

Blog của Ken Shirriff

Lịch sử máy tính, khôi phục máy tính cũ, kỹ thuật đảo ngược vi mạch và bắt cú thứ gì

Thiết kế ngược thuật toán nhân trong bộ xử lý Intel 8086

Mặc dù các lập trình viên ngày nay coi phép nhân là điều hiển nhiên, nhưng hầu hết các bộ vi xử lý trong những năm 1970 chỉ có thể cộng và trừ — phép nhân đòi hỏi một vòng lặp chậm và tốn nhạt được triển khai trong mã hợp ngữ.¹ Một trong những tính năng hay của bộ xử lý Intel 8086 (1978) là nó cung cấp các lệnh nhân cho máy,² có thể nhân các số 8-bit hoặc 16-bit với một lệnh duy nhất. Bên trong, 8086 vẫn thực hiện một vòng lặp, nhưng vòng lặp được thực hiện bằng vi mã: nhanh hơn và minh bạch hơn đối với người lập trình. Mặc dù vậy, phép nhân là một thao tác chậm, chậm hơn khoảng 24 đến 30 lần so với phép cộng.

Trong bài đăng trên blog này, tôi giải thích quá trình nhân bên trong 8086, phân tích vi mã mà nó đã sử dụng và thảo luận về mạch phần cứng đã hỗ trợ nó.³ Phân tích của tôi dựa trên kỹ thuật đảo ngược 8086 từ ảnh khuôn. Ảnh khuôn dưới đây cho thấy con chip dưới kính hiển vi. Tôi đã dán nhãn các khối chức năng chính; những cái quan trọng đối với bài đăng này tôi hơn. Ở bên trái, ALU (Đơn vị Số học/Logic) thực hiện các phép toán số học cốt lõi của phép nhân: cộng và dịch chuyển. Phép nhân cũng sử dụng một số tính năng phần cứng khác: thanh ghi X, cờ F1 và bộ đếm vòng lặp. ROM vi mã ở phía dưới bên phải điều khiển quá trình.

Nhận bài viết mới qua email:

Nhập email của bạn

Đặt mua

Liên hệ

Giới thiệu về Ken Shirriff
Mastodon

Bài viết phổ biến



Bên trong Globus INK: một máy tính điều hướng cơ học cho các chuyến bay vũ trụ của Liên Xô



Thiết kế ngược thuật toán nhân trong bộ xử lý Intel 8086

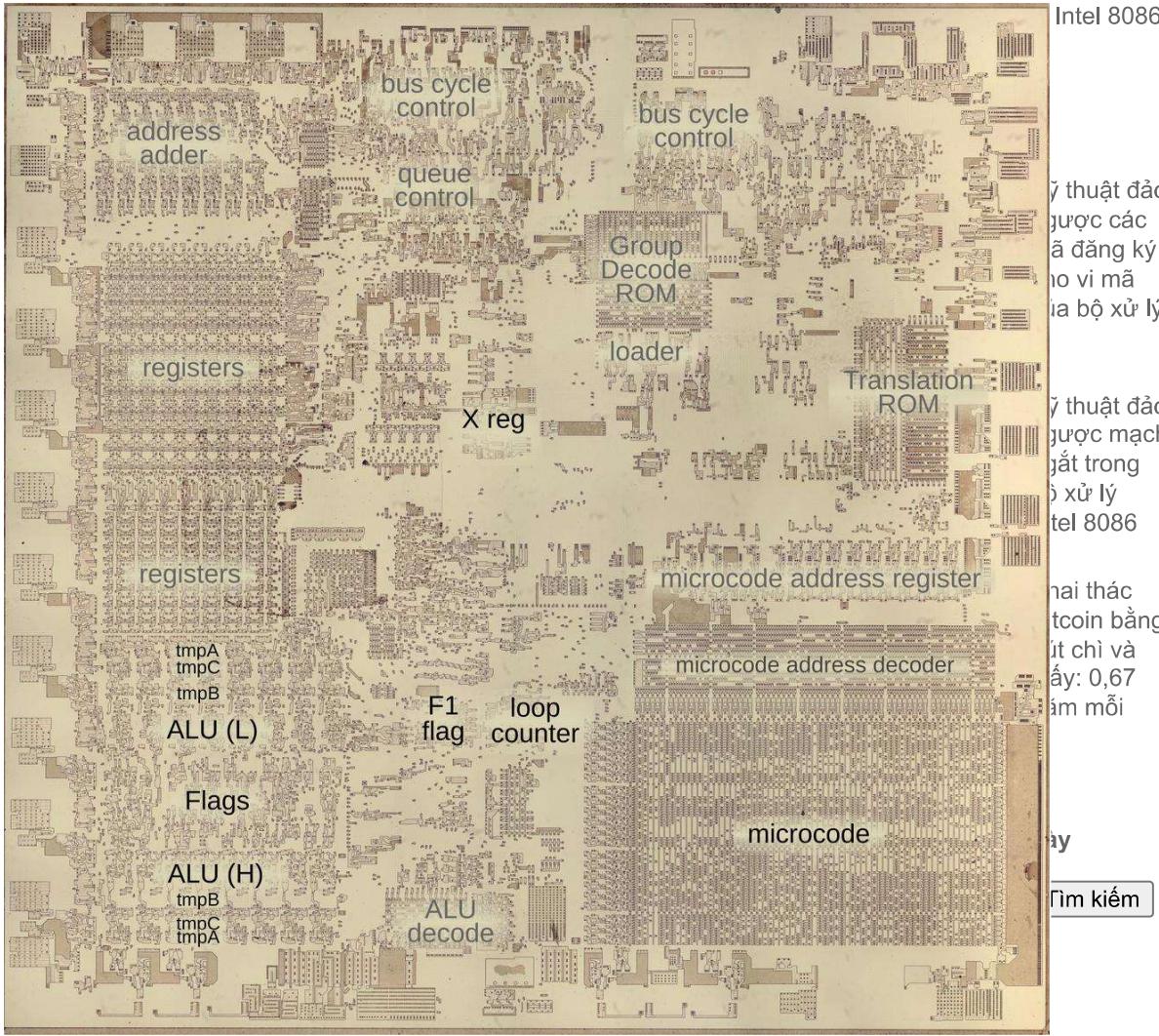


Cách bộ xử lý 8086 xác định độ dài của một lệnh



Kỹ thuật đảo ngược các thiết bị điện tử trong máy tính điều hướng tương tự Globus

Thiết kế ngược vi mã địa chỉ ModR/M trong bộ xử



Khuôn 8086 dưới kính hiển vi, với các khối chức năng chính được dán nhãn. Bức ảnh này cho thấy con chip đã được loại bỏ kim loại và polysilicon, để lộ lớp silicon bên dưới. Nhấp vào hình ảnh này (hoặc bất kỳ hình ảnh nào khác) để có phiên bản lớn hơn.

vi mã

Các thủ tục nhân trong 8086 được thực hiện trong vi mã. Hầu hết mọi người nghĩ về hướng dẫn của máy là các bước cơ bản mà máy tính thực hiện. Tuy nhiên, nhiều bộ xử lý (kể cả 8086) có một lớp phần mềm khác bên dưới: vi mã. Với vi mã, thay vì xây dựng mạch điều khiển từ các cổng logic phức tạp, phần lớn logic điều khiển được thay thế bằng mã. Để thực hiện một lệnh máy, bên trong máy tính sẽ thực thi một số lệnh vi mô đơn giản hơn, được chỉ định bởi vi mã. Điều này đặc biệt hữu ích cho một lệnh máy chẳng hạn như phép nhân, đòi hỏi nhiều bước trong một vòng lặp.

Một vi lệnh trong 8086 được mã hóa thành 21 bit như hình bên dưới. Mỗi lệnh vi mô có một bước di chuyển từ thanh ghi nguồn sang thanh ghi đích, mỗi lệnh được chỉ định bằng 5 bit. Ý nghĩa của các bit còn lại phụ thuộc vào trường hợp và có thể là bất kỳ thứ gì từ thao tác ALU đến đọc hoặc ghi bộ nhớ đến

thay đổi luồng điều khiển vi mã. Do đó, một vi lệnh 8086 thường thực hiện song song hai việc: di chuyển và hành động. Để biết thêm về vi mã 8086, hãy xem [bài đăng trên blog về vi mã](#) của tôi .

	Type	a	b	Flag
Short jump	Source	Destination	0 0	Condition
ALU operation	Source	Destination	0 1	ALU operation
Bookkeeping	Source	Destination	1 0 0	Operation 1
Memory read/write	Source	Destination	1 1 0	r/w IAK RNI Seg Reg Addr Factor
Long jump	Source	Destination	1 0 1	Condition target tag
Long call	Source	Destination	1 1 1	Condition target tag

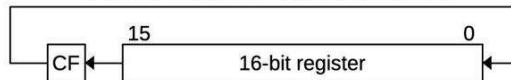
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

Việc mã hóa một lệnh vi mô thành 21 bit. Dựa trên [NEC v. Intel: Phần cứng sẽ bị cuốn vào Hố đen Bản quyền?](#)

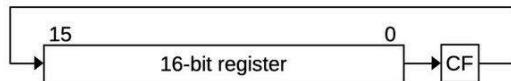
Hành vi của một hoạt động vi mô ALU rất quan trọng đối với phép nhân. ALU có ba thanh ghi tạm thời vô hình đối với người lập trình: tmpA, tmpB và tmpC. Một hoạt động ALU lấy đối số đầu tiên của nó từ bất kỳ thanh ghi tạm thời nào, trong khi đối số thứ hai luôn đến từ tmpB. Một hoạt động ALU yêu cầu hai hướng dẫn vi mô. Vi lệnh đầu tiên chỉ định hoạt động của ALU và thanh ghi nguồn, cấu hình ALU. Chẳng hạn, ADD tmpA để thêm tmpA vào tmpB mặc định. Trong vi lệnh tiếp theo (hoặc sau này), kết quả ALU có thể được truy cập thông qua thanh ghi Σ và chuyển sang thanh ghi khác.

Trước khi bắt đầu với quy trình vi mã, tôi nên giải thích hai thao tác ALU đóng vai trò trung tâm trong phép nhân: LRCY và RRCY, Xoay trái qua Mang và Xoay phải qua Mang. (Những thao tác này tương ứng với các lệnh máy RCL và RCR xoay qua thực hiện sang trái hoặc phải.) Các thao tác này dịch chuyển các bit trong từ 16 bit, tương tự như thao tác dịch chuyển bit << và trong các ngôn ngữ cấp cao, nhưng có thêm một tính năng. >> Thay vì loại bỏ bit ở cuối, bit đó được chuyển vào cờ nhớ (CF). Trong khi đó, bit trước đây trong cờ carry di chuyển vào từ. Bạn có thể coi đây là việc xoay các bit trong khi coi cờ nhớ là bit thứ 17 của từ.

LRCY micro-instruction



RRCY micro-instruction



nhãn

6502 8008 8085 8086 8087
alto analog Apollo apple arc
arduino arm beaglebone
bitcoin c# chip máy tính cadc
css dx7 thiết bị điện tử fpga
fractals bộ gen globus haskell
html5 ibm ibm1401 intel ipv6
ir java javascript toán vi mã
dao động ký ảnh cung cấp
năng lượng ngẫu nhiên
sheevaplug snark space kỹ
thuật đảo ngược
tiếng Tây Ban Nha tổng hợp lý
thuyết xé nhỏ unicode Z-80

Lưu trữ blog

▼ 2023 (16)

▼ Tháng 3 (3)

Xoay trái thông qua mang và xoay phải thông qua vi hướng dẫn mang.

