

HỌC VIỆN KỸ THUẬT MẬT MÃ

**KHOA CÔNG NGHỆ THÔNG TIN**

¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯

TỐI ƯU PHẦN MỀM NHÚNG

**TỐI ƯU KÍCH THƯỚC CHƯƠNG TRÌNH.**

**TỐI ƯU TỐI THIỂU 20 CHƯƠNG TRÌNH C, 20 CHƯƠNG TRÌNH HỢP NGỮ 80X86**

**VÀ 20 CHƯƠNG TRÌNH HỢP NGỮ ARM**

Ngành: Công nghệ thông tin

Chuyên ngành: Kỹ thuật phần mềm nhúng

Sinh viên thực hiện Cao Thị Thùy Linh CT020328

Nguyễn Trung Kiên

Giảng viên: **Phạm Văn Hưởng**

Khoa Công nghệ thông tin – Học viện Kỹ thuật mật mã

Hà Nội, 2021



**NHẬN XÉT VÀ CHO ĐIỂM CỦA GIÁO VIÊN**

**MỤC LỤC**

[1.1 Chọn lệnh ngắn hơn 8](#_Toc91742495)

[1.2 Sử dụng các hằng số và địa chỉ ngắn hơn 10](#_Toc91742496)

[1.3 Sử dụng lại hằng số 12](#_Toc91742497)

[1.4 Hằng số ở chế độ 64-bit 13](#_Toc91742498)

[1.5 Địa chỉ và con trỏ ở chế độ 64 bit 13](#_Toc91742499)

[1.6 Thực hiện lệnh lâu hơn vì lợi ích của việc căn chỉnh 16](#_Toc91742500)

[1.7 Demo 20](#_Toc91742501)

[**CHƯƠNG 2: TỐI ƯU MÃ C** 40](#_Toc91742502)

[2.1. Thời gian thử nghiệm trung bình của các chương trình 40](#_Toc91742503)

[2.2. Bảng thống kê kết quả mã C 41](#_Toc91742504)

[CHƯƠNG 3: TỐI ƯU MÃ HỢP NGỮ 42](#_Toc91742505)

[3.1. Mã 8086 42](#_Toc91742506)

[3.1.1. Thời gian thử nghiệm trung bình của các chương trình 8086 42](#_Toc91742507)

[3.1.2. Bảng thống kê kết quả mã 8086 43](#_Toc91742508)

[3.2. Mã arm 44](#_Toc91742509)

[3.2.1. Thời gian thử nghiệm trung bình của các chương trình arm 44](#_Toc91742510)

[3.2.2. Bảng thống kê kết quả mã arm 45](#_Toc91742511)

[**KẾT LUẬN** 46](#_Toc91742512)

[**TÀI LIỆU THAM KHẢO** 47](#_Toc91742513)

**DANH MỤC HÌNH ẢNH**

**DANH MỤC BẢNG**

**LỜI NÓI ĐẦU**

.

*Hà Nội, ngày tháng năm 2021*

*Sinh viên thực hiện*

# CHƯƠNG 1: CƠ SỞ LÝ THUYẾT: Tối ưu hóa kích thước

Bộ đệm ẩn mã có thể chứa từ 8 đến 64 kb mã. Nếu có vấn đề khi giữ vòng lặp quan trọng bên trong của mã trong bộ đệm mã, bộ đệm pop hoặc bộ đệm lặp lại, thì có thể xem xét giảm kích thước của mã. Giảm kích thước mã cũng có thể cải thiện việc giải mã các lệnh. Các vòng lặp rất nhỏ đặc biệt nhanh vì chúng có thể vừa với bộ đệm lặp lại hoặc bộ giải mã có thể sử dụng lại các dòng bộ đệm mã đã tải.

Đôi lúc khi không quan trọng về tốc độ, ta có thể ưu tiên về giảm tải kích cỡ mã dù cho nó gây ra giảm tốc độ thực thi. Mã 64 bit không cần nhiều byte cho địa chỉ hơn mã 32 bit vì nó có thể sử dụng địa chỉ tương đối RIP 32-bit. Mã 64-bit có thể lớn hơn một chút so với mã 32-bit do có tiền tố REX và các khác biệt nhỏ khác, nhưng nó cũng có thể nhỏ hơn mã 32-bit vì số lượng thanh ghi tăng lên làm giảm nhu cầu về các biến bộ nhớ.

## 1.1 Chọn lệnh ngắn hơn

Một số lệnh có hình thức ngắn. Các lệnh PUSH và POP với một thanh ghi số nguyên chỉ chiếm một byte. XCHG EAX, reg32 cũng là một lệnh byte đơn và do đó chiếm ít không gian hơn lệnh MOV, nhưng XCHG chậm hơn MOV. INC và DEC với thanh ghi 32 bit ở chế độ 32 bit. Dạng ngắn gọn của INC và DEC không khả dụng ở chế độ 64 bit.

Các lệnh sau mất ít hơn một byte khi sử dụng bộ tích lũy so với khi sử dụng bất kỳ thanh ghi nào khác: ADD, ADC, SUB, SBB, AND, OR, XOR, CMP, TEST với toán hạng tức thì không có phần mở rộng dấu. Điều này cũng áp dụng cho lệnh MOV với toán hạng bộ nhớ và không có thanh ghi con trỏ ở chế độ 16 và 32 bit, nhưng không áp dụng ở chế độ 64 bit. Ví dụ:

Graphical user interface, application

Description automatically generated

Các lệnh với con trỏ chiếm ít hơn một byte khi chúng chỉ có con trỏ cơ sở (ngoại trừ ESP, RSP hoặc R12) và sự dịch chuyển so với khi chúng có thanh ghi chỉ mục được chia tỷ lệ hoặc cả con trỏ cơ sở và thanh ghi chỉ mục, hoặc ESP, RSP hoặc R12 làm cơ sở con trỏ. Ví dụ:

Text

Description automatically generated with medium confidence

Các lệnh với EBP, RBP hoặc R13 làm con trỏ cơ sở và không có dịch chuyển và không có chỉ mục nào chiếm nhiều hơn một byte so với các thanh ghi khác:

Text

Description automatically generated

Các lệnh có con trỏ chỉ mục được chia tỷ lệ và không có con trỏ cơ sở phải có độ dịch chuyển bốn byte, ngay cả khi nó bằng 0:



lệnh ở chế độ 64 bit cần tiền tố REX nếu sử dụng ít nhất một trong các thanh ghi R8 - R15 hoặc XMM8 - XMM15. Do đó, các lệnh sử dụng các thanh ghi này dài hơn một byte so với các lệnh sử dụng các thanh ghi khác, trừ khi dù sao thì tiền tố REX cũng là cần thiết vì những lý do khác:

Graphical user interface, text, application

Description automatically generated

Trong ví dụ 10.5a, có thể tránh tiền tố REX bằng cách sử dụng thanh ghi RBX thay vì R8 làm con trỏ.

