CISC là kiến trúc x86, được tiên phong bởi Intel và cũng được AMD sử dụng (với phần mở rộng AMD64).1
* **Ví dụ Lịch sử:** Các gia đình CISC quan trọng khác trong lịch sử bao gồm Motorola 68k (được sử dụng trong Apple Macintosh và Amiga thời kỳ đầu), DEC VAX và PDP-11.4
* **Lĩnh vực Ứng dụng:** Ngày nay, CISC (x86) thống trị thị trường máy tính cá nhân (PC), máy tính xách tay và máy chủ doanh nghiệp.4
* **Lý do:** Sự thống trị này ít liên quan đến lợi thế kỹ thuật hiện tại mà liên quan nhiều hơn đến lợi thế lịch sử và "tương thích ngược" (backward compatibility).8 Hệ sinh thái phần mềm khổng lồ được xây dựng xung quanh Windows và x86 3 đã tạo ra một rào cản khổng lồ đối với việc gia nhập, vì người dùng mong đợi phần mềm cũ của họ vẫn chạy trên các máy mới.

### 6.2 Sự trỗi dậy của RISC: ARM, RISC-V và Sự thống trị trong Di động và Nhúng

* **Ví dụ:** Kiến trúc RISC phổ biến nhất hiện nay là ARM.1 Các ví dụ quan trọng khác bao gồm MIPS (phổ biến trong các bộ định tuyến mạng), RISC-V (một kiến trúc mã nguồn mở đang phát triển nhanh chóng), PowerPC, SPARC, và DEC Alpha.1
* **Lĩnh vực Ứng dụng:** RISC hoàn toàn thống trị trong các thiết bị di động (điện thoại thông minh, máy tính bảng), các hệ thống nhúng (từ vi điều khiển trong lò vi sóng đến hệ thống điều khiển ô tô) và thị trường Internet of Things (IoT).4
* **Lý do:** Lợi thế kỹ thuật cốt lõi của RISC trong các lĩnh vực này là *tiêu thụ điện năng thấp*.4
* **Ứng dụng Mới nổi:** RISC không còn chỉ giới hạn ở các thiết bị công suất thấp. Nó đang tích cực thâm nhập vào các thành trì truyền thống của CISC, bao gồm máy tính hiệu năng cao (ví dụ: các chip M1/M2/M3 của Apple, dựa trên ARM) 8 và các trung tâm dữ liệu đang tìm kiếm hiệu quả năng lượng tốt hơn.29

Sự phân chia thị trường hiện tại (x86 trong PC, ARM trong Di động) là kết quả của chiến lược kinh doanh và thời điểm lịch sử cũng như là về kỹ thuật. Như tài liệu 3 giải thích, RISC ban đầu thất bại trong việc thâm nhập thị trường PC do "thiếu hỗ trợ phần mềm" (cụ thể là Windows) và sự thống trị về sản xuất của Intel. x86 giành chiến thắng trong cuộc chiến PC nhờ *hệ sinh thái* và *tương thích ngược*.8 Trong khi đó, tài liệu 28 lưu ý "sự coi thường hoàn toàn của Intel đối với các nền tảng di động". Điều này tạo ra một *khoảng trống* thị trường khổng lồ khi điện thoại thông minh xuất hiện. Kiến trúc RISC, với lợi thế *tiêu thụ điện năng thấp* 4, và mô hình kinh doanh *cấp phép* (licensing) của ARM (cho phép các công ty như Apple, Qualcomm và Samsung thiết kế chip của riêng họ) 28, là sự phù hợp hoàn hảo cho thị trường di động mới nổi.

Sự trỗi dậy của RISC-V 28 đại diện cho một mặt trận thứ ba, được thúc đẩy không phải bởi kỹ thuật hay hiệu quả năng lượng, mà bởi một mô hình kinh doanh *mã nguồn mở*. x86 là độc quyền.28 ARM là độc quyền (yêu cầu phí cấp phép).28 Ngược lại, RISC-V là "Mã nguồn mở" và "miễn phí bản quyền".28 Điều này cho phép bất kỳ ai, từ một sinh viên đến một nhà sản xuất vi điều khiển 10 cent 29 hay một tập đoàn siêu máy tính châu Âu 29, được tự do sử dụng, sửa đổi và tùy chỉnh ISA mà không cần trả phí bản quyền. Điều này cho thấy cuộc chiến trong tương lai không chỉ là RISC vs. CISC, mà còn là Độc quyền vs. Mã nguồn mở.

