

ĐẠI HỌC QUỐC GIA TP HỒ CHÍ MINH  
TRƯỜNG ĐẠI HỌC BÁCH KHOA  
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH



LUẬN VĂN TỐT NGHIỆP

---

PHÁT TRIỂN  
HỆ THỐNG BE-PUM

---

**Giáo Viên Hướng Dẫn:**

PGS.TS. QUẢN THÀNH THƠ

ThS. NGUYỄN MINH HẢI

ThS. LÊ ĐÌNH THUẬN

**Sinh viên thực hiện:**

NGUYỄN XUÂN KHÁNH - MSSV: 51101594

NGUYỄN LÂM HOÀNG YÊN - MSSV: 51104402

**Giáo Viên Phản Biện:**

TS. BÙI HOÀI THẮNG

TP.Hồ Chí Minh, Ngày 19 tháng 12 năm 2015



# LỜI CAM KẾT

Chúng tôi xin cam đoan rằng mọi thông tin và công việc được trình bày trong bài báo cáo này ngoài việc tham khảo các nguồn tài liệu khác có ghi đầy đủ trong phần phụ lục các tài liệu tham khảo, thì đều do chính chúng tôi thực hiện và chưa có phần nội dung nào được sao chép từ các đề tài thực tập tốt nghiệp, luận văn tốt nghiệp của trường này và trường khác. Nếu có bất kì sai phạm hay gian lận nào, chúng tôi xin chịu hoàn toàn trách nhiệm trước Ban Chủ Nhiệm Khoa và Ban Giám Hiệu Nhà Trường.

TP. Hồ Chí Minh, Ngày 19 tháng 12 năm 2015

**Nhóm sinh viên thực hiện đề tài**

# LỜI CẢM ƠN

Đầu tiên em xin gửi lời cảm ơn sâu sắc đến thầy PGS.TS. Quản Thành Thơ, thầy đã có những sự định hướng, động viên, giải đáp và quan tâm to lớn đến em và cho cả mọi người đang cùng dưới sự dẫn dắt của thầy. Thầy luôn theo sát mọi người qua từng ngày, từng tuần thực hiện nghiên cứu và đưa ra những gợi ý để công việc được suôn sẻ cũng như đạt kết quả như mong muốn. Những lời chỉ bảo và răn đe của thầy luôn là động lực để em và mọi người hoàn thành nhiệm vụ. Chính phong thái đó của thầy là lý do để em chọn vào tham gia nghiên cứu trong nhóm của thầy; và càng ngày em càng cảm ơn về sự lựa chọn đúng đắn đó.

Kế đến em xin dành sự biết ơn to lớn dành cho ThS. Nguyễn Minh Hải, với em anh không chỉ là một người hướng dẫn đề tài, mà còn là một người anh gần gũi vì những sự trợ giúp ân cần từ những ngày đầu em tham gia cùng mọi người thực hiện nghiên cứu. Làm việc với anh không phải là một sự căng thẳng và rập khuôn, bởi anh luôn thân thiện thảo luận cùng mọi người để đưa ra những ý kiến, công việc và thời hạn hợp tình, hợp lý. Anh luôn dành sự quan tâm và tận tình cho từng thành viên dưới sự hướng dẫn của anh; kèm theo đó là những lời động viên, chia sẻ để từ đó em có thêm động lực để hoàn thành được đề tài này.

Em cũng xin cảm ơn ThS. Lê Đình Thuận đã giúp em hoàn thiện bài báo cáo này qua việc theo dõi sát sao mỗi ngày trong thời gian viết bài để hướng dẫn, đóng góp ý kiến và giúp em sửa chữa những sai sót có trong báo cáo.

---

Em xin dành những sự tri ân chân thành và to lớn đến toàn thể quý thầy cô của khoa Khoa học và Kỹ thuật Máy tính mà em đang theo học. Nhờ những sự tận tình giảng dạy, truyền đạt kiến thức và kinh nghiệm quý báu qua từng bài giảng mà em có thêm được sự vun đắp, vững chắc trong vốn hiểu biết của mình; để từ đó làm nền tảng cho em hoàn thành được đề tài ngày hôm nay.

Em cũng không quên bày tỏ lòng biết ơn đến gia đình, người thân của mình, những người đã luôn quan tâm, động viên và chăm sóc em qua từng ngày để em có được sức khỏe, tinh thần và môi trường phát triển, giúp em có được như ngày hôm nay. Cuối cùng em xin cảm ơn những người bạn đã hỗ trợ, chia sẻ để em hoàn thành tốt được đề tài này.

Với toàn bộ sự biết ơn sâu sắc đó, em xin gửi lời chúc đến mọi người đã giúp em làm được những điều đến ngày hôm nay. Chúc cho mọi quý thầy cô, đặc biệt là những người em đã nêu tên ở trên, của khoa Khoa học và Kỹ thuật Máy tính luôn có được một sức khỏe dồi dào để luôn hạnh phúc trong cuộc sống và có khả năng để cống hiến thêm cho khoa, cho trường và cho thế hệ đi sau.

Trân trọng.

TP. Hồ Chí Minh, 19/12/2015

**Nhóm sinh viên thực hiện đề tài**

# TÓM TẮT LUẬN VĂN

# Mục lục

<b>LỜI CAM KẾT</b>	<b>ii</b>
<b>LỜI CẢM ƠN</b>	<b>iii</b>
<b>TÓM TẮT LUẬN VĂN</b>	<b>v</b>
<b>1 Giới Thiệu</b>	<b>1</b>
1.1 Giới thiệu về BE-PUM	1
1.1.1 BE-PUM là gì	1
1.1.2 Mục tiêu hướng đến của BE-PUM	1
1.1.3 Các hướng tiếp cận	2
1.1.4 Đồ thị luồng điều khiển	3
1.1.5 Những đòi hỏi trong quá trình hiện thực	4
1.2 Tổng quan về hợp ngữ	5
1.3 Tổng quan về Windows API	6
1.4 Mục tiêu đề tài	7
1.5 Giới hạn đề tài	8
1.6 Cấu trúc của báo cáo	8
<b>2 Phân Tích Vấn Đề</b>	<b>11</b>
2.1 Các câu lệnh hợp ngữ	11
2.1.1 Assembly ngôn ngữ dùng để phân tích	11
2.1.2 BE-PUM và Assembly	11
2.1.3 Phân tích Assembly trong BE-PUM	12

---

2.2	Windows API .....	13
2.2.1	Windows API trong những phần mềm độc hại .....	13
2.2.2	BE-PUM và Windows API .....	14
2.2.3	Truy xuất Windows API bên trong BE-PUM thông qua JNA .....	15
<b>3</b>	<b>Kiến Thức Nền</b>	<b>16</b>
3.1	Làm việc với BE-PUM .....	16
3.2	Các câu lệnh hợp ngữ .....	17
3.2.1	Assembly .....	17
3.2.2	Floating-Point Unit (FPU) .....	29
3.2.3	Phân nhánh theo mã điều kiện FPU .....	71
3.2.4	Xử lý ngoại lệ dấu chấm động .....	76
3.3	Windows API .....	88
3.3.1	Một vài bộ thư viện cần hỗ trợ .....	88
3.3.2	Trình tự các bước để ứng dụng JNA vào BE-PUM .....	89
3.3.3	Những kiểu dữ liệu và ánh xạ của chúng vào JNA .....	89
3.3.4	Dữ liệu kiểu cấu trúc (structure) .....	91
<b>4</b>	<b>Thiết Kế Và Xây Dựng</b>	<b>93</b>
4.1	Các câu lệnh hợp ngữ .....	93
4.1.1	Sơ đồ chương trình BE-PUM .....	93
4.1.2	Phân tích class mô phỏng câu lệnh Assembly .....	94
4.1.3	Quy trình thực hiện .....	106
4.2	Windows API .....	108
4.2.1	Các thành phần chính của bộ xử lý Windows API .....	108
4.2.2	Những khai báo ánh xạ của bộ xử lý Windows API .....	111
4.2.3	Kiến trúc xây dựng và làm việc .....	113
4.2.4	Quản lý môi trường tương tác vật lý .....	117
<b>5</b>	<b>Kết Quả</b>	<b>118</b>
5.1	Thí nghiệm và đánh giá kết quả .....	118
5.2	Các câu lệnh hợp ngữ đã được hỗ trợ .....	119

---



---

5.3	Các Windows API đã được hỗ trợ .....	121
<b>6</b>	<b>Hướng Phát Triển Trong Tương Lai</b>	<b>127</b>
6.1	Tăng số lượng các câu lệnh hợp ngữ được hỗ trợ .....	127
6.2	Tăng số lượng các Windows API được hỗ trợ .....	128
6.3	Hiện thực hóa việc tự động sinh mã cho công tác hỗ trợ Windows API....	128

# Danh sách hình ảnh

1.1	Ví dụ mô hình luồng điều khiển được sinh ra từ tập tin ở Bảng 1.1 . . . . .	4
3.1	Các thanh ghi <sup>1</sup> . . . . .	17
3.2	Các thanh ghi <sup>2</sup> . . . . .	17
3.3	Hệ thống mã nhị phân biểu diễn số thực . . . . .	30
3.4	Định dạng mã nhị phân của Floating-point . . . . .	31
3.5	Số thực và NaN . . . . .	33
3.6	Mối quan hệ giữa bộ xử lý FPU và bộ xử lý Integer . . . . .	36
3.7	Môi trường thực thi FPU . . . . .	37
3.8	Ngăn xếp thanh ghi dữ liệu FPU . . . . .	38
3.9	Ví dụ xử lý câu lệnh trong stack của FPU . . . . .	39
3.10	Thanh ghi trạng thái FPU . . . . .	41
3.11	Sao lưu có điều kiện . . . . .	45
3.12	Thanh ghi điều khiển FPU . . . . .	46
3.13	Thanh ghi điều khiển FPU . . . . .	50
3.14	Nội dung thanh ghi opcode FPU . . . . .	52
3.15	Chế độ bảo vệ (protected) FPU trong bộ nhớ, định dạng 32-bit . . . . .	53
3.16	Chế độ địa chỉ thực FPU trong bộ nhớ , định dạng 32-bit . . . . .	53
3.17	Chế độ bảo vệ (protected) FPU trong bộ nhớ, định dạng 16-bit . . . . .	54
3.18	Chế độ địa chỉ thực FPU trong bộ nhớ , định dạng 16-bit . . . . .	54
3.19	Định dạng loại dữ liệu dấu chấm động . . . . .	55
3.20	Cách sắp xếp và lưu trữ bộ nhớ của structure . . . . .	91
4.1	Sơ đồ class chương trình BE-PUM . . . . .	93

---

4.2	Sơ đồ class mô phỏng câu lệnh Assembly .....	95
4.3	Class Environment .....	96
4.4	Class .....	97
4.5	Class Stack .....	98
4.6	Class Register .....	99
4.7	Class Memory .....	100
4.8	Class Flag .....	101
4.9	Class X86TransitionRule .....	102
4.10	Class FPUControlWord .....	103
4.11	Class FPUStatusWord .....	104
4.12	Class .....	105
4.13	Quy trình hiện thực một câu lệnh assembly .....	106
4.14	So sánh kết quả với Onlydbg .....	107
4.15	Kiến trúc chính của bộ xử lý Windows API dành cho BE-PUM .....	109
4.16	Mô hình hiện thực một số Windows API trong <i>kernel32.dll</i> .....	110
4.17	Lớp chứa thông tin ánh xạ những Windows API trong <i>user32.dll</i> .....	111
4.18	Một số dữ liệu kiểu cấu trúc được xây dựng thêm .....	112
4.19	Cây cấu trúc mô tả những thành phần trong bộ xử lý Windows API .....	113
4.20	Giản đồ thể hiện mối liên hệ giữa các thành phần .....	116
6.1	Biểu diễn số thực 32 bits .....	130
6.2	Giao diện công cụ Ollydbg .....	134
6.3	Mở file trong Ollydbg .....	135
6.4	Ví dụ Ollydbg .....	135
6.5	Cửa sổ Register trong Ollydbg .....	136
6.6	Cửa sổ Disassembler trong Ollydbg .....	137
6.7	Cửa sổ Stack trong Ollydbg .....	138
6.8	Cửa sổ Dump trong Ollydbg .....	139
6.9	Chức năng trong Ollydbg .....	140
6.10	Giao diện công cụ MASM32 .....	143

---

# Danh sách các bảng

1.1	Ví dụ về nội dung của một tập tin nhị phân dưới dạng hợp ngữ . . . . .	4
1.2	Một số phiên bản chính của Windows API . . . . .	7
3.1	Bảng chia bộ nhớ 32-bit, 16-bit, 8-bit . . . . .	18
3.2	Bảng chia bộ nhớ 32-bit, 16-bit . . . . .	18
3.3	Bảng ký hiệu toán hạng: . . . . .	21
3.4	Bảng kiểu dữ liệu: . . . . .	23
3.5	Bảng ký hiệu kiểu số tự nhiên: . . . . .	23
3.6	Bảng phạm vi kiểu dữ liệu số không dấu . . . . .	24
3.7	Bảng phạm vi kiểu dữ liệu số có dấu . . . . .	24
3.8	Thứ tự ưu tiên biểu thức toán . . . . .	25
3.10	Xử lý số denormalized . . . . .	34
3.12	Độ chính xác trong thanh ghi điều khiển FPU . . . . .	46
3.13	Bảng giá trị phương thức làm tròn . . . . .	47
3.14	Làm tròn số dương khi overflow xảy ra . . . . .	49
3.15	Làm tròn số dương khi Underflow xảy ra . . . . .	49
3.16	Làm tròn số dương khi Underflow xảy ra . . . . .	56
3.17	Mã hóa số thực và NaN . . . . .	58
3.18	Mã hóa nhị phân số nguyên . . . . .	59
3.19	Mã hóa số nguyên đã được nén (BCD) . . . . .	62
3.20	Mã hóa số không được hỗ trợ theo định dạng extended-real . . . . .	64
3.21	Câu lệnh sao lưu dữ liệu . . . . .	65
3.22	Câu lệnh sao chép có điều kiện . . . . .	66

---

3.23	Thiết lập cờ điều kiện FPU theo phép so sánh số thực .....	70
3.24	Thiết lập thanh ghi trạng thái EFLAGS theo phép so sánh số thực .....	70
3.25	Hằng số câu lệnh Test cho lệnh rẽ nhánh .....	72
3.26	Thao tác số học không hợp lệ và kết quả trả về .....	80
3.27	Điều kiện xảy ra ngoại lệ chia cho số 0 và kết quả trả về .....	80
3.28	Kết quả khi tràn trên số học xảy ra .....	82
3.29	Những bộ thư viện được hỗ trợ bởi BE-PUM .....	88
3.30	Một số ánh xạ kiểu dữ liệu cơ bản .....	90
5.1	Một số kết quả của thí nghiệm sinh mô hình .....	119
5.2	Thống kê các câu lệnh assembly đã được hiện thực .....	121
5.3	Thống kê các Windows API đã được xây dựng .....	126
6.1	Các kiểu định dạng chuẩn IEEE .....	130
6.3	Ví dụ biểu diễn số dấu chấm động .....	131
6.4	Ví dụ về expoment .....	132
6.5	Ví dụ biểu diễn số mantissa .....	133
6.6	Ví dụ chuẩn IEEE .....	133



# Chương 1

## Giới Thiệu

### 1.1 Giới thiệu về BE-PUM

#### 1.1.1 BE-PUM là gì

BE-PUM tên đầy đủ là Binary Emulation for Pushdown Model generation, là một công cụ dùng để phân tích động mã nhị phân của một chương trình bất kỳ chạy trên kiến trúc X86 của hệ điều hành Microsoft Windows nền tảng 32-bit. Sau khi phân tích, BE-PUM sẽ sinh ra hợp ngữ – mã assembly và đồ thị luồng điều khiển (control flow graph – CFG) của chương trình đầu vào.