Nhưng trong ví dụ 10.5b, dù sao cũng cần một tiền tố REX cho kích thước toán hạng 64-bit và lệnh không thể có nhiều hơn một tiền tố REX.

Các tính toán điểm động có thể được thực hiện với các lệnh kiểu x87 cũ với các thanh ghi ngăn xếp dấu chấm động ST (0) - ST (7) hoặc các lệnh kiểu SSE mới hơn với các thanh ghi XMM. Các lệnh kiểu x87 nhỏ gọn hơn so với lệnh sau, ví dụ:

Graphical user interface, text, application

Description automatically generated

Việc sử dụng mã kiểu x87 có thể có lợi ngay cả khi nó yêu cầu thêm lệnh FXCH. Chỉ có sự khác biệt nhỏ về tốc độ thực thi giữa hai loại lệnh dấu phẩy động trên các bộ xử lý hiện tại. Tuy nhiên, có khả năng là các lệnh kiểu x87 sẽ bị coi là lỗi thời và sẽ kém hiệu quả hơn trên các bộ vi xử lý trong tương lai.

Bộ xử lý hỗ trợ tập lệnh AVX có thể mã lệnh XMM theo hai cách khác nhau, với tiền tố VEX hoặc với tiền tố cũ. Đôi khi phiên bản VEX ngắn hơn và đôi khi phiên bản cũ ngắn hơn. Tuy nhiên, có một hình phạt hiệu suất nghiêm trọng khi trộn các lệnh XMM không có tiền tố VEX với các lệnh sử dụng thanh ghi YMM trên hầu hết các bộ xử lý Intel.

Tập lệnh AVX-512 sử dụng tiền tố 4 byte mới được gọi là EVEX. Mặc dù tiền tố EVEX dài hơn một hoặc hai byte so với tiền tố VEX, nó cho phép mã hóa các toán hạng bộ nhớ hiệu quả hơn với con trỏ và bù đắp. Các toán hạng bộ nhớ có độ lệch 4 byte đôi khi có thể được thay thế bằng độ lệch tỷ lệ 1 byte khi tiền tố EVEX được sử dụng. Do đó tổng chiều dài lệnh trở nên nhỏ hơn.

## 1.2 Sử dụng các hằng số và địa chỉ ngắn hơn

Nhiều địa chỉ bước nhảy, địa chỉ dữ liệu và hằng số dữ liệu có thể được biểu thị dưới dạng hằng số 8 bit mở rộng dấu hiệu. Điều này giúp tiết kiệm rất nhiều không gian. Chỉ có thể sử dụng byte mở rộng dấu hiệu nếu giá trị nằm trong khoảng từ -128 đến +127.

Đối với địa chỉ bước nhảy, điều này có nghĩa là các bước nhảy ngắn mất hai byte mã, trong khi các bước nhảy vượt quá 127 byte sẽ mất 5 byte nếu không có điều kiện và 6 byte nếu có điều kiện.

Tương tự như vậy, địa chỉ dữ liệu chiếm ít không gian hơn nếu chúng có thể được biểu thị dưới dạng một con trỏ và sự dịch chuyển giữa -128 và +127. Ví dụ sau giả định rằng [mem1] và [mem2] là địa chỉ bộ nhớ tĩnh trong phân đoạn dữ liệu và khoảng cách giữa chúng nhỏ hơn 128 byte:

A picture containing text

Description automatically generated

Giảm xuống:

Graphical user interface

Description automatically generated with low confidence

Ở chế độ 64-bit, cần thay thế mov eax, offset mem1 bằng lea rax, [mem1], dài hơn một byte. Lợi thế của việc sử dụng một con trỏ rõ ràng sẽ tăng lên nếu có thể sử dụng cùng một con trỏ nhiều lần. Do đó, việc lưu trữ dữ liệu trên ngăn xếp và sử dụng EBP hoặc ESP làm con trỏ sẽ làm cho mã nhỏ hơn so với việc sử dụng các vị trí bộ nhớ tĩnh và địa chỉ tuyệt đối, với điều kiện là dữ liệu nằm trong phạm vi +/- 127 byte của con trỏ. Sử dụng PUSH và POP để ghi và đọc dữ liệu số nguyên tạm thời thậm chí còn ngắn hơn.

Hằng số dữ liệu cũng có thể chiếm ít dung lượng hơn nếu chúng nằm trong khoảng từ -128 đến +127. Hầu hết các lệnh với toán hạng ngay lập tức có dạng ngắn trong đó toán hạng là một byte đơn mở rộng dấu hiệu. Ví dụ:

Text

Description automatically generated

Các lệnh duy nhất có toán hạng tức thì không có dạng rút gọn với hằng số 8 bit có dấu mở rộng là MOV, TEST, CALL và RET. Một lệnh TEST với toán hạng tức thời 32 bit có thể được thay thế bằng các lựa chọn thay thế ngắn hơn khác nhau, tùy thuộc vào logic trong trường hợp. Vài ví dụ:

Text, table

Description automatically generated

Không nên sử dụng các phiên bản có hằng số 16 bit ở chế độ 32 bit hoặc 64 bit, chẳng hạn như TEST AX, 800H vì nó sẽ gây ra hình phạt cho việc giải mã tiền tố thay đổi độ dài trên một số bộ xử lý, như được giải thích trong sách lệnh 3: "Vi kiến ​​trúc của CPU Intel, AMD và VIA".