Những phép dịch này thực hiện một phần quan trọng của quá trình nhân vì phép dịch có thể được xem như phép nhân hai. LRCY cũng cung cấp một cách thuận tiện để di chuyển bit quan trọng nhất sang cờ mang, nơi nó có thể được kiểm tra cho một bước nhảy có điều kiện. (Điều này rất quan trọng vì bit trên cùng được sử dụng làm bit dấu.) Tương tự, RRCY cung cấp quyền truy cập vào bit ít quan trọng nhất, rất quan trọng đối với quá trình nhân. Một thuộc tính quan trọng khác là việc thực hiện RRCY trên một từ cao hơn và sau đó RRCY trên một từ thấp hơn sẽ thực hiện dịch chuyển 32 bit, vì bit thấp của từ cao hơn sẽ được chuyển sang bit cao của từ thấp hơn thông qua bit carry.

phép nhân nhị phân

Phương pháp cộng và dịch của phép nhân (bên dưới) tương tự như phép nhân dài ở trường phổ thông, ngoại trừ nó sử dụng hệ nhị phân thay vì số thập phân. Trong mỗi hàng, số bị nhân được nhân với một chữ số của số bị nhân. (Bội số là giá trị được thêm nhiều lần và hệ số nhân kiểm soát số lần nó được thêm vào.) Các hàng liên tiếp được dịch chuyển sang trái một chữ số. Ở dưới cùng, các hàng được cộng lại với nhau để tạo ra sản phẩm. Ví dụ dưới đây cho thấy cách tính 6×5 ở dạng nhị phân bằng cách sử dụng phép nhân dài.

$$\begin{array}{r} 0110 \\ \times 0101 \\ \hline 0110 \\ 0000 \\ 0110 \\ 0000 \\ \hline 00011110 \end{array}$$

Phép nhân nhị phân dài đơn giản hơn nhiều so với phép nhân thập phân: ở mỗi bước, bạn đang nhân với 0 hoặc 1. Do đó, mỗi hàng hoặc bằng 0 hoặc bội số được dịch chuyển thích hợp (trong trường hợp này là 0110). (Không giống như phép nhân dài thập phân, bạn không cần biết bảng cửu chương.) Điều này đơn giản hóa việc triển khai phần cứng, vì mỗi bước có thể thêm hoặc không thêm bội số. Nói cách khác, mỗi bước kiểm tra một bit của hệ số nhân, bắt đầu với bit thấp, để xác định xem có nên thêm hay không. Bit này có thể thu được bằng cách dịch chuyển hệ số nhân sang bên phải một bit trong mỗi bước.

Mặc dù sơ đồ trên hiển thị tổng ở cuối, nhưng việc triển khai thực tế sẽ thực hiện phép cộng ở mỗi bước của vòng lặp, giữ cho tổng chạy được. Hơn nữa, trong 8086, thay vì dịch chuyển số bị chia sang trái trong mỗi bước, tổng sẽ dịch chuyển sang

Thiết kế ngược thuật toán nhân i...

Thiết kế ngược các mã thanh ghi cho 808...

Kỹ thuật đảo ngược các thiết bị điện tử trong Globus ...

- ▶ Tháng hai (5)
- ▶ Tháng Một (8)
- ▶ 2022 (18)
- ▶ 2021 (26)
- ▶ 2020 (33)
- ▶ 2019 (18)
- ▶ 2018 (17)
- ▶ 2017 (21)
- ▶ 2016 (34)
- ▶ 2015 (12)
- ▶ 2014 (13)
- ▶ 2013 (24)
- ▶ 2012 (10)
- ▶ 2011 (11)
- ▶ 2010 (22)
- ▶ 2009 (22)
- ▶ 2008 (27)

phải. (Kết quả giống nhau nhưng giúp việc thực hiện dễ dàng hơn) Như vậy, phép nhân 6×5 sẽ trải qua các bước dưới đây.

```

0101
×0110
-----
00000
001010
0011110
00011110

```

Tại sao bạn chuyển kết quả sang bên phải? Có một lý do thông minh cho việc này. Giả sử bạn đang nhân hai số 16 bit, kết quả là 32 bit. Điều đó yêu cầu bốn từ lưu trữ 16 bit nếu bạn sử dụng phương pháp đơn giản. Nhưng nếu bạn xem xét kỹ hơn, tổng đầu tiên phù hợp với 16 bit, và sau đó bạn cần thêm một bit ở mỗi bước. Trong khi đó, bạn đang "sử dụng hết" một bit của hệ số nhân ở mỗi bước. Vì vậy, nếu bạn gộp tổng và số nhân lại với nhau, bạn có thể ghép chúng thành hai từ. Dịch chuyển sang phải hoàn thành việc này, như sơ đồ bên dưới minh họa cho $0xffff \times 0xf00f$. Tổng (màu xanh lam) bắt đầu trong thanh ghi 16 bit được gọi `tmpA` trong khi số nhân (màu xanh lá cây) được lưu trữ trong thanh ghi 16 bit `tmpB` đăng ký. Trong mỗi bước, cả hai đều được dịch chuyển sang phải, do đó tổng tăng một bit và số nhân mất một bit. Cuối cùng, tổng chiếm tất cả 32 bit, được chia cho cả hai thanh ghi.

tổng (tmpA)	số nhân (tmpC)
0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1 1
1 0 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1 1
1 1 0 1 1 1 1 1 1	0 1 1 1 1 1 1 1 0
1 1 1 0 1 1 1 1 1	0 0 1 1 1 1 1 1 0
0 1 1 1 0 1 1 1 1	0 0 0 1 1 1 1 1 1
0 0 1 1 1 0 1 1 1	1 0 0 0 1 1 1 1 1
0 0 0 1 1 1 0 1 1	1 1 0 0 0 1 1 1 1
0 0 0 0 1 1 1 1 0	1 1 1 0 0 0 1 1 1
0 0 0 0 0 1 1 1 1	1 1 1 1 0 0 0 1 1
0 0 0 0 0 0 1 1 1	1 1 1 1 1 0 0 0 1
0 0 0 0 0 0 0 1 1	1 1 1 1 1 1 0 0 0
0 0 0 0 0 0 0 0 1	0 0 1 1 1 1 1 1 0
0 0 0 0 0 0 0 0 0	0 0 0 1 1 1 1 1 1
1 0 0 0 0 0 0 0 0	0 1 1 1 0 1 1 1 1
1 1 0 0 0 0 0 0 0	0 0 1 1 1 0 1 1 1
1 1 1 0 0 0 0 0 0	0 0 0 1 1 1 0 1 1
1 1 1 1 0 0 0 0 0	0 0 0 0 1 1 1 0 1

vì mã nhân

8086 có bốn lệnh nhân để xử lý phép nhân có dấu và không dấu của toán hạng byte và từ. Các hướng dẫn máy này được thực hiện trong vi mã. Tôi sẽ bắt đầu bằng cách mô tả phép nhân từ không dấu, phép nhân hai giá trị 16 bit và tạo ra kết

quả 32 bit. Từ nguồn được cung cấp bởi thanh ghi hoặc bộ nhớ. Nó được nhân với AX, thanh ghi tích lũy. Kết quả 32-bit được trả về trong thanh ghi DX và AX.

Vi mã bên dưới là quy trình chính để nhân từ, cả có dấu và không dấu. Mỗi vi lệnh chỉ định một thanh ghi di chuyển ở bên trái và một hành động ở bên phải. Các bước di chuyển các từ giữa các thanh ghi hiển thị và các thanh ghi tạm thời của ALU, trong khi các hành động chủ yếu là các lệnh gọi chương trình con đến các quy trình vi mô khác.

di chuyển hành động	
AX → tmpC LRCY tmpC	<i>iMUL rmw:</i>
M → tmpB CALL X0 PREIMUL	được gọi cho phép nhân có dấu
CALL CORX	thủ tục cốt lõi
CALL F1 NEGATE	được gọi cho kết quả âm
CALL X0 IMULCOF	được gọi cho phép nhân có dấu
tmpC → AX JMPS X0 7	
GỌI MULCOF	được gọi cho phép nhân không dấu
tmpA → DX RNI	

Vi mã bắt đầu bằng cách di chuyển một đối số AX vào thanh ghi C tạm thời của ALU và thiết lập ALU để thực hiện Xoay trái qua Mang trên thanh ghi này, để truy cập bit dấu. Tiếp theo, nó di chuyển đối số thứ hai M vào thanh ghi B tạm thời; M tham chiếu thanh ghi hoặc bộ nhớ được chỉ định trong byte thứ hai của lệnh, byte "ModR/M". Đối với một lệnh nhân có chữ ký, PREIMUL vi chương trình con được gọi, nhưng tôi sẽ bỏ qua lệnh đó ngay bây giờ. (X0 Điều kiện kiểm tra bit 3 của lệnh, trong trường hợp này phân biệt MUL với IMUL.) Tiếp theo, CORX chương trình con được gọi, là trung tâm của phép nhân.

⁴ Nếu kết quả cần phải được phủ định (được biểu thị bằng F1 điều kiện), NEGATE vi chương trình con được gọi. Đối với phép nhân có dấu, IMULCOF sau đó được gọi để đặt cờ nhớ và cờ tràn, trong khi MULCOF được gọi cho phép nhân không dấu. Trong khi đó, các byte kết quả được di chuyển từ các thanh ghi tạm thời C và tạm thời sang các thanh ghi AX và DX. Cuối cùng, RNI chạy hướng dẫn máy tiếp theo, kết thúc quy trình vi mã.

CORX

Trọng tâm của mã nhân là CORX quy trình thực hiện vòng lặp nhân, tính toán tích thông qua các ca và cộng. Hai dòng đầu tiên thiết lập vòng lặp, khởi tạo tổng (tmpA) thành 0. Số lượng vòng lặp được điều khiển bởi bộ đếm vòng lặp có mục đích

đặc biệt. Lệnh MAXCvi mô khởi tạo bộ đếm thành 7 hoặc 15, tương ứng với một byte hoặc từ được nhân lên. Ca đầu tiên của tmpC được thực hiện, đưa bit thấp vào cờ nhớ.

Thân vòng lặp thực hiện bước dịch chuyển và cộng. Nó kiểm tra cờ mang, bit thấp của bội số. Nó bỏ qua ADDnếu không có carry (NCY). Mặt khác, tmpB được thêm vào tmpA. (Khi tmpA được dịch chuyển sang phải, kết quả là tmpB được thêm vào các vị trí ngày càng cao hơn.) Các thanh ghi tmpA và tmpC được xoay sang phải. Điều này cũng đặt bit tiếp theo của bội số vào cờ mang cho chu kỳ tiếp theo. Vì mã sẽ nhảy lên đầu vòng lặp nếu bộ đếm không bằng 0 (NCZ). Nếu không, chương trình con trả về với kết quả là tmpA và tmpC.

$\Sigma \rightarrow \text{tmpA RRCY tmpC}$

CORX: khởi tạo xoay phải

$\Sigma \rightarrow \text{tmpC MAXC}$

nhận kết quả xoay, khởi tạo bộ đếm thành giá trị tối đa

JMPS NCY 8 **5: đầu vòng lặp**

THÈM tmpA cộng *theo điều kiện*

$\Sigma \rightarrow \text{tmpA F}$

tổng vào tmpA, cập nhật cờ để nhận

RRCY tmpA **8: Dịch chuyển 32-bit của tmpA/tmpC**

$\Sigma \rightarrow \text{tmpA RRCY tmpC}$

$\Sigma \rightarrow \text{tmpC JMPS NCZ 5}$

lặp thành 5 nếu bộ đếm không bằng 0

RTN

MULCOF

Chương trình con cuối cùng là MULCOF, cấu hình cờ mang và cờ tràn. 8086 sử dụng quy tắc nếu nửa trên của kết quả khác 0, cờ mang và cờ tràn được đặt, nếu không chúng sẽ bị xóa. Hai dòng đầu tiên chuyển tmpA (nửa trên của kết quả) qua ALU để đặt cờ 0 cho bước nhảy có điều kiện. Là một tác dụng phụ, các cờ trạng thái khác sẽ được đặt nhưng các giá trị này là "không xác định" trong tài liệu.⁶ Nếu kết quả kiểm tra khác không, thì cờ mang và cờ tràn được đặt (SCOF), nếu không chúng sẽ bị xóa (CCOF).⁵ Các SCOF và CCOF vi hoạt động được triển khai chỉ để sử dụng bằng phép nhân, minh họa cách vi mã có thể được thiết kế theo các nhu cầu cụ thể.