## Phần 7: Sự hội tụ Hiện đại và Tương lai của Thiết kế CPU

Cuộc tranh luận "RISC vs. CISC" cổ điển, vốn rất gay gắt trong những năm 1980 và 1990, phần lớn đã trở nên lỗi thời trong bối cảnh hiện đại. Trên thực tế, các thuật ngữ này đang trở thành một "lăng kính sai lầm" để so sánh các CPU hiện đại.11

### 7.1 Sự "mờ nhạt" của ranh giới

"Ranh giới đã bị mờ" 30 bởi vì cả hai bên đã áp dụng các đặc điểm của nhau để giải quyết những điểm yếu của mình.

* **RISC không còn "rút gọn":** Một "hiểu lầm phổ biến" là RISC có nghĩa là ít lệnh hơn. Trên thực tế, "ngày nay nhiều tập lệnh RISC có bộ lệnh lớn hơn nhiều CPU CISC".9 Các kiến trúc RISC hiện đại như ARM đã phát triển để bao gồm hàng trăm lệnh, bao gồm các phần mở rộng phức tạp cho xử lý vector (ví dụ: NEON) và các lệnh chuyên dụng khác.21
* **CISC không còn "phức tạp" (bên trong):** Như sẽ được phân tích dưới đây, các CPU CISC hiện đại không còn thực thi các lệnh phức tạp của chúng một cách trực tiếp. Chúng đã áp dụng một lõi giống như RISC.

"RISC" và "CISC" ngày nay mô tả chính xác hơn là *di sản* ISA (ISA heritage) hoặc *ISA có thể thấy được* (visible ISA) mà các lập trình viên và trình biên dịch nhắm đến, chứ không phải là *vi kiến trúc* (microarchitecture) bên trong thực thi mã đó. Bên trong, tất cả các chip hiệu suất cao hiện đại đều là "con lai" (hybrid). Có lẽ đặc điểm xác định thực sự duy nhất còn lại của một ISA RISC thuần túy là kiến trúc load-store nghiêm ngặt của nó.9

### 7.2 Phân tích "RISC bên trong CISC": Vai trò của Micro-operations (µops)

Sự phát triển quan trọng nhất làm mờ ranh giới là việc các bộ xử lý CISC (x86) hiện đại áp dụng các nguyên tắc của RISC. Các CPU CISC hiện đại "dịch các lệnh thành các vi hoạt động (micro-operations) đơn giản hơn bên trong".30 Đây là ý tưởng "RISC bên trong CISC".32

Thay vì cố gắng thực thi một lệnh x86 phức tạp, có độ dài thay đổi trực tiếp, CPU x86 hiện đại có một "bộ dịch" (translator) hoặc giai đoạn "crack" 32 ở phía trước (frontend). Giai đoạn này lấy lệnh CISC phức tạp và chia nó thành một "tập hợp các lệnh RISC tương đương".32 Những lệnh nội bộ, đơn giản, giống RISC này được gọi là **micro-operations** (hoặc µops).

Những µops này sau đó được đưa vào một lõi thực thi (execution core) hiệu suất cao, có đường ống (pipelined), giống như RISC, với nhiều thanh ghi và thực thi ngoài trật tự (out-of-order execution).15

Quá trình chuyển đổi này bắt đầu một cách nghiêm túc với "Pentium Pro vào năm 1995" 15, vốn về cơ bản là "một CPU RISC với một bộ giải mã CISC được gắn vào".15

Mô hình hybrid này mang lại cho CISC (x86) *điều tốt nhất của cả hai thế giới*:

1. **Tương thích Ngược (CISC):** Nó có thể chạy hàng thập kỷ phần mềm kế thừa được viết cho ISA x86.
2. **Hiệu suất Cao (RISC):** Nó được hưởng lợi từ tất cả các lợi thế về hiệu suất của một lõi thực thi có đường ống, giống như RISC.

Đây là một kỹ thuật tuyệt vời, nhưng nó có giá của nó. Bộ giải mã x86 để thực hiện việc "crack" này cực kỳ phức tạp, tiêu tốn một phần đáng kể năng lượng của chip 29, và là một nút thắt cổ chai cố hữu so với bộ giải mã RISC vốn đã đơn giản.

### 7.3 Sự khác biệt then chốt: Microcode (Vi mã) so với Micro-operations (Vi hoạt động)

Điều quan trọng là phải phân biệt giữa khái niệm CISC "cũ" (microcode) và khái niệm hybrid "hiện đại" (micro-operations).

