Cơ bản về lập trình nhúng

Biểu diễn số và dữ liệu

- Đơn vị cơ bản nhất trong biểu diễn thông tin của hệ thống số được gọi là *bit*, chính là ký hiệu viết tắt của thuật ngữ *binary digit*.
- Năm 1964, IBM đã thiết kế và chế tạo máy tính số sử dụng một nhóm 8 bit để đánh địa chỉ bộ nhớ và định nghĩa ra thuật ngữ *8 bit = 1 byte*.
- Ngày nay sử dụng rộng rãi thuật ngữ *word* là một từ dữ liệu dùng để biểu diễn kích thước dữ liệu mà được xử lý một cách hiệu quả nhất đối với mỗi loại kiến trúc xử lý số cụ thể. Chính vì vậy một từ có thể là 16 bits, 32 bits, hoặc 64 bits…
- Mỗi một byte có thể được chia ra thành hai nửa 4 bit và được gọi là các *nibble*. *Nibble* chứa các bit trọng số lớn được gọi là *nibble* bậc cao, và *nibble* chứa các bit trọng số nhỏ được gọi là *nibble* bậc thấp.

Các hệ thống cơ số

Trong các hệ thống biểu diễn số hiện nay đều được biểu diễn ở dạng tổng quát là tổng luỹ thừa theo cơ số, và được phân loại theo giá trị cơ số. Một cách tổng quát một hệ biểu diễn số cơ số b và a được biểu diễn như sau:

A=a**n**b**n**+ a**n-1**b**n-1**+…+ $a_0 = \sum_{i=0}^{n} ai * bi$

Ví dụ: như cơ số *binary* (nhị phân), cơ số *decimal* (thập phân, cơ số 8 *Octal* (bát phân).

Ví dụ về biểu diễn các giá trị trong các hệ cơ số khác nhau:

- 243.51**10** = 2 x 10**2** + 4 x 10**1** + 3 x 10**0** + 5 x 10**-1** + 1 x 10**-2**
- 212**3** = 2 x 3**2** + 1 x 3**1** + 2 x 3**0**
- 10110**2** = 1 x 2**4** + 0 x 2**3** + 1 x 2**2** + 1 x 2**1** + 0 x 2**0** = 22**10**

Hai loại cơ số biểu diễn thông dụng nhất hiện nay cho các hệ thống xử lý số là *cơ số nhị phân* và *cơ số mười sáu*.

Số nguyên

Trong biểu diễn số có dấu để phân biệt số dương và số âm người ta sử dụng bit trọng số lớn nhất qui ước làm bit dấu và các bit còn lại được sử dụng để biểu diễn giá trị độ lớn của số. Ví dụ một từ 8 bit được sử dụng để biểu diễn giá trị -1 sẽ có dạng nhị phân là 10000001, và giá trị +1 sẽ có dạng 00000001. Như vậy với một từ 8 bit có thể biểu diễn

được các số trong phạm vi từ -127 đến +127. Một cách tổng quát một từ *N* bit sẽ biểu diễn được *-2(N-1)-1* đến *+2(N-1)-1*.

Chú ý khi thực hiện cộng hai số có dấu:

- Nếu hai số cùng dấu thì thực hiện phép cộng phần biểu diễn giá trị và sử dụng bit dấu cùng dấu với hai số đó.
- Nếu hai số khác dấu thì kết quả sẽ nhận dấu của toán tử lớn hơn, và thực hiện phép trừ giữa toán tử có giá trị lớn hơn với toán tử bé hơn.

Ví dụ 1: Cộng hai số có dấu 010011111**2** và 001000111**2**.

1 1 1 1 ⇐ carries (Số nhớ)

0 1 0 0 1 1 1 1 (79)

0 + 0 1 0 0 0 1 1 + (35)

0 1 1 1 0 0 1 0 (114)

Ví dụ 2: Trừ hai số có dấu 010011111**2** và 011000111**2**.