PASS tmpA

MULCOF: vượt qua tmpA để kiểm tra nếu không

$\Sigma \rightarrow$ không có cờ **cập nhật JMPS 12 F đích**

JMPS Z 8

12: nhảy nếu

SCOF RTN

bằng 0 , nếu không thì đặt mang và tràn

CCOF RTN

8: xóa gánh và tràn

phép nhân 8 bit

8086 có các lệnh riêng cho phép nhân 8 bit. Quá trình nhân 8 bit tương tự như phép nhân 16 bit, ngoại trừ các giá trị dài bằng một nửa và vòng lặp dịch chuyển và cộng thực hiện 8 lần thay vì 16. Như minh họa bên dưới, tổng 8 bit bắt đầu ở mức thấp một nửa thanh ghi A tạm thời và được chuyển sang trái vào tmpC. Trong khi đó, hệ số nhân 8 bit bắt đầu ở nửa thấp của tmpC và được dịch chuyển sang bên phải. Cuối cùng, kết quả được phân chia giữa tmpA và tmpC.

tổng (tmpA)								số nhân (tmpC)							
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0	1	0	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1
0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1
0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0

8086 hỗ trợ nhiều lệnh với phiên bản byte và word, sử dụng đổi số 8 bit hoặc 16 bit. Trong hầu hết các trường hợp, lệnh byte và từ sử dụng cùng một vi mã, với ALU và phần cứng đăng ký sử dụng byte hoặc từ dựa trên lệnh. Tuy nhiên, các lệnh nhân byte và từ sử dụng các thanh ghi khác nhau, yêu cầu thay đổi vi mã. Đặc biệt, hệ số nhân nằm trong AL, nửa thấp của bộ tích lũy. Cuối cùng, kết quả 16 bit được trả về trong AX, bộ tích lũy 16 bit đầy đủ; hai hướng dẫn vi mô lắp ráp kết quả từ tmpC và tmpA vào hai byte của bộ tích lũy, 'AL' và 'AH' tương ứng. Ngoài những thay đổi đó, vi mã giống như từ vi mã nhân đã thảo luận trước đó.

AL → tmpC LRCY tmpC

iMUL rmb:

M → tmpB GOI X0 PREIMUL

GOI CORX

GỌI F1 PHỦ ĐỊNH

GỌI XÓ IMULCOF

tmpC → AL JMP.S X0 7

GỌI MULCOF

tmpA → AH RNI

phép nhân có dấu

8086 (giống như hầu hết các máy tính) đại diện cho các số đã ký bằng cách sử dụng định dạng được gọi là phần bù của hai. Trong khi một byte thông thường chứa một số từ 0 đến 255, một byte có dấu chứa một số từ -128 đến 127. Một số âm

được hình thành bằng cách lật tất cả các bit (được gọi là phần bù của một) và sau đó thêm 1, thu được phần bù của hai giá trị.⁷ Chẳng hạn, +5 là 0x05 trong khi -5 là 0xfb. (Lưu ý rằng bit trên cùng của một số được đặt cho một số âm; đây là bit dấu.) Điều thú vị về các số bù của hai là các phép toán cộng và trừ giống nhau hoạt động trên cả giá trị có dấu và không dấu. Thật không may, đây không phải là trường hợp của phép nhân có dấu, vì các giá trị có dấu và không dấu mang lại các kết quả khác nhau do phần mở rộng của dấu.

8086 có các lệnh nhân riêng IMUL(Integer Multiply) để thực hiện phép nhân có dấu. 8086 thực hiện phép nhân có dấu bằng cách chuyển đổi các đối số thành giá trị dương, thực hiện phép nhân không dấu và sau đó phủ định kết quả nếu cần. Như đã trình bày ở trên, cả phép nhân có dấu và không dấu đều sử dụng cùng một vi mã, nhưng vi mã gọi một số chương trình con cho phép nhân có dấu một cách có điều kiện. Tôi sẽ thảo luận về các chương trình con vi mô dưới đây.

PREIMUL

Chương trình con đầu tiên cho phép nhân có dấu là PREIMUL, thực hiện các thao tác sơ bộ cho phép nhân số nguyên. Nó chuyển đổi hai đối số, được lưu trữ trong tmpC và tmpB, thành các giá trị dương. Nó theo dõi các dấu hiệu bằng cách sử dụng một cờ nội bộ có tên F1, chuyển đổi cờ này cho một đối số phủ định. Điều này thuận tiện xử lý quy tắc rằng hai tiêu cực tạo ra một tích cực vì bổ sung F1 cờ hai lần sẽ xóa nó.

Vi mã này, bên dưới, minh họa mức độ phức tạp của vi mã và cách các hoạt động vi mô được sắp xếp cẩn thận để có được giá trị phù hợp vào đúng thời điểm. Vi lệnh đầu tiên thực hiện một thao tác ALU và thiết lập thao tác thứ hai. Mã gọi đã thiết lập ALU để thực hiện LRCY tmpC, do đó, đó là kết quả được trả về bởi Σ (và bị loại bỏ). Thực hiện xoay trái và loại bỏ kết quả có vẻ vô nghĩa, nhưng tác dụng phụ quan trọng là bit trên cùng (tức là bit dấu) kết thúc bằng cờ nhớ. Vi mã không có bước nhảy có điều kiện dựa trên dấu hiệu, nhưng có bước nhảy có điều kiện dựa trên giá trị mang, vì vậy vấn đề là kiểm tra xem tmpC có âm hay không. Vi lệnh đầu tiên cũng thiết lập phủ định (NEG tmpC) cho hoạt động ALU tiếp theo.

$\Sigma \rightarrow \text{no dest NEG tmpC}$ **PREIMUL:** thiết lập phủ định bước nhảy tmpC
JMPS NCY 7 nếu tmpC dương

$\Sigma \rightarrow \text{tmpC CF1}$ nếu âm, phủ nhận tmpC, chuyển bước nhảy F1
JMPS 7 sang mã dùng chung

LRCY tmpB **7:**

$\Sigma \rightarrow \text{no dest NEG tmpB}$ thiết lập phủ định tmpB
JMPS NCY 11 nhảy nếu tmpB dương

$\Sigma \rightarrow \text{tmpB CF1 RTN}$ nếu âm, phủ định tmpB, lật F1
RTN **11: quay lại**

Đối với các dòng còn lại, nếu bỏ trống (NCY), hai dòng tiếp theo sẽ bị bỏ qua. Mặt khác, kết quả ALU (Σ) được ghi vào tmpC, làm cho nó dương và F1còn được bổ sung bằng CF1. (Lệnh nhảy ngắn thứ hai (JMPS) có vẻ không cần thiết, nhưng tôi đã sắp xếp lại mã cho rõ ràng.) Nửa sau của vi mã thực hiện kiểm tra tương tự trên tmpB. Nếu tmpB âm, nó bị phủ định và F1được bật.

NEGATE

Mã vi mô bên dưới được gọi sau khi tính toán kết quả, nếu kết quả cần phải được thực hiện âm. Việc phủ định khó hơn bạn mong đợi vì kết quả được phân chia giữa các thanh ghi tmpA và tmpC. Hoạt động bổ sung của hai (NEG) được áp dụng cho từ thấp, trong khi phần bù của 2 hoặc phần bù của một (COM1) được áp dụng cho từ trên, tùy thuộc vào việc thực hiện các lý do toán học.⁸ Đoạn mã này cũng chuyển đổi F1 và làm cho tmpB trở nên dương tính; Tôi nghĩ rằng mã này chỉ hữu ích cho phép chia, cũng sử dụng NEGATEchương trình con.

Phủ định tmpC **Phủ định:** phủ định tmpC

$\Sigma \rightarrow \text{tmpC COM1 tmpA F}$ có thể bổ sung cho tmpA

JMPS CY 6

NEG tmpA phủ nhận tmpA nếu không có carry

$\Sigma \rightarrow \text{tmpA CF1}$ **6: chuyển đổi F1 vì lý do nào đó**

LRCY tmpB **7: kiểm tra dấu hiệu của tmpB**

$\Sigma \rightarrow \text{không có NEG đích tmpB}$ có thể phủ định tmpB
JMPS NCY 11 bỏ qua nếu tmpB dương

$\Sigma \rightarrow \text{tmpB CF1 RTN}$ khác phủ định tmpB, chuyển đổi F1
RTN **11: quay lại**

IMULCOF

Quy trình này **IMULCOF** tương tự như **MULCOF**, nhưng việc tính toán phức tạp hơn một chút đối với kết quả đã ký. Quy trình này đặt cờ nhớ và cờ tràn nếu nửa trên của kết quả là có ý nghĩa, nghĩa là nó không chỉ là phần mở rộng về dấu của nửa dưới.⁹ Nói cách khác, byte trên cùng không đáng kể nếu nó trùng lặp bit trên cùng (bit dấu) của byte dưới. Thủ thuật trong vi mã là thêm bit trên cùng của byte thấp hơn vào byte trên bằng cách đặt nó vào cờ carry và thực hiện phép cộng với carry (ADC) bằng 0. Nếu kết quả là 0, thì byte trên không đáng kể, xử lý các trường hợp tích cực và tiêu cực. (Điều này cũng áp dụng cho các từ thay vì byte.)

ZERO → tmpB LRCY tmpC $\Sigma \rightarrow$ không có ADC đích tmpA nào $\Sigma \rightarrow$ không có	IMULCOF: nhận bit trên cùng của tmpC <i>thêm vào tmpA và 0 (tmpB)</i> cờ cập nhật F đích 12: nhảy nếu kết quả bằng 0 <i>nếu không thì đặt mang và tràn</i>
JMPS Z 8 SCOF RTN	8: xóa gánh và tràn
	CCOF RTN

Phần cứng để nhân

Phần lớn, 8086 sử dụng phép cộng ALU thông thường và dịch chuyển cho thuật toán nhân. Một số tính năng phần cứng đặc biệt cung cấp hỗ trợ.

Bộ đếm vòng lặp

8086 có bộ đếm vòng lặp 4 bit đặc biệt để nhân. Bộ đếm này bắt đầu từ 7 đối với phép nhân byte và 15 đối với phép nhân từ, dựa trên lệnh. Bộ đếm vòng lặp này cho phép vi mã giảm bộ đếm, kiểm tra điểm cuối và thực hiện nhánh có điều kiện trong một thao tác vi mô. Bộ đếm được thực hiện với bốn flip-flop, cùng với logic để tính toán giá trị sau khi giảm đi một. Lệnh MAXCvi mô (Số lượng tối đa) đặt bộ đếm thành 7 hoặc 15 cho các hoạt động byte hoặc từ tương ứng. Vi NCZlệnh (Not Counter Zero) có hai hành động. Đầu tiên, nó thực hiện một bước nhảy có điều kiện nếu bộ đếm khác không. Thứ hai, nó giảm bộ đếm.

đăng ký X

Vi mã nhân sử dụng một thanh ghi bên trong được gọi là Xthanh ghi để phân biệt giữa lệnh **MUL** và **IMUL** lệnh. Thanh Xghi là một thanh ghi 3 bit chứa mã lệnh ALU, được biểu thị bằng các bit 5–3 của lệnh.¹⁰ Vì lệnh được giữ trong Thanh ghi lệnh, bạn có thể thắc mắc tại sao cần có một thanh ghi riêng. Động lực là một số opcode chỉ định loại hoạt động ALU trong

byte thứ hai của lệnh, byte ModR/M, bit 5–3.¹¹ Do hoạt động của ALU đôi khi được chỉ định ở byte đầu tiên và đôi khi ở byte thứ hai, thanh Xghi đã được thêm vào để xử lý cả hai trường hợp này.