Các lựa chọn thay thế ngắn hơn cho thanh ghi MOV, hằng số thường hữu ích. Ví dụ:

Graphical user interface, text, application

Description automatically generated

cũng có thể xem xét giảm kích thước của dữ liệu tĩnh. Rõ ràng, một mảng có thể được làm nhỏ hơn bằng cách sử dụng kích thước dữ liệu nhỏ hơn cho các phần tử. Ví dụ: số nguyên 16 bit thay vì số nguyên 32 bit nếu dữ liệu chắc chắn phù hợp với kích thước dữ liệu nhỏ hơn. Mã để truy cập số nguyên 16 bit lớn hơn một chút so với để truy cập số nguyên 32 bit, nhưng sự gia tăng kích thước mã là nhỏ so với sự giảm kích thước dữ liệu cho một mảng lớn. Nên tránh các lệnh có toán hạng dữ liệu tức thời 16 bit ở chế độ 32 bit và 64 bit vì có vấn đề với các tiền tố thay đổi độ dài giải mã.

## 1.3 Sử dụng lại hằng số

Nếu cùng một địa chỉ hoặc hằng số được sử dụng nhiều lần thì có thể tải nó vào một thanh ghi. Một MOV với toán hạng tức thời 4 byte đôi khi có thể được thay thế bằng một lệnh số học nếu giá trị của thanh ghi trước MOV được biết đến. Thí dụ:

Text

Description automatically generated with low confidence

Background pattern

Description automatically generated with low confidence

Graphical user interface, text, application

Description automatically generated

## 1.4 Hằng số ở chế độ 64-bit

Ở chế độ 64-bit, có ba cách để di chuyển một hằng số vào thanh ghi 64-bit: với hằng số 64-bit , với hằng số mở rộng dấu 32-bit và với hằng số mở rộng bằng 0 32-bit:

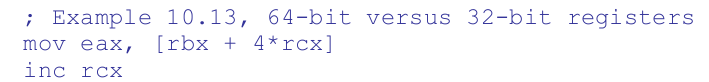
Table

Description automatically generated with low confidence

Một số nhà lắp ráp sử dụng phiên bản mở rộng dấu hiệu thay vì phiên bản mở rộng bằng 0 ngắn hơn, ngay cả khi hằng số nằm trong phạm vi phù hợp với hằng số không mở rộng. có thể buộc trình hợp dịch sử dụng phiên bản không mở rộng bằng cách chỉ định một thanh ghi đích 32-bit. Các ghi vào thanh ghi 32-bit luôn không được mở rộng vào thanh ghi 64-bit.

## 1.5 Địa chỉ và con trỏ ở chế độ 64 bit

Mã 64-bit tốt hơn nên sử dụng kích thước thanh ghi 64-bit cho cơ sở và chỉ mục trong địa chỉ và kích thước thanh ghi 32-bit cho mọi thứ khác. Thí dụ:



Tại đây, có thể tiết kiệm một byte bằng cách thay đổi inc rcx thành inc ecx. Điều này sẽ hoạt động vì giá trị của thanh ghi chỉ mục chắc chắn phải nhỏ hơn 2 32. Tuy nhiên, con trỏ cơ sở có thể lớn hơn 2 32 trong một số hệ thống, vì vậy không thể thay thế thêm rbx, 4 bằng cách thêm ebx, 4. Không bao giờ sử dụng thanh ghi 32-bit làm cơ sở hoặc chỉ mục bên trong dấu ngoặc vuông ở chế độ 64-bit. Quy tắc sử dụng thanh ghi 64 bit bên trong dấu ngoặc vuông của địa chỉ gián tiếp và thanh ghi 32 bit ở mọi nơi khác cũng áp dụng cho lệnh LEA. Ví dụ:

Graphical user interface, text, application

Description automatically generated

Biểu mẫu có đích 32 bit và địa chỉ 64 bit được ưu tiên trừ khi cần kết quả 64 bit. Phiên bản này không mất nhiều thời gian để thực thi hơn phiên bản có đích 64-bit.

Không bao giờ được sử dụng các biểu mẫu có tiền tố kích thước địa chỉ. Một mảng con trỏ 64 bit trong chương trình 64 bit có thể được làm nhỏ hơn bằng cách sử dụng con trỏ 32 bit liên quan đến cơ sở hình ảnh hoặc đến một số điểm tham chiếu. Điều này làm cho mảng con trỏ nhỏ hơn với chi phí làm cho mã sử dụng con trỏ lớn hơn vì nó cần thêm cơ sở hình ảnh. Điều này có mang lại lợi thế ròng hay không phụ thuộc vào kích thước của mảng.

Thí dụ:

Graphical user interface, text, application

Description automatically generatedViệc triển khai với con trỏ hình ảnh tương đối chỉ khả dụng trong Windows:

Timeline

Description automatically generatedMột khả năng khác là sử dụng chính bảng nhảy làm điểm tham chiếu. Phương pháp này thường được sử dụng bởi trình biên dịch cho hệ điều hành Mac OS, nhưng nó cũng có thể được sử dụng trong các hệ thống 64-bit khác. Lưu ý rằng không phải tất cả các trình lắp ráp đều có thể tạo tham chiếu tương đối từ phần dữ liệu đến phần mã. Ví dụ này hoạt động với trình hợp dịch NASM.

Text, timeline

Description automatically generated with medium confidenceMột giải pháp thay thế đơn giản là sử dụng con trỏ tuyệt đối 32 bit. Phương pháp này chỉ có thể được sử dụng nếu chắc chắn rằng tất cả các địa chỉ đều nhỏ hơn 2 31:

Graphical user interface, text, application

Description automatically generatedTrong ví dụ 10.15d, địa chỉ của JumpTable là địa chỉ có thể định vị lại 32 bit được ký hiệu mở rộng thành 64 bit. Điều này hoạt động nếu địa chỉ nhỏ hơn 2 31. Các địa chỉ của Label1, v.v., không được mở rộng, vì vậy điều này sẽ hoạt động nếu các địa chỉ nhỏ hơn 2 32. Phương pháp của ví dụ 10.15d có thể được sử dụng nếu chắc chắn rằng cơ sở hình ảnh cộng với kích thước chương trình nhỏ hơn 2 31. Điều này sẽ hoạt động trong các chương trình ứng dụng trong Linux và BSD và trong một số trường hợp trong Windows, nhưng không hoạt động trong Mac OS X (xem trang 23).

Thậm chí có thể thay thế các con trỏ 64 bit hoặc 32 bit bằng các hiệu số 16 bit liên quan đến điểm tham chiếu phù hợp:

Text

Description automatically generated with medium confidenceVí dụ 10.15d sử dụng Label1 làm điểm tham chiếu. Nó chỉ hoạt động nếu tất cả các nhãn nằm trong khoảng thời gian Nhãn1  2 15. Bảng chứa các hiệu số 16 bit được mở rộng dấu hiệu và được thêm vào điểm tham chiếu.