BE-PUM được xây dựng chính trên mã nguồn của JakStab, nhưng không hạn hẹp ở việc chỉ phân tích tĩnh, BE-PUM có thể phân tích động và chỉ ra lại mỗi dòng lệnh của mã assembly môi trường làm việc của nó là như thế nào. Việc này sẽ giải quyết được những hạn chế mà phương pháp phân tích tĩnh không thể xử lý được.

#### 1.1.2 Mục tiêu hướng đến của BE-PUM

Với cuộc sống ngày càng hiện đại của con người, sự phổ biến và tầm quan trọng của máy tính ngày càng nâng cao. Các công việc quan trọng nay hầu hết đều có thể được quản lý và xử lý bởi các phần mềm máy tính, từ các hệ thống quản lý nhân công hay các

quy trình tài chính, ngân hàng. Sự lệ thuộc nhiều vào máy tính đó đã kéo theo cho sự ra đời của hàng loạt những phần mềm độc hại, với các mục đích xấu, đôi lúc chỉ là để mua vui hay nghiêm trọng hơn là tư lợi cho bản thân người viết ra chúng.

Bằng việc phân tích mã nhị phân, BE-PUM đang được phát triển để tập trung vào việc phân tích những phần mềm bị nghi ngờ, sau đó dựa vào hành vi sẽ phát hiện được những kỹ thuật tấn công, và cuối cùng là xác định xem đây có thực là phần mềm gây hại đến máy tính (malware - malicious software) hay không?

### **1.1.3 Các hướng tiếp cận**

Với một công cụ phân tích tập trung vào phần mềm độc hại, ta có một số hướng tiếp cận như sau:

#### **1.1.3.1 Phát hiện dựa trên dấu hiệu**

Phát hiện dựa trên dấu hiệu (signature detection) là một hướng tiếp cận theo kiểu công nghiệp, nó được dùng trong đại đa số các phần mềm phát hiện malware vì đặc tính là nhanh chóng xác định một malware; nhưng đi kèm là hạn chế chỉ phát hiện được trong giới hạn một lượng dấu hiệu đã nạp sẵn.

Khi một chương trình phát hiện malware thông thường thực hiện việc quét tìm mối nguy hiểm, chúng sẽ kiểm tra nội dung của từng tập tin, so trùng nội dung đó với bộ từ điển các dấu hiệu đã có sẵn và nhận biết. Việc này sẽ đòi hỏi bộ từ điển kia được cập nhật liên tục từ nhà phát hành, nhưng việc cập nhật đó cũng chỉ có giới hạn, trong khi các malware được gia tăng qua từng ngày. Việc một malware sinh ra biến thể nhỏ có thể khiến cho một dấu hiệu nhận dạng trước đó không hoạt động chính xác nữa.

#### **1.1.3.2 Phát hiện dựa vào giả lập**

Dựa vào bộ giả lập (emulation) là một cách tiếp cận với việc xây dựng và thực thi tập tin malware trong một môi trường sandbox. Môi trường sandbox là môi trường ảo hóa được dựng lên với các tính năng tương tự như một hệ thống thật, nhưng cách ly và bị



hạn chế không thể tạo ra một thay đổi lâu dài. Điều này có nghĩa là tất cả những gì xảy ra trong sandbox sẽ chỉ ở trong sandbox.

Nhưng một malware tinh vi có thể phát hiện được rằng chúng đang chạy trong sandbox và ta sẽ không thu được kết quả gì. Cộng thêm với việc xây dựng một môi trường sandbox đòi hỏi bỏ ra rất nhiều công sức, và hành vi của một malware sẽ có thể bị khác nhau (do thực thi thật) ở nhiều thời điểm; do đó phương pháp này không được nhắm tới.

#### 1.1.3.3 Phát hiện dựa vào sinh mô hình

Sinh mô hình (model generation) là một cách tiếp cận hình thức với việc mọi câu lệnh được mô hình hóa và biểu diễn bằng mô hình. Sau khi xử lý một chuỗi các mô hình qua các câu lệnh có trong tập tin, ta sẽ có được đồ thị luồng điều khiển và giải quyết bài toán trên đó bằng phương pháp kiểm tra mô hình (model checking).

#### 1.1.4 Đồ thị luồng điều khiển

*Đồ thị luồng điều khiển (control flow graph - CFG)* trong khoa học máy tính là một phép biểu diễn của tất cả các đường có thể được duyệt qua trong quá trình thực thi của một chương trình. [1]

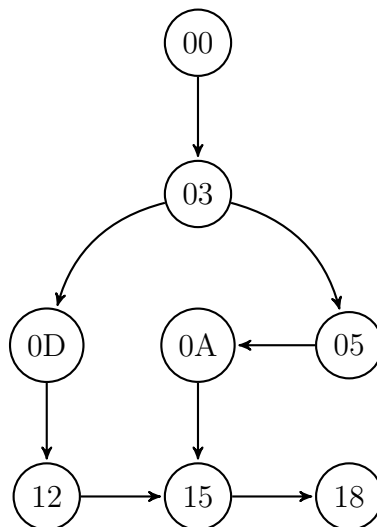
Trong BE-PUM, một địa chỉ trong chương trình được ánh xạ tương ứng với 1 nút (node) trong đồ thị. BE-PUM sẽ xử lý theo luồng thực thi của chương trình và sinh ra từng nút một tương ứng. Trong quá trình xử lý, BE-PUM sẽ quyết định được điểm kế tiếp cần đi đến trong lúc duyệt nhánh. Tất cả việc đó được thực hiện thông qua giải thuật *on-the-fly model generation*.

*On-the-fly model generation* là giải thuật chính trong hệ thống BE-PUM. Quá trình thực thi của tập tin mã nhị phân là một chuỗi thực thi động, bắt đầu từ một giá trị trong ở phân vùng mã (code section) của bộ nhớ, giá trị địa chỉ này được chỉ rõ bởi giá trị thanh ghi EIP. BE-PUM sẽ đặt khởi điểm từ đây, và lần dò theo từng câu lệnh hợp ngữ,

tìm điểm kế tiếp sẽ được thực thi. Cứ như vậy, BE-PUM sẽ đi qua hết các nhánh có khả năng thực thi của chương trình và cho ra một đồ thị luồng điều khiển đầy đủ.

Địa chỉ câu lệnh	Nội dung câu lệnh
0x00401000	cmp eax, 0
0x00401003	je 0x0040100d
0x00401005	mov eax, 0x00401001
0x0040100a	jmp 0x00401015
0x0040100c	ret
0x0040100d	mov eax, 0x00401018
0x00401012	add eax, 10
0x00401015	sub eax, 4
0x00401015	ret

Bảng 1.1: Ví dụ về nội dung của một tập tin nhị phân dưới dạng hợp ngữ



Hình 1.1: Ví dụ mô hình luồng điều khiển được sinh ra từ tập tin ở Bảng 1.1

### 1.1.5 Những đòi hỏi trong quá trình hiện thực

Đối với những tập tin nhị phân đơn giản, ta có thể dễ dàng quyết định được nút kế tiếp. Nhưng với những malware tinh vi, chúng sẽ có cách để ta không dễ dàng hiểu được ý

định chúng sẽ làm gì, ra sao. Phương thức trên được gọi là làm rối mã (obfuscation). Một trong những cách diễn hình có thể kể đến là nhảy gián tiếp (indirect jump – thực hiện lệnh nhảy đến một giá trị không phải là hằng số như giá trị thanh ghi, giá trị ô nhớ...) và tự thay đổi mã (self-modification code – trong quá trình thực thi sẽ tự ý thay đổi nội dung mã thực thi của chính mình).

Và BE-PUM đã được dựng lên để có thể chống lại việc làm rối mã này. Đó là điểm mạnh và là ưu thế vượt trội của BE-PUM so với các công cụ phân tích tĩnh mã thực thi khác như Jackstab, IDA-Pro, Capstone, Unicorn, METASM, HOOPER bằng việc áp dụng phương pháp *dynamic symbolic execution*.

*Dynamic symbolic execution* là một phương pháp nhằm tính toán các giá trị thanh ghi và bộ nhớ ở một vị trí cụ thể, khi chúng ta không thể biết được điểm thực thi kế tiếp của chương trình là gì. Nhờ đó ta có thể giải quyết được các cách thức làm rối mã như nhảy gián tiếp, tự thay đổi mã như đã nói ở trên.

Để có thể tính toán được các giá trị thanh ghi cũng như bộ nhớ, ta cần xây dựng một hệ thống mô phỏng các câu lệnh mà một tập tin nhị phân thực hiện được. Các câu lệnh đó bao gồm 2 phần chính đó là *hợp ngữ* và *Windows API*.

## 1.2 Tổng quan về hợp ngữ

Ngôn ngữ assembly (hay hợp ngữ) là ngôn ngữ cấp thấp được biên dịch để thực thi một chương trình nào đó. Ngôn ngữ assembly có tính gợi nhớ, mỗi câu lệnh thực thi một chức năng, hay nhiệm vụ riêng biệt tương ứng với một lệnh yêu cầu máy thực thi.

Các chương trình sau khi biên dịch ra thành nhiều dạng file khác nhau để thực thi. Mỗi chương trình được biên dịch thành ngôn ngữ assembly trước khi biên dịch qua ngôn ngữ máy để máy hiểu và thực thi câu lệnh. Khi biên dịch qua mã assembly, đọc đoạn mã assembly để phân tích xem chương trình này có phải là virus máy tính không, có những

hành vi bất thường từ đó xem xét đưa ra kết quả chương trình này có an toàn với máy tính.

### 1.3 Tổng quan về Windows API

Với tên gọi đầy đủ là Microsoft Windows application programming interface, đôi lúc được gọi một cách ngắn gọn là WinAPI, đây là một bộ giao diện lập trình ứng dụng (API) lõi có sẵn trong hệ điều hành Microsoft Windows (hay thường được gọi ngắn gọn là Windows).

Windows API là tên gọi chung của những dự án được triển khai cho những nền tảng khác nhau được áp dụng trong hệ điều hành Windows; mà thông thường chúng có những tên gọi riêng. Một số tên gọi và mô tả cho những phiên bản xin được trình bày trong Bảng 1.2.

Tên của phiên bản Windows API	Mô tả
Win16	Đây là tên gọi bộ API đầu tiên có trong phiên bản Microsoft Windows 16-bit. Thuở ban đầu bộ API này có tên gọi là “Windows API”, nhưng sau khi phiên bản kế tiếp là Win32 ra đời, nó được đổi tên thành Win16. Lúc này các chức năng của Win16 API chủ yếu nằm trong các tập tin lõi của hệ điều hành như kernel.exe (hoặc krnl286.exe hoặc krnl386.exe), user.exe và gdi.exe. Dù rằng phần mở rộng là exe, nhưng thực tế đây là các thư viện liên kết động.

Win32	Đây là tên gọi cho bộ Windows API trong phiên bản Windows 32-bit, được giới thiệu lần đầu cùng hệ điều hành Windows NT. Các chức năng của Win32 API cũng bao gồm cả những chức năng của Win16 API, nhưng khác với phiên bản trước, chúng được đóng gói trong các tập tin DLL; trong đó có thể kể đến những tập tin DLL cốt lõi của Win32 API như: kernel32.dll, user32.dll, và gdi32.dll.
Win64	Đây là một biến thể sau đó của bộ API được hiện thực trong nền tảng Windows 64-bit. Cả chương trình 32-bit hoặc 64-bit đều có thể được biên dịch với cùng một mã nền duy nhất, dù rằng một số API trong Win32 đã được thay thế và loại bỏ. Tất cả con trỏ trong phiên bản này mặc định là 64-bit, do đó cần kiểm tra khả năng tương thích trong mã nguồn và viết lại nếu cần thiết.

Bảng 1.2: Một số phiên bản chính của Windows API

## 1.4 Mục tiêu đề tài

Trong phạm vi của đề tài luận tốt nghiệp, mục tiêu nhắm tới là phát triển hệ thống xử lý các câu lệnh hợp ngữ và Windows API cho BE-PUM.

Với số lượng các API rất lớn hiện có trong hệ điều hành Windows, hiện tại đề tài đang tập trung vào xử lý các API ở phiên bản Win32 API, do hầu hết các phần mềm độc hại mà BE-PUM hướng tới vẫn đang dùng bộ API này; với sự ưu tiên từng bước xây dựng cho các API được dùng phổ biến trước.

Bên cạnh việc nhận thông tin đầu vào từ vùng nhớ đã được xây dựng của BE-PUM và trả về kết quả sau khi gọi API vào đúng địa chỉ cần thiết, điều quan trọng là phải đảm bảo không gây ngắt quãng cũng như tránh nguy hại hệ thống đang chạy. Và như vậy

với những tương tác vật lý từ lời gọi API (bộ lưu trữ máy tính, cơ sở dữ liệu registry... ) hay tương tác người dùng (API tạo cửa sổ message box, lệnh cho một thread “ngủ đông” trong một khoảng thời gian,... ) cần được kiểm soát để không làm ảnh hưởng tới kết quả thực thi của BE-PUM.

*Lưu ý: do nội dung đề tài tập trung vào xử lý cho Win32 API, nên kể từ đây, khi báo cáo nhắc đến Windows API tức là nói đến Win32 API.*

Việc hiện thực các câu lệnh hợp ngữ góp phần phân tích một số loại malware đang nghiên cứu. Dựa trên một số bộ malware hiện nay và những câu lệnh phổ biến thường gặp, BE-PUM sẽ hỗ trợ tốt công việc phân tích đúng môi trường làm việc của từng câu lệnh từ đó xác định chính xác và phân tích đúng các kỹ thuật mà malware đang tiến hành.

## 1.5 Giới hạn đề tài

Trong phạm vi của đề tài luận văn tốt nghiệp, mục tiêu nhắm tới là hiện thực các câu lệnh hợp ngữ và Windows API với số lượng đạt mức như sau:

- Số lượng câu lệnh hợp ngữ được hỗ trợ đạt khoảng 250 câu lệnh.
- Số lượng câu lệnh Windows API được hỗ trợ đạt khoảng 400 câu lệnh.

## 1.6 Cấu trúc của báo cáo

Bài báo cáo này bao gồm những đề mục sau đây:

### Chương 1

Giới thiệu tổng quan về BE-PUM, yếu tố quyết định để cho ra đề tài này; dẫn nhập về Windows API, thành phần sẽ được áp dụng để phát triển cho BE-PUM; dẫn nhập về hợp ngữ assembly, và cuối cùng nêu ra được mục đích chính của đề tài sẽ cần làm gì.

**Chương 2**

Dem đến những cái nhìn về những vấn đề đã và đang được lưu tâm khi thực hiện đề tài này; sự phổ biến của Windows API trong những phần mềm độc hại để thấy sự cần thiết của việc xây dựng một bộ xử lý Windows API cho BE-PUM; những khó khăn khi thực hiện điều đó và giải pháp cho vấn đề. Đồng thời phân tích vấn đề đặt ra là hiện thực các câu lệnh trong hợp ngữ assembly, giải thích tại sao sử dụng ngôn ngữ assembly để tiến hành phân tích chương trình.

**Chương 3**

Trình bày những kiến thức cần thiết cho quá trình thực hiện đề tài; từ những kiến thức phải nắm được về hệ thống BE-PUM do đây là một đề tài làm việc dựa trên đó; và mỗi khi làm việc với một thư viện bất kỳ, đòi hỏi ta phải tìm hiểu cách thức làm việc với thư viện đó và cả những kiến thức cần thiết do bộ thư viện ấy yêu cầu. Đồng thời cũng trình bày các kiến thức cơ bản về hợp ngữ assembly, từ đó có cái nhìn tổng quan về assembly để tiến hành xây dựng chương trình BE-PUM.