* **Microcode (Vi mã):** Như đã thảo luận trong Phần 3.4, đây là *phần mềm* (firmware) được lưu trữ trong ROM.26 Đơn vị điều khiển hoạt động như một *trình thông dịch* (interpreter) để thực thi vi mã này. Đây là cách làm "cũ", chậm chạp, dựa trên trình thông dịch.22
* **Micro-operations (µops):** Đây là các lệnh cấp thấp, giống RISC.35 Chúng là *đầu ra* (output) của một bộ giải mã *nối dây cứng* (hardwired "crack" decoder).32 Chúng được *dịch* (translated), không phải được *thông dịch*, và được đưa trực tiếp vào pipeline phần cứng để thực thi.25

Các CPU x86 hiện đại thực sự sử dụng *cả hai*:

1. Đối với các lệnh *phổ biến*, đơn giản (ví dụ: ADD, MOV, JMP), chúng sử dụng một bộ giải mã nối dây cứng nhanh để tạo ra µops (thường là 1:1 hoặc 1:vài).36
2. Đối với các lệnh *hiếm, rất phức tạp* (ví dụ: các lệnh string, các hàm toán học phức tạp), bộ giải mã nối dây cứng sẽ từ bỏ và "quay trở lại... ROM vi mã" để thực thi một chương trình vi mã chậm hơn.36

Điều thú vị là, các CPU RISC (như ARM) *cũng* sử dụng µops 32, nhưng chúng *không* sử dụng microcode (trình thông dịch ROM).32 Đối với RISC, việc dịch từ lệnh ISA sang µop thường là "1:1".25 µop chỉ đơn giản là một định dạng *bên trong* "rộng hơn và đơn giản hơn" giúp đơn giản hóa các giai đoạn sau của pipeline.25 Đối với các kiến trúc như RISC-V, người ta cho rằng các lệnh ISA "chính là các vi hoạt động" (are the micro-ops).29

### 7.4 Các khái niệm liên quan: Mô hình Harvard và Bộ đệm L1 Phân tách (I-Cache/D-Cache)

Một khái niệm kiến trúc thường liên quan đến các thiết kế RISC là Mô hình Harvard.

* **Mô hình Harvard:** Là một thiết kế máy tính có *bộ nhớ và bus riêng biệt* cho Lệnh (instructions) và Dữ liệu (data).38
* **Mô hình Von Neumann:** Ngược lại, đây là mô hình truyền thống nơi Lệnh và Dữ liệu chia sẻ cùng một bộ nhớ và một bus duy nhất.
* **Lợi ích của Harvard:** Mô hình Harvard cho phép CPU "vừa đọc một lệnh vừa thực hiện truy cập bộ nhớ dữ liệu cùng một lúc".38 Điều này tránh được "nút thắt cổ chai Von Neumann" 39, nơi CPU phải chờ đợi vì nó không thể tìm nạp lệnh và tìm nạp dữ liệu cùng một lúc.

Các thiết kế RISC "có nhiều khả năng" sử dụng mô hình Harvard 2, mặc dù đây không phải là một yêu cầu bắt buộc (các CPU ARM cũ hơn đã sử dụng Von Neumann).40 Lý do của sự liên kết này là vì mô hình Harvard là một *yếu tố hỗ trợ quan trọng* cho pipeline thông lượng cao mà triết lý RISC yêu cầu.

Hãy xem xét một pipeline RISC.16 Nó cần thực hiện nhiều việc cùng một lúc. Giai đoạn "Tìm nạp Lệnh" (Instruction Fetch - IF) và giai đoạn "Truy cập Bộ nhớ/Dữ liệu" (Memory Access - MEM) là hai giai đoạn quan trọng.

* Trong một hệ thống Von Neumann thuần túy (bus chung), các giai đoạn IF và MEM sẽ *xung đột* (contend) cho cùng một bus.39 CPU không thể tìm nạp lệnh *tiếp theo* (giai đoạn IF) trong khi nó đang đọc/ghi dữ liệu cho lệnh *hiện tại* (giai đoạn MEM).
* Điều này sẽ tạo ra một "xung đột cấu trúc" (structural hazard) và làm *đình trệ* pipeline mỗi chu kỳ, phá hủy mục tiêu một lệnh mỗi chu kỳ (CPI=1).