Các ví dụ trên cho thấy các phương pháp khác nhau để lưu trữ con trỏ mã. Các phương pháp tương tự có thể được sử dụng cho con trỏ dữ liệu. Một con trỏ có thể được lưu trữ dưới dạng địa chỉ tuyệt đối 64 bit, địa chỉ tương đối 32 bit, địa chỉ tuyệt đối 32 bit hoặc độ lệch 8 bit hoặc 16 bit so với điểm tham chiếu thích hợp. Các phương pháp sử dụng con trỏ liên quan đến cơ sở hình ảnh hoặc một điểm tham chiếu chỉ có giá trị là mã bổ sung nếu có nhiều con trỏ. Điều này thường xảy ra trong các câu lệnh switch lớn và trong danh sách được liên kết.

## 1.6 Thực hiện lệnh lâu hơn vì lợi ích của việc căn chỉnh

Có những tình huống có thể thuận lợi nếu đảo ngược lời khuyên của các đoạn trước để làm cho lệnh dài hơn. Quan trọng nhất là trường hợp mục nhập vòng lặp cần được căn chỉnh (xem trang 85). Thay vì chèn các NOP để căn chỉnh nhãn mục nhập vòng lặp, có thể làm cho các lệnh trước dài hơn độ dài tối thiểu của chúng theo cách sao cho mục nhập vòng lặp trở nên căn chỉnh chính xác. Các phiên bản dài hơn của lệnh không mất nhiều thời gian hơn để thực thi, vì vậy chúng tôi có thể tiết kiệm thời gian thực hiện các NOP.

Trình hợp dịch thông thường sẽ chọn dạng lệnh ngắn nhất có thể. Thường có thể chọn một hình thức dài hơn của cùng một lệnh hoặc tương đương. Điều này có thể được thực hiện theo một số cách.

Sử dụng dạng tổng quát thay vì dạng ngắn của lệnh Các dạng ngắn của INC, DEC, PUSH, POP, XCHG, ADD, MOV không có byte mod-reg-r / m (xem trang 25). Các lệnh tương tự có thể được mã hóa ở dạng chung với một byte mod-reg-r / m.

Ví dụ:

Text

Description automatically generated

Sử dụng một lệnh tương đương dài hơn

Ví dụ:

Graphical user interface, text, application

Description automatically generated with medium confidence

Sử dụng toán hạng 4 byte ngay lập tức

Có thể thay thế các lệnh có toán hạng tức thì 8 bit mở rộng bằng dấu hiệu bằng phiên bản có toán hạng tức thì 32 bit:

Graphical user interface, text, application

Description automatically generatedThêm độ dời 0 vào con trỏ

Một lệnh với con trỏ có thể có độ dịch chuyển 1 hoặc 4 byte ở chế độ 32 bit hoặc 64 bit (1 hoặc 2 byte ở chế độ 16 bit). Một chuyển vị giả bằng 0 có thể được sử dụng để làm cho lệnh dài hơn:

Text, letter

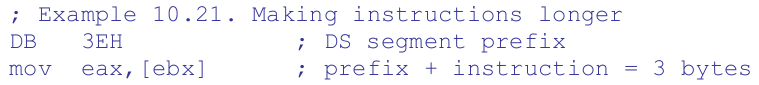
Description automatically generatedĐiều tương tự có thể được thực hiện với LEA EAX, [EBX + 0] thay thế cho MOV EAX, EBX. Sử dụng byte SIB

Một lệnh có toán hạng bộ nhớ có thể có một byte SIB (xem trang 25). Một byte SIB có thể được thêm vào một lệnh chưa có để làm cho lệnh dài hơn một byte. Một byte SIB không thể được sử dụng ở chế độ 16 bit hoặc ở chế độ 64 bit với địa chỉ tương đối RIP. Thí dụ:

A picture containing text

Description automatically generatedSử dụng tiền tố

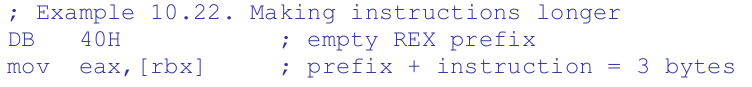
Một cách dễ dàng để tạo một chỉ dẫn dài hơn là thêm các tiền tố không cần thiết. Tất cả các lệnh có toán hạng bộ nhớ có thể có tiền tố phân đoạn. Tiền tố phân đoạn DS hiếm khi cần thiết, nhưng nó có thể được thêm vào mà không làm thay đổi ý nghĩa của lệnh:

Tất cả các lệnh có toán hạng bộ nhớ có thể có tiền tố phân đoạn, bao gồm cả LEA. Thực sự có thể thêm tiền tố phân đoạn vào các lệnh mà không cần toán hạng bộ nhớ.

Các tiền tố vô nghĩa như vậy chỉ đơn giản là bị bỏ qua. Nhưng không có gì đảm bảo tuyệt đối rằng tiền tố vô nghĩa sẽ không có một số ý nghĩa đối với các bộ xử lý trong tương lai. Ví dụ: Pentium 4 đã sử dụng các tiền tố phân đoạn trên các lệnh rẽ nhánh làm gợi ý dự đoán rẽ nhánh. Tôi có thể nói rằng xác suất rất thấp là các tiền tố phân đoạn sẽ có bất kỳ ảnh hưởng xấu nào đến các bộ xử lý trong tương lai đối với các lệnh có thể có toán hạng bộ nhớ, tức là các lệnh có byte mod-reg-r / m.

Các tiền tố phân đoạn CS, DS, ES và SS không có hiệu lực ở chế độ 64-bit, nhưng chúng vẫn được phép, theo lệnh lập trình viên kiến ​​trúc AMD64, Tập 3: Mục đích chung và lệnh Hệ thống, 2003.