**Chương 4**

Mỗi chương trình bất kỳ đều cần một thiết kế tốt để giúp cho việc xây dựng dễ dàng và quy chuẩn hơn. Mục này sẽ trình bày cách mà bộ xử lý Windows API đã được hiện thực để tương lai sau này có thể dễ dàng sửa chữa, bảo trì và bổ sung thêm vào kiến trúc đó. Để hiểu rõ cấu chương trình mô phỏng câu lệnh assembly, sơ đồ class trình bày trong chương này thể hiện được mối quan hệ giữa các class, cấu trúc của chương trình BE-PUM được phát triển dựa trên dự án JakStab, giới thiệu các class quan trọng của chương trình BE-PUM

**Chương 5**

Trình bày về kết quả mà bộ xử lý Windows API đã đạt được với những Windows API đã được hỗ trợ cho hệ thống BE-PUM. Đồng thời trình bày các bước để hiện thực một câu lệnh của hợp ngữ assembly, cách đánh giá kết quả đúng hay sai bằng cách so sánh kết quả với chương trình OnlyDbg, giới thiệu về công cụ

MASM.

## **Chương 6**

Sau khi có được những kết quả ở thời điểm hiện tại của quá trình thực hiện đề tài, chương cuối này sẽ trình bày về những phương hướng cần tiếp tục phát triển trong tương lai đối với đề tài hiện tại.

## **Phụ Lục**

Liệt kê về những tài liệu và nguồn tham khảo có liên quan đến đề tài này.



## Chương 2

# Phân Tích Vấn Đề

### 2.1 Các câu lệnh hợp ngữ

#### 2.1.1 Assembly ngôn ngữ dùng để phân tích

Với các chương trình cần được phân tích, việc biết mã nguồn của chương trình là một điều rất khó khăn và hầu như là không thể biết được do đó để phân tích chương trình, BE-PUM dịch ngược chương trình cần phân tích từ mã chương trình đó, dựa vào các opcode để chuyển sang mã assembly để có thể đọc được và hiểu được chương trình đang thực thi gì?

Tại sao lại chọn mã assembly để phân tích? Vì lý do không thể tìm được mã nguồn của chương trình cần phân tích, để tiến hành dịch ngược, và trên cơ sở mọi chương trình thực thi đều phải biên dịch qua ngôn ngữ máy là mã nhị phân. Từ đó quyết định chọn ngôn ngữ assembly là ngôn ngữ cấp thấp nhưng vẫn có thể đọc hiểu và phân tích. Từ đó, dựa trên đoạn mã assembly của chương trình cần phân tích, BE-PUM tiến hành phân tích đoạn mã assembly để trả lời câu hỏi “Chương trình này có phải là virus hay không?”

#### 2.1.2 BE-PUM và Assembly

Hợp ngữ assembly là một ngôn ngữ lập trình cấp thấp. Trên thực tế đã có rất nhiều chương trình được viết bằng ngôn ngữ assembly được áp dụng cho những chiếc máy tính

đầu tiên. Công việc lập trình bằng ngôn ngữ assembly rất cực nhọc, dễ xảy ra lỗi trong quá trình lập trình và tốn rất nhiều chi phí, công sức và thời gian. Các chương trình viết bằng hợp ngữ phụ thuộc rất nhiều vào kiến trúc máy tính của nó, và trong phạm vi của công cụ BE-PUM, dựa trên kiến trúc X86 của Microsoft Windows để tiến hành phân tích chương trình.

Hợp ngữ assembly là tập hợp các câu lệnh gọi nhớ giúp người lập trình có thể hiểu được, là một ngôn ngữ cấp thấp nhưng vẫn có thể dễ đọc. Số lượng câu lệnh assembly là khá nhiều do đó trong quá trình xây dựng công cụ phân tích BE-PUM tập trung vào những câu lệnh mà virus thường hay sử dụng và hiện thực thêm các câu lệnh để hỗ trợ trong quá trình phân tích virus.

BE-PUM là một dự án được phát triển lên từ dự án JakStab và được viết hoàn toàn bằng ngôn ngữ JAVA. Việc hiện thực câu lệnh assembly trên JAVA có đôi chút khó khăn khi không có thư viện hỗ trợ, với mỗi câu lệnh assembly có biến môi trường khác nhau, nhiệm vụ khác nhau vì vậy để hiện thực một câu lệnh hoàn chỉnh cần phải có tài liệu tham khảo, có ví dụ từng trường hợp cụ thể, tránh sai sót hoặc xử lý thiếu trường hợp. Việc tìm tài liệu với từng câu lệnh cũng gặp đôi chút khó khăn khi tài liệu một số câu lệnh rất ít. Do đó để hạn chế những thiếu sót, các câu lệnh assembly thường được kiểm tra, nếu có trường hợp sai xảy ra sẽ được tìm hiểu và khắc phục.

### **2.1.3 Phân tích Assembly trong BE-PUM**

Mục đích của chương trình Be-Pum đặt ra trả lời câu hỏi “Chương trình này có phải là virus hay không?”. Để trả lời câu hỏi này có rất nhiều cách khác nhau để hiện thực chương trình, Be-Pum dựa trên phân tích hành vi của chương trình để trả lời vấn đề trên. Phân tích mã assembly là một trong những bước đầu để trả lời câu hỏi đó. Assembly là ngôn ngữ cấp thấp nhưng người lập trình vẫn có thể đọc được nó, và mọi chương trình khi thực thi đều có thể biên dịch ra mã assembly. Do đó, dựa vào hành vi của chương trình và tiến hành phân tích mã assembly mà có thể đưa ra kết luận.

Để giải quyết vấn đề hỗ trợ, phân tích từng câu lệnh của assembly, BE-PUM tiến hành xây dựng các class để tiến hành phân tích, hiện thực câu lệnh. Chia nhỏ vấn đề của từng câu lệnh thành các hàm trong từng class để hiện thực. Việc chia nhỏ vấn đề có thể giải quyết những câu lệnh có tính lặp lại, xử lý cùng một tính năng hay mục đích nào đó của câu lệnh. Giúp quản lý tốt hơn về hiện thực câu lệnh.

Có rất nhiều bước để hoàn thành chương trình Be-Pum, trong đó đọc hiểu mã assembly đầu vào là một trong những bước cơ bản. Có rất nhiều câu lệnh assembly mà BE-PUM chưa hỗ trợ, do đó cần đặt ra nhiệm vụ là hiện thực các câu lệnh assembly trong BE-PUM. Tính tới thời điểm hiện nay, Be-Pum đã hiện thực 200 câu lệnh thực thi số nguyên và 65 câu lệnh thực thi số thực và đã được kiểm tra.

## **2.2 Windows API**

### **2.2.1 Windows API trong những phần mềm độc hại**

Để cung cấp sức mạnh và sự tiện lợi cho lập trình viên trong việc viết ứng dụng chạy trên hệ điều hành Windows, các API trong bộ Windows API mở ra nhiều cách thức nhanh chóng và mạnh mẽ cho lập trình viên trong việc tương tác với hệ thống.

Và vấn đề gì cũng có hai mặt của nó, sự hỗ trợ mạnh mẽ đó cũng là con đường đơn giản để các tin tặc áp dụng vào việc xây dựng nên các phương pháp tấn công, cũng như cho ra đời những phần mềm nguy hại (malware), để lại bao hậu quả xấu cho hệ thống máy vi tính trên toàn cầu.

Trong quá trình xây dựng BE-PUM và qua việc phân tích hàng ngàn mẫu malware chạy trên môi trường Windows được phát tán ở khắp nơi trên thế giới, hầu hết những mẫu malware trên đều áp dụng lời gọi Windows API vào cách thức tấn công của chúng. Những phương pháp tấn công phổ biến như SEH hay phương pháp chống phát hiện đều có sự tồn tại của Windows API trong đó.

Do đó, việc xây dựng một bộ công cụ xử lý những thông tin trả về từ Windows API là rất cần thiết cho việc phát triển hệ thống BE-PUM, một hệ thống tập trung vào phân tích mã nhị phân của malware.

### 2.2.2 BE-PUM và Windows API

Mã nguồn của những API trong bộ Windows API được tập đoàn Microsoft giữ kín và không hề công bố. Chỉ có những đặc tả và hướng dẫn sử dụng được Microsoft phổ biến rộng rãi cho lập trình viên. Nghĩa rằng ta chỉ có thể biết được đầu vào của lời gọi và mong muốn đầu ra sẽ như ý, chứ không thể nắm rõ lô-gíc xử lý bên trong của chúng. Điều đó khiến cho việc xử lý đúng đắn một cách tổng quát đối với mọi đầu vào của mỗi API bằng cách viết lại bộ mã xử lý tương ứng của chúng vào trong BE-PUM dường như trở nên không thể.

Hướng tiếp cận hiện tại là tiến hành lấy nội dung bộ nhớ, nội dung các đối số nằm trên stack bên trong BE-PUM và tiến hành gọi thực sự với Windows API, nhận kết quả trả về và nạp lại vào trong BE-PUM để tiếp tục tiến hành phân tích các câu lệnh tiếp theo.

BE-PUM là một dự án được phát triển lên từ nhân của dự án JakStab và được viết hoàn toàn trên ngôn ngữ lập trình Java. Với Windows API thì lại là một câu chuyện hoàn toàn khác, Windows API được phát triển chủ yếu tập trung vào ngôn ngữ lập trình C kèm với các mô tả và cấu trúc dữ liệu được viết trên đó. Thêm lần nữa, việc hiện thực ý tưởng gọi để lấy kết quả Windows API từ trong lòng BE-PUM gặp nhiều khó khăn. Đặc biệt là việc ánh xạ các dữ liệu kiểu cấu trúc giữa hai thành phần trên cũng là một trở ngại.

Vì những lý do trên, cần tìm hiểu một cách thức giải quyết vấn đề nhanh chóng và đơn giản hơn bằng một bộ công cụ nào đó để xử lý rào cản ngôn ngữ giữa Java và C. Thêm vào đó, bộ công cụ này cũng cần có tính linh hoạt và mềm dẻo để cho việc phát triển về sau được dễ dàng.

### 2.2.3 Truy xuất Windows API bên trong BE-PUM thông qua JNA

Vấn đề trên được giải quyết thông qua bộ thư viện Java Native Access (JNA).

Java Native Access là một thư viện được cộng đồng phát triển, nhằm giúp cho các chương trình được viết bằng ngôn ngữ lập trình Java dễ dàng truy cập vào các thư viện native shared mà không cần thông qua Java Native Interface. Thiết kế của JNA cũng cung cấp khả năng này mà không cần bỏ ra nhiều công sức.

Với khả năng ánh xạ dễ dàng giao diện lập trình giữa hai ngôn ngữ Java và C; bao gồm ánh xạ tên hàm, kiểu dữ liệu trả về, kiểu dữ liệu của các thông số đầu vào; từ những kiểu dữ liệu cơ bản đến những kiểu dữ liệu cấu trúc và kể cả con trỏ; đó là những ưu điểm để lựa chọn JNA áp dụng vào trong việc giải quyết yêu cầu của đề tài nêu trên.

## Chương 3

# Kiến Thức Nền

### 3.1 Làm việc với BE-PUM

Trong BE-PUM có xây dựng một hệ thống mô tả môi trường làm việc tương tự như trên một hệ thống máy tính khởi chạy ngoài thực tế; chúng bao gồm các thành phần chính mà đề tài sẽ ảnh hưởng đến như sau:

- **Chồng (stack)** là một đơn vị dùng để lưu trữ giá trị theo nguyên tắc xếp chồng lên nhau, hoạt động theo nguyên lý vào sau ra trước (Last In First Out (LIFO)). Nó được dùng với nhiều mục đích khác nhau, ví dụ như để ghi nhận địa chỉ cần trở về khi thực hiện gọi một chương trình con, hay để ghi nhận các giá trị của các tham số truyền vào khi gọi một hàm Windows API.
- **Bộ nhớ (memory)** dùng để lưu trữ toàn bộ giá trị của các vùng nhớ mà chương trình đang làm việc. Windows API cũng có khả năng tương tác với phần này; do vậy, đòi hỏi khi xây dựng xử lý API cần đảm bảo kết quả chính xác cho thành phần này.
- **Thanh ghi (register)** là một vùng nhớ có dung lượng nhỏ nhưng lại có khả năng truy xuất rất nhanh, chúng được dùng để làm vùng nhớ tạm, trung gian cho các câu lệnh chạy trong hệ thống. Mỗi khi gọi một hàm bất kỳ, nếu như kiểu

trả về khác void thì giá trị trả về sẽ được lưu trữ vào thanh ghi EAX.

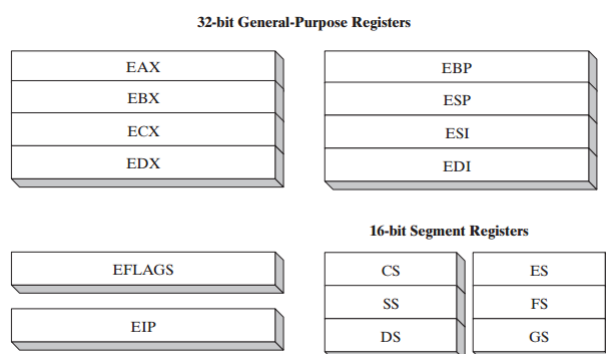
## 3.2 Các câu lệnh hợp ngữ

### 3.2.1 Assembly

#### 3.2.1.1 Bộ nhớ assembly

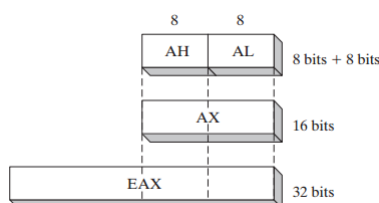
Biến và hằng trong assembly có tính chất và mục đích sử dụng khác nhau. Thông qua các câu lệnh liên kết chặt chẽ với bộ nhớ là các thanh ghi bộ nhớ, trạng thái, các cờ được đánh dấu mà thông qua đó để thực thi chương trình.

Các thanh ghi bộ nhớ:



Hình 3.1: Các thanh ghi <sup>1</sup>

Trong đó các thanh ghi General-Purpose gồm những thanh ghi có số bit nhỏ hơn:



Hình 3.2: Các thanh ghi <sup>2</sup>

<sup>1</sup> Trích dẫn hình ảnh trong sách Assembly Language

<sup>2</sup> Trích dẫn hình ảnh trong sách Assembly Language

Ví dụ: thanh ghi EAX(32 bit địa chỉ) có thể được chia ra gồm thanh ghi AX(16 bit địa chỉ). Thanh ghi AX tiếp tục chia địa chỉ thành AH(8 bit) và AL(8 bit). Việc chia địa chỉ như thế giúp tối ưu bộ nhớ khi sử dụng.

32 Bit	16 Bit	8Bit	8 Bit
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

Bảng 3.1: Bảng chia bộ nhớ 32-bit, 16-bit, 8-bit

Đối với một số thanh ghi có 32-bit địa chỉ có thể chia ra 16-bit địa chỉ, không thể chia nhỏ hơn. Bảng chi bộ nhớ thanh ghi có thể chia ra 16-bit địa chỉ.

32 Bit	16 Bit
ESI	SI
EDI	DI
EBP	BP
ESX	SP

Bảng 3.2: Bảng chia bộ nhớ 32-bit,16-bit

Mỗi thanh ghi có một chức năng khác nhau:

- EAX: thường được sử dụng trong các phép tính số học: cộng, trừ, nhân, chia, các phép toán logic, chuyển dữ liệu.
- EBX: được sử dụng như thanh ghi địa chỉ .
- ECX: được sử dụng trong các phép lặp, chuyển bit địa chỉ, xoay bit địa chỉ.
- EDX: thường được dùng để lưu dữ liệu, kết hợp với thanh ghi EAX thực hiện các phép toán nhân, cộng.
- ESP: luôn trở thối địa chỉ đỉnh stack hiện thời.



- EBP: thanh ghi dùng để truy cập giá trị, nhập dữ liệu stack. Khác với thanh ghi ESP, thanh ghi EBP còn được sử dụng để truy cập trong các đoạn chương trình khác nhau.
- EIP: thanh ghi con trỏ chỉ dẫn, giá trị của thanh ghi không thể thay đổi một cách trực tiếp, chỉ thay đổi khi trỏ tới câu lệnh tiếp theo. Thanh ghi EIP lưu giá trị trỏ tới của câu lệnh tiếp theo sẽ được gọi.
- ESI, EDI: thanh ghi chỉ số nguồn và chỉ số đích thường được dùng trong các thao tác, xử lý mảng và chuỗi.