**Biểu hiện Hiện đại (Kiến trúc Harvard Sửa đổi):** Hầu hết các CPU hiệu suất cao hiện đại (cả RISC và CISC) đều sử dụng *Kiến trúc Harvard Sửa đổi* (Modified Harvard Architecture). Trong mô hình này, Bộ đệm L1 (L1 cache) được *phân tách* thành một Bộ đệm Lệnh (Instruction Cache / I-cache) và một Bộ đệm Dữ liệu (Data Cache / D-cache).2 Phần còn lại của hệ thống bộ nhớ (L2, L3, RAM) được thống nhất (vẫn là Von Neumann).38

Bằng cách phân tách L1 43, giai đoạn IF có thể truy cập I-cache 42 trong khi giai đoạn MEM truy cập D-cache 42 *trong cùng một chu kỳ đồng hồ*.45 Quyền truy cập song song (parallel access) 39 này là *cần thiết* để đạt được thông lượng "một lệnh mỗi chu kỳ" 9 mà toàn bộ triết lý RISC hướng tới.

### 7.5 Kết luận: Cuộc tranh luận về ISA và Tương lai của Điện toán

Cuộc chiến triết học giữa RISC và CISC, vốn được tranh luận gay gắt trong nhiều thập kỷ, đã đi đến một kết luận rõ ràng: các nguyên tắc cốt lõi của RISC đã *thắng lợi*.

Các khái niệm cơ bản được RISC tiên phong—một ISA load-store đơn giản, thực thi có đường ống sâu, điều khiển nối dây cứng, sự phụ thuộc nặng nề vào trình biên dịch thông minh, và một vi kiến trúc bên trong giống RISC—đã được chứng minh là con đường vượt trội để đạt được hiệu suất cao. Tất cả các CPU hiệu suất cao hiện đại, bất kể ISA bên ngoài của chúng là gì, đều được xây dựng trên các nguyên tắc này.15 Kiến trúc x86 của Intel và AMD đã sống sót và phát triển thịnh vượng không phải bằng cách bác bỏ RISC, mà bằng cách *hấp thụ* nó, giấu nó bên dưới một lớp dịch để duy trì tương thích ngược.

Cuộc chiến *thương mại* và *thị trường* (x86 vs. ARM vs. RISC-V) vẫn tiếp diễn. RISC đã "chiến thắng một cách dứt khoát" ở mọi thị trường không yêu cầu hỗ trợ kế thừa của Windows.29 x86, mặc dù có ISA "tồi tệ" 29, vẫn duy trì hiệu suất cạnh tranh nhờ "nhiều tỷ đô la" đầu tư vào kỹ thuật vi kiến trúc. Tuy nhiên, nó phải trả giá bằng hiệu quả năng lượng; sự phức tạp của bộ giải mã x86 có nghĩa là nó "kém hiệu quả về năng lượng" hơn ARM hoặc RISC-V, và đây là một bất lợi "vốn có" 29 trong một thế giới ngày càng quan tâm đến năng lượng.

Tương lai của thiết kế CPU sẽ không còn được định hình bởi cuộc tranh luận "RISC vs. CISC" cũ kỹ. Thay vào đó, nó sẽ được định hình bởi các trận chiến về *hệ sinh thái kế thừa* (x86), *hiệu quả năng lượng* và *mô hình cấp phép* (ARM), và *sự tự do của mô hình kinh doanh mở* (RISC-V).