Ở chế độ 64-bit, cũng có thể sử dụng tiền tố REX trống để thực hiện lệnh dài hơn:

Các tiền tố REX trống có thể được áp dụng một cách an toàn cho hầu hết các lệnh ở chế độ 64-bit chưa có tiền tố REX, ngoại trừ các lệnh sử dụng AH, BH, CH hoặc DH. Tiền tố REX không thể được sử dụng ở chế độ 32 bit hoặc 16 bit. Tiền tố REX phải đứng sau bất kỳ tiền tố nào khác và không có lệnh nào có thể có nhiều hơn một tiền tố REX. Sách lệnh tối ưu hóa của AMD khuyến nghị sử dụng tối đa ba tiền tố kích thước toán hạng (66H) làm bộ đệm. Nhưng tiền tố này chỉ có thể được sử dụng trên các lệnh không bị ảnh hưởng bởi tiền tố này, tức là các lệnh dấu phẩy động NOP và x87. Tiền tố phân đoạn được áp dụng rộng rãi hơn và có cùng tác dụng - hay đúng hơn là không có tác dụng.

Có thể thêm nhiều tiền tố giống hệt nhau vào bất kỳ lệnh nào miễn là tổng độ dài của lệnh không vượt quá 15. Ví dụ, có thể có một lệnh với hai hoặc ba tiền tố đoạn DS. Nhưng các lệnh có nhiều tiền tố sẽ mất thêm thời gian để giải mã trên nhiều bộ xử lý. không nên sử dụng tiền tố kích thước địa chỉ làm bộ đệm vì điều này có thể làm chậm quá trình giải mã lệnh.

Không đặt tiền tố giả ngay trước nhãn nhảy để căn chỉnh nó:

Text

Description automatically generated with medium confidenceTrong ví dụ này, lệnh MOV EAX, [ESI] sẽ được giải mã bằng tiền tố đoạn DS khi đến từ L1, nhưng không có tiền tố khi đến từ L2. Về nguyên tắc, điều này hoạt động, nhưng một số bộ vi xử lý nhớ vị trí của ranh giới lệnh và các bộ xử lý như vậy sẽ bị nhầm lẫn khi cùng một lệnh bắt đầu ở hai vị trí khác nhau. Có thể có một hình phạt thực hiện cho điều này.

Các bộ xử lý hỗ trợ tập lệnh AVX sử dụng tiền tố VEX, dài 2 hoặc 3 byte. Tiền tố VEX 2 byte luôn có thể được thay thế bằng tiền tố VEX 3 byte. Tiền tố VEX có thể được đặt trước tiền tố phân đoạn nhưng không được đặt trước bất kỳ tiền tố nào khác. Không có tiền tố nào khác được phép sau tiền tố VEX. Hầu hết các lệnh sử dụng thanh ghi XMM hoặc YMM có thể có tiền tố VEX. Không trộn các lệnh vectơ YMM với tiền tố VEX và các lệnh XMM không có tiền tố VEX (xem chương 13.1). Một số lệnh mới hơn về thanh ghi mục đích chung cũng sử dụng tiền tố VEX.

Tập lệnh AVX512 sử dụng các tên tiền tố 4 byte EVEX. lệnh có thể được thực hiện lâu hơn bằng cách thay thế tiền tố VEX bằng tiền tố EVEX khi AVX512VL được hỗ trợ.

nên kiểm tra các lệnh được mã hóa bằng tay bằng trình gỡ lỗi hoặc trình tháo gỡ để đảm bảo chúng chính xác.

## 1.7 Demo

|  |
| --- |
| Viết chương trình file.s  Biên dịch:   * Hợp dịch tập tin .s => .o: nasm -f elf32 vd.s =>(32 bit)   nasm -f elf64 vd.s =>(64 bit)   * + Nếu không có lỗi => vd.o: * Liên kết: ld -m elf\_i386 -s -o vd vd.o (32bit)   ld -m elf\_x86\_64 -s -o vd vd.o (64 bit)   * Thực thi ./vd * Check time: time ./vd * Check size: du vd |

VD10.1

|  |  |  |
| --- | --- | --- |
| Làm  Code 1   |  | | --- | | section .text  global \_start    \_start:  add ebx,1000  section .data |   Text  Description automatically generated  Code TU 1   |  | | --- | | section .text  global \_start    \_start:  add eax,1000  section .data |   Text  Description automatically generated  So sánh kích thước: |

VD10.2

|  |  |  |
| --- | --- | --- |
| Text  Description automatically generated  Làm  Code 2   |  | | --- | | section .text  global \_start    \_start:  mov eax,[esp+12]  section .data |   Text  Description automatically generated  Code TU 2   |  | | --- | | section .text  global \_start    \_start:  mov eax,[ebp+12]  section .data |   Text  Description automatically generated with medium confidence  Text  Description automatically generated  So sánh kích thước |

VD10.3

|  |  |  |
| --- | --- | --- |
| Text  Description automatically generated  Làm  Code 1   |  | | --- | | section .text  global \_start    \_start:  mov eax,[ebp]  section .data |   Text  Description automatically generated  Code TU 1   |  | | --- | | section .text  global \_start    \_start:  mov eax,[ebx]  section .data |   Text  Description automatically generated  So sánh kích thước |

VD10.4

|  |  |  |
| --- | --- | --- |
| A screenshot of a computer  Description automatically generated with medium confidence  Làm  Code   |  | | --- | | section .text  global \_start    \_start:  lea eax,[ebx\*2]  section .data |   Text  Description automatically generated  Code TU   |  | | --- | | section .text  global \_start    \_start:  lea eax,[ebx + ebx]  section .data |   Text  Description automatically generated with low confidence  Text  Description automatically generated  So sánh kích thước |

VD10.5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Graphical user interface, text, application  Description automatically generated  Làm  Code a   |  | | --- | | section .text  global \_start  \_start:  mov eax,[r8]  section .data |   Text  Description automatically generated  Code a TU:   |  | | --- | | section .text  global \_start  \_start:  mov eax,[rbx]  section .data |   Text  Description automatically generated  Code b   |  | | --- | | section .text  global \_start  \_start:  mov rax,[r8]  section .data |   Text  Description automatically generated  Code b TU   |  | | --- | | section .text  global \_start  \_start:  mov rax,[rbx]  section .data |   Text  Description automatically generated  Text  Description automatically generated  So sánh kích thước  Text  Description automatically generated |

VD10.6

|  |  |
| --- | --- |
| Text  Description automatically generated  Làm   |  | | --- | | section .text  global \_start  \_start:  fadd st0,st1  addsd xmm0,xmm1  section .data |   Text  Description automatically generated |