Thanh ghi EFLAGS bao gồm các chuỗi nhị phân được điều khiển bởi CPU hoặc phản ánh một số phép toán của CPU. Trạng thái của CPU. Bao gồm 8 cờ hiệu với mỗi loại cờ có chức năng khác nhau:

- CF(Carry Flag cờ nhớ): cờ được bật khi phép tính có nhớ bit.
- ZF(Zero Flag cờ 0): cờ được bật khi phép tính vừa thực hiện là 0.
- SF(Sign Flag cờ dấu): cờ này được bật khi kết quả thực phép tính có bit dấu.
- OF(Overflow Flag cờ tràn): cờ được bật khi kết quả thực hiện phép tính bị tràn số học.
- PF(Parity Flag cờ chẵn lẻ): cờ được bật khi kết quả của phép tính có chẵn bit 1.
- AF(Auxiliary Flag cờ nhớ phụ): cờ được bật khi phép tính được thực hiện có sử dụng bit nhớ phụ.
- IF(Interrupt Flag cờ ngắt): cờ được bật khi có thông báo cho phép ngắt xảy ra.
- DF(Direction Flag cờ hướng): cờ được bật và sử dụng khi thao tác với mảng và chuỗi, sử dụng để giảm chỉ số tự động thì thao tác.

### 3.2.1.2 Câu lệnh assembly

Một câu lệnh assembly có cú pháp đầy đủ như sau:

[ *Nhãn lệnh:* ] < *Tên lệnh* < [ *Các toán hạng* ] [ ;*lời chú thích* ]

Trong đó:

- [ *Nhãn lệnh:* ] là một chuỗi ký tự kết thúc bằng dấu “:”, được thanh thế cho địa chỉ câu lệnh, được sử dụng trong câu lệnh If, else hoặc cần gọi thay bằng địa

chỉ câu lệnh. Trong một chương trình không thể có hai nhãn lệnh trùng tên, tên nhãn lệnh không được trùng với tên thủ tục.

- <Tên lệnh > là một trong số những lệnh gọi nhớ của mã assembly, không phân biệt chữ hoa, chữ thường. Mỗi dòng lệnh chỉ đảm nhận một câu lệnh, mỗi câu lệnh phải được đặt trên một dòng.
- [Các toán hạng] là các đối tượng mà câu lệnh sẽ tác động vào. Tùy theo từng câu lệnh mà có 0 toán hạng, 1 toán hạng, 2 toán hạng, 3 toán hạng. Toán hạng đầu tiên gọi là toán hạng đích, toán hạng thứ 2 gọi là toán hạng nguồn. Với câu lệnh có 3 toán hạng thì chỉ có 1 toán hạng đích trong câu lệnh đó.
- [; lời giải thích] khi biên dịch thì lời giải thích không được biên dịch sang mã máy, chỉ có tác dụng với người đọc chương trình. Giúp người đọc dễ hiểu các câu lệnh.

Ví dụ 1: xét câu lệnh sau đây:

VD1: mov ax, bx ;lưu giá trị thanh ghi ax từ thanh ghi bx

Trong đó:

- VD1 : là chuỗi ký tự nhãn lệnh.
- mov: tên lệnh với chức năng lưu giá trị đích(ax) từ giá trị nguồn (bx).
- ax, bx: các toán hạng, trong trường hợp này là các thanh ghi bộ nhớ 16 bit.
- ; lưu giá trị thanh ghi ax từ thanh ghi bx : lời giải thích, chú thích thêm.

Xem các câu lệnh sau đây:

- NOT : đây là câu lệnh không có toán hạng.
- Mov ax, bh: câu lệnh này có 2 toán hạng ax (toán hạng đích) thanh ghi 16 bit, bh (toán hạng nguồn) thanh ghi 8 bit.
- Add ah, sqt: câu lệnh này có 2 toán hạng ah (toán hạng đích) thanh ghi 8 bit, sqt (toán hạng nguồn) một biến được gán trước có kiểu dữ liệu là byte.
- Mov ax, [SI]: câu lệnh này có 2 toán hạng ax (toán hạng đích) thanh ghi 16 bit, [SI] (toán hạng nguồn) là một ô nhớ.
- Imul ax, bx, 10: câu lệnh này có 3 toán hạng ax (toán hạng đích) thanh ghi 16 bit, bx (toán hạng nguồn) thanh ghi 16 bit, và một hằng số (10).

Câu lệnh Imul có tới 3 toán hạng, toán hạng đích (ax) sẽ được lưu kết quả của phép nhân từ toán hạng nguồn (bx) nhân với hằng số (10: tức toán hạng thứ 3).

Bảng ký hiệu toán hạng:

Toán hạng	16 Mô tả
reg8	Thanh ghi có 8 bit địa chỉ: AH, AL, BH, BL, CH, CL, DH, DL
reg16	Thanh ghi có 16 bit địa chỉ: AX, BX, CX, DX, SI, DI, SP, BP
reg32	Thanh ghi có 32 bit địa chỉ: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EB
reg	Bất kỳ thanh ghi nào
sreg	Thanh ghi segment có 16 bit địa chỉ: CS, DS, SS, ES, FS, GS
imm	8-, 16-, hoặc 32- bit giá trị được truyền vào
imm8	Giá trị hằng số truyền vào kiểu byte 8-bit
imm16	Giá trị hằng số truyền vào kiểu word 16-bit
imm32	Giá trị hằng số truyền vào kiểu doubleword 32-bit
reg/mem8	Toán hạng 8 bit, có thể là thanh ghi có 8-bit địa chỉ hoặc bộ nhớ byte
reg/mem16	Toán hạng 16 bit, có thể là thanh ghi có 16-bit địa chỉ hoặc bộ nhớ word
reg/mem32	Toán hạng 32 bit, có thể là thanh ghi có 32-bit địa chỉ hoặc bộ nhớ doubleword
mem	Bất kỳ bộ nhớ 8-, 16-, 32- bit

Bảng 3.3: Bảng ký hiệu toán hạng:

Phân tích câu lệnh mov:

Cú pháp:

- Mov [toán hạng đích], [toán hạng nguồn].
- Mov reg/mem8, reg8
- Mov reg/mem16, reg16
- Mov reg/mem32, reg32

- Mov reg8, reg/mem8
- Mov reg16, reg/mem16
- Mov reg32, reg/mem32
- Mov reg/mem16, sreg
- Mov sreg, reg/mem16
- Mov reg8, imm8
- Mov reg16, imm16
- Mov reg32, imm32
- Mov reg/mem8, imm8
- Mov reg/mem16, imm16
- Mov reg/mem32, imm32

Trong đó:

- [ Toán hạng đích]: có thể là thanh ghi (8 bit, 16 bit, 32 bit), ô nhớ (địa chỉ của ô nhớ) hay một biến nào đó.[Toán hạng đích]không thể là hằng số.
- [ Toán hạng nguồn ]: có thể là hằng số, biến, thanh ghi, ô nhớ.

Xem ví dụ sau:

- Mov ax, bx ; đặt giá trị của thanh ghi bx vào ax.
- Mov ax, 5 ; đặt giá trị 5 vào thanh ghi ax.
- Mov bx, 5\*3 ; đặt giá trị 5\*3 vào thanh ghi bx.
- Mov DI, 'A' ; đặt mã ASCII của 'A' vào thanh ghi DI
- Mov ch, var ; đặt giá trị của biến var kiểu byte vào thanh ghi ch.
- Mov bh, 300 ; giá trị không hợp lệ vì bh có kiểu byte giới hạn 255 nên không thể gán giá trị 300 vào bh.
- Mov ch, ax ; giá trị không hợp lệ vì giá trị thanh ghi ax (16 bit) không thể gán giá trị vào thanh ghi ch (16 bit).

### 3.2.1.3 Tập giá trị

Khác với các ngôn ngữ lập trình khác, kiểu giá trị của assembly có đôi chút khác biệt, được tính theo bit, tùy theo yêu cầu sử dụng của người lập trình.

Các kiểu dữ liệu:

Kiểu	Cách sử dụng
BYTE	kiểu số nguyên 8-bit không dấu.
SBYTE	kiểu số nguyên 8-bit có dấu.
WORD	kiểu số nguyên 16-bit không dấu.
SWORD	kiểu số nguyên 16-bit có dấu.
DWORD	kiểu số nguyên 32-bit không dấu.
SDWORD	kiểu số nguyên 32-bit có dấu.
FWORD	kiểu số nguyên 48-bit.
QWORD	kiểu số nguyên 64-bit
TBYTE	kiểu số nguyên 80-bit (10-byte).
REAL4	kiểu số thực 32-bit (4-byte) chuẩn IEEE.
REAL8	kiểu số thực 64-bit (8-byte) chuẩn IEEE.
REAL10	kiểu số thực 80-bit (10-byte) chuẩn IEEE.

Bảng 3.4: Bảng kiểu dữ liệu:

Ngoài ra còn có một số ký hiệu kiểu số tự nhiên.

Kiểu	Cách sử dụng
DB	kiểu số nguyên 8-bit.
DW	kiểu số nguyên 16-bit.
DD	kiểu số nguyên hoặc số thực 32-bit.
DQ	kiểu số nguyên hoặc số thực 64-bit.
DT	định nghĩa kiểu số nguyên 80-bit (10 byte).

Bảng 3.5: Bảng ký hiệu kiểu số tự nhiên:

Trên cơ sở các đơn vị dữ liệu được lưu trên kiến trúc X86, một byte tương ứng với 8-bit, từng từ word tương ứng với 16-bit (2 byte), doubleword tương ứng với 32-bit (4 byte), quadword tương ứng với 64-bit (8 byte). Từ đó ta tính toán được phạm vi giới hạn số học của các kiểu số nguyên không dấu và có dấu:

Kiểu lưu trữ	Phạm vi (từ thấp đến cao)	Dung lượng
Byte không dấu	0 đến 255	1 byte
Word không dấu	0 đến 65,535	2 bytes
Double word không dấu	0 đến 4,294,967,295	4 bytes
Quadword không dấu	0 đến 18,446,744,073,709,551,615	8 bytes

Bảng 3.6: Bảng phạm vi kiểu dữ liệu số không dấu

Kiểu lưu trữ	Phạm vi (từ thấp đến cao)
Byte có dấu	-128 đến +127
Word có dấu	-32,768 đến +32,767
Double word có dấu	-2,147,483,648 đến +2,147,483,647
Quadword có dấu	-9,223,372,036,854,775,808 đến +9,223,372,036,854,775,807

Bảng 3.7: Bảng phạm vi kiểu dữ liệu số có dấu

#### 3.2.1.4 Kiểu giá trị

##### Hằng số

Kiểu hằng số nguyên được dùng trong hợp ngữ assembly là một chuỗi số đứng trước nó là dấu của số nguyên đó, tiếp theo là chuỗi số và cuối cùng là cơ số để xác định chuỗi số đó. Cú pháp của chuỗi số:

$$[+|-] \text{ chuỗi số } [\text{cơ số}]$$

Cơ số là một trong những dạng cơ số dưới đây (không phân biệt chữ hoa hay chữ thường):

- h : Hexadecimal hệ cơ số 16
- q/o : Octal hệ cơ số 8
- d : Decimal hệ cơ số 10
- b : Binary hệ cơ số 2

Nếu không có ký hiệu cơ số thì mặc định số đó là hệ cơ số 10. Dưới đây là một số ví dụ:

- 26 : 26 hệ cơ số 10
- 26d : 26 hệ cơ số 10
- 11010011b : 11010011 hệ cơ số 2
- 42q : 42 hệ cơ số 8
- 42o : 42 hệ cơ số 8
- 1Ah : 1A hệ cơ số 16

### Biểu thức số nguyên

Biểu thức số nguyên là các biểu thức toán học mà các toán hạng thuộc kiểu số nguyên. Các biểu thức toán học được xếp theo thứ tự ưu tiên, việc xếp thứ tự này khá quan trọng vì nó ảnh hưởng đến kết quả của một biểu thức. Dưới đây là bảng biểu thức toán học được sắp thứ tự ưu tiên cao nhất là 1 và thấp nhất là 4:

Toán tử	Tên	Thứ tự ưu tiên
()	Dấu ngoặc	1
+, −	Dấu dương, âm	2
*, /	Nhân, chia	3
MOD	Chia lấy dư	3
+, −	Cộng, trừ	4

Bảng 3.8: Thứ tự ưu tiên biểu thức toán

Xem các ví dụ dưới đây để hiểu rõ hơn:

- $4 + 5 * 3$  : Thực hiện phép nhân trước khi thực hiện phép cộng
- $12 - 2 \text{ MOD } 3$  : Thực hiện phép chia lấy dư trước khi thực hiện phép trừ

- $-7 + 2$  : Lấy dấu của số 7 trước khi thực hiện phép cộng với 2
- $(5 + 3) * 7$  : Thực hiện biểu thức trong ngoặc trước khi thực hiện phép nhân.

### Kiểu ký tự và chuỗi

Kiểu ký tự và chuỗi ký tự trong hợp ngữ assembly dựa trên bảng mã ASCII để xây dựng. Biến có kiểu ký tự được lưu dưới dạng mã nhị phân của bảng mã ASCII. Cách khai báo biến có kiểu ký tự:

- 'A'
- "b"

Một chuỗi ký tự được đặt trong dấu nháy đơn (') hoặc nháy kép (""):

- 'ABCDEF'
- "AAAA"
- "Hom nay la thu hai"
- '123456'

#### 3.2.1.5 Cấu trúc một chương trình assembly

Hiện nay, hầu hết các hệ điều hành hiện nay, đặc biệt là hệ điều hành Microsoft đều hỗ trợ hai dạng cấu trúc tập tin thực thi là COM và EXE. Nhưng trong BE-PUM tập trung phân tích cấu trúc tập tin EXE. Cấu trúc tập tin EXE và COM có sự khác biệt rất lớn. Cấu trúc tập tin EXE được chia thành ba đoạn: Mã lệnh (Code), dữ liệu (Data), ngăn xếp (Stack). Trong khi đó cấu trúc tập tin COM chỉ có một đoạn mã lệnh được gom từ ba đoạn trong cấu trúc tập tin EXE.

Ngoài ra, cấu trúc tập tin EXE có thể bố trí hơn ba đoạn bộ nhớ. Do đó, khi thiết kế chương trình, hay gặp phải chương trình được bố trí hơn ba đoạn thì cần phải quan tâm đến các modul của chương trình, sự liên kết giữa các đoạn với nhau.

Cấu trúc chương trình được giới thiệu, phân tích dưới đây là cấu trúc tập tin thực thi EXE, nêu lên những khái niệm cơ bản của một chương trình assembly.

. Model < chế độ bộ nhớ >



```
.Stack 100h  
<khai báo dữ liệu>  
.Code  
<Thủ tục chính > PROC  
  
<các câu lệnh của chương trình>  
  
< Thủ tục chính>  
Endp  
<Các thủ tục khác được khai>  
END
```

Trong cấu trúc được đưa ra, các từ *.Model*, *.Stack*, *.Data*, *.Code*, *PROC*, *Endp*, *END* là các từ để hướng dẫn thực thi một chương trình assembly.

Nhìn vào cấu trúc chương trình, ta thấy rõ một chương trình assembly được phân tích ra gồm 3 đoạn chính: đoạn *Code*, chứa toàn bộ mã lệnh, nơi thực thi của chương trình, đoạn *Data* nơi chứa các biến dữ liệu được khai báo của chương trình, đoạn *Stack* nơi chứa stack của chương trình khi chương trình được nạp vào bộ nhớ để thực thi.

Hướng dẫn *.Model* được đặt ngay trên đầu của cấu trúc chương trình nhằm mục đích khai báo chế độ nhớ mà chương trình sử dụng để thực thi.

Hướng dẫn *.Stack* đặt ở đầu chương trình mục đích để khai báo kích thước stack được sử dụng để thực thi chương trình. Kích thước stack thường được khai báo là 100h (256) byte.