#### Nguồn trích dẫn

1. RISC vs CISC - GeeksforGeeks, truy cập vào tháng 11 5, 2025, <https://www.geeksforgeeks.org/computer-organization-architecture/computer-organization-risc-and-cisc/>
2. RISC AND CISC - arXiv, truy cập vào tháng 11 5, 2025, <https://arxiv.org/pdf/1101.5364>
3. RISC vs. CISC - CS Stanford, truy cập vào tháng 11 5, 2025, <https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/>
4. RISC vs CISC Architecture: Key Differences Explained 2025, truy cập vào tháng 11 5, 2025, <https://www.stromasys.com/resources/decoding-risc-vs-cisc-architecture/>
5. Comparing RISC and CISC in CPU Architecture - Total Phase, truy cập vào tháng 11 5, 2025, <https://www.totalphase.com/blog/2024/08/comparing-risc-and-cisc-in-cpu-architecture/>
6. What Is RISC and CISC Architecture & Their Differences | Glossary | Conclusive Engineering, truy cập vào tháng 11 5, 2025, <https://conclusive.tech/glossary/what-is-risc-and-cisc-architecture-their-differences/>
7. truy cập vào tháng 11 5, 2025, <https://www.totalphase.com/blog/2024/08/comparing-risc-and-cisc-in-cpu-architecture/#:~:text=RISC%20focuses%20on%20simplicity%2C%20and,multifaceted%20tasks%20in%20fewer%20steps.>
8. CISC vs. RISC: The Differences Between x86 and ARM ... - Medium, truy cập vào tháng 11 5, 2025, <https://medium.com/@samah.ikramfarez/cisc-vs-risc-the-differences-between-x86-and-arm-architectures-639a64ad7c76>
9. Reduced instruction set computer - Wikipedia, truy cập vào tháng 11 5, 2025, <https://en.wikipedia.org/wiki/Reduced_instruction_set_computer>
10. RISC vs. CISC | Baeldung on Computer Science, truy cập vào tháng 11 5, 2025, <https://www.baeldung.com/cs/risc-vs-cisc>
11. RISC vs CISC Microprocessor Philosophy in 2022 : r/programming - Reddit, truy cập vào tháng 11 5, 2025, <https://www.reddit.com/r/programming/comments/uigvtg/risc_vs_cisc_microprocessor_philosophy_in_2022/>
12. RISC vs. CISC Processors - Microsystems, truy cập vào tháng 11 5, 2025, <https://microsystems.electricalandcontrol.com/risc-vs-cisc-processors/>
13. performance - Why do compilers have to be "smarter" for RISC ..., truy cập vào tháng 11 5, 2025, <https://superuser.com/questions/725892/why-do-compilers-have-to-be-smarter-for-risc-instruction-set-architecture>
14. Reduced instruction set computer (RISC) architecture - IBM, truy cập vào tháng 11 5, 2025, <https://www.ibm.com/history/risc>
15. What caused/started the "CISC vs RISC" in the 1980s? : r/hardware - Reddit, truy cập vào tháng 11 5, 2025, <https://www.reddit.com/r/hardware/comments/z61995/what_causedstarted_the_cisc_vs_risc_in_the_1980s/>
16. truy cập vào tháng 11 5, 2025, <https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/pipelining/index.html#:~:text=Because%20RISC%20instructions%20are%20simpler,fetched%20in%20a%20single%20operation.>
17. RISC vs CISC: What's the Difference? - Patsnap Eureka, truy cập vào tháng 11 5, 2025, <https://eureka.patsnap.com/blog/risc-vs-cisc/>
18. Dissecting Windows Malware Series – RISC vs CISC Architectures ..., truy cập vào tháng 11 5, 2025, <https://8ksec.io/dissecting-windows-malware-series-risc-vs-cisc-architectures-part-4/>
19. Load–store architecture - Wikipedia, truy cập vào tháng 11 5, 2025, <https://en.wikipedia.org/wiki/Load%E2%80%93store_architecture>
20. Register-register vs register-memory - Stack Overflow, truy cập vào tháng 11 5, 2025, <https://stackoverflow.com/questions/66699334/register-register-vs-register-memory>
21. ARM vs RISC and x86 vs CISC - Stack Overflow, truy cập vào tháng 11 5, 2025, <https://stackoverflow.com/questions/72962725/arm-vs-risc-and-x86-vs-cisc>
22. Hardwired Vs. Microprogrammed Control Unit: Differences & Examples - Unstop, truy cập vào tháng 11 5, 2025, <https://unstop.com/blog/differences-between-hardwired-control-unit-and-microprogrammed-control-unit>
23. Difference between Hardwired and Micro-programmed Control Unit | Set 2 - GeeksforGeeks, truy cập vào tháng 11 5, 2025, <https://www.geeksforgeeks.org/computer-organization-architecture/difference-between-hardwired-and-micro-programmed-control-unit-set-2/>
24. CPUs from the past versus now (question about hardwired control unit vs microprogrammed control unit) : r/beneater - Reddit, truy cập vào tháng 11 5, 2025, <https://www.reddit.com/r/beneater/comments/1ntqujo/cpus_from_the_past_versus_now_question_about/>