0 1 1 2 ⇐ borrows (số vay)

0 1 1 0 0 0 1 1 (99)

0 - 1 0 0 1 1 1 1 - (79)

0 0 0 1 0 1 0 0 (20)

- Thuật toán thực hiện phép tính có dấu:
- (1) Khai báo và xóa các biến lưu giá trị và dấu để chuẩn bị thực hiện phép tính.
- (2) Kiểm tra dấu của toán tử thứ nhất để xem có phải số âm không. Nếu là số âm thì thực hiện bù dấu và bù toán tử. Nếu không thì chuyển qua thực hiện bước 3.
- (3) Kiểm tra dấu của toán tử thứ hai để xem có phải số âm không. Nếu là số âm thì thực hiện bù dấu và bù toán tử. Nếu không thì chuyển sang thực hiện bước 4.
- (4) Thực hiện phép nhân hoặc chia với các toán tử vừa xử lý.
- (5) Kiểm tra dấu. Nếu zero thì coi như đã kết thúc. Nếu bằng -1 (0ffh) thì thực hiện phép tính bù hai với kết quả thu được và kết thúc.

Hiện nay người ta sử dụng hai qui ước biểu diễn số nguyên phân biệt theo thứ tự của byte trọng số trong một từ được biểu diễn:

- *Litte edian*: byte trọng số nhỏ nhất đứng trước thuận lợi cho phép cộng hoặc trừ.
- *Big endian*: byte trọng số lớn nhất đứng trước thuận lợi cho phép nhân hoặc chia.

Ví dụ: xét một số nhị phân 4*byte*

Byte 3	Byte 2	Byte 1	Byte 0

Theo qui ước biểu diễn *litte edian* thì thứ tự địa chỉ lưu trong bộ nhớ sẽ là:

Địa chỉ cở sở + 0 = Byte 0

Địa chỉ cơ sở + 1 = Byte 1

Địa chỉ cơ sở + 2 = Byte 2

Địa chỉ cơ sở + 3 = Byte 3

Và theo qui ước biểu diễn số *big edian* sẽ là:

Địa chỉ cở sở + 0 = Byte 3

Địa chỉ cơ sở + 1 = Byte 2

Địa chỉ cơ sở + 2 = Byte 1

Địa chỉ cơ sở + 3 = Byte 0

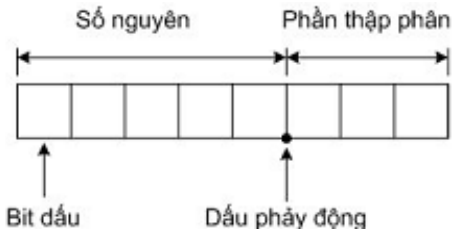
Số dấu phẩy tĩnh

Chúng ta có thể sử dụng một ký hiệu dấu chấm ảo để biểu diễn một số thực. Dấu chấm ảo được sử dụng trong từ dữ liệu dùng để phân biệt và ngăn cách giữa phần biểu diễn giá trị nguyên của dữ liệu và một phần lẻ thập phân. Ví dụ về một từ 8bit biểu diễn số dấu phảy động được chỉ ra như trong Hình 6.1. Với cách biểu diễn này, giá trị thực của số được tính như sau:

$N = a_4 2^4 + a_3 2^3 + a_2 2^2 + a_1 2^1 + a_0 2^0 + a_{-1} 2^{-1} + a_{-2} 2^{-2} + a_{-3} 2^{-3}$

$= 0.2^4+1.2^3+0.2^2+1.2^1+0.2^0+1.2^{-1}+0.2^{-2}+1.2^{-3}$

$= 8+2+1+1/2+1/8 = 11.625$



Định dạng biểu diễn số dấu phảy tĩnh 8 bit

Nhược điểm của phương pháp biểu diễn số dấu phảy tĩnh là vùng biểu diễn số nguyên bị hạn chế bởi dấu phảy tĩnh được gán cố định. Điều này dễ xảy ra hiện tượng tràn số khi thực hiện các phép nhân hai số lớn.

Số dấu phẩy động

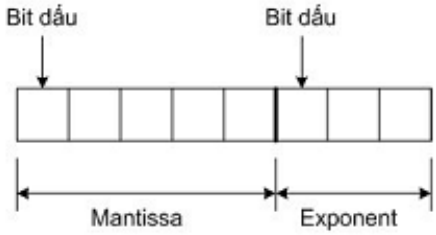
Phương pháp biểu diễn số chính xác và linh hoạt được sử dụng rộng rãi hiện nay là hệ thống biểu diễn số dấu phảy động. Đây cũng là một phương pháp biểu diễn số khoa học bao gồm 2 phần: phần biểu diễn lưu trữ số *mantissa* và một phần lưu trữ biểu diễn số *exponent*. Ví dụ trong hệ cơ số thập phân, một số nguyên bằng 5 có thể được biểu diễn hoặc là 0.5*10**1** , 50 * 10**−1** , hoặc 0.05*10**2** …Trong máy tính số hoặc hệ thống số nói chung, các số dấu phảy động nhị phân thường được biểu diễn dạng:

$N = M * 2^{\mathbf{E}}$

Trong đó:

- M là phần giá trị số *mantissa*,
- E là phần lũy thừa của số N. M thường là các giá trị lẻ mà phần thập phân của nó thường nằm trong khoảng $0.5 \leq M \leq 1$.