Ví dụ chương trình được viết theo cấu trúc EXE:

```
.model small

    include |masm32 |include |windows.inc
include |masm32 |include |windows.inc
include |masm32 |include |windows.inc
include |masm32 |include |windows.inc

include |masm32 |include |windows.inc
include |masm32 |include |windows.inc
include |masm32 |include |windows.inc
include |masm32 |include |windows.inc
stack 100h

.data

var1 WORD 200

.code
start:

main proc
    mov ax, 775
    aad
    mov eax, 100
    aad
    mov bl, 5
    div bl
    mov eax, 10
    bsf eax, var1
    ret
main endp

end start
```

Ở phần đầu chương trình có khai báo các đường dẫn thư viện hỗ trợ chương trình bằng cú pháp:

*include <đường dẫn>*

Có thể thấy đoạn chương trình trên sử dụng chế độ bộ nhớ Small. Khai báo kích thước Stack là 100h byte. Trong phần khai báo dữ liệu Data chỉ khai báo một biến duy nhất là var1 có kiểu dữ liệu là word, giá trị của biến là 200. Trong phần Code, tên thủ tục là main, tên thủ tục có thể được đặt tùy ý.

### 3.2.2 Floating-Point Unit (FPU)

Kiến trúc Floating-Point Unit (FPU) của Intel cung cấp hiệu quả cao trong khả năng xử lý floating-point. Kiến trúc FPU hỗ trợ xử lý các số nguyên, số thực và Binary Coded Decimal (BCD)-số nguyên kiểu dữ liệu, và các floating-point được xử lý bằng thuật toán được định nghĩa theo tiêu chuẩn IEEE 754 và 854. Các câu lệnh FPU được thực hiện từ bộ vi xử lý của các dòng lệnh bình thường và được cải thiện đáng kể hiệu quả của các bộ vi xử lý Intel trong xử lý các phép tính có độ chính xác cao, các hoạt động xử lý dấu chấm động.

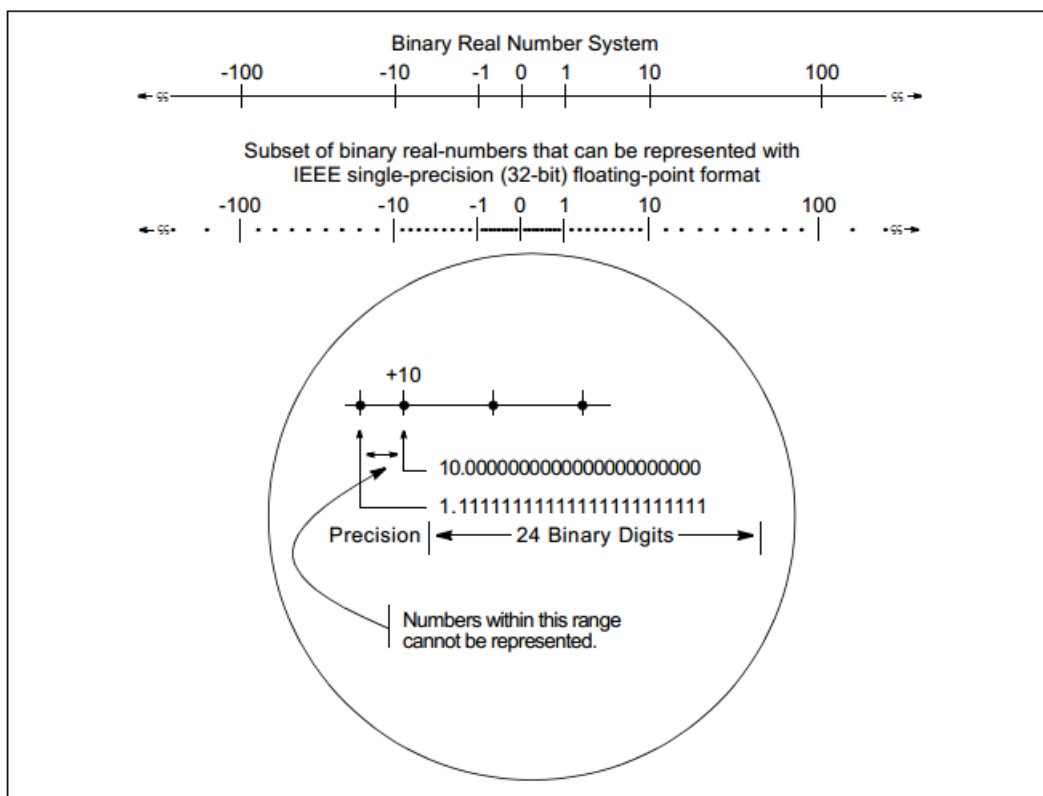
#### 3.2.2.1 Số thực và định dạng Floating-point

Phần này sẽ mô tả cách các số thực được biểu diễn ở dạng dấu chấm động của FPU trong kiến trúc Intel. Đồng thời giới thiệu các thuật ngữ như số normalized, số denormalized, số mũ, số không, và NaN (not a number).

#### Hệ thống số thực

Như thể hiện trong hình 3, hệ thống số thực bao gồm sự liên tục của các số thực từ trừ vô cực ( $-\infty$ ) đến cộng vô cực ( $+\infty$ ).

Bởi vì kích thước và số lượng thanh ghi của bất kỳ máy tính nào có thể có hạn chế, chỉ có một đoạn mã nhị phân của số thực có thể được sử dụng trong tính toán số thực.



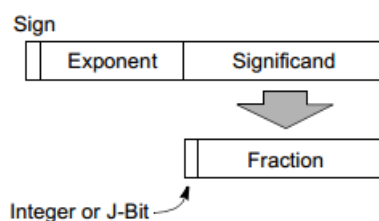
Hình 3.3: Hệ thống mã nhị phân biểu diễn số thực

Hình 3, phần số thực mà FPU hỗ trợ biểu diễn một giá trị xấp xỉ của một số thực. Phạm vi và độ chính xác của các phần mã nhị phân của số thực được biểu diễn theo định dạng FPU sử dụng để hiển thị cho số thực được sử dụng.

### Định dạng Floating-point

Để tăng tốc độ và hiệu quả của các phép tính số thực, các FPU hiện thị đại diện cho một số thực mà được biểu diễn theo định dạng floating-point. Trong định dạng này, một số thực có ba phần: bit dấu, định trị và số mũ. Định dạng này phù hợp với chuẩn IEEE.

Bit dấu là một giá trị nhị phân cho biết xem số đó là số âm (1) hay số dương (0). Phần định trị gồm hai phần: bit số nguyên (J-bit) và phần phân số. Phần bit số nguyên thường không được hiển thị, nhưng thay vào đó có giá trị mặc định. Phần số mũ là một số nguyên biểu diễn lũy thừa 2, điều này làm cho phần định trị được tăng lên.



Hình 3.4: Định dạng mã nhị phân của Floating-point

### Số thông thường (normalized)

Trong hầu hết mọi trường hợp, thanh ghi FPU hiển thị giá trị số thực định dạng thông thường. Ngoại trừ số 0 thì phần định trị luôn được tạo bởi một số thực và theo sau là phần phân số.

- 1.fff...fff

Ký hiệu	Giá trị
Số thập phân	178.125
Số thập phân khoa học	1.78124E102
Số nhị phân khoa học	1.0110010001E2111
Số nhị phân khoa học (biểu diễn số mũ)	1.0110010001E210000110
Định dạng số thực	0(Bit dấu) 10000110(Phần mũ) 011001000100000000000001(Phần định trị)

Đối với các giá trị thấp hơn 1 thì số 0 ở phía trước được loại bỏ (mỗi số 0 được loại bỏ, số mũ được tăng lên 1).

Để hiển thị số lớn nhất của phần định trị theo định dạng thông thường thì phải cung cấp độ dài của phần định trị này. Để hiển thị số thực theo định dạng thông thường này thì phần định trị sẽ biểu diễn giá trị giữa 1 và 2, phần số mũ sẽ chỉ ra độ tăng của số thực đó.

### Số mũ (baised expoment)

Thanh ghi FPU biểu diễn số mũ theo định dạng cơ bản. Có nghĩa là một hằng số sẽ được thêm vào số mũ được biểu diễn vì vậy<sup>31</sup> số mũ luôn là một số dương. Giá trị hằng số phụ thuộc vào số lượng bit được dùng để biểu diễn số thực trong định dạng dấu chấm phẩy động được sử dụng. Hằng số được chọn sao cho có thể thay đổi giá trị nhưng vẫn

### Số thực và mã hóa phi số thực

Có nhiều loại số thực và giá trị đặc biệt có thể được mã hóa theo định dạng dấu phẩy động của FPU:

Những số đặc biệt và giá trị đặt biệt được chia thành các loại sau:

- Số 0.
- Số hữu hạn không thông thường (denormalized).
- Số hữu hạn thông thường (normalized).
- Số vô cực
- Không phải là một số (NaNs: Not a number )
- Số không định nghĩa.

Hình 3.5 sẽ chỉ rõ cách mã hóa số thực và phi số thực phù hợp với các biểu diễn số thực theo chuẩn IEEE. Ví dụ được sử dụng ở đây miêu tả chuẩn IEEE chính xác đơn (32-bit), các ký hiệu “S” chỉ bit dấu, “E” chỉ số mũ, “F” chỉ phần phân số. ( Giá trị của phần số mũ là một số nguyên).

Các thanh ghi FPU có thể thực hiện các loại and/or trả về đúng giá trị, phụ thuộc vào phép toán được thực hiện. Các phần dưới đây sẽ mô tả các biểu diễn số thực và phi số thực.

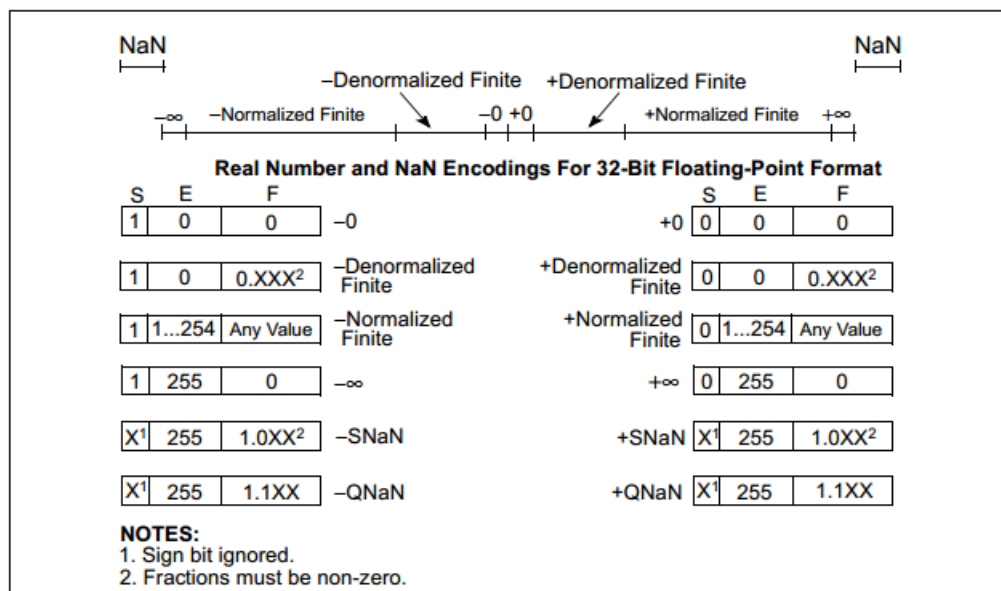
### Số 0

Số 0 có thể được biểu diễn là +0 và -0 phụ thuộc vào bit dấu. Việc mã hóa +0 và -0 tương đương nhau về mặt giá trị. Bit dấu của số 0 phụ thuộc vào phép toán được thực thi và cách thức làm tròn được sử dụng. Bit dấu số 0 có thể xác định tràn số dưới có đang xảy ra hay không, ngoài ra còn xác định bit dấu của giá trị vô cực.

### Số normalized và denormalized

Trừ số 0, số thực được chia thành 2 dạng: normalized (bình thường) và denormalized (không bình thường). Số normalized bao gồm các giá trị khác 0, có thể mã hóa theo định

dạng IEEE trong phạm vi từ số 0 đến vô cực. Trong hình 5, định dạng chính xác đơn với chỉ số mũ từ 0 đến 254 (phạm vi tham số -126 đến 126).



Hình 3.5: Số thực và NaN

Khi số thực gần về số 0, chiều dài bit dùng để biểu diễn số thực không đủ dài để biểu diễn. Bởi vì chỉ số mũ (trong ví dụ này là 32-bit) không đủ lớn để biểu diễn bit dịch phải để loại bỏ các bit 0 ở đầu (xem thêm phụ lục IEEE).

Khi số mũ là số 0, những số nhỏ có thể được biểu diễn bằng những bit số nguyên (và những chuỗi bit ở đầu) của số 0. Số thực trong phạm vi này được gọi là denormalized (hay số rất nhỏ). Việc sử dụng các bit 0 ở đầu cho phép biểu diễn các số rất nhỏ nhưng những số rất nhỏ này là nguyên nhân mất đi sự chính xác (số bit trong phần định trị được giảm đi sẽ được tính toán dựa trên số bit 0 ở phần số mũ).

Khi thực hiện các phép tính trên dấu chấm động, mỗi thanh ghi FPU thực hiện phép toán với số normalized và đưa ra kết quả là một số normalized. Số denormalized biểu diễn cho điều kiện tràn số dưới.

Số denormalized được tính bằng kỹ thuật “gradual underflow”. Bảng () sẽ đưa ra một ví dụ của kỹ thuật “gradual underflow” trong xử lý tính toán số denormalized. Ở đây, sử

dụng định dạng đơn nên số nhỏ nhất của số mũ là -126. Kết quả trong ví dụ này yêu cầu số mũ là -129. Nếu nhỏ hơn -129 là nằm ngoài phạm vi số mũ cho phép, kết quả của số denormalized bằng cách thêm các bit 0 ở đầu đến khi số mũ nhỏ nhất là -129.

Toán hạng	Dấu	Số mũ*	Phần định trị
Kết quả đúng	0	-129	1.01011100000..00
Số Denormalize	0	-128	0.10101110000..00
Số Denormalize	0	-127	0.01010111000..00
Số Denormalize	0	-126	0.00101011100..00
Kết quả Denormalize	0	-126	0.00101011100..00

Bảng 3.10: Xử lý số denormalized

Lưu ý: \* Số mũ được biểu diễn trong phạm vi (-126 đến 126)

Trong trường hợp tốt nhất, tất cả các bit của phần định trị được dịch phải bằng cách chèn thêm bit 0 ở đầu, tạo ra kết quả là số 0.

FPU xử lý với số denormal theo các cách sau đây:

- Không được phép tạo số denormalized.
- Cung cấp xử lý ngoại lệ underflow để người lập trình có thể kiểm tra khi số denormalized được tạo ra.
- Cung cấp các toán hạng xử lý số denormalized, cho phép chương trình kiểm tra khi mà các số denormalized được sử dụng như là một toán hạng.

Khi một số denormalized ở định dạng chính xác đơn hoặc chính xác kép được sử dụng như một toán hạng, và xử lý ngoại lệ denormal được đánh dấu, thanh ghi FPU sẽ tự động chuyển giá trị của số denormalized thành normalized bằng cách chuyển sang định dạng mở rộng.



### Số vô cực

Có 2 số vô cực là âm vô cực và dương vô cực được dùng để biểu diễn số thực âm nhỏ nhất và số thực dương lớn nhất vì vậy có thể biểu diễn dạng dấu chấm động. Số vô cực luôn được biểu diễn bằng bit 0 của phần định trị (phần phân số và bit số nguyên) và số mũ tối đa cho phép của mỗi định dạng (ví dụ: số mũ tối đa 255 đối với định dạng kép).

Bit dấu của số vô cực có thể dùng để so sánh. Số vô cực luôn được chỉ định rõ là âm vô cực luôn nhỏ hơn so với bất kỳ số thực cụ thể nào, dương vô cực luôn lớn hơn so với bất kỳ số thực cụ thể nào. Việc tính toán số vô cực luôn chính xác. Ngoại lệ được đánh dấu khi có phép toán xử dụng toán hạng là số vô cực, số vô cực được coi là toán hạng không hợp lệ.