VD10.7

|  |  |
| --- | --- |
| Text, letter  Description automatically generated  Làm  Code a   |  | | --- | | section .text  global \_start  \_start:  mov ebx,100  add ebx,200  section .data |   Text  Description automatically generated    Code b |

VD10.8

|  |  |
| --- | --- |
| Text  Description automatically generated  Làm   |  | | --- | | section .text  global \_start  \_start:  push 200  push 100  add ebx, 128  sub ebx, -128  section .data |   Text  Description automatically generated |

VD10.9

|  |  |
| --- | --- |
| Text  Description automatically generated  Làm   |  | | --- | | section .text  global \_start  \_start:  test eax,8  test ebx,8  test al,8  test bl,8  and ebx,8  bt ebx,3  cmp ebx,8  section .data |   Text  Description automatically generated |

VD10.10

|  |  |
| --- | --- |
| Table  Description automatically generated with low confidence  Làm   |  | | --- | | section .text  global \_start  \_start:  mov eax,0  xor eax,eax  mov eax,1  xor eax,eax  push 1    mov eax,-1  or eax,-1  section .data |   Text  Description automatically generated |

VD10.11

|  |
| --- |
| A picture containing text  Description automatically generated  Graphical user interface, text  Description automatically generated  Làm |

VD10.12

|  |  |
| --- | --- |
| Text  Description automatically generated with medium confidence  Làm   |  | | --- | | section .text  global \_start  \_start:  mov rax,123456789abcdef0h  mov rax,-100  mov eax,100  section .data |   Text  Description automatically generated  Text  Description automatically generated |

VD10.13

|  |  |  |
| --- | --- | --- |
| Graphical user interface  Description automatically generated with low confidence  Làm  Code vs 64bit   |  | | --- | | section .text  global \_start  \_start:  mov eax, [rbx + 4\*rcx]  inc rcx  section .data |   Text  Description automatically generated  Text  Description automatically generated    Code vs 32bit   |  | | --- | | section .text  global \_start  \_start:  mov eax, [rbx + 4\*rcx]  inc rcx  section .data |   Error  Text  Description automatically generated |

VD10.14

|  |  |
| --- | --- |
| Text  Description automatically generated  Làm   |  | | --- | | section .text  global \_start  \_start:  lea eax,[ebx+ecx]  lea eax,[rbx+rcx]  lea rax,[ebx+ecx]  lea rax,[rbx+rcx]  section .data |   Text  Description automatically generated |

VD10.15

|  |
| --- |
| Text, timeline  Description automatically generated with medium confidence  Text, timeline  Description automatically generated  Text  Description automatically generated  Làm  Code a |

VD10.16

|  |  |
| --- | --- |
| Text  Description automatically generated  Làm   |  | | --- | | section .text  global \_start  \_start:  inc eax  DB 0FFH, 0C0H  push ebx  DB 0FFH, 0F3H  section .data |   Text  Description automatically generated |

VD10.17

|  |  |
| --- | --- |
| Graphical user interface, text  Description automatically generated  Làm   |  | | --- | | section .text  global \_start  \_start:  inc eax  add eax,1    mov eax,ebx  lea eax, [ebx]  section .data |   Text  Description automatically generated |

VD10.18

|  |  |
| --- | --- |
| Graphical user interface  Description automatically generated with medium confidence  Làm   |  | | --- | | section .text  global \_start  \_start:  add ebx,1  DB 0x81, 0xC3  DD 1  section .data |   Text  Description automatically generated |

VD10.19

|  |  |  |
| --- | --- | --- |
| Text, letter  Description automatically generated  Làm  Code   |  | | --- | | section .text  global \_start  \_start:  mov eax,[ebx]  mov eax,[ebx + 00]  mov eax,[ebx + 00000000]  section .data |   Text  Description automatically generated  Code TU:   |  | | --- | | section .text  global \_start  \_start:  mov eax,[ebx]  DB 0x8B, 0x43, 0x00  DB 0x8B, 0x83,0,0,0,0  section .data |   Text  Description automatically generated  So sánh kích thước |

VD10.20

|  |
| --- |
| Text  Description automatically generated with medium confidence  Làm |

VD10.21

|  |  |
| --- | --- |
| Graphical user interface, text, application  Description automatically generated  Làm   |  | | --- | | section .text  global \_start  \_start:  DB 3EH  mov eax,[ebx]  section .data |   Text  Description automatically generated |

VD20.22

|  |  |
| --- | --- |
| Làm   |  | | --- | | section .text  global \_start  \_start:  DB 40H  mov eax,[rbx]  section .data |   Text  Description automatically generated |

VD20.23

|  |  |
| --- | --- |
| Text  Description automatically generated  Làm   |  | | --- | | section .text  global \_start  \_start:  L1: mov ecx,1000  DB 0x3E  L2: mov eax,[esi]  section .data |   Text  Description automatically generated |

# 

# **CHƯƠNG 2: TỐI ƯU MÃ C**

## 2.1. Các kỹ thuật tối ưu

* Đưa các hàm con ra khỏi hàm main sẽ giảm thiểu được thời gian và kích thước
* Dữ liệu được khởi tạo khi khai báo sẽ giảm thiểu được thời gian và kích thước
* Sắp xếp các biến theo chiều giảm dần của kích thước để tránh thừa nhiều Padding(ô nhớ k được sử dụng) để tối ưu kích thước mã
* Dùng biến cục bộ thay biến toàn cục để tối ưu kích thước
* Nên định nghĩa hàm “Void” để giảm thiểu phần xử lý trong hàm(vì các hàm int, float... đều phải có giá trị trả về)
* Dùng các biến có kiểu dũ liệu nhỏ nếu bài toàn nhỏ để tối ưu kích thước
* Tối ưu thủ công để tối ưu thời gian: Chuyển phép nhân thành cộng, phép chia thành phép nhân
* Dùng khai báo dynamic, dùng xong free để tối ưu kích thước
* Tự động tính kích thước mảng tự động để cấp phát vừa đủ ô nhớ cho mảng:

arr\_size = sizeof(array)/sizeof(array[0]);

* Gcc dùng cờ -Os -flto để tối ưu kích thước:
  + -Os: để giảm thiểu kích thước nhị phân
  + -flto: Cho phép gcc giữ thông tin nguồn bổ sung để thực hiện tối ưu hóa thời gian liên kết chuẩn, liên kết nhiều tệp đối tượng với nhau. Đặc biệt, điều này cho phép loại bỏ mã không sử dụng
* Gcc dùng strip: khi chương trình chạy ở chế độ debug, nó có các hàm, câu lệnh không ảnh hưởng đến chương trình, các câu lệnh lưu hết và in ra quá trình , thông tin tại thời gian thực thi, strip bỏ qua các cái trên.