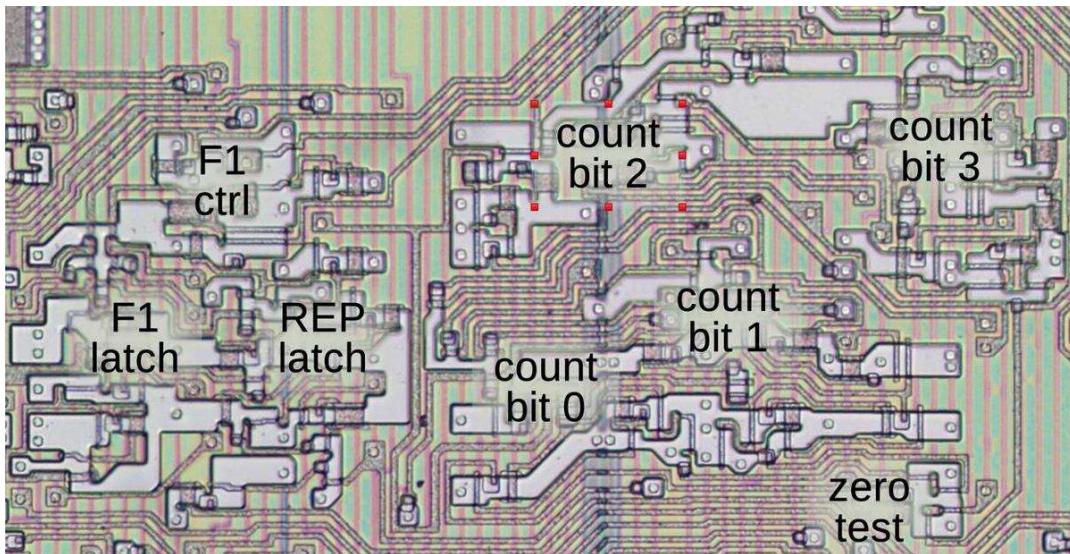
Đối với hầu hết các phần, Xthan ghi cho biết hoạt động nào trong số tám hoạt động ALU tiêu chuẩn được chọn (ADD, OR, ADC, SBB, , AND, SUB, XOR, CMP). Tuy nhiên, một vài lệnh sử dụng bit 0 của Xthan ghi để phân biệt giữa các cặp lệnh khác. Chẳng hạn, nó phân biệt giữa MULand IMUL, DIVand IDIV, CMPSand SCAS, MOVSand LODS, or AAAvà AAS. Mặc dù các cặp hướng dẫn này có thể có mã tùy ý, nhưng chúng đã được chỉ định cẩn thận. Vì mã có thể kiểm tra bit này bằng cách sử dụng X0điều kiện và thực hiện các bước nhảy có điều kiện.

Việc thực hiện Xthan ghi rất đơn giản, bao gồm ba flip-flop để giữ ba bit của lệnh. Các flip-flop được tải từ xe buýt hàng đợi tìm nạp trước trong Đồng hồ thứ nhất và trong Đồng hồ thứ hai để có các hướng dẫn thích hợp, khi các byte lệnh di chuyển trên xe buýt. Kiểm tra bit 0 của Xthan ghi với X0điều kiện được hỗ trợ bởi mạch đánh giá điều kiện vi mã, vì vậy nó có thể được sử dụng cho các bước nhảy có điều kiện trong vi mã.

Iá cờ F1

Vì mã nhân sử dụng cờ nội bộ có tên F1,¹² có hai cách sử dụng riêng biệt. Cờ theo dõi tiền REPtố để sử dụng với thao tác chuỗi. Nhưng F1cờ cũng được sử dụng bởi phép nhân và chia có dấu để theo dõi dấu hiệu. Cờ F1có thể được chuyển đổi bằng vi mã thông qua CF1hướng dẫn vi mô (Bổ sung F1). Cờ F1được thực hiện với một flip-flop, cùng với bộ ghép kênh để chọn giá trị. Nó bị xóa khi một lệnh mới bắt đầu, được đặt theo REPtiền tố và được bật bởi CF1lệnh vi mô.

Sơ đồ dưới đây cho thấy chốt F1 và bộ đếm vòng lặp xuất hiện trên khuôn như thế nào. Trong hình ảnh này, lớp kim loại đã được loại bỏ, cho thấy silicon và hệ thống dây polysilicon bên dưới.



Bộ đếm và chốt F1 khi chúng xuất hiện trên khuôn. Chốt cho trạng thái REP cũng ở đây.

Những tiến bộ sau này trong phép nhân

8086 nhân lên khá chậm so với các bộ xử lý Intel sau này.¹³

8086 mất tới 133 chu kỳ xung nhịp để nhân các giá trị 16 bit không dấu do các vòng vi mã phức tạp. Đến năm 1982, bộ xử lý Intel 286 đã cắt giảm thời gian này xuống còn 21 chu kỳ xung nhịp. Intel 486 (1989) đã sử dụng thuật toán cải tiến có thể kết thúc sớm, do đó, nhân với một số nhỏ có thể chỉ mất 9 chu kỳ.

Mặc dù những tối ưu hóa này đã cải thiện hiệu suất, nhưng chúng vẫn phụ thuộc vào việc lặp qua các bit. Với việc chuyển sang bộ xử lý 32 bit, thời gian vòng lặp trở nên khó sử dụng. Giải pháp là thay thế vòng lặp bằng phần cứng: thay vì thực hiện 32 vòng lặp thay đổi và cộng, một mảng các bộ cộng có thể tính toán phép nhân trong một bước. Số lượng phần cứng này là không hợp lý trong thời đại 8086, nhưng khi định luật Moore làm cho bóng bán dẫn nhỏ hơn và rẻ hơn, việc nhân lên phần cứng trở nên thiết thực. Chẳng hạn, Cyrix Cx486SLC (1992) có hệ số nhân phần cứng 16-bit giúp cắt từ nhân xuống còn 3 chu kỳ. Intel Core 2 (2006) thậm chí còn nhanh hơn, có thể hoàn thành phép nhân 32 bit trong mỗi chu kỳ xung nhịp.

Nhân phần cứng là một chủ đề khá phức tạp, với nhiều tối ưu hóa để tối đa hóa hiệu suất trong khi giảm thiểu phần cứng.¹⁴

Chỉ thay thế vòng lặp bằng một chuỗi gồm 32 bộ cộng là quá chậm vì kết quả sẽ bị trễ trong khi truyền qua tất cả các bộ cộng. Giải pháp là sắp xếp các bộ cộng dưới dạng cây để cung cấp tính song song. Lớp đầu tiên có 16 bộ cộng để cộng các cặp số hạng. Lớp tiếp theo thêm các cặp tổng riêng này, v.v. Cây kết quả của bộ cộng có độ sâu 5 lớp thay vì 32 lớp, giúp giảm thời gian tính tổng. Các bộ nhân thực sự đạt được những cải thiện về hiệu suất hơn nữa bằng cách chia nhỏ các

bộ cộng và tạo ra một cây phức tạp hơn: cây [Wallace](#) đáng kính (1964) và [bộ nhân Dadda](#)(1965) là hai cách tiếp cận phổ biến. Một tối ưu hóa khác là [thuật toán Booth](#) (1951), thực hiện trực tiếp phép nhân có dấu mà không cần chuyển đổi các đối số thành giá trị dương trước. Pentium 4 (2000) đã sử dụng bộ mã hóa Booth và cây Wallace ([ref](#)), nhưng nghiên cứu vào đầu những năm 2000 cho thấy cây Dadda nhanh hơn và hiện phổ biến hơn.

kết luận

Phép nhân khó tính toán hơn nhiều so với phép cộng hoặc phép trừ. Bộ xử lý 8086 đã che giấu sự phức tạp này khỏi người lập trình bằng cách cung cấp bốn lệnh nhân cho phép nhân byte và từ của các giá trị có dấu hoặc không dấu. Các hướng dẫn này đã được thực hiện phép nhân trong vi mã, thực hiện các ca và cộng trong một vòng lặp. Bằng cách sử dụng chương trình con vi mã và thực thi có điều kiện, bốn hướng dẫn máy này chia sẻ hầu hết vi mã. Vì khả năng vi mã của 8086 là rất nhỏ, đây là một tính năng quan trọng của việc triển khai.

Nếu bạn đã hoàn thành tất cả các cuộc thảo luận về vi mã, xin chúc mừng! Microcode thậm chí còn khó hiểu hơn mã hợp ngữ. Một phần của vấn đề là vi mã rất chi tiết, thậm chí các hoạt động của ALU cũng được chia thành nhiều bước. Một vấn đề phức tạp khác là vi mã 8086 thực hiện song song một thao tác di chuyển thanh ghi và một thao tác khác, vì vậy thật khó để theo dõi những gì đang diễn ra. Vi mã có vẻ hơi giống một trò chơi ghép hình, với các mảnh được ghép cẩn thận với nhau sao cho gọn nhất có thể. Tôi hy vọng những lời giải thích ở đây có ý nghĩa hoặc ít nhất là cho bạn cảm nhận về cách hoạt động của vi mã.

Cho đến nay , tôi đã viết nhiều [bài đăng về 8086](#) và dự định tiếp tục thiết kế ngược khuôn 8086, vì vậy hãy theo dõi tôi trên Twitter [@kenshirriff](#) hoặc [RSS](#) để cập nhật. Tôi cũng đã bắt đầu thử nghiệm với Mastodon gần đây với tên [@kenshirriff@oldbytes.space](#) .

Ghi chú và tài liệu tham khảo

1. Các máy tính lớn quay trở lại ENIAC có các lệnh nhân và chia. Tuy nhiên, các bộ vi xử lý đời đầu đã lùi một bước và không hỗ trợ các hoạt động phức tạp hơn này. (Lý thuyết của tôi là giá bộ nhớ giảm khiến việc thực hiện nhân và chia trong phần mềm hiệu quả hơn về mặt chi phí so với phần cứng.) National Semiconductor [IMP-16](#) , một bộ vi xử lý 16-bit bit-slice từ năm 1973, có thể là sản phẩm đầu tiên có hướng dẫn nhân và chia. Bộ xử lý 8 bit [Motorola 6809](#) (1978) bao gồm phép nhân 8 bit

nhưng không chia. Tôi nghĩ 8086 là bộ xử lý Intel đầu tiên hỗ trợ phép nhân. ↵

2. 8086 cũng hỗ trợ phép chia. Mặc dù các hướng dẫn chia tương tự như phép nhân theo nhiều cách, tôi đang tập trung vào phép nhân và bỏ qua phép chia cho bài đăng trên blog này. ↵
3. Phân tích vi mã của tôi dựa trên [quá trình tháo gỡ vi mã 8086](#) của Andrew Jenner . ↵
4. Tôi nghĩ CORX là viết tắt của Core Multiply và CORD viết tắt của Core Divide. ↵
5. Các định nghĩa về mang và tràn là khác nhau đối với phép nhân so với phép cộng và phép trừ. Lưu ý rằng kết quả của phép toán nhân sẽ luôn vừa với không gian kết quả có sẵn, lớn gấp đôi đối số. Chẳng hạn, giá trị lớn nhất bạn có thể nhận được bằng cách nhân các giá trị 16 bit là $0xffff \times 0xffff = 0xffffe0001$ khớp với 32 bit. (Phép nhân có dấu và 8 bit tương tự nhau.) Điều này trái ngược với phép cộng và phép trừ, có thể vượt quá dung lượng khả dụng của chúng. Một lần mang cho biết rằng một phần bổ sung đã vượt quá khoảng trống của nó khi được coi là không dấu, trong khi một phần bổ sung tràn cho biết rằng một phần bổ sung đã vượt quá khoảng trống của nó khi được coi là không dấu. ↵
6. Tài liệu của Intel nói rằng các cờ dấu, mang, tràn và chẵn lẻ không được xác định sau thao MULtac, mặc dù vi mã khiến chúng được tính toán. Ý nghĩa của "không xác định" là các lập trình viên không nên tính vào các giá trị cờ vì Intel có thể thay đổi hành vi trong các chip sau này. [Chủ đề này](#) thảo luận về tác động của MULcờ và hành vi khác nhau như thế nào trên chip NEC V20 . ↵
7. Có thể giải thích tại sao phần bù hai của một số được xác định bằng cách thêm 1 vào phần bù một. Phần bù của một số chỉ đơn giản là lật tất cả các bit. Nếu bạn lấy một giá trị byte n, $0xff - n$ là phần bù của một, vì bit 1 trong kết quả là bit 0.