25. Do microcode and micro-operation mean the same? : r/arm - Reddit, truy cập vào tháng 11 5, 2025, <https://www.reddit.com/r/arm/comments/oi4dk1/do_microcode_and_microoperation_mean_the_same/>
26. What is microcode? How does it work? Any articles? : r/arm - Reddit, truy cập vào tháng 11 5, 2025, <https://www.reddit.com/r/arm/comments/pa9ojc/what_is_microcode_how_does_it_work_any_articles/>
27. Why is RISC architecture better suited for pipeline processing than CISC? - Quora, truy cập vào tháng 11 5, 2025, <https://www.quora.com/Why-is-RISC-architecture-better-suited-for-pipeline-processing-than-CISC>
28. What are the reasons for the transition from CISC (x86) to RISC-V processors? Is there a significant difference between the two? - Quora, truy cập vào tháng 11 5, 2025, <https://www.quora.com/What-are-the-reasons-for-the-transition-from-CISC-x86-to-RISC-V-processors-Is-there-a-significant-difference-between-the-two>
29. Is there any differnce in instruction pipelining of an risc and cisc processor contributing to better effecient processor than cisc? : r/RISCV - Reddit, truy cập vào tháng 11 5, 2025, <https://www.reddit.com/r/RISCV/comments/1cz4aqi/is_there_any_differnce_in_instruction_pipelining/>
30. RISC vs CISC – Clayton Cafiero, truy cập vào tháng 11 5, 2025, <https://www.uvm.edu/~cbcafier/cs2210/content/02_basics_of_architecture/risc_vs_cisc.html>
31. Is RISC vs. CISC distinction without a difference? : r/hardware - Reddit, truy cập vào tháng 11 5, 2025, <https://www.reddit.com/r/hardware/comments/uk6us8/is_risc_vs_cisc_distinction_without_a_difference/>
32. arm - Difference between Micro-Operations in RISC and CISC ..., truy cập vào tháng 11 5, 2025, <https://electronics.stackexchange.com/questions/188268/difference-between-micro-operations-in-risc-and-cisc-processors>
33. What exactly is microcode and how does it differ from firmware? - Super User, truy cập vào tháng 11 5, 2025, <https://superuser.com/questions/1283788/what-exactly-is-microcode-and-how-does-it-differ-from-firmware>
34. Is the concept of microcode incompatible with an RISC design? Why is it inherently slower and only suited to CISC? - Quora, truy cập vào tháng 11 5, 2025, <https://www.quora.com/Is-the-concept-of-microcode-incompatible-with-an-RISC-design-Why-is-it-inherently-slower-and-only-suited-to-CISC>
35. Difference between an instruction and a micro-op - Stack Overflow, truy cập vào tháng 11 5, 2025, <https://stackoverflow.com/questions/33026830/difference-between-an-instruction-and-a-micro-op>
36. Microcode - Wikipedia, truy cập vào tháng 11 5, 2025, <https://en.wikipedia.org/wiki/Microcode>
37. Does ARM have no microcode updates? : r/arm - Reddit, truy cập vào tháng 11 5, 2025, <https://www.reddit.com/r/arm/comments/1nx8ke7/does_arm_have_no_microcode_updates/>
38. Harvard architecture - Wikipedia, truy cập vào tháng 11 5, 2025, <https://en.wikipedia.org/wiki/Harvard_architecture>
39. Harvard Architecture - GeeksforGeeks, truy cập vào tháng 11 5, 2025, <https://www.geeksforgeeks.org/computer-organization-architecture/harvard-architecture/>
40. CISC and RISC architectures - Stack Overflow, truy cập vào tháng 11 5, 2025, <https://stackoverflow.com/questions/35226663/cisc-and-risc-architectures>
41. The RISC-V Processor - CS@Cornell, truy cập vào tháng 11 5, 2025, <https://www.cs.cornell.edu/courses/cs3410/2019sp/schedule/slides/06-cpu-pre-bw.pdf>
42. Cache Memory Performance - GeeksforGeeks, truy cập vào tháng 11 5, 2025, <https://www.geeksforgeeks.org/computer-organization-architecture/cache-memory-performance/>
43. What does a 'Split' cache means. And how is it useful(if it is)? - Stack Overflow, truy cập vào tháng 11 5, 2025, <https://stackoverflow.com/questions/55752699/what-does-a-split-cache-means-and-how-is-it-usefulif-it-is>
44. CPU cache - Wikipedia, truy cập vào tháng 11 5, 2025, <https://en.wikipedia.org/wiki/CPU_cache>
45. why are separate icache and dcache needed [duplicate] - Stack Overflow, truy cập vào tháng 11 5, 2025, <https://stackoverflow.com/questions/8707041/why-are-separate-icache-and-dcache-needed>