## 2.2. Thời gian thử nghiệm trung bình của các chương trình



## 2.3. Bảng thống kê kết quả mã C



Code 11 đến 20

* Tối ưu thời gian

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **STT** | **Tên chương trình** | **Kỹ thuật tối ưu thực hiện** | **Ct gốc** | **Ct tối ưu** |
| 11 | Tính tổng bình phương các số lẻ từ 1 đến < n | Rút gọn câu lệnh, thay đổi câu khai báo. | 2.204s | 1.1475s |
| 12 | Tìm max của 3 số a,b,c nhập từ bàn phím | Rút gọn câu lệnh, thay đổi câu khai báo và loại bỏ hàm con. | 2.104 | 1.576 |
| 13 | Nhập một số nguyên từ bàn phím, kiểm tra xem đó là số chẵn hay lẻ | Rút gọn câu lệnh, thay đổi câu khai báo. | 1.021 | 0.592 |
| 14 | Tìm ước số chung lớn nhất và bội số chung nhỏ nhất của 2 số nguyên nhập từ bàn phím | Rút gọn câu lệnh, thay đổi câu khai báo. Ko dùng biến global | 3.499 | 3.165 |
| 15 | Nhập một số nguyên từ bàn phím. Kiểm tra một số có phải là số hoàn hảo? | Rút gọn chương trình, ko dùng hàm con | 1.628 | 1.217 |
| 16 | Nhập vào dãy số tính tổng các số lẻ trong dãy. | Rút gọn lệnh , không dùng hàm con | 3.049 | 2.221 |
| 17 | Nhập vào 1 mảng và in các số nguyên tố trong mảng và vị trí của nó | Rút gọn lệnh , không dùng biến global | 2.013 | 1.462 |
| 18 | Nhập vào 1 dãy số và in ra vị trí lớn nhất | Rút gọn lệnh , không dùng biến global | 2.144 | 1.823 |
| 19 | Nhập vào 1 mảng và xuất mảng đó ra console. | Rút gọn lệnh , không dùng hàm con | 2.401 | 1.666 |
| 20 | Nhập vào 1 mảng và tìm kiếm 1 giá trị bất kỳ trong mảng và in ra. | Rút gọn lệnh , không dùng hàm con | 4.208 | 2.988 |

* Tối ưu kích thước

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **STT** | **Tên chương trình** | **Kỹ thuật tối ưu thực hiện** | **Ct gốc** | **Ct tối ưu** |
| 11 | Tính tổng bình phương các số lẻ từ 1 đến < n | Rút gọn câu lệnh, thay đổi câu khai báo. | 8424 | 8400 |
| 12 | Tìm max của 3 số a,b,c nhập từ bàn phím | Rút gọn câu lệnh, thay đổi câu khai báo và loại bỏ hàm con. | 8480 | 8400 |
| 13 | Nhập một số nguyên từ bàn phím, kiểm tra xem đó là số chẵn hay lẻ | Rút gọn câu lệnh, thay đổi câu khai báo. | 8376 | 8284 |
| 14 | Tìm ước số chung lớn nhất và bội số chung nhỏ nhất của 2 số nguyên nhập từ bàn phím | Rút gọn câu lệnh, thay đổi câu khai báo. Ko dùng biến global | 8448 | 8386 |
| 15 | Nhập một số nguyên từ bàn phím. Kiểm tra một số có phải là số hoàn hảo? | Rút gọn chương trình, ko dùng hàm con | 8496 | 8448 |
| 16 | Nhập vào dãy số tính tổng các số lẻ trong dãy. | Rút gọn lệnh , không dùng hàm con | 8504 | 8400 |
| 17 | Nhập vào 1 mảng và in các số nguyên tố trong mảng và vị trí của nó | Rút gọn lệnh , không dùng biến global | 8472 | 8400 |
| 18 | Nhập vào 1 dãy số và in ra vị trí lớn nhất | Rút gọn lệnh , không dùng biến global | 8528 | 8400 |
| 19 | Nhập vào 1 mảng và xuất mảng đó ra console. | Rút gọn lệnh , không dùng hàm con | 8568 | 8400 |
| 20 | Nhập vào 1 mảng và tìm kiếm 1 giá trị bất kỳ trong mảng và in ra. | Rút gọn lệnh , không dùng hàm con | 8536 | 8408 |

# CHƯƠNG 3: TỐI ƯU MÃ HỢP NGỮ

## 3.1. Mã 8086

### 3.1.1. Các kỹ thuật tối ưu

* Đáng số lại các nhãn .L:
* Thay đổi thanh ghi đăng ký mục đích đặc biệt $rip thành $rbp:
  + $rip: được sử dụng trong con trỏ ,là địa chỉ tương đối toàn cục, Hướng dẫn(Bộ đếm chương trình; được gọi là $pc trong GDB)
  + $rbp: được sử dụng trong con trỏ, nó chỉ vào đáy khung ngăn xếp hiện tại và các biến cục bộ, cơ sở(mục đích chung trong một số chế độ biên dịch)
* Thay đổi địa chỉ đăng ký cfi\_offset, cfi\_def\_cfa\_register: "Địa chỉ khung cuộc gọi": CFI được sử dụng để gỡ lỗi. Nó cho phép trình gỡ lỗi để thư giãn một ngăn xếp.
* Gcc dùng cờ -Os -flto để tối ưu kích thước:
  + -Os: để giảm thiểu kích thước nhị phân
  + -flto: Cho phép gcc giữ thông tin nguồn bổ sung để thực hiện tối ưu hóa thời gian liên kết chuẩn, liên kết nhiều tệp đối tượng với nhau. Đặc biệt, điều này cho phép loại bỏ mã không sử dụng
* Gcc dùng strip: khi chương trình chạy ở chế độ debug, nó có các hàm, câu lệnh không ảnh hưởng đến chương trình, các câu lệnh lưu hết và in ra quá trình , thông tin tại thời gian thực thi, strip bỏ qua các cái trên.