Bây giờ, giả sử chúng ta muốn biểu thị -5 dưới dạng byte đã ký. Việc thêm $0x100$ sẽ giữ nguyên giá trị byte với việc thực hiện hết byte. Nhưng $0x100 - 5 = (1 + 0xff) - 5 = 1 + (0xff - 5) = 1 + (\text{phần bù của } 5)$. Do đó, về mặt toán học, việc biểu thị -5 bằng cách thêm 1 vào phần bù của 5 là hợp lý và điều này đúng với bất kỳ giá trị nào. ↵

8. Mã phủ định hơi phức tạp vì kết quả được chia thành hai từ. Trong hầu hết các trường hợp, từ phía trên được bổ sung theo từng bit. Tuy nhiên, nếu từ dưới bằng 0, thì từ trên bị phủ định (phần bù của hai). Tôi sẽ chứng minh bằng các giá trị 16 bit để giữ cho các ví dụ nhỏ. Số 257 (0x0101) bị phủ định thành -257 (0xffeff). Lưu ý rằng byte trên là phần bù của một (0x01 so với 0xfe) trong khi byte dưới là phần bù của hai (0x01 so với 0xff). Mặt khác, số 256 (0x0100) bị phủ định thành -256 (0xff00). Trong trường hợp này, byte trên là phần bù của hai (0x01 so với 0xff) và byte dưới cũng là phần bù của hai (0x00 so với 0x00).

(Giải thích toán học: phần bù của hai được hình thành bằng cách lấy phần bù của một và thêm 1. Trong hầu hết các trường hợp, sẽ không có phần mang từ byte thấp sang byte trên, vì vậy byte trên sẽ vẫn là phần bù của một. Tuy nhiên, nếu byte thấp là 0, thì phần bù là 0xff và việc thêm 1 sẽ tạo thành một giá trị mang. Việc thêm giá trị mang này vào byte trên sẽ tạo ra phần bù hai của byte đó.)

Để hỗ trợ phủ định nhiều từ, NEGlệnh của 8086 sẽ xóa cờ nhớ nếu toán hạng là 0 và nếu không thì đặt cờ nhớ. (Điều này ngược lại với điều trên vì các phép trừ (bao gồm cả NEG) coi cờ nhớ là cờ mượn, với ý nghĩa ngược lại.) NEGHoạt động của vi mã có hành vi giống hệt với lệnh của máy, vì nó được sử dụng để thực hiện lệnh của máy.

Do đó, để thực hiện phủ định hai từ, vi mã sẽ phủ định từ thấp (tmpC) và cập nhật các cờ (F). Nếu mang được đặt, phần bù của một được áp dụng cho từ trên (tmpA). Nhưng nếu phần mang bị xóa, phần bù của hai phần được áp dụng cho tmpA. ↵

9. Quy IMULCOFtrình coi nửa trên của kết quả là có ý nghĩa nếu nó không phải là phần mở rộng về dấu của nửa dưới. Chẳng hạn, bỏ byte trên cùng của 0x0005(+5) mang lại 0x05 (+5). Bỏ byte trên cùng của 0xffffb(-5) mang lại 0xfb (-5). Do đó, byte trên không có ý nghĩa trong những trường hợp này. Ngược lại, bỏ byte trên cùng của 0x00fb(+251) mang lại 0xfb(-5), vì vậy byte trên là đáng kể. ↵

10. Thật kỳ lạ, bằng sáng chế 8086 nói rằng Xthan ghi là thanh ghi 4 bit chứa các bit 3–6 của byte (cột 9, dòng 20). Nhưng nhìn vào khuôn, nó là một thanh ghi 3 bit giữ các bit 3–5 của byte. ↵

11. Một số lệnh được chỉ định bởi các bit 5–3 trong byte ModR/M thay vì trong byte opcode đầu tiên. Động lực là để tránh lãng phí bit cho các hướng dẫn sử dụng byte ModR/M nhưng không cần đặc tả thanh ghi. Ví dụ, hãy xem xét hướng dẫn ADD [BX], 0x1234. Lệnh này sử dụng byte ModR/M để chỉ định địa chỉ bộ nhớ. Tuy nhiên, vì nó sử dụng một toán hạng ngay lập tức nên nó không cần đặc tả thanh ghi thường được cung cấp bởi các bit 5–3 của byte ModR/M. Điều này giải phóng các bit để chỉ định hướng dẫn. Từ một khía cạnh, đây là một vụ hack tồi tệ, trong khi từ một khía cạnh khác, đây là một sự tối ưu hóa thông minh. ↩

12. Andrew Jenner thảo luận về cờ F1 và sự tương tác giữa REP và phép nhân [tại đây](#). ↩

13. Dưới đây là một số con số hiệu suất chi tiết. Bộ xử lý 8086 mất 70–77 chu kỳ xung nhịp để nhân các giá trị 8 bit và 118–133 chu kỳ xung nhịp để nhân các giá trị 16 bit. Các bội số có dấu chậm hơn một chút do các phép tính về dấu: lần lượt là 80–98 và 128–154 chu kỳ đồng hồ. Thời gian có thể thay đổi do các bước nhảy có điều kiện trong quá trình nhân.

Intel 186 (1982) đã tối ưu hóa phép nhân một chút, đưa từ thanh ghi nhân xuống còn 35–37 chu kỳ. Intel 286 (cũng năm 1982) đã giảm con số này xuống còn 21 xung nhịp. 486 (1989) đã sử dụng chức năng nhân thay đổi cộng nhưng nó có thuật toán "ra sớm" dừng khi các bit còn lại bằng 0, do đó, phép nhân 16 bit có thể mất từ 9 đến 22 đồng hồ. Bộ đồng xử lý dấu phẩy động 8087 (1980) đã sử dụng phép nhân cơ số 4, nhân với các cặp bit tại một thời điểm và cộng hoặc trừ. Điều này mang lại một nửa chu kỳ bổ sung. Kiến trúc vi mô P5 của Pentium (1993) đã sử dụng cách tiếp cận khác thường là sử dụng lại hệ số nhân phần cứng của đơn vị dấu phẩy động để nhân số nguyên, mất 10 chu kỳ cho phép nhân 32 bit. ↩

14. [Phản trình bày này](#) cung cấp một cái nhìn tổng quan về việc triển khai phép nhân trong phản ứng. ↩

11 bình luận: 

Nhãn: [8086](#), [chip](#), [vi mă](#), [kỹ thuật đảo ngược](#)

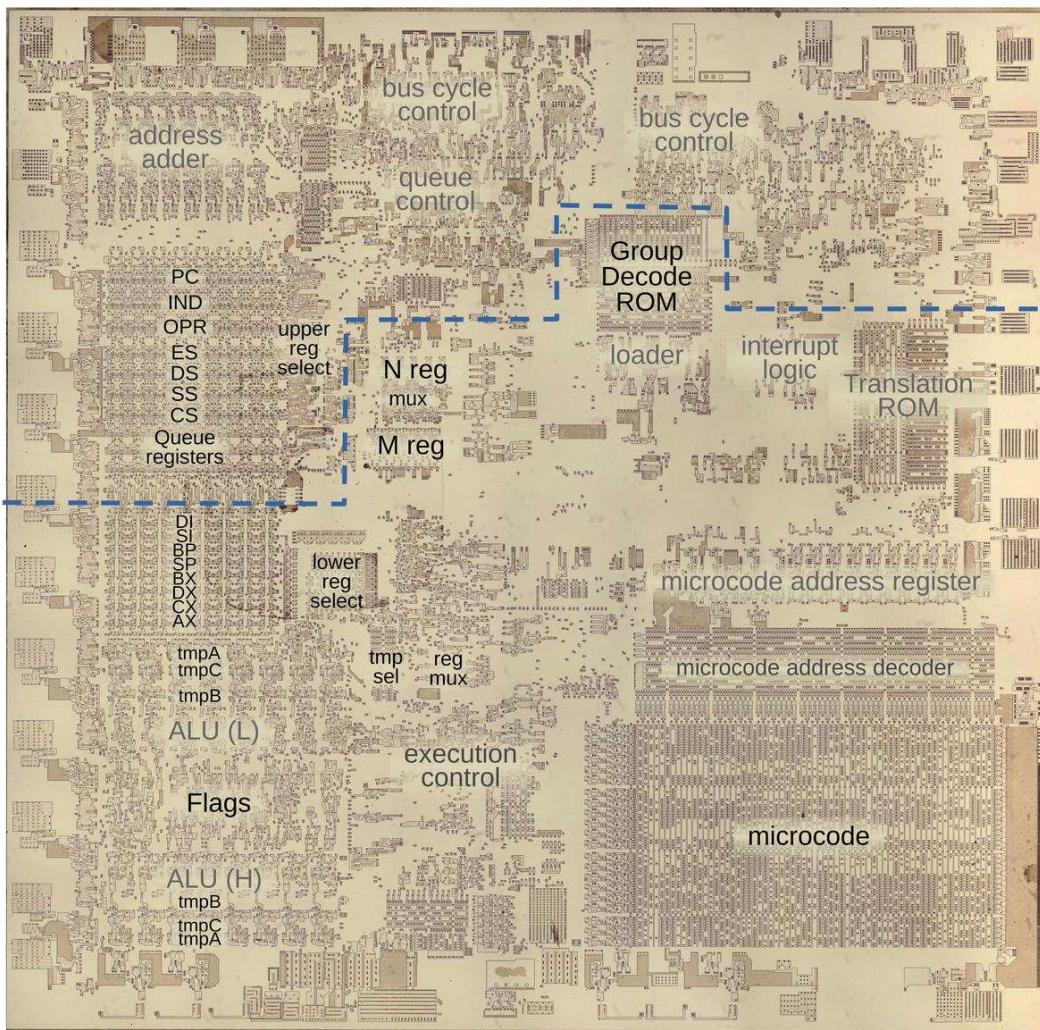
Kỹ thuật đảo ngược các mã đăng ký cho vi mă của bộ xử lý 8086

Giống như hầu hết các bộ xử lý, Intel 8086 (1978) cung cấp các thanh ghi nhanh hơn bộ nhớ chính. Cũng như các thanh ghi mà lập trình viên có thể nhìn thấy, 8086 có một số thanh

ghi bên trong bị ẩn đói với người dùng. Bên trong, 8086 có một lược đồ phức tạp để chọn thanh ghi nào sẽ sử dụng, với sự kết hợp giữa vi mã và phần cứng. Các thanh ghi được gán một số nhận dạng 5 bit, từ lệnh máy hoặc từ vi mã. Trong bài đăng trên blog này, tôi giải thích cách hệ thống đăng ký này hoạt động.

Phân tích của tôi dựa trên kỹ thuật đảo ngược 8086 từ ảnh chét. Ảnh khuôn dưới đây cho thấy con chip dưới kính hiển vi. Đối với ảnh khuôn này, tôi đã loại bỏ các lớp kim loại và polysilicon, để lộ lớp silicon bên dưới. Tôi đã dán nhãn các khối chức năng chính; những cái quan trọng đối với bài đăng này tối hơn. Đặc biệt, các thanh ghi và Đơn vị Số học/Logic (ALU) ở bên trái và vi mã ROM ở phía dưới bên phải. Về mặt kiến trúc, chip được phân vùng thành Đơn vị giao diện xe buýt (BIU) ở trên cùng và Đơn vị thực thi (EU) bên dưới. BIU xử lý hoạt động của bus và bộ nhớ cũng như tìm nạp trước lệnh, trong khi Đơn vị Thực thi (EU) thực thi lệnh.

Bus Interface Unit (BIU)



Execution Unit (EU)

Khuôn 8086 dưới kính hiển vi, với các khối chức năng chính được dán nhãn.
Nhấp vào hình ảnh này (hoặc bất kỳ hình ảnh nào khác) để có phiên bản lớn

hon.

vi mă

Hầu hết mọi người nghĩ về hướng dẫn của máy là các bước cơ bản mà máy tính thực hiện. Tuy nhiên, nhiều bộ xử lý (kể cả 8086) có một lớp phần mềm khác bên dưới: vi mă. Với vi mă, thay vì xây dựng mạch điều khiển từ các công logic phức tạp, phần lớn logic điều khiển được thay thế bằng mã. Để thực hiện một lệnh máy, bên trong máy tính sẽ thực thi một số lệnh vi mô đơn giản hơn, được chỉ định bởi vi mă.