Hình 6.2 mô tả biểu diễn một số dấu phảy động của từ 8 bit gồm 5 bit biểu diễn phần số có nghĩa *mantissa*, và 3 bit biểu diễn phần lũy thừa. Vì các phần *mantissa* và lũy thừa đều có thể nhận các giá trị âm vì vậy các bit đầu tiên của các phần giá trị đó đều có thể được sử dụng để biểu diễn dấu khi cần thiết.



Biểu diễn dấu phảy động 8bit

Trong một số VXL, VĐK do độ rộng từ nhị phân nhỏ nên có thể sử dụng 2 từ để biểu diễn một số dấu phảy động. Một từ sẽ dùng để biểu diễn giá trị *mantissa*, và một phần biểu diễn giá trị *exponent*.

Nếu phần *mantissa* được chuẩn hóa thành một số lẻ có giá trị trong khoảng $0.5 \leq M \leq 1$ thì bit đầu tiên sau bit dấu thường là một và sẽ có một dấu phảy nhị phân ẩn ngay sau bit dấu.

Phần biểu diễn *exponent* E sẽ quyết định vị trí của dấu phảy động sẽ dịch sang trái (E>0) hay sang phải (E<0) bao nhiêu vị trí. Ví dụ biểu diễn một số thập phân 6.5 bằng một từ 8 bit dấu phảy động như sau:

*N = .1101 * 2* **11(2)** *= (1/2+1/4+1/16)*2 3 = 6.5*

Trong trường hợp này phần *mantissa* gồm 4 bit và phần *exponent* gồm 3 bit. Nếu ta dịch dấu phảy sang phải 3 vị trí bit thì chúng ta sẽ có một số nhị phân dấu phảy động biểu diễn được sẽ là *110.1*.

Tổng quát hóa trong trường hợp một số nhị phân dấu phảy động n bit gồm m bit biểu diễn phần *mantissa* và *e* bit biểu diễn phần *exponent* thì giá trị của số lớn nhất có thể biểu diễn được sẽ là:

$$N_{\max} = (1 - 2^{-m+1})2^{(2^{e-1}-1)}$$

Và số dương nhỏ nhất có thể biểu diễn là:

$$N_{\min} = 0.5 * 2^{-(2^{e-1}-1)}$$

Theo tiêu chuẩn IEEE 754 và 854 có 2 định dạng chính cho số dấu phảy động là số thực dài (*long*) và số thực ngắn (*short*) chúng khác nhau về dải biểu diễn và độ lớn lưu trữ yêu cầu. Theo chuẩn này, số thực dài được định dạng 8 byte bao gồm 1 bit dấu, 11 bit *exponent* và 53 bit lưu giá trị số có nghĩa. Một số thực ngắn được định dạng 4 byte bao gồm 1 bit dấu, 8 bit lũy thừa và 24 bit lưu giá trị số có nghĩa. Một số thực ngắn có thể biểu diễn và xử lý được số có giá trị nằm trong dải 10**38** to 10**-38** và số thực dài có thể biểu diễn và xử lý được số có giá trị thuộc dải 10**308** to 10**-308** . Để biểu diễn một giá trị tương đương như vậy bằng số dấu phảy tĩnh thì cần tới 256 bit hay 32 byte dữ liệu.

Ngôn ngữ lập trình

Một trong những ngôn ngữ lập trình có lẽ phổ cập rộng rãi nhất hiện nay là ngôn ngữ *C*. So với bất kỳ ngôn ngữ lập trình nào khác đang tồn tại C thực sự phù hợp và trở thành một ngôn ngữ phát triển của hệ nhúng. Điều này không phải là cố hữu và sẽ tồn tại mãi, nhưng tại thời điểm này thì C có lẽ là một ngôn ngữ gần gũi nhất để trở thành một chuẩn

ngôn ngữ trong thế giới hệ nhúng. Trong phần này chúng ta sẽ cùng tìm hiểu tại sao C lại trở thành một ngôn ngữ phổ biến đến vậy và tại sao chúng ta lựa chọn nó như một ngôn ngữ minh họa cho việc lập trình hệ nhúng.