Số denormalized biểu diễn cho điều kiện tràn số dưới, 2 số vô cực biểu diễn cho điều kiện tràn số trên. Kết quả số normlized của một phép toán có số mũ nằm trong phạm vi cho phép của số mũ phụ thuộc theo định dạng được sử dụng.

### NaNs (Not a number: không phải là một số)

NaN không phải là một, NaN là phần không nằm trên trục số thực trong hình 5. Việc mã hóa số NaN được chỉ ra trên trục số thực nằm ở cuối và đầu trục số. Khoảng trống này biểu diễn bất kỳ giá trị nào lớn với số mũ tối đa và phần phân số có bit khác không. (Bit dấu được bỏ qua đối với số NaN).

Theo chuẩn IEEE có 2 định dạng cho NaN là: quiet NaNs(QNaNs) và signaling NaNs (SNaNs). Số QNaNs là NaN với phần phân số có hầu hết các bit là 1. Số SNaNs là NaN với phần phân số hầu hết các bit là 0. QNaNs cho phép tạo ra thông qua các phép toán mà không đánh dấu ngoại lệ nào. SNaNs thường thông báo một ngoại lệ không hợp lệ xảy ra khi thực hiện phép toán học.

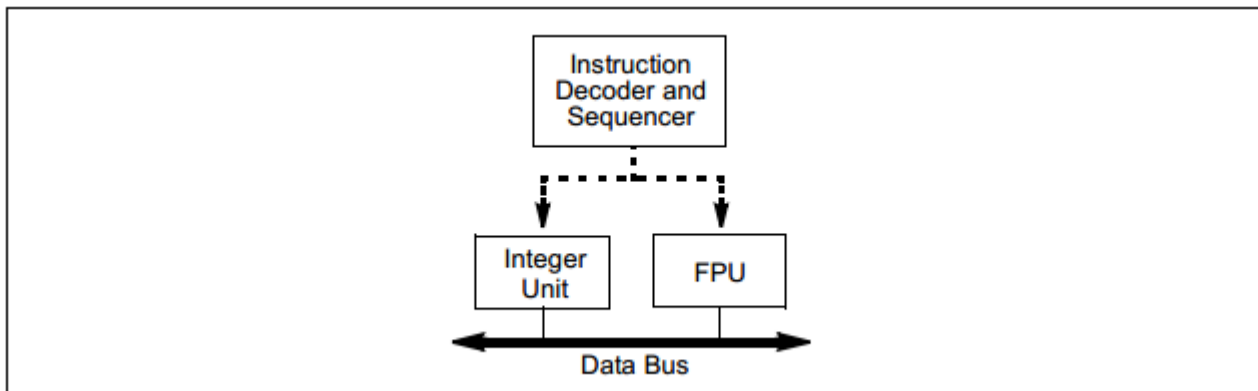
### Số không định nghĩa

Đối với mỗi loại dữ liệu của FPU, có một cách để mã hóa giá trị đặc biệt được gọi là giá trị không định nghĩa. Ví dụ, khi tính toán trên trường số thực, số không được định

nghĩa là số QNaNs. FPU tạo ra giá trị không định nghĩa khi giá trị được đánh dấu xử lý ngoại lệ.

### 3.2.2.2 Kiến trúc FPU

Nhìn tổng quan kiến trúc, FPU là một bộ xử lý các thao tác hoạt động song song với bộ xử lý số nguyên. FPU có các câu lệnh cũng giống như các mã lệnh khác và được thực hiện tuần tự như bộ xử lý số nguyên và chia sẻ đường truyền với bộ xử lý số nguyên. Bộ xử lý FPU, bộ xử lý số nguyên hoạt động độc lập nhau và song song.



Hình 3.6: Mối quan hệ giữa bộ xử lý FPU và bộ xử lý Integer

Những câu lệnh được thực thi trong môi trường của FPU hình 3.6 bao gồm 8 thanh ghi dữ liệu (gọi là thanh ghi dữ liệu FPU) và sau đây là những thanh ghi với mục đích khác:

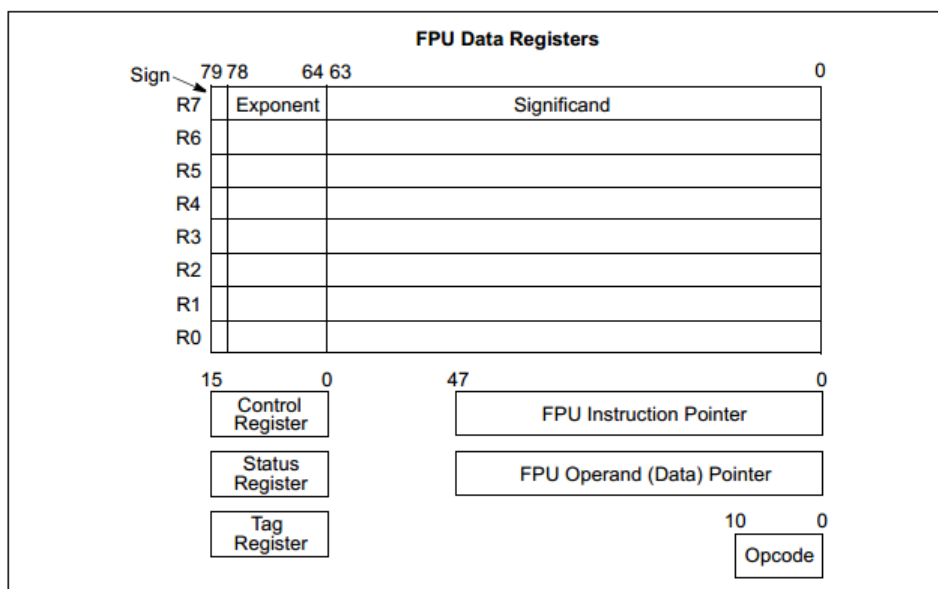
- Thanh ghi trạng thái (the status register).
- Thanh ghi điều khiển (the control register).
- Thanh ghi thẻ (the tag word register).
- Thanh ghi con trỏ lệnh (instruction pointer register).
- Thanh ghi toán hạng (last operand register).
- Thanh ghi Opcode (Opcode register).

Những thanh ghi này được miêu tả trong phần tiếp theo.

## Thanh ghi dữ liệu FPU

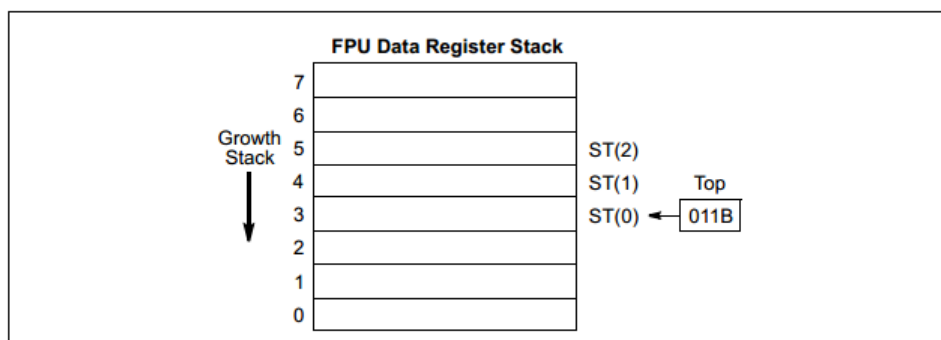
Các thanh ghi dữ liệu FPU (hình 3.7) bao gồm 8 thanh ghi 80-bit. Giá trị được lưu trữ trong các thanh ghi theo định dạng mở rộng. Khi các số thực, số nguyên hoặc giá trị BCD được nén được nạp vào từ bộ nhớ vào trong thanh ghi dữ liệu FPU, các giá trị sẽ tự động chuyển về định dạng mở rộng. Khi tính toán, kết quả được chuyển lại vào bộ nhớ từ bất kỳ thanh ghi FPU, kết quả có thể đã được định dạng mở rộng hoặc được chuyển về định dạng FPU khác (số thực, số nguyên, giá trị BCD được nén).

Những tập lệnh của FPU sẽ xử lý 8 thanh ghi dữ liệu như là một ngăn xếp (stack) các thanh ghi (hình 3.7) tất cả địa chỉ các thanh ghi dữ liệu là tương đối so với thanh ghi đỉnh của ngăn xếp. Số lượng thanh ghi ở đỉnh ngăn xếp hiện tại được lưu trữ trong trường TOP (stack TOP) trong thanh ghi trạng thái của FPU. Khi có 1 thao tác nạp thì TOP giảm đi 1 và nạp 1 giá trị vào thanh ghi ở đỉnh mới của ngăn xếp, và thao tác lưu trữ sẽ lưu trữ giá trị thanh ghi từ thanh ghi TOP hiện tại vào bộ nhớ.



Hình 3.7: Môi trường thực thi FPU

Nếu hoạt động nạp giá trị được thực hiện khi  $TOP = 0$  giá trị mới của TOP được thiết lập là 7. Ngoại lệ ngăn xếp bị tràn trên xảy ra giá trị được nạp không được ghi đè lên thanh ghi.



Hình 3.8: Ngăn xếp thanh ghi dữ liệu FPU

Có nhiều câu lệnh xử lý dấu chấm động cho phép người lập trình thao tác hoàn toàn trên đỉnh stack hoặc thao tác trên các thanh ghi cụ thể mà có quan hệ với đỉnh stack. Assembly hỗ trợ hóa địa chỉ thanh ghi, việc sử dụng thanh ghi ST(0) hoặc đơn giản là ST để biểu diễn thanh ghi hiện tại và ST(i) dùng để biểu diễn thanh ghi thứ i kể từ đỉnh stack ( $0 \leq i \leq 7$ ). Ví dụ, nếu đỉnh stack hiện tại là 011B (thanh ghi thứ 3 là đỉnh của stack), khi thực hiện câu lệnh toán học cộng thì nội dung của đỉnh stack sẽ thay đổi theo phép tính cộng hai thanh ghi trong stack ( thanh ghi 3 và 5).

*FADD ST, ST(2);*

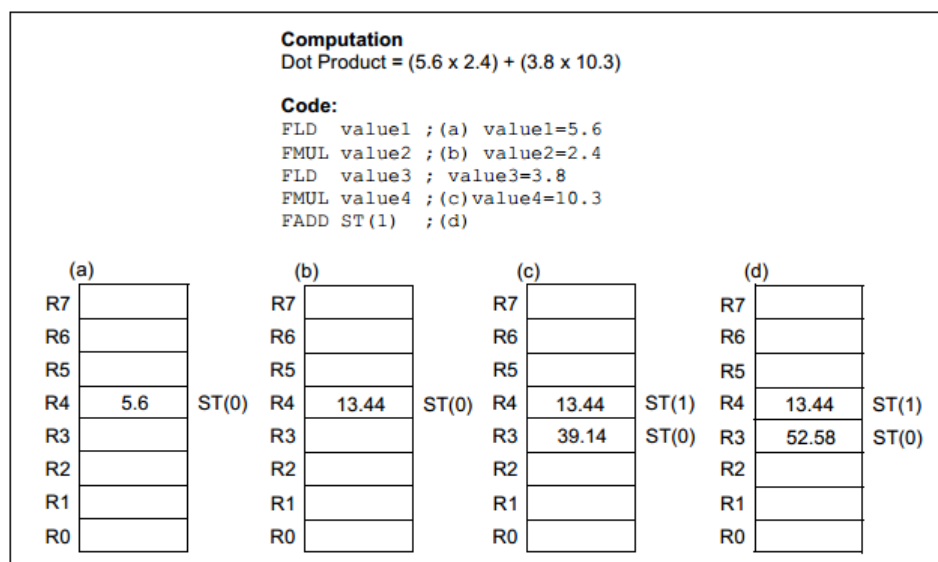
Hình 3.9 cho thấy một ví dụ về cấu trúc ngăn xếp của thanh ghi FPU và câu lệnh thường được sử dụng để thực hiện một phép toán.

- 1 Lệnh đầu tiên (FLD value1) tăng thanh ghi con trỏ của stack (hiện tại là đỉnh stack) và nạp giá trị 5.6 từ bộ nhớ vào ST(0). Kết quả của câu lệnh này được chỉ ra ở hình (a).
- 2 Lệnh thứ 2 nhân giá trị trong ST(0) với 2.4 từ bộ nhớ và lưu kết quả vào trong ST(0) như hình (b).
- 3 Lệnh thứ 3 tăng đỉnh stack lên và nạp giá trị 3.8 vào ST(0).
- 4 Lệnh thứ 4 nhân giá trị trong ST(0) với 10.3 từ bộ nhớ chính và lưu trữ kết quả trong ST(0) như hình (c).
- 5 Lệnh thứ 5 cộng thêm giá trị của đỉnh stack ST(0) với giá trị trong ST(1) và lưu kết quả vào trong ST(0) như hình (d).

Đây là một ví dụ trong việc thực thi một câu lệnh tính toán xử lý dấu chấm động.

### Truyền tham số vào stack của thanh ghi FPU

Cũng giống như các thanh ghi xử lý số nguyên, thanh ghi dữ liệu FPU không bị ảnh hưởng bởi các lời gọi thủ tục, hay cách gọi khác, các giá trị được giữ nguyên đến khi có câu lệnh muốn thực thi trên thanh ghi dữ liệu này. Một thủ tục có thể gọi để sử dụng giá trị cho một thủ tục hay câu lệnh khác. Việc gọi thanh ghi bằng cách thông qua con trỏ đỉnh stack hiện tại và cách đặt tên ST(0) và ST(i). Đây là cách gọi thông thường nhất khi muốn lấy giá trị của một thanh ghi trong stack và trả về kết quả trong đỉnh ST(0) khi thực thi câu lệnh hoặc của chương trình.



Hình 3.9: Ví dụ xử lý câu lệnh trong stack của FPU

## Thanh ghi trạng thái FPU

Thanh ghi trạng thái FPU 16 bit (hình 3.10) chỉ ra trạng thái hiện tại của FPU. Bit cờ của thanh ghi trạng thái bao gồm các cờ báo của FPU, con trỏ đỉnh stack, mã điều kiện, cờ báo lỗi, cờ báo lỗi ngoại lệ và cờ báo lỗi stack. FPU thiết lập những bit cờ trong thanh ghi để chỉ ra kết quả của phép toán được thực hiện.

Nội dung của thanh ghi trạng thái FPU có thể được lưu vào bộ nhớ bằng cách sử dụng câu lệnh FSTSW/FNSTSW, FSTEN/FNSTENV và FSAVE/FNSAVE. Cũng có thể lưu trữ trong thanh ghi AX của bộ xử lý số nguyên bằng cách sử dụng câu lệnh FSTSW/FNSTSW.