8086 sử dụng một cách tiếp cận hỗn hợp: mặc dù nó sử dụng vi mă nhưng phần lớn chức năng lệnh được thực hiện với logic cổng. Cách tiếp cận này đã loại bỏ sự trùng lặp khỏi vi mă và giữ cho vi mă đủ nhỏ cho công nghệ năm 1978. Theo một nghĩa nào đó, vi mă được tham số hóa. Chẳng hạn, vi mă có thể chỉ định thao tác Đơn vị Số học/Logic (ALU) chung và một thanh ghi chung. Logic cổng kiểm tra hướng dẫn để xác định thao tác cụ thể nào cần thực hiện và thanh ghi thích hợp.

Một vi lệnh trong 8086 được mã hóa thành 21 bit như hình bên dưới. Mỗi vi lệnh đều có một lần di chuyển từ thanh ghi nguồn sang thanh ghi đích, mỗi lệnh được chỉ định bằng 5 bit; mã hóa này là chủ đề chính của bài đăng trên blog này. Ý nghĩa của các bit còn lại phụ thuộc vào trường loại và có thể là bất kỳ thứ gì từ thao tác ALU đến đọc hoặc ghi bộ nhớ đến thay đổi luồng điều khiển vi mă. Để biết thêm về vi mă 8086, hãy xem bài đăng trên blog về vi mă của tôi .

	Type	a	b	Flag
Short jump	Source	Destination	0 0	Condition
ALU operation	Source	Destination	0 1	ALU operation
Bookkeeping	Source	Destination	1 0 0	Operation 1
Memory read/write	Source	Destination	1 1 0	r/w IAK RNI Seg Reg Addr Factor
Long jump	Source	Destination	1 0 1	Condition
Long call	Source	Destination	1 1 1	target tag
	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21			

Việc mã hóa một lệnh vi mô thành 21 bit. Dựa trên [NEC v. Intel: Phần cứng sẽ bị cuốn vào Hố đen Bản quyền?](#)

Hãy xem cách hướng dẫn máy XCHG AX, regđược triển khai trong vi mă. Lệnh này trao đổi AXvới thanh ghi được chỉ định trong 3 bit thấp của lệnh.¹ Vi mă cho hướng dẫn này bao gồm ba hướng dẫn vi mô, vì vậy hướng dẫn mất ba chu kỳ đồng hồ. Mỗi hướng dẫn vi mô chứa một động thái, đó là phần thú vị.² Thanh ghi đã chỉ định được chuyển đến thanh ghi B tạm thời của ALU, thanh AXghi được chuyển đến thanh ghi đã

chỉ định và cuối cùng B tạm thời được chuyển đến AX, hoàn thành việc hoán đổi.

di chuyển hành động

M → tmpB

XCHG AX,rw : di chuyển reg tới tmpB

AX → M NXT

di chuyển AX tới reg, Tiếp theo

tmpB

cuối cùng → AX RNI di chuyển tmpB tới AX, Chạy Hướng dẫn Tiếp theo

Phần quan trọng cho cuộc thảo luận này là cách M chỉ ra thanh ghi mong muốn. Giả sử hướng dẫn là XCHG AX,DX. Ba bit dưới cùng của hướng dẫn là 010, cho biết DX. Trong chu kỳ xung nhịp đầu tiên của việc thực hiện lệnh, byte opcode được chuyển từ hàng đợi tìm nạp trước qua bus hàng đợi. Thanh Mghi được tải với DXsô (có thể là 26), dựa trên ba bit dưới cùng của lệnh. Sau chu kỳ đồng hồ thứ hai, vi mã bắt đầu. Lệnh vi mô đầu tiên đặt Mgiá trị của (26) lên bus nguồn và số cho tmpB(13) trên bus đích, khiến quá trình chuyển từ DX sang tmpB. Vi lệnh thứ hai đặt AXsô (24) lên bus nguồn và Mgiá trị (26) lên xe buýt đích, khiến quá trình chuyển từ AX sang DX. Vi lệnh thứ ba đặt tmpBsô (13) lên bus nguồn và AXsô (24) lên bus đích, gây ra quá trình chuyển từ tmpBsang AX.

Do đó, các giá trị trên bus nguồn và đích kiểm soát việc truyền dữ liệu trong mỗi vi lệnh. Vi mã có thể chỉ định rõ ràng các giá trị này (như for AX và tmpB) hoặc có thể chỉ định Mthan ghi để sử dụng thanh ghi được xác định trong lệnh. Do đó, cùng một vi mã thực hiện tất cả các XCHG hướng dẫn và vi mã không cần biết thanh ghi nào có liên quan.

Mã hóa thanh ghi

Vi mã ở trên minh họa cách các số khác nhau chỉ định các thanh ghi khác nhau. Bảng dưới đây cho thấy cách số 0-31 ánh xạ vào một thanh ghi. Một số số có ý nghĩa khác đối với thanh ghi nguồn hoặc thanh ghi đích; một dấu gạch chéo phân tách các mục này.

0	ES	số 8	AL	16	AH	24	CÂY RÌU
1	CS	9	CL	17	CH	25	CX
2	SS	10	ĐL	18	ĐH (M)	26	ĐX
3	DS	11	BL	19	BH (N)	27	BX
4	máy tính	12	tmpA	20	Σ/tmpAL	28	SP
5	ÂN	13	tmpB	21	ONES/tmpBL	29	BP
6	OPR	14	tmpC	22	CR/tmpAH	30	VÀ

7 Hồi-	15 F	23 ZERO/tmpBH	31 TÙ
--------	------	---------------	-------

Hầu hết các mục nhập này là các thanh ghi mà lập trình viên có thể nhìn thấy: các thanh ghi phân đoạn có màu xanh lá cây, các thanh ghi 8 bit có màu xanh lam và các thanh ghi 16 bit có màu đỏ. Một số thanh ghi bên trong và thanh ghi giả cũng có thể truy cập được: IND(Thanh ghi gián tiếp), giữ địa chỉ bộ nhớ để đọc hoặc ghi; OPR(Operand register), giữ dữ liệu để đọc hoặc ghi; Q(Hàng đợi), đọc một byte từ hàng đợi tìm nạp trước lệnh; Các thanh ghi tạm thời ALU A, B và C, cùng với các byte thấp (L) và (H); F, Cờ đăng ký; Σ, đầu ra ALU; ONES, tất cả những cái ; CR, ba bit thấp của địa chỉ vi mã; và ZERO, giá trị bằng không. Các mục M và N chỉ có thể được chỉ định từ vi mã, thay thế cho DH và BH.

Cái bàn hơi phức tạp, nhưng có những lý do cho cấu trúc của nó. Đầu tiên, hướng dẫn máy trong thanh ghi mã hóa 8086 theo hệ thống bên dưới. Số thanh ghi 5 bit ở trên thực chất là phần mở rộng của mã hóa lệnh. Ngoài ra, các thanh ghi AX/CX/DX/BX (màu đỏ) được sắp xếp theo phiên bản byte trên và byte thấp hơn (màu xanh). Điều này đơn giản hóa phần cứng vì ba bit thấp của số thanh ghi chọn thanh ghi, ³ trong khi hai bit trên thực hiện lựa chọn byte so với từ. Các thanh ghi bên trong khớp với các vị trí có sẵn trong bảng.

16-Bit (w = 1)	8-Bit (w = 0)	Segment
000 AX	000 AL	00 ES
001 CX	001 CL	01 CS
010 DX	010 DL	10 SS
011 BX	011 BL	11 DS
100 SP	100 AH	
101 BP	101 CH	
110 SI	110 DH	
111 DI	111 BH	

Các bài tập thanh ghi, từ [MCS-86 Hướng dẫn tham khảo ngôn ngữ hội](#).

Byte ModR/M

Nhiều lệnh của 8086 sử dụng một byte thứ hai được gọi là byte ModR/M để chỉ định các chế độ đánh địa chỉ.⁴ Byte ModR/M mang lại cho 8086 rất nhiều tính linh hoạt trong cách một lệnh truy cập các toán hạng của nó. Byte chỉ định một thanh ghi cho một toán hạng và một thanh ghi hoặc bộ nhớ cho toán hạng kia. Sơ đồ bên dưới cho thấy cách byte được chia thành ba trường: modchọn chế độ tổng thể, regchọn một thanh ghi và r/mchọn một thanh ghi hoặc chế độ bộ nhớ. Đối với một byte ModR/M, các trường reg và r/m được đọc vào

các thanh ghi N và M tương ứng, vì vậy các thanh ghi được chỉ định trong byte ModR/M có thể được truy cập bằng vi mã.

chế độ	đăng ký	r/m
7	6	5 4 3 2 1 0

Hãy xem hướng dẫn SUB AX, BX trừ BX từ AX. Trong 8086, một số bước xử lý quan trọng diễn ra trước khi bắt đầu vi mã. Đặc biệt, "ROM giải mã nhóm" phân loại lệnh thành hơn một chục danh mục ảnh hưởng đến cách nó được xử lý, chẳng hạn như lệnh được triển khai không có vi mã, lệnh một byte hoặc lệnh có byte ModR/M. ROM giải mã nhóm cũng chỉ ra cấu trúc của các lệnh, chẳng hạn như các lệnh có một Wbit chọn byte so với các thao tác từ hoặc một Dbit đảo ngược hướng của các toán hạng. Trong trường hợp này, Group Decode ROM phân loại lệnh có chứa một Dbit, một Wbit, một thao tác ALU và một byte ModR/M.

Dựa trên các tín hiệu của Group Decode ROM, các trường từ opcode và các byte ModR/M được trích xuất và lưu trữ trong các thanh ghi bên trong khác nhau. Loại hoạt động ALU (SUB) được lưu trữ trong ALU_opr thanh ghi. Byte ModR/M chỉ định BX trong regtrường và AX trong r/mtrường để regso thanh ghi (BX, 27) được lưu trong N thanh ghi và r/msô thanh ghi (AX, 24) được lưu trong M thanh ghi.

Khi quá trình giải mã sơ bộ hoàn tất, vi mã bên dưới cho lệnh ALU này được thực thi.⁵ Có ba vi lệnh, vì vậy lệnh mất ba chu kỳ đồng hồ. Đầu tiên, thanh ghi được chỉ định bởi M(tức là AX) được chuyển sang thanh ghi A tạm thời của ALU (tmpA).

Trong khi đó, XI định cấu hình ALU để thực hiện thao tác được chỉ định bởi các bit lệnh, tức là SUB. Vi lệnh thứ hai di chuyển thanh ghi được chỉ định bởi N(tức là BX) sang thanh ghi của ALU tmpB. Vi lệnh cuối cùng lưu trữ kết quả của ALU (Σ , số 20) trong thanh ghi được biểu thị bởi M(tức là AX).

di chuyển hành động	
M → tmpA	XI tmpA ALU rm→r : AX đến tmpA
N → tmpB	NXT BX đến tmpB
Σ → M	RNI F kết quả thành AX, cập nhật cờ

Một trong những tính năng thú vị của 8086 là nhiều lệnh chứa một Dbit đảo ngược hướng hoạt động, hoán đổi nguồn và đích. Nếu chúng ta giữ byte ModR/M nhưng sử dụng SUB lệnh với D bộ bit, lệnh sẽ trở thành SUB BX, AX, trừ đi AX,

BX ngược lại với trước đó. (Hoán đổi nguồn và đích sẽ hữu ích hơn khi một đối số nằm trong bộ nhớ. Nhưng tôi sẽ sử dụng một ví dụ với hai thanh ghi để giữ cho nó đơn giản.) Hướng dẫn này chạy chính xác cùng một vi mã như trước đây. Sự khác biệt là khi vi mã truy cập M, do bit định hướng, nó nhận giá trị trong N, tức là BX thay vì AX. Quyền truy cập N được hoán đổi tương tự. Kết quả là AX được trừ khỏi BX và sự thay đổi hướng là trong suốt đối với vi mã.⁶

Thanh ghi M và N

Bây giờ chúng ta hãy xem xét kỹ hơn cách các thanh ghi M và N được triển khai. Mỗi thanh ghi giữ một số thanh ghi 5 bit, được mở rộng từ ba bit của lệnh. Thanh ghi M được tải từ ba bit có ý nghĩa nhỏ nhất của lệnh hoặc byte ModR/M, trong khi thanh ghi N được tải với các bit từ ba đến năm. Thông thường nhất, các thanh ghi được chỉ định bởi byte ModR/M, nhưng một số hướng dẫn chỉ định thanh ghi trong opcode.⁶

Bảng dưới đây cho thấy cách các bit trong opcode của lệnh hoặc byte ModR/M (i5, i4, i3) được chuyển đổi thành số 5 bit cho thanh ghi N. Có ba trường hợp: thanh ghi 16 bit, thanh ghi 8 bit và thanh ghi đoạn. Các ảnh xạ bên dưới có vẻ ngẫu nhiên, nhưng chúng dẫn đến các mục được hiển thị trong bảng mã hóa thanh ghi 5 bit trước đó. Tôi đã tô màu các mục để bạn có thể thấy sự tương ứng.