Sự thành công về phát triển phần mềm thường là nhờ vào sự lựa chọn ngôn ngữ phù hợp nhất cho một dự án đặt ra. Cần phải tìm một ngôn ngữ để có thể *đáp ứng được yêu cầu lập trình cho các bộ xử lý từ 8bit đến 64bit*, trong các hệ thống chỉ có hữu hạn về bộ nhớ vài Kbyte hoặc Mbyte. Cho tới nay, điều này chỉ có C là thực sự có thể thỏa mãn và phù hợp nhất.

Rõ ràng C có một số ưu điểm nổi bật tiêu biểu như khá nhỏ và *dễ dàng cho việc học, các chương trình biên dịch thường khá sẵn cho hầu hết các bộ xử lý đang sử dụng hiện nay*, và có *rất nhiều người đã biết và làm chủ được ngôn ngữ này rồi*, hay nói cách khác cũng *đã được phổ cập từ lâu*. Hơn nữa C có lợi thế là *không phụ thuộc vào bộ xử lý thực thi mã nguồn*. Người lập trình chỉ phải tập trung chủ yếu vào việc xây dựng thuật toán, ứng dụng và thể hiện bằng ngôn ngữ thân thiện thay vì phải tìm hiểu sâu về kiến thức phần cứng, cũng như rất nhiều các ưu điểm nổi bật khác của ngôn ngữ bậc cao nói chung.

Có lẽ một thế mạnh lớn nhất của C là một ngôn ngữ bậc cao mức thấp nhất. Tức là với ngôn ngữ C chúng ta vẫn có thể điều khiển và truy nhập trực tiếp phần cứng khá thuận tiện mà không hề phải hy sinh hay đánh đổi bất kỳ một thế mạnh nào của ngôn ngữ bậc cao. Thực chất đây cũng là một trong những tiêu chí xây dựng của những người sáng lập ra ngôn ngữ C muốn hướng tới. Thực tế điều này đã được đề cập đến khi hai nhà sáng lập ra ngôn ngữ C, *Kernighan*và*Ritchie* đã đưa vào trong phần giới thiệu của cuốn sách của họ "*The C Programming Language*" như sau: "*C is a relatively "low level" language. This characterization is not pejorative; it simply means that C deals with the same sort of objects that most computers do. These may be combined and moved about with the arithmetic and logical operators implemented by real machines…"*

Tất nhiên là C không phải là ngôn ngữ duy nhất cho các nhà lập trình nhúng. Ít nhất hiện nay người ta cũng có thể biết tới ngoài ngôn ngữ C là Assembly, C++, và Ada.

Trong những buổi đầu phát triển hệ nhúng thì ngôn ngữ Assembly chủ yếu được sử dụng cho các vi xử lý đích. Với ngôn ngữ này cho phép người lập trình điều khiển và kiểm soát hoàn toàn vi xử lý cũng như phần cứng hệ thống trong việc thực thi chương trình. Tuy nhiên ngôn ngữ Assembly có nhiều nhược điểm mà cũng chính là lý do tại sao hiện nay nó ít được phổ cập và sử dụng. Đó là, việc học và sử dụng ngôn ngữ Assembly rất khó khăn và đặc biệt khó khăn trong việc phát triển các chương trình ứng dụng lớn phức tạp. Hiện nay nó chỉ được sử dụng chủ yếu như điểm nối giữa ngôn ngữ bậc cao và bậc thấp và được sử dụng khi có yêu cầu đặc biệt về hiệu suất thực hiện và tối ưu về tốc độ mà không thể đạt được bằng ngôn ngữ khác. Ngôn ngữ Assembly chỉ thực sự phù hợp cho những người có kinh nghiệm và hiểu biết tốt về cấu trúc phần cứng đích cũng như nguyên lý thực hiện của bộ lệnh và chíp xử lý.