### 3.1.2. Thời gian thử nghiệm trung bình của các chương trình 8086



### 3.1.3. Bảng thống kê kết quả mã 8086



## 3.2. Mã arm

### 3.2.1. Các kỹ thuật tối ưu

* Dùng các giải pháp giải quyết cho ngăn xếp: stmia, ldmia(hậu tố ia(increment after): tăng dần): đẩy(lưu trữ)/ bật(tải) nhiều thanh ghi đến/từ một “trống chồng lên” tức là, con trỏ ngăn xếp trỏ đến từ tự do tiếp theo trên ngăn xếp khi nó phát triển từ địa chỉ thấp đến cao
* Gán địa chỉ gián tiếp: Đưa các thanh ghi vào khung ngăn cục bộ fp phù hợp
  + Sp: Địa chỉ của phần tử được cấp phát gần nhất (tức phần tử được cấp phát cuối cùng cho đến thời điểm hiện tại
  + Fp: dùng làm con trỏ frame
* Giảm các frame, frame\_needed
* Thay đổi vị trí các thành địa chỉ LC
* Đáng số lại các nhãn .L
* arm-linux-gnueabi-gcc dùng cờ -O2 để tối ưu thời gian

### 3.2.2. Thời gian thử nghiệm trung bình của các chương trình arm



### 3.2.3. Bảng thống kê kết quả mã arm



**Code 11 đến 20**

* **Tối ưu thời gian**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **STT** | **Tên chương trình** | **Kỹ thuật tối ưu thực hiện** | **Ct gốc** | **Ct tối ưu** |
| 11 | Tính tổng bình phương các số lẻ từ 1 đến < n | Dùng cờ O2 | 0.016s | 0.031s |
| 12 | Tìm max của 3 số a,b,c nhập từ bàn phím | Dùng cờ O2 | 0.016s | 0.016s |
| 13 | Nhập một số nguyên từ bàn phím, kiểm tra xem đó là số chẵn hay lẻ | Dùng cờ O2 | 0.016s | 0.016s |
| 14 | Tìm ước số chung lớn nhất và bội số chung nhỏ nhất của 2 số nguyên nhập từ bàn phím | Dùng cờ O2 | 0.016s | 0.000s |
| 15 | Nhập một số nguyên từ bàn phím. Kiểm tra một số có phải là số hoàn hảo? | Dùng cờ O2 | 0.031s | 0.016s |
| 16 | Nhập vào dãy số tính tổng các số lẻ trong dãy. | Dùng cờ O2 | 0.031s | 0.016s |
| 17 | Nhập vào 1 mảng và in các số nguyên tố trong mảng và vị trí của nó | Dùng cờ O2 | 0.016s | 0.031s |
| 18 | Nhập vào 1 dãy số và in ra vị trí lớn nhất | Dùng cờ O2 | 0.031s | 0.016s |
| 19 | Nhập vào 1 mảng và xuất mảng đó ra console. | Dùng cờ O2 | 0.031s | 0.016s |
| 20 | Nhập vào 1 mảng và tìm kiếm 1 giá trị bất kỳ trong mảng và in ra. | Dùng cờ O2 | 0.046s | 0.031s |

* **Tối ưu kích thước**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **STT** | **Tên chương trình** | **Kỹ thuật tối ưu thực hiện** | **Ct gốc** | **Ct tối ưu** |
| 11 | Tính tổng bình phương các số lẻ từ 1 đến < n | Dùng cờ O2 | 1552 bytes | 1446 bytes |
| 12 | Tìm max của 3 số a,b,c nhập từ bàn phím | Dùng cờ O2 | 1820 bytes | 1931 bytes |
| 13 | Nhập một số nguyên từ bàn phím, kiểm tra xem đó là số chẵn hay lẻ | Dùng cờ O2 | 1088 bytes | 1157 bytes |
| 14 | Tìm ước số chung lớn nhất và bội số chung nhỏ nhất của 2 số nguyên nhập từ bàn phím | Dùng cờ O2 | 1957 bytes | 1778 bytes |
| 15 | Nhập một số nguyên từ bàn phím. Kiểm tra một số có phải là số hoàn hảo? | Dùng cờ O2 | 2097 bytes | 1936 bytes |
| 16 | Nhập vào dãy số tính tổng các số lẻ trong dãy. | Dùng cờ O2 | 3370 bytes | 2625 bytes |
| 17 | Nhập vào 1 mảng và in các số nguyên tố trong mảng và vị trí của nó | Dùng cờ O2 | 2568 bytes | 2428 bytes |
| 18 | Nhập vào 1 dãy số và in ra vị trí lớn nhất | Dùng cờ O2 | 2615 bytes | 2357 bytes |
| 19 | Nhập vào 1 mảng và xuất mảng đó ra console. | Dùng cờ O2 | 4660 bytes | 3588 bytes |
| 20 | Nhập vào 1 mảng và tìm kiếm 1 giá trị bất kỳ trong mảng và in ra. | Dùng cờ O2 | 4150 bytes | 3368 bytes |

# **KẾT LUẬN**

# **TÀI LIỆU THAM KHẢO**

[1] [Hội đồng 1: Cơ bản – CS 61 2018 (harvard.edu)](https://cs61.seas.harvard.edu/site/2018/Asm1/)

[2] [Optimize Options (Using the GNU Compiler Collection (GCC))](https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html)

[3] [opdenacker-embedded-linux-size-reduction-techniques\_0.pdf](file:///C:\Users\ADMIN\OneDrive\Desktop\Nam5\ToiUuPMNhung\cuoiky\documents\opdenacker-embedded-linux-size-reduction-techniques_0.pdf)

[4] [Data Structure Alignment trong C/C++ | CppDeveloper](https://cppdeveloper.com/c-nang-cao/data-alignment-trong-c-c/)

[5] [Bí kíp optimize code C/C++ | CppDeveloper](https://cppdeveloper.com/c-nang-cao/bi-kip-toi-uu-hoa-code-c-c/)

[6] [assembly - What are CFI directives in Gnu Assembler (GAS) used for? - Stack Overflow](https://stackoverflow.com/questions/2529185/what-are-cfi-directives-in-gnu-assembler-gas-used-for)

[7] [What does the ! character do in ARM assembly? - Stack Overflow](https://stackoverflow.com/questions/60395608/what-does-the-character-do-in-arm-assembly)