Cách thức	4	3	2	1	0
đăng ký 16-bit	1	1	i5	i4	i3
đăng ký 8 bit	i5	i5'	0	i4	i3
phân đoạn reg	0	0	0	i4	i3

Tôi sẽ đi qua ba trường hợp chi tiết hơn. Nhiều lệnh 8086 có hai phiên bản, một phiên bản hoạt động theo byte và một phiên bản hoạt động theo từ, được phân biệt bằng bit W(bit 0) trong lệnh. Nếu ROM giải mã nhóm chỉ ra rằng lệnh có một Wbit và Wbit đó là 0, thì lệnh đó là lệnh byte.⁷ Nếu lệnh có một byte ModR/M và lệnh hoạt động trên một byte, thì thanh Nghi được nạp với số 5 bit cho thanh ghi byte đã chỉ định. Điều này xảy ra trong "Đồng hồ thứ hai", chu kỳ đồng hồ khi byte ModR/M được tìm nạp từ hàng đợi lệnh. Trường hợp thứ hai cũng tương tự; nếu lệnh hoạt động trên một từ, thì N thanh ghi được nạp số cho thanh ghi từ được chỉ định trong byte ModR/M.

Trường hợp thứ ba xử lý một thanh ghi đoạn. Thanh Nghi được tải với số thanh ghi đoạn trong Đồng hồ thứ hai nếu

ROM giải mã nhóm cho biết lệnh có byte ModR/M với trường thanh ghi đoạn (cụ thể là lệnh thanh ghi đoạn MOV). Một chút ngạc nhiên, một số đăng ký phân đoạn cũng được tải trong Đồng hồ đầu tiên. Điều này hỗ trợ các hướng dẫn thanh ghi phân đoạn PUSH và POP, có thanh ghi phân đoạn được mã hóa trong các bit 3 và 4 của opcode. ^{số 8}

Bảng dưới đây cho thấy cách các bit được gán trong thanh ghi M, thanh ghi này sử dụng các bit lệnh **i2**, **i1**, và **i0**. Các trường hợp phức tạp hơn một chút so với N thanh ghi. Đầu tiên, số đăng ký 16 bit được tải từ byte opcode trong Đồng hồ đầu tiên để hỗ trợ các lệnh chỉ định thanh ghi ở các bit thấp. Trong Đồng hồ thứ hai, giá trị này có thể được thay thế.

Đối với byte ModR/M sử dụng chế độ thanh ghi, M thanh ghi được nạp lại bằng thanh ghi 8 bit hoặc 16 bit đã chỉ định, tùy thuộc vào tín hiệu chế độ byte được mô tả trước đó. Tuy nhiên, đối với byte ModR/M sử dụng chế độ bộ nhớ, thanh M ghi được nạp với OPR (Toán hạng), thanh ghi bên trong chứa từ được đọc hoặc ghi vào bộ nhớ.

Cách thức	4	3	2	1	0
đăng ký 16-bit	1	1	i2	i1	i0
đăng ký 8 bit	i2	i2'	0	i1	i0
OPR	0	0	1	1	0
AX/AL	byte'	1	0	0	0
chuyển sang 8-bit	m2	m2'	0	m1	m0

Nhiều hướng dẫn sử dụng thanh ghi AX hoặc AL, chẳng hạn như hướng dẫn ngay lập tức ALU, hướng dẫn đầu vào và đầu ra và hướng dẫn chuỗi. Đối với những điều này, ROM giải mã nhóm kích hoạt AX hoặc AL số thanh ghi cụ thể được tải vào thanh ghi M trong Đồng hồ thứ hai. Bit trên cùng được đặt cho thao tác từ và xóa cho thao tác cung cấp byte AX hoặc AL khi thích hợp.

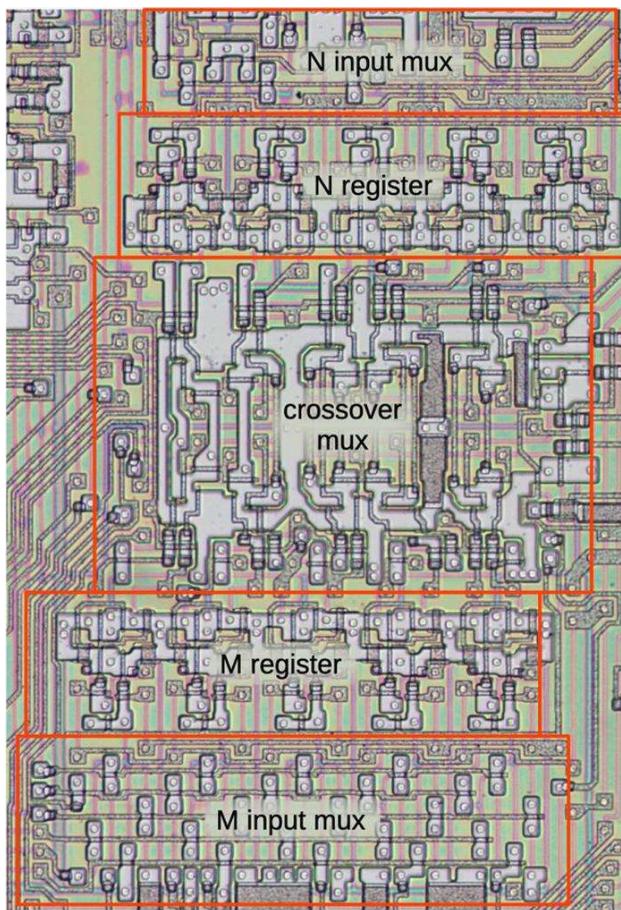
Trường hợp đăng ký cuối cùng M là một chút khó khăn. Đối với một lệnh di chuyển ngay lập tức, chẳng hạn như MOV BX, imm, bit 3 chuyển đổi giữa một byte và một từ di chuyển (chứ không phải bit 0), bởi vì các bit 2-0 chỉ định thanh ghi. Thật không may, đầu ra ROM Giải mã Nhóm không khả dụng trong Đồng hồ đầu tiên để chỉ ra trường hợp này. Thay vào đó, M được tải trong Đồng hồ đầu tiên với giả định là thanh ghi 16 bit. Nếu điều đó sai, M thanh ghi được chuyển đổi thành số thanh ghi 8 bit trong Đồng hồ thứ hai bằng cách xáo trộn một vài bit.

Sản xuất các giá trị nguồn và đích

Có ba trường hợp đối với số đi trên bus nguồn hoặc đích: số thanh ghi có thể đến từ vi lệnh, giá trị có thể đến từ hoặc thanh M. Giá N như được chỉ định trong vi lệnh hoặc giá trị có thể đến từ M và N đăng ký với các vai trò được hoán đổi bởi Dbit. (Lưu ý rằng nguồn và đích có thể là các trường hợp khác nhau và được chọn bằng mạch riêng.)

Trường hợp đầu tiên là trường hợp mặc định, trong đó 5 bit từ nguồn vi lệnh hoặc đích chỉ định thanh ghi trực tiếp. Chẳng hạn, trong vi lệnh `tmpB→AX`, vi mã biết thanh ghi nào đang được sử dụng và chỉ định trực tiếp chúng.

Trường hợp thứ hai và thứ ba liên quan đến logic nhiều hơn. Xem xét nguồn trong `M→tmpB`. Đối với lệnh không có Dbit, số thanh ghi được lấy từ M. Tương tự như vậy nếu Dbit là 0. Nhưng nếu lệnh sử dụng một Dbit và Dbit đó là 1, thì số thanh ghi được lấy từ N. Bộ ghép kênh giữa các thanh ghi M và N chọn thanh ghi thích hợp để đưa lên bus.



Thanh ghi M và N khi chúng xuất hiện trên khuôn. Lớp kim loại đã được loại bỏ khỏi hình ảnh này để hiển thị silicon và polysilicon bên dưới.

Sơ đồ trên cho thấy mạch thanh ghi M và N mà mạch được thực hiện như thế nào trên khuôn, với N thanh ghi ở trên cùng và M thanh ghi ở dưới. Mỗi thanh ghi có một bộ ghép kênh đầu vào thực hiện các bảng trên, chọn 5 bit thích hợp tùy thuộc vào chế độ. Bản thân các thanh ghi được thực hiện như các chốt động được điều khiển bởi đồng hồ. Ở giữa, một bộ ghép kênh chéo điều khiển các bus nguồn và đích, chọn M và N đăng ký khi thích hợp và khuếch đại tín hiệu bằng các bóng bán dẫn tương đối lớn. Đầu ra thứ ba từ bộ ghép kênh, các bit từ vi lệnh, được triển khai trong mạch được tách biệt về mặt vật lý và gần với ROM vi mã hơn.

Phần cứng lựa chọn thanh ghi

Làm thế nào để số 5 bit chọn một thanh ghi? 8086 có một loạt logic biến một số thanh ghi thành một dòng điều khiển cho phép đọc hoặc ghi thanh ghi. Phần lớn, logic này được thực hiện với các cổng NOR khớp với một số thanh ghi cụ thể và tạo tín hiệu chọn. Tín hiệu đi qua một [trình điều khiển bootstrap](#) đặc biệt để tăng điện áp vì nó cần điều khiển 16 bit thanh ghi.

Các thanh ghi của 8086 được tách thành hai nhóm chính. "Các thanh ghi phía trên" nằm ở phía trên bên trái của chip, trong Đơn vị Giao diện Bus. Đây là những thanh ghi liên quan trực tiếp đến việc truy cập bộ nhớ. "Các thanh ghi thấp hơn" nằm ở phía dưới bên trái của chip, trong Đơn vị thực thi. Từ dưới lên trên, chúng là AX, CX, DX, BX, SP, BP, SI, và DI; thứ tự vật lý của chúng khớp với thứ tự của chúng trong tập lệnh.⁹ Một PLA (Mảng logic có thể lập trình) riêng biệt chọn các thanh ghi hoặc cờ tạm thời ALU làm đích. Ngay bên dưới nó, một PLA chọn nguồn từ các thanh ghi tạm thời ALU, cờ hoặc kết quả ALU (Σ).¹⁰ Tôi đã viết về các thanh ghi của 8086 và cách triển khai mức thấp của chúng [tại đây](#) nếu bạn muốn biết thêm thông tin.

Một số lịch sử

Hệ thống chọn các thanh ghi có mã 3 bit của 8086 bắt nguồn từ Datapoint 2200,¹¹ một máy tính để bàn được công bố vào năm 1970. Bộ xử lý của Datapoint 2200 được triển khai với một bảng mạch tích hợp TTL, vì đây là trước bộ vi xử lý. Nhiều hướng dẫn của Datapoint đã sử dụng mã 3 bit để chọn thanh ghi, với đặc tả thanh ghi đích ở bit 5-3 của lệnh và thanh ghi nguồn ở bit 2-0. (Bố cục này về cơ bản giống như trong các lệnh 8086 và byte ModR/M.)¹² Tám giá trị của mã này đã chọn một trong 7 thanh ghi, với giá trị thứ tám biểu thị quyền truy cập bộ nhớ. Intel sao chép kiến trúc Datapoint 2200 cho 8008

¹³ bộ vi xử lý (1972) và làm sạch nó cho 8080 (1974), nhưng vẫn giữ cách bố trí lệnh cơ bản và các bit lựa chọn thanh ghi/bộ nhớ.