C++ là một ngôn ngữ kế thừa từ C để nhằm vào các lớp ứng dụng và tư duy lập trình hướng đối tượng và cũng bắt đầu chiếm được số lượng lớn quan tâm trong việc ứng dụng cho phát triển hệ nhúng. Tất cả các đặc điểm cốt lõi của C vẫn được kế thừa hoàn toàn trong ngôn ngữ C++ và ngoài ra còn hỗ trợ khả năng mới về trừu tượng hóa dữ liệu và phù hợp với tư duy lập trình hiện đại; hướng đối tượng. Tuy nhiên điều này bị đánh đổi bởi hiệu suất và thời gian thực thi do đó chỉ phù hợp với các dự án phát triển chương trình lớn và không chịu sức ép lớn về thời gian thực thi.

Ada cũng là một ngôn ngữ hướng đối tượng mặc dù nó không được phổ cập rộng rãi như C++. Ada được xây dựng bởi cơ quan quốc phòng Mỹ để phục vụ phát triển các phần mềm quân sự chuyên dụng đặc biệt. Mặc dù cũng đã được chuẩn hóa quốc tế (Ada83 và Ada95) nhưng nó vẫn không được phổ cập rộng rãi ngoài việc ứng dụng chủ yếu trong các lĩnh vực quân sự và hàng không vũ trụ. Và nó cũng dần dần bị mất ưu thế và sự phổ cập trong thời gian gần đây, đây cũng là một điều đáng tiếc vì bản thân Ada cũng là một ngôn ngữ có nhiều đặc điểm phù hợp cho việc phát triển phần mềm hệ nhúng thay vì phải sử dụng C++.

Các kiến trúc phần mềm nhúng thông dụng

Hiện nay tồn tại một số loại kiến trúc phần mềm thông dụng trong các hệ thống nhúng như sau:

Vòng lặp kiểm soát đơn giản

Theo thiết kế này, phần mềm được tổ chức thành một vòng lặp đơn giản. Vòng lặp gọi đến các chương trình con, mỗi chương trình con quản lý một phần của hệ thống phần cứng hoặc phần mềm.

Hệ thống ngắt điều khiển

Các hệ thống nhúng thường được điều khiển bằng các ngắt. Có nghĩa là các tác vụ của hệ thống nhúng được kích hoạt bởi các loại sự kiện khác nhau. Ví dụ, một ngắt có thể được sinh ra bởi một bộ định thời sau một chu kỳ được định nghĩa trước, hoặc bởi sự kiện khi cổng nối tiếp nhận được một byte nào đó.

Loại kiến trúc này thường được sử dụng trong các hệ thống có bộ quản lý sự kiện đơn giản, ngắn gọn và cần độ trễ thấp. Hệ thống này thường thực hiện một tác vụ đơn giản trong một vòng lặp chính. Đôi khi, các tác vụ phức tạp hơn sẽ được thêm vào một cấu trúc hàng đợi trong bộ quản lý ngắt để được vòng lặp xử lý sau đó. Lúc này, hệ thống gần giống với kiểu nhân đa nhiệm với các tiến trình rời rạc.

Đa nhiệm tương tác

Một hệ thống đa nhiệm không ưu tiên cũng gần giống với kỹ thuật vòng lặp kiểm soát đơn giản ngoại trừ việc vòng lặp này được ẩn giấu thông qua một giao diện lập trình API. Các nhà lập trình định nghĩa một loạt các nhiệm vụ, mỗi nhiệm vụ chạy trong một môi trường riêng của nó. Khi không cần thực hiện nhiệm vụ đó thì nó gọi đến các tiến trình con tạm nghỉ (bằng cách gọi "pause", "wait", "yeild" …).

Ưu điểm và nhược điểm của loại kiến trúc này cũng giống với kiểm vòng lặp kiểm soát đơn giản. Tuy nhiên, việc thêm một phần mềm mới được thực hiện dễ dàng hơn bằng cách lập trình một tác vụ mới hoặc thêm vào hàng đợi thông dịch (queue-interpreter).

Đa nhiệm ưu tiên

Ở loại kiến trúc này, hệ thống thường có một đoạn mã ở mức thấp thực hiện việc chuyển đổi giữa các tác vụ khác nhau thông qua một bộ định thời. Đoạn mã này thường nằm ở mức mà hệ thống được coi là có một hệ điều hành và vì thế cũng gặp phải tất cả những phức tạp trong việc quản lý đa nhiệm.