### Con trỏ đỉnh stack (TOP)

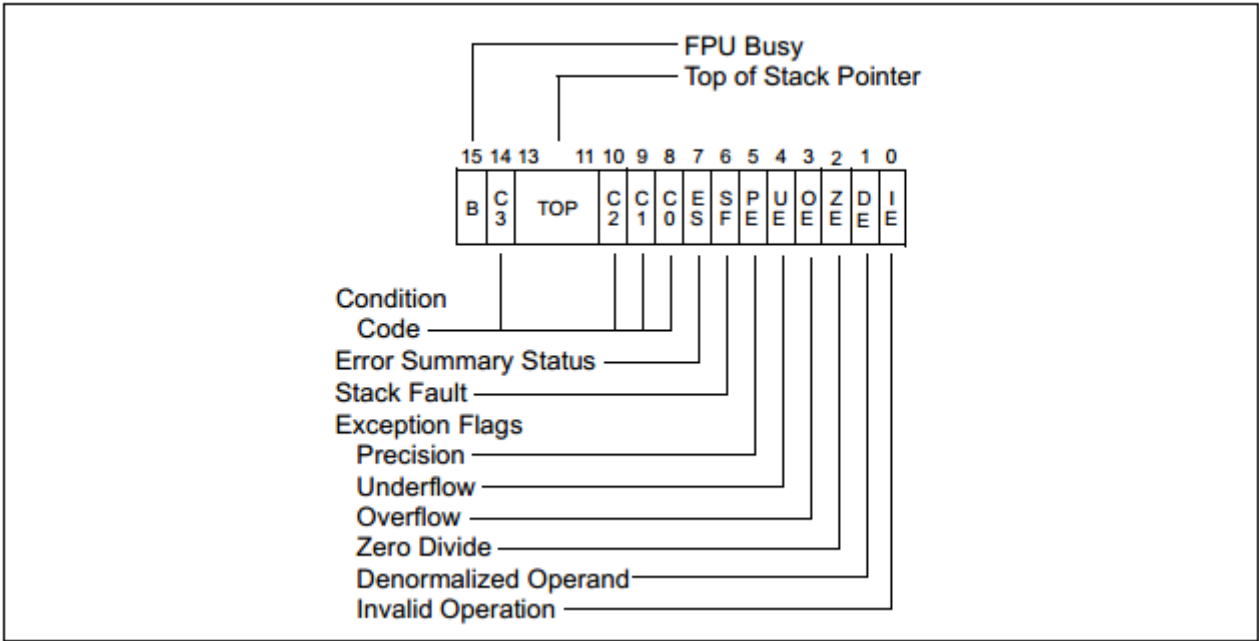
Con trỏ trỏ tới thanh ghi dữ liệu hiện tại ở đỉnh stack được chứa trong bit 11 đến 13 của thanh ghi trạng thái FPU. Con trỏ thường được gọi là TOP (đỉnh stack) có giá trị từ 0 đến 7.

### Cờ điều kiện

Bốn bit cờ điều kiện của thanh ghi FPU (từ C0 đến C3) chỉ ra kết quả của sự so sánh dấu chấm động và thao tác toán học. Bảng () tóm tắt các loại lệnh mà dấu chấm động sẽ thiết lập bit cờ điều kiện phục vụ cho việc phân loại xử lý ngoại lệ.

Như thể hiện trong bảng (), bit cờ điều kiện C1 được sử dụng trong nhiều câu lệnh. Khi bit cờ IE và SF được bật, để chỉ ra lỗi tràn trên hoặc tràn dưới của stack (IS), bit C1 sẽ giúp phân biệt giữa tràn trên ( $C1 = 1$ ) và tràn dưới ( $C1 = 0$ ). Khi bit cờ PE trong thanh ghi trạng thái được bật, để cho thấy kết quả đã được làm tròn, bit C1 sẽ được bật nếu số được làm tròn bằng cách làm tròn lên. Câu lệnh FXAM bật C1 theo bit dấu của giá trị được kiểm tra

.



Hình 3.10: Thanh ghi trạng thái FPU

Bit cờ điều kiện C2 được sử dụng bởi câu lệnh FPREM và FPREM1 để thể hiện phép tính khi sử dụng câu lệnh này không thực hiện được. Khi thực hiện được phép tính này, các bit cờ điều kiện C0, C1 và C3 sẽ được thay đổi theo kết quả của phép tính, dựa vào các bit cuối của kết quả tính được.

Các câu lệnh FPTAN, FSIN, FCOS và FSINCOS bật bit điều kiện C2 khi toán hạng được sử dụng trong câu lệnh này vượt quá phạm vi cho phép từ -263 đến +263.

Câu lệnh	C0	C1	C2	C3
FCOM, FCOMP, FCOMPP, FICOM, FI-COMP, FTST, FUCOM, FUCOMP, FUCOMPP	Kết quả so sánh		Không phụ thuộc phép so sánh	0 hoặc #IS
FCOMI, FCOMP, FU-COMI, FUCOMP	Không thiết lập (Các câu lệnh này thiết lập cờ EFLAGS)			#IS
FAXM	Toán hạng			Bit dấu

FPREM, FPREM1	Q2	Q1	0=Thực hiện được 1=Không thực hiện được	Q0 hoặc #IS
F2XM1, FADD, FADDP, FBSTP, FCMOV <sub>cc</sub> , FI- ADD, FDIV, FDIVP, FDIVR, FDIVRP, FIDIV, FIDIVR, FIMUL, FIST, FISTP, FISUB, FISUBR, FMUL, FMULP, FPATAN, FRNDINT, FSCALE, FST, FSTP, FSUB, FSUBP, FSUBR, FSUBRP, FSQRT, FYL2X, FYL2XP1	Không thiết lập			Làm tròn hoặc #IS
FCOS, FSIN, FSINCOS, FPTAN	Không thiết lập		1=Toán hạng nguồn nằm ngoài phạm vi giá trị.	Làm tròn hoặc #IS(Không thiết lập nếu C2=1)
FABS, FBLD, FCHS, FDECSTP, FILD, FINC- STP, FLD, Nạp hằng số, FSTP, FXCH, FXTRACT	Không thiết lập			0 hoặc #IS
FLDENV, FRSTOR	Mỗi bit được sao lưu vào trong bộ nhớ			



FFREE, FLDCW, FCLEX/FNCLEX, FNOP, FSTCW/FNSTCW, FSTENV/FNSTENV, FSTSW/FNSTSW	Không thiết lập			
FINIT/FNINIT, FSAVE/FNSAVE	0	0	0	0

### Cờ ngoại lệ

Sáu bit ngoại lệ (bit 0 đến 5) của thanh ghi trạng thái thể hiện một hoặc nhiều trường hợp ngoại lệ xử lý dấu chấm động sẽ được phát hiện từ khi giá trị cuối cùng của bit được chưa bật. (Cờ ngoại lệ sẽ được mô tả trong phần Xử lý ngoại lệ dấu chấm động).

Bit cờ tóm tắt trạng thái ngoại lệ (ES bit 7) được bật khi có bất kỳ cờ ngoại lệ nào đã được bật. Khi cờ ES được bật, FPU xử lý ngoại lệ được gọi sử dụng phần mềm xử lý ngoại lệ để xử lý. (Lưu ý, nếu một cờ xử lý ngoại lệ được bật, FPU sẽ vẫn tiếp tục bật những cờ xử lý ngoại lệ nếu có ngoại lệ xảy ra, nhưng cờ ES sẽ vẫn không được bật).

Khi cờ ngoại lệ được bật, những cờ được bật sẽ vẫn được bật đến khi cờ được tắt. Để tắt cờ xử lý ngoại lệ có thể sử dụng câu lệnh FCLEX/FNCELX, khởi tạo FPU với câu lệnh FINIT/FNINIT hoặc ghi đè cờ với câu lệnh FRSTOR, FLDENV.

B-bit (bit 15) phản ánh nội dung của cờ ES.

### Cờ lỗi stack

Cờ báo lỗi stack (bit 6 của thanh ghi trạng thái FPU) thể hiện stack đang bị tràn trên hoặc tràn dưới. FPU bật cờ SF khi phát hiện tràn trên hoặc tràn dưới, nhưng việc bật cờ SF không rõ ràng khi phát hiện một toán hạng không hợp lệ. Khi cờ SF được bật, cờ điều kiện C1 chỉ ra lỗi: tràn trên ( $C1 = 1$ ) và tràn dưới ( $C1 = 0$ ). Khi cờ SF được bật sẽ

được giữ nguyên đến khi nào được tắt. Để tắt thì thực thi các câu lệnh FINIT/FNINT, FCLEX/FNCLEX, hoặc FSAVE/FNSAVE).

## Phân nhánh và sao lưu có điều kiện dựa trên cờ điều kiện FPU

Kiến trúc Intel FPU (bắt đầu với các bộ xử lý Pentium Pro) hỗ trợ hai kỹ thuật nhánh và sao lưu có điều kiện theo cách so sánh hai giá trị dấu chấm động. Kỹ thuật được đề cập ở đây là kỹ thuật cũ và kỹ thuật mới.

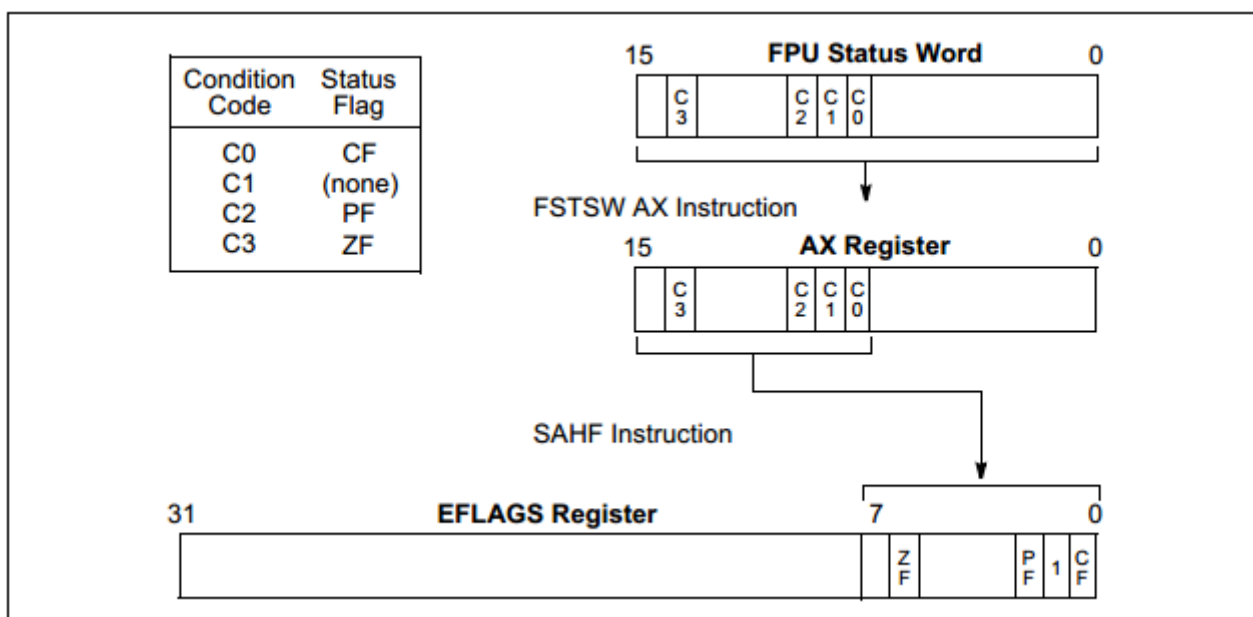
Kỹ thuật cũ có sẵn trong FPU của họ bộ xử lý Pentium. Kỹ thuật này sử dụng các câu lệnh so sánh dấu chấm động (FCOM, FCOMP, FCOMPP, FTST, FUCOMPP, và FICOMP) để so sánh hai giá trị dấu chấm động và thiết lập các cờ điều kiện (C0 đến C3) theo kết quả. Nội dung của cờ điều kiện sau đó được sao chép vào cờ trạng thái trong thanh ghi EFLAGS theo hai bước sau (hình 3.11) :

- 1 Câu lệnh sao lưu FSTSW AX chuyển trạng thái FPU vào trong thanh ghi AX.
- 2 Câu lệnh SAHF sao lưu 8-bit trên của thanh ghi AX, bao gồm cả cờ điều kiện, vào trong 8-bit thấp của thanh ghi cờ EFLAGS.

Khi cờ điều kiện đã được sao lưu vào thanh ghi cờ EFLAGS, điều kiện nhảy hoặc điều kiện sao lưu có thể được thực hiện dựa trên các thiết lập mới của cờ trạng thái trong thanh ghi EFLAGS.

Kỹ thuật mới chỉ có trong bộ vi xử lý Pentium Pro. Kỹ thuật này sử dụng các câu lệnh `FCMOVcc` để so sánh hai giá trị dấu chấm động và thiết lập cờ ZF, PF và CF của thanh ghi EFLAGS một cách trực tiếp. Để thực hiện việc thay đổi này cần một câu lệnh sử dụng kỹ thuật mới, nếu dùng kỹ thuật cũ cần tới ba câu lệnh.

Lưu ý rằng các câu lệnh `FCMOVcc` (chỉ có trong bộ xử lý Pentium Pro) cho phép sao lưu có điều kiện các giá trị dấu chấm động (giá trị này nằm trong thanh ghi dữ liệu FPU). Để sao lưu có điều kiện các giá trị dấu chấm động thì những câu lệnh này không xét đến cờ IF.



Hình 3.11: Sao lưu có điều kiện

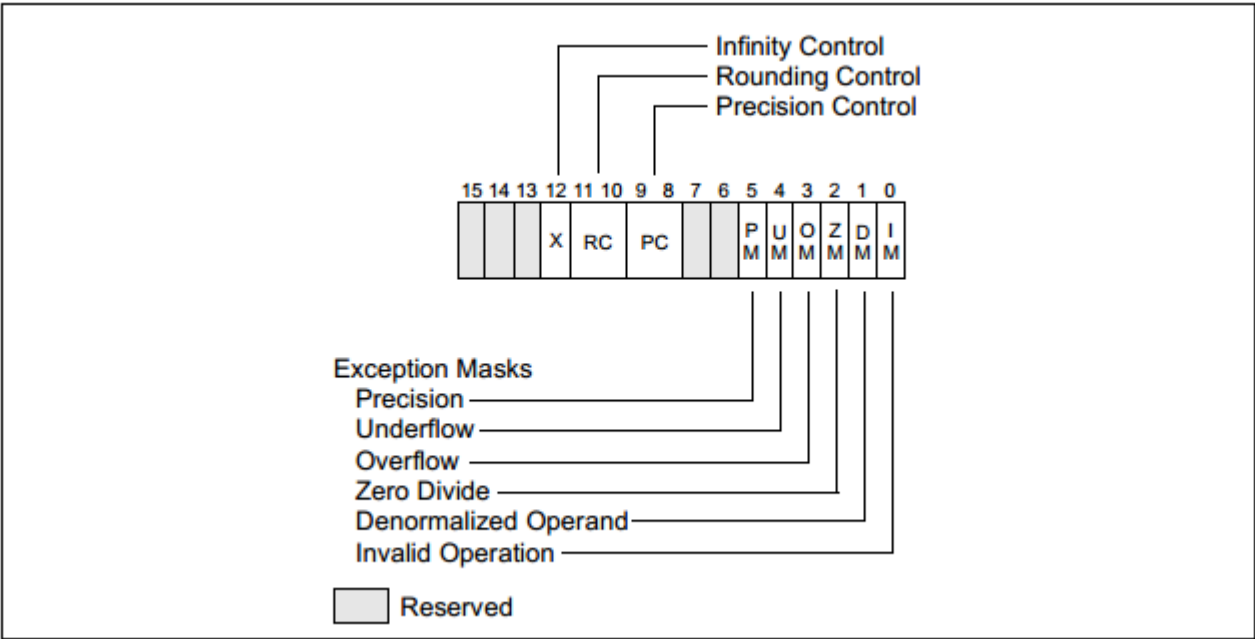
## Thanh ghi điều khiển FPU

Thanh ghi điều khiển FPU 16-bit (hình 3.12) điều khiển độ chính xác của FPU và phương thức làm tròn. Thanh ghi này cũng chứa các bit cờ xử lý ngoại lệ. Nội dung của thanh ghi điều khiển FPU có thể được nạp vào bằng câu lệnh FLDCW và được sao lưu vào bộ nhớ với câu lệnh FSTCW/FNSTCW.

Khi FPU được khởi tạo với một trong hai câu lệnh FINIT/FNINIT hoặc FSAVE/FNSAVE, thanh ghi điều khiển FPU tự động thiết lập thông số là 037h, các cờ xử lý ngoại lệ sẽ được bật, cờ làm tròn được thiết lập làm tròn đến số gần nhất, cờ độ chính xác được thiết lập là chính xác kép 64-bits.

### Cờ ngoại lệ

Cờ ngoại lệ (bit 0 đến bit 5 của thanh ghi điều khiển FPU) có 6 cờ ngoại lệ trong thanh ghi điều khiển FPU giống với 6 cờ ngoại lệ trong thanh ghi trạng thái FPU (bit 0 đến bit 5). Khi một trong các cờ này được bật, tương ứng ngoại lệ của giá trị dấu chấm động sẽ bị chặn khi giá trị được tạo. .