Việc 8086 sử dụng một hệ thống đánh số cho tất cả các thanh ghi vượt xa khuôn mẫu này một cách đáng kể, một phần vì các thanh ghi của nó hoạt động như các thanh ghi mục đích chung và các thanh ghi mục đích đặc biệt.¹⁴ Nhiều lệnh có thể hoạt động thay thế cho các thanh ghi AX, BX, v.v., coi chúng là các thanh ghi mục đích chung. Nhưng mỗi thanh ghi này đều có mục đích đặc biệt riêng cho các hướng dẫn khác, vì vậy vi mã phải có khả năng truy cập chúng một cách cụ thể. Điều này thúc đẩy cách tiếp cận của 8086 trong đó các thanh ghi có thể được coi là các thanh ghi mục đích chung được chọn từ các bit lệnh hoặc như các thanh ghi mục đích đặc biệt được chọn bởi vi mã.

Motorola 68000 (1979) là một so sánh thú vị với 8086 vì chúng là đối thủ cạnh tranh. 68000 sử dụng vi mã rộng hơn nhiều (vi lệnh 32 bit so với 24 bit trong 8086). Nó có hai bus nội bộ chính, nhưng thay vì cung cấp các chuyển nguồn và đích chung chung như 8086, 68000 có một hệ thống phức tạp hơn nhiều: khoảng hai tá trường vi mã kết nối các thanh ghi và các thành phần khác với bus theo nhiều cách khác nhau.¹⁵

kết luận

Bên trong, 8086 đại diện cho các thanh ghi có số 5 bit. Điều này là bất thường so với các bộ vi xử lý trước đây, vốn thường chọn các thanh ghi trực tiếp từ mạch lệnh hoặc mạch điều khiển. Ba yếu tố đã thúc đẩy thiết kế này trong 8086. Đầu tiên, nó sử dụng vi mã, do đó, một phương pháp thống nhất để chỉ định các thanh ghi (cả bên trong và bên trong lập trình viên) là hữu ích. Thứ hai, khả năng hoán đổi nguồn và đích trong một lệnh đã thúc đẩy mức độ gián tiếp trong đặc tả thanh ghi, được cung cấp bởi thanh ghi M và N. Cuối cùng, tính linh hoạt của byte ModR/M, cụ thể là hỗ trợ các thanh ghi byte, từ và đoạn, có nghĩa là đặc tả thanh ghi cần 5 bit.

Cho đến nay, tôi đã viết nhiều bài đăng về 8086 và dự định tiếp tục thiết kế ngược khuôn 8086, vì vậy hãy theo dõi tôi trên Twitter [@kenshirriff](#) hoặc RSS để cập nhật. Tôi cũng đã bắt đầu thử nghiệm với Mastodon gần đây với tên [@oldbytes.space@kenshirriff](#).

Ghi chú và tài liệu tham khảo

1. Bên cạnh đó, NOP hướng dẫn (không hoạt động) trong 8086 thực sự là XCHG AX, AX. Trao đổi AX thanh ghi

với chính nó không làm được gì ngoài việc mất 3 chu kỳ đồng hồ. ↵

2. Phần hành động của vi lệnh biểu thị vi lệnh cuối cùng thứ hai (NXT, next) và cuối cùng (RNI, Run Next Instruction), vì vậy việc thực thi lệnh máy tiếp theo có thể bắt đầu. ↵
3. Lưu ý rằng thanh ghi #18 có thể tham chiếu đến cả thanh ghi đích DH và thanh ghi đích. Điều này không gây ra xung đột vì nó đề cập đến DH nếu được tải từ lệnh và đề cập đến thanh ghi đích nếu được chỉ định trong lệnh vi mô. Vấn đề duy nhất là một lệnh vi mô không thể đề cập đến thanh DH với một cách rõ ràng (hoặc BH thanh ghi tương tự). Hạn chế này không phải là vấn đề vì vi mô không bao giờ cần thực hiện việc này. ↵
4. Tôi thảo luận chi tiết về byte ModR/M của 8086 [tại đây](#). ↵
5. Danh sách vi mã dựa trên [sự tháo gỡ](#) của Andrew Jenner. Tôi đã thực hiện một số sửa đổi để (hy vọng) làm cho nó dễ hiểu hơn. ↵
6. Có một vài hướng dẫn chỉ định một thanh ghi trong opcode chứ không phải byte ModR/M. Đối với các thanh ghi 16 bit, các INCI lệnh DEC, XCHG, PUSH, và POP chỉ định thanh ghi trong ba bit thấp của mã lệnh. Các MOV hướng dẫn ngay lập tức chỉ định thanh ghi 8 bit hoặc 16 bit trong ba bit thấp. Mặt khác, đoạn được chỉ định bởi bit 3 và 4 của tiền tố đoạn, PUSH, và POP hướng dẫn. ↵
7. Một số hướng dẫn chỉ có phiên bản byte (DAA, AAA, DAS, AAS, AAM, AAD, XLAT). Điều này được biểu thị bằng đầu ra ROM Giải mã Nhóm và buộc thực thi lệnh vào chế độ byte. Do đó, các hướng dẫn này sẽ tải một thanh ghi byte vào N, nhưng vì các hướng dẫn này không có đặc tả thanh ghi nên điều đó không thành vấn đề. ↵
8. Các tiền tố phân đoạn sử dụng cùng các bit lệnh (3 và 4) như PUSH và POP để chọn thanh ghi phân đoạn, vì vậy bạn có thể mong đợi các tiền tố cũng tải N thanh ghi. Tuy nhiên, các tiền tố được triển khai trong phần cứng, thay vì vi mã. Vì vậy, chúng không sử dụng N thanh ghi và N thanh ghi không được nạp số thanh ghi đoạn. ↵
9. Bạn có thể thắc mắc tại sao BX thanh ghi này không theo thứ tự với các thanh ghi khác, cả về mặt vật lý trên chip

và trong tập lệnh. 8086 được thiết kế sao cho mã hợp ngữ 8080 có thể dịch sang mã 8086. Ban đầu, các thanh ghi 8086 có các tên khác nhau: XA, BC, DE, HL, SP, MP, IJ, và IK. Bốn tên đầu tiên khớp với các thanh ghi trong bộ xử lý Intel 8080, trong khi đó MP là Con trỏ bộ nhớ và IJ và IK là các thanh ghi Chỉ mục. Tuy nhiên, khi 8086 được phát hành, các thanh ghi được đặt tên tương ứng với chức năng của chúng trong 8086, loại bỏ tên 8080. XA đã trở thành Accumulator AX, BC thanh ghi đã được sử dụng để đếm, vì vậy nó đã trở thành thanh ghi Đếm CX. Thanh DE ghi là một thanh ghi dữ liệu, vì vậy nó trở thành Thanh ghi dữ liệu DX. Thanh HL ghi được sử dụng làm cơ sở để truy cập bộ nhớ, vì vậy nó trở thành Thanh ghi cơ sở BX. Kết quả là BX đã ký kết thúc cuối cùng.

Một chương trình CONV-86 cho phép dịch các chương trình hợp ngữ 8080 thành các chương trình hợp ngữ 8086, với các thanh ghi 8080 được thay thế bằng các thanh ghi 8086 tương ứng. Các tên đăng ký cũ của 8086 có thể được nhìn thấy trong bảng [sáng chế 8086](#), trong khi các tên Accumulator, Base, Count, Data nằm trong [MCS-86 Hướng dẫn tham khảo hợp ngữ](#). Xem thêm [cuộc thảo luận Stack Exchange này](#).

10. Nguồn tất cả không có bất kỳ giải mã nào; bus ALU được nạp trước ở trạng thái cao, do đó, theo mặc định, tất cả đều ở trạng thái cao.
11. Tất nhiên, hệ thống sử dụng các trường bit trong hướng dẫn để chọn các thanh ghi đã cũ hơn nhiều. Ví dụ, kiến trúc IBM System/360 đột phá (1964) đã sử dụng các trường 4 bit trong hướng dẫn để chọn một trong 16 thanh ghi mục đích chung.
12. Lưu ý rằng với cách bố trí lệnh này, tập lệnh ánh xạ hoàn toàn vào hệ bát phân. Datapoint 2200 đã sử dụng hệ bát phân để mô tả tập lệnh, nhưng Intel đã chuyển sang hệ thập lục phân cho các bộ xử lý của mình. Hệ thập lục phân đang trở nên phổ biến hơn bát phân vào thời điểm đó, nhưng việc chuyển sang hệ thập lục phân che giấu hầu hết cấu trúc vốn có của hướng dẫn. Xem [x86 là một máy bát phân](#) để biết chi tiết.
13. Nhà sản xuất Datapoint đã nói chuyện với Intel và Texas Instruments về việc thay thế bảng chip bằng một chip xử lý duy nhất. Texas Instruments sản xuất chip vi xử lý TMX 1795 và Intel sản xuất 8008 ngay sau đó, cả hai đều sao chép kiến trúc và tập lệnh của Datapoint 2200. Datapoint không thích hiệu suất của những con chip này

và quyết định gắn bó với bộ xử lý dựa trên TTL. Texas Instruments không thể tìm được khách hàng cho TMX 1795 và đã từ bỏ nó. Mặt khác, Intel đã bán 8008 dưới dạng bộ vi xử lý 8 bit, tạo ra thị trường bộ vi xử lý trong quá trình này. Việc lựa chọn thanh ghi trong các bộ xử lý này khá đơn giản: 3 bit lệnh được giải mã thành 8 dòng chọn để chọn thanh ghi (hoặc bộ nhớ) thích hợp. Vì các bộ xử lý này có điều khiển được mã hóa cứng thay vì vi mã. ↵

14. Trong khi 8086 có tám thanh ghi có thể được xem như mục đích chung, tất cả chúng đều có một số mục đích cụ thể. Thanh AXghi đóng vai trò là bộ tích lũy và có một số chức năng đặc biệt, chẳng hạn như sử dụng nó trong XCHGthao tác (Exchange), thao tác I/O, phép nhân và phép chia. Thanh BXghi có vai trò đặc biệt là thanh ghi cơ sở để truy cập bộ nhớ. Thanh CXghi đóng vai trò là bộ đếm cho các thao tác chuỗi và cho lệnh JCXZ(Jump if CXZero). Thanh DXghi có thể chỉ định cồng cho các hoạt động I/O và được sử dụng cho CWD(Chuyển từ thành từ kép) và trong phép nhân và phép chia. Thanh SPghi có một vai trò duy nhất là con trỏ ngăn xếp. Các SIvàDIcác thanh ghi được sử dụng làm thanh ghi chỉ mục cho các thao tác chuỗi và truy cập bộ nhớ. Cuối cùng, BPthanh ghi có vai trò duy nhất là con trỏ cơ sở vào phân đoạn ngăn xếp. Ở phía 8 bit, AX, BX, CX, và DXcó thể được truy cập dưới dạng các thanh ghi 8 bit, trong khi các thanh ghi khác thì không thể. Thanh ghi 8 bit ALđược sử dụng riêng cho XLAT(Dịch) trong khi AHđược sử dụng cho các hoạt động cờ LAHFvà SAHF. Do đó, các thanh ghi của 8086 không hoàn toàn trực giao và mỗi thanh ghi có một số trường hợp đặc biệt, thường là vì các lý do lịch sử. ↵
15. Một cách khác để xem xét vi mã của Motorola 68000 là các điều khiển thanh ghi đến từ vi mã "ngang", một vi lệnh có nhiều bit và các trường điều khiển trực tiếp các phần tử chức năng. Vi mã của 8086 "dọc" hơn; các lệnh vi mô có tương đối ít bit và các trường được mã hóa cao. Đặc biệt, các trường thanh ghi nguồn và đích của 8086 được mã hóa cao, trong khi 68000 có các trường điều khiển kết nối của một thanh ghi riêng lẻ với bus. ↵

2 nhận xét:



Nhãn: [8086](#) , [chip](#) , [vi mã](#) , [kỹ thuật đảo ngược](#)