Bất kỳ tác vụ nào có thể phá hủy dữ liệu của một tác vụ khác đều cần phải được tách biệt một cách chính xác. Việc truy cập tới các dữ liệu chia sẻ có thể được quản lý bằng một số kỹ thuật đồng bộ hóa như hàng đợi thông điệp (message queues), semaphores … Vì những phức tạp nói trên nên một giải pháp thường được đưa ra đó là sử dụng một hệ điều hành thời gian thực. Lúc đó, các nhà lập trình có thể tập trung vào việc phát triển các chức năng của thiết bị chứ không cần quan tâm đến các dịch vụ của hệ điều hành nữa.

Vi nhân (Microkernel) và nhân ngoại (Exokernel)

Khái niệm vi nhân (microkernel) là một bước tiếp cận gần hơn tới khái niệm hệ điều hành thời gian thực. Lúc này, nhân hệ điều hành thực hiện việc cấp phát bộ nhớ và chuyển CPU cho các luồng thực thi. Còn các tiến trình người dùng sử dụng các chức năng chính như hệ thống file, giao diện mạng lưới,… Nói chung, kiến trúc này thường được áp dụng trong các hệ thống mà việc chuyển đổi và giao tiếp giữa các tác vụ là nhanh.

Còn nhân ngoại (exokernel) tiến hành giao tiếp hiệu quả bằng cách sử dụng các lời gọi chương trình con thông thường. Phần cứng và toàn bộ phần mềm trong hệ thống luôn đáp ứng và có thể được mở rộng bởi các ứng dụng.

Nhân khối (monolithic kernels)

Trong kiến trúc này, một nhân đầy đủ với các khả năng phức tạp được chuyển đổi để phù hợp với môi trường nhúng. Điều này giúp các nhà lập trình có được một môi trường giống với hệ điều hành trong các máy để bàn như Linux hay Microsoft Windows và vì thế rất thuận lợi cho việc phát triển. Tuy nhiên, nó lại đòi hỏi đáng kể các tài nguyên phần cứng làm tăng chi phí của hệ thống. Một số loại nhân khối thông dụng là Embedded Linux và Windows CE. Mặc dù chi phí phần cứng tăng lên nhưng loại hệ thống nhúng này đang tăng trưởng rất mạnh, đặc biệt là trong các thiết bị nhúng mạnh như Wireless router hoặc hệ thống định vị GPS. Lý do của điều này là:

- Hệ thống này có cổng để kết nối đến các chip nhúng thông dụng.
- Hệ thống cho phép sử dụng lại các đoạn mã sẵn có phổ biến như các trình điều khiển thiết bị, Web Servers, Firewalls, …
- Việc phát triển hệ thống có thể được tiến hành với một tập nhiều loại đặc tính, chức năng còn sau đó lúc phân phối sản phẩm, hệ thống có thể được cấu hình để loại bỏ một số chức năng không cần thiết. Điều này giúp tiết kiệm được những vùng nhớ mà các chức năng đó chiếm giữ.
- Hệ thống có chế độ người dùng để dễ dàng chạy các ứng dụng và gỡ rối. Nhờ đó, qui trình phát triển được thực hiện dễ dàng hơn và việc lập trình có tính linh động hơn.
- Có nhiều hệ thống nhúng thiếu các yêu cầu chặt chẽ về tính thời gian thực của hệ thống quản lý. Còn một hệ thống như Embedded Linux có tốc độ đủ nhanh để trả lời cho nhiều ứng dụng. Các chức năng cần đến sự phản ứng nhanh cũng có thể được đặt vào phần cứng.

Tập lệnh

Cấu trúc tập lệnh CISC và RISC

Hầu hết các vi điều khiển và VXL nhúng có cấu trúc được phát triển dựa theo kiến trúc máy tính tập lệnh phức hợp *CISC (Complex Instruction Set Computer).* CISC là một cấu trúc xử lý các lệnh phức hợp, tức là một lệnh phức hợp sẽ bao gồm một vài lệnh đơn. Theo nguyên lý này có thể giảm bớt được thời gian dùng để truy nhập và đọc mã chương trình từ bộ nhớ. Điều này rất có ý nghĩa với các kiến trúc thiết kế xử lý tính toán theo kiểu tuần tự. Lý do cho sự ra đời của tập lệnh phức hợp nhằm giảm thiểu dung lượng bộ nhớ cần thiết để lưu giữ chương trình thực hiện, và sẽ giảm được giá thành về bộ nhớ cần cung cấp cho CPU. Các lệnh càng gọn và phức hợp thì sẽ cần càng ít không gian bộ nhớ chương trình. Kiến trúc tập lệnh phức hợp sử dụng các lệnh với độ dài biến đổi tuỳ thuộc vào độ phức hợp của các lệnh từ đơn giản đến phức tạp. Trong đó sẽ có một số lượng lớn các lệnh có thể truy nhập trực tiếp bộ nhớ. Vì vậy với kiến trúc tập lệnh phức hợp chúng ta sẽ có được một tập lệnh đa dạng phức hợp, gọn, với độ dài lệnh thay đổi và dẫn đến chu kỳ thực hiện lệnh cũng thay đổi tuỳ theo độ phức hợp trong từng lệnh.

Một vài lệnh phức hợp, đặc biệt là các lệnh truy nhập bộ nhớ cần tới vài chục chu kỳ để thực hiện. Trong một số trường hợp các nhà thiết kế VXL thấy rằng cần phải giảm chu kỳ nhịp lệnh để có đủ thời gian cho các lệnh hoàn thành điều này cũng dẫn đến thời gian thực hiện bị kéo dài hơn.

Một số VĐK được phát triển theo kiến trúc máy tính tập lệnh rút gọn *RISC (Reduced Instruction Set Computer)*. RISC phù hợp với các thiết kế kiến trúc xử lý các lệnh đơn. Thuật ngữ "rút gọn" (reduced) đôi khi bị hiểu không thật chính xác theo nghĩa đen của nó thực chất ý tưởng gốc xuất phát từ khả năng cung cấp một tập lệnh tối thiểu để thực hiện tất cả các hoạt động chính như: chuyển dữ liệu, các hoạt động *ALU* và rẽ nhánh điều khiển chương trình. Chỉ có các lệnh nạp (*load*), lữu trữ (*store*) là được phép truy nhập trực tiếp bộ nhớ.

CISC	RISC
Bất kỳ lệnh nào cũng có thể tham chiếu tới bộ nhớ	Chỉ có các lệnh Nạp (Load) hoặc Lưu trữ (Store) là có thể tham chiếu tới bộ nhớ
Tồn tại nhiều lệnh và kiểu địa chỉ	Tồn tại ít lệnh và kiểu địa chỉ
Khuôn dạng lệnh đa dạng	Khuôn dạng lệnh cố định
Chỉ có một tập thanh ghi	Có nhiều tập thanh ghi
Các lệnh thực hiện trong nhiều nhịp chu kỳ	Các lệnh thực hiện trong một nhịp chu kỳ
Có một chương trình nhỏ để thông dịch lệnh	Lệnh được thực hiện trực tiếp ngay bởi phần cứng
Chương trình thông dịch lệnh phức tạp	Chương trình biên dịch mã nguồn phức tạp
Không hỗ trợ cơ chế pipeline	Hỗ trợ cơ chế pipeline
Kích thước mã chương trình nhỏ gọn	Kích thước mã chương trình lớn

So sánh đặc điểm của CISC và RISC

Các kiểu truyền địa chỉ toán tử lệnh

Các kiểu đánh/truyền địa chỉ cho phép chúng ta chỉ ra/truyền toán tử tham gia trong các lệnh thực thi. Kiểu địa chỉ có thể chỉ ra là một hằng số, một thanh ghi hoặc một khu vực cụ thể trong bộ nhớ. Một số kiểu đánh địa chỉ cho phép sử dụng địa chỉ ngắn và một số loại thì cho phép chúng ta xác định khu vực chứa toán tử lệnh, và thường được gọi là địa chỉ hiệu dụng của toán tử và thường là động. Chúng ta sẽ xét một số loại hình đánh địa chỉ cơ bản hiện đang được sử dụng rộng rãi trong cơ chế thực hiện lệnh.

- *Đánh địa chỉ tức thì (Immediate Addressing)*: Phương pháp này cho phép truyền giá trị toán tử lệnh một cách tức thì như một phần của câu lệnh được thực thi. Ví dụ nếu sử dụng kiểu đánh địa chỉ tức thời cho câu lệnh Load 0x0008 thì giá trị 0x0008 sẽ được nạp ngay vào AC. Trường bit thường dùng để chứa toán tử lệnh sẽ chứa giá trị thực của toán tử chứ không phải địa chỉ của toán tử cần truyền cho lệnh thực thi. Kiểu địa chỉ tức thời cho phép thực thi