Hình 3.12: Thanh ghi điều khiển FPU

Cờ điều khiển độ chính xác

Cờ điều khiển độ chính xác (PC từ bit 8 đến bit 9 của thanh ghi điều khiển FPU) xác định độ chính xác (64, 53 hay 24-bits) của phép tính dấu chấm động được thực hiện bởi FPU (bảng 3.12 ). Mặc định độ chính xác là chính xác mở rộng, sử dụng 64 bits phần định trị làm định dạng cho các thanh ghi giá trị của FPU nhưng cũng có thể thiết lập lại bởi người lập trình, trình biên dịch hoặc hệ điều hành. Thiết lập này phù hợp cho hầu hết các ứng dụng bởi vì cho phép các ứng dụng sự thuận tiện và độ chính xác của định dạng chính xác mở rộng.

Độ chính xác	Cờ PC
Chính xác đơn (24-bits)*	00B
Dành riêng	01B
Chính xác kép (52-bits)*	10B
Chính xác mở rộng (64-bits)*	11B

Bảng 3.12: Độ chính xác trong thanh ghi điều khiển FPU

Lưu ý: \* Các bit đã bao gồm phần bit số nguyên.

Độ chính xác kép và chính xác đơn được thiết lập sẽ giảm kích thước bit của phần định trị xuống thành 53-bits và 24-bits. Những thiết lập này được cung cấp để hỗ trợ chuẩn IEEE và cho phép tạo lại chính xác kết quả của sự tính toán được thực hiện ở kiểu dữ liệu theo định dạng chính xác thấp hơn. Việc sử dụng độ chính xác thấp hơn, giá trị

**Cờ điều khiển làm tròn**

Cờ điều khiển làm tròn (RC) trong thanh ghi điều khiển FPU (bit 10 và 11) sẽ điều khiển cách kết quả của câu lệnh dấu phẩy động được làm tròn. Có 4 phương thức làm tròn được hỗ trợ (bảng 3.13): làm tròn đến số gần nhất, làm tròn lên, làm tròn xuống, làm tròn gần tới 0. Làm tròn đến số gần nhất là phương thức mặc định và phù hợp với hầu hết các ứng dụng, ước lượng được kết quả gần chính xác nhất.

Phương thức làm tròn	Cờ RC	Mô tả
Làm tròn đến số gần nhất	00B	Kết quả được làm tròn là gần nhất với kết quả chính xác.
Làm tròn lên (hướng về $+\infty$ )	01B	Kết quả được làm tròn tới số gần nhất, lớn hơn so với kết quả chính xác.
Làm tròn xuống (hướng về $-\infty$ )	10B	Kết quả được làm tròn tới số gần nhất, nhỏ hơn so với kết quả chính xác.
Làm tròn về 0	11B	Kết quả được làm tròn tới số gần nhất, giá trị tuyệt đối của kết quả làm tròn không lớn hơn so với kết quả chính xác

Bảng 3.13: Bảng giá trị phương thức làm tròn

Phương thức làm tròn lên và làm tròn xuống được giới hạn trực tiếp trong lúc làm tròn số và được sử dụng để xác định khoảng giá trị chính xác. Khoảng giá trị được sử dụng để xác định giới hạn trên và giới hạn dưới của kết quả khi thực hiện với nhiều phép tính, khi các kết quả trung gian của phép tính đã được làm tròn số.

Phương thức làm tròn tới 0 (hay được gọi là “chop”) thường được sử dụng khi thực hiện các phép tính số nguyên trong FPU.

Bất cứ khi nào thực hiện các câu lệnh toán học của FPU đưa ra một kết quả chính xác theo định dạng (chính xác đơn, chính xác kép, chính xác mở rộng). Tuy nhiên, thường thì kết quả chính xác của một phép toán hoặc sao lưu không thể mã hóa chính xác theo định dạng của toán hạng.

Lấy ví dụ giá trị  $a$  có 24 bit phân số. Bit thấp nhất của giá trị  $a$  (được gạch chân) không thể mã hóa chính xác theo định dạng chính xác đơn (lúc này chỉ có 23 bit phần phân số).

$$(a) 1.000100001000001110010111\text{E}_2101$$

Để làm tròn kết quả (a), FPU đầu tiên chọn ra hai chuỗi bit có phần phân số gần  $a$  nhất ( $b \leq a \leq c$ ).

$$(b) 1.00010000100000111001011\text{E}_2101$$

$$(c) 1.00010000100000111001100\text{E}_2101$$

Sau đó FPU sẽ thiết lập giá trị  $b$  và  $c$  theo phương thức làm tròn đã được lựa chọn trong cờ RC. Việc làm tròn sẽ tạo ra lỗi trong kết quả mà giá trị được làm tròn nhỏ hơn một đơn vị trong bit cuối của kết quả được làm tròn.

Kết quả được làm tròn được gọi là kết quả không chính xác. Khi kết quả không chính xác của câu lệnh FPU được tạo ra, cờ chính xác (PE) trong thanh ghi trạng thái FPU được bật.

Khi cờ xử lý ngoại lệ overflow được bật, kết quả chính xác nằm giữa giá trị cụ thể và dương vô cực, FPU sẽ làm tròn kết quả theo bảng 3.14.

Phương thức làm tròn	Kết quả
Làm tròn đến số gần nhất	$+\infty$

Làm tròn về 0	Giá trị lớn nhất gần nhất mà số thực dương có thể làm tròn.
Làm tròn lên (hướng về $+\infty$ )	$+\infty$
Làm tròn xuống (hướng về $-\infty$ )	Giá trị lớn nhất gần nhất mà số thực dương có thể làm tròn.

Bảng 3.14: Làm tròn số dương khi overflow xảy ra

Khi cờ xử lý ngoại lệ overflow được bật, kết quả chính xác nằm giữa giá trị cụ thể và âm vô cực, FPU sẽ làm tròn kết quả theo bảng 3.15

Phương thức làm tròn	Kết quả
Làm tròn đến số gần nhất	$-\infty$
Làm tròn về 0	Giá trị lớn nhất gần nhất mà số thực âm có thể làm tròn.
Làm tròn lên (hướng về $+\infty$ )	Giá trị lớn nhất gần nhất mà số thực âm có thể làm tròn.
Làm tròn xuống (hướng về $-\infty$ )	$-\infty$

Bảng 3.15: Làm tròn số dương khi Underflow xảy ra

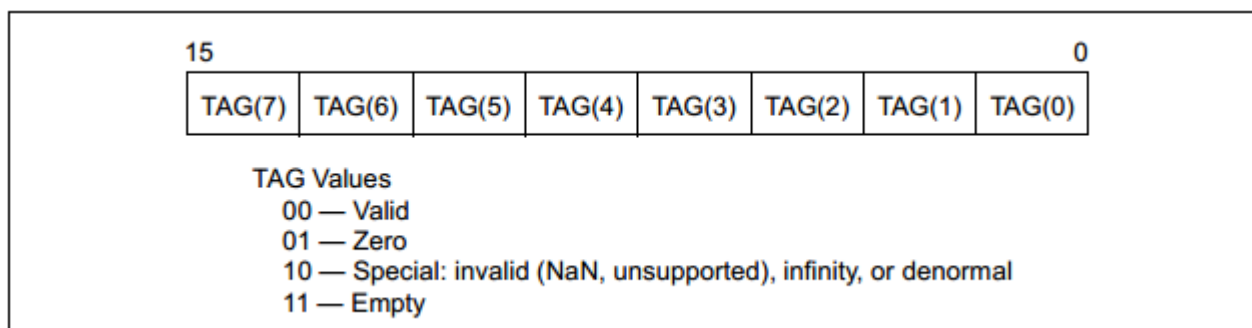
Phương thức làm tròn không ảnh hưởng đến câu lệnh so sánh, câu lệnh sử dụng kết quả chính xác, hoặc câu lệnh có kết quả là NaN.

## Cờ điều khiển vô cùng

Cờ điều khiển vô cùng (bit 10 của thanh ghi điều khiển FPU) được cung cấp cho bộ xử lý toán học Intel 281, không có ý nghĩa với bộ xử lý FPU của Pentium Pro hoặc bộ xử lý FPU của Intel 486, hoặc bộ xử lý Intel 387 NPX.

## Thẻ FPU

Thanh ghi thẻ 16-bit ( hình 3.13 ) chỉ ra nội dung của 8 thanh ghi dữ liệu FPU (2-bit một biểu diễn cho một thanh ghi). Mỗi thẻ cho biết một thanh ghi chứa một số hợp lệ, bằng không, hoặc một số dấu chấm động đặc biệt (NaN, số vô cùng, denormal, hoặc không hỗ trợ định dạng), hoặc rỗng. Các thẻ FPU được lưu trữ trong FPU trong thanh ghi thẻ FPU. Khi FPU được khởi tạo bằng câu lệnh FINIT/FNINIT hoặc FSAV/FNSAVE, thẻ FPU của mỗi thanh ghi được thiết lập là FFFFh, thể hiện tất cả dữ liệu hiện có trong thanh ghi là rỗng.



Hình 3.13: Thanh ghi điều khiển FPU

Mỗi thẻ trong thanh ghi thẻ FPU tương ứng với một thanh ghi vật lý (từ 0 đến 7). Đỉnh stack hiện tại (TOP) được lưu trong thanh ghi trạng thái FPU được sử dụng để kết hợp với thanh ghi thẻ liên quan đến ST(0).

FPU sử dụng giá trị của thẻ để phát hiện điều kiện tràn trên và tràn dưới của stack. Stack bị tràn trên khi con trỏ TOP giảm đi (do tải dữ liệu vào một thanh ghi hoặc một phép toán được thêm vào) trở đến một thanh ghi có giá trị không rỗng. Stack bị tràn dưới khi con trỏ TOP tăng lên (do sao lưu dữ liệu hoặc phép toán lấy dữ liệu ra) trở tới một thanh ghi rỗng hoặc thanh ghi rỗng được tham chiếu là một toán hạng. Thanh ghi không rỗng được xác định trong thanh ghi thẻ có chứa nội dung biểu diễn giá trị số 0 (01b), giá trị hợp lệ là (00b), hoặc một giá trị đặc biệt (10b).

Các chương trình ứng dụng hoặc xử lý ngoại lệ có thể sử dụng thông tin thẻ để kiểm tra nội dung của thanh ghi dữ liệu FPU mà không cần giải mã dữ liệu của mỗi thanh



ghi dữ liệu. Để đọc thanh ghi thẻ, thanh ghi thẻ phải được lưu trong bộ nhớ sử dụng câu lệnh FSTENV/FNSTENV hoặc FSAVE/FNSAVE. Địa chỉ của thanh ghi thẻ trong bộ nhớ sau khi lưu khi thực hiện một câu lệnh được thể hiện trong hình () đến hình ().

Phần mềm không thể trực tiếp tải hoặc sửa đổi giá trị trong thanh ghi thẻ. Câu lệnh FLDENV và FRSTOR tải một hình ảnh của thanh ghi thẻ vào trong FPU. Tuy nhiên, FPU sử dụng giá trị thanh ghi thẻ để xác định xem các thanh ghi dữ liệu là rỗng (11b) hoặc không rỗng (00b, 01b hoặc 10b). Nếu hình ảnh thanh ghi thẻ thể hiện các dữ liệu trong thanh ghi là không rỗng, FPU đọc các giá trị trong thanh ghi dữ liệu và thiết lập thanh ghi thẻ. Hành động này ngăn chặn một chương trình thiết lập các giá trị trong thanh ghi thẻ biểu diễn không chính xác nội dung hiện tại của thanh ghi dữ liệu.

### Con trỏ câu lệnh FPU và con trỏ toán hạng (dữ liệu)

Con trỏ lưu trữ trong lệnh FPU và toán hạng (dữ liệu) cho câu lệnh không điều khiển được thực thi trong thanh ghi 48-bit: con trỏ lệnh FPU và con trỏ trỏ tới thanh ghi toán hạng FPU (dữ liệu). (Thông tin được lưu để cung cấp trạng thái thông tin cho xử lý ngoại lệ). Con trỏ lưu trữ trong lệnh FPU và thanh ghi con trỏ toán hạng bao gồm một offset (lưu trữ trong bit 0 đến bit 31) và một bộ lọc câu lệnh (lưu trong bits 32 đến 41).

Nội dung của câu lệnh FPU và thanh ghi con trỏ toán hạng được giữ nguyên khi có bất kỳ câu lệnh nào của câu lệnh điều khiển (FINIT/FNINIT, FCLEX/FNCLEX, FLDCW, FSTCW/FNSTCW, FSTSW/FNSTSW, FSTENV/FNSTENV, FLDENV, FSAVE/FNSAVE, FRSTOR, và WAIT/FWAIT) được thực thi. Nội dung của thanh ghi toán hạng FPU không xác định được nếu như câu lệnh điều khiển không có bộ nhớ toán hạng.

Những thanh ghi này có thể truy cập bởi các câu lệnh FSTENV/FNSTENV, FLDENV, FINIT/FNINIT, FSAVE/FNSAVE và FRSTOR. Lệnh FINIT/FNINIT và FSAVE/FNSAVE sẽ xóa nội dung của những thanh ghi này.

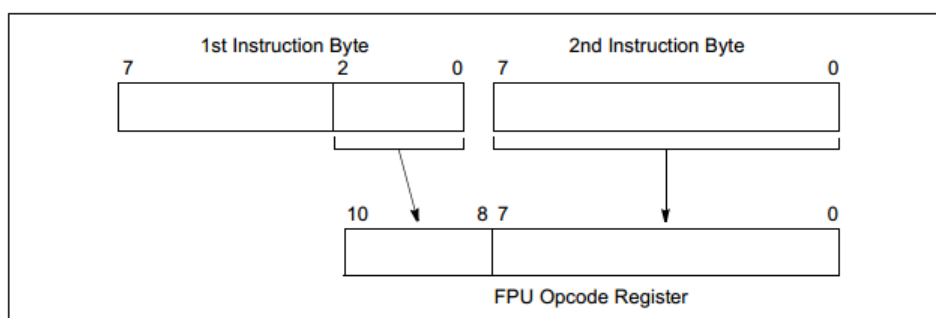
Tất cả kiến trúc Intel FPU và NPX trừ bộ xử lý 8087, con trỏ lệnh FPU trở tới bất kỳ tiền tố nào mà nó đứng trước câu lệnh. Đối với bộ xử lý 8087, con trỏ lệnh FPU chỉ trở tới opcode hiện tại.

## Thanh ghi opcode

FPU lưu trữ các opcode của câu lệnh không điều khiển trước khi thực hiện trong thanh ghi opcode 11-bits. (Thông tin này cung cấp trạng thái thông tin cho các xử lý ngoại lệ). Chỉ bytes thứ 1 và 2 của opcode (trước tất cả tiền tố) được lưu trữ trong thanh ghi opcode FPU. Hình () chỉ ra việc mã hóa những byte này.

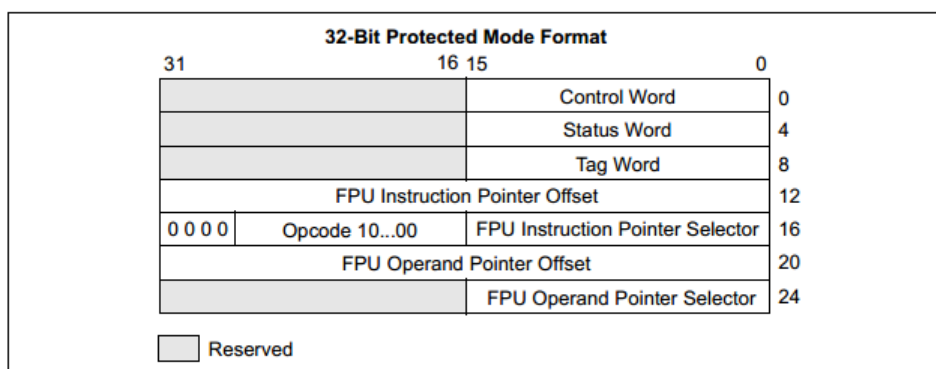
## Lưu trạng thái FPU

Các câu lệnh FSTENV/FNSTENV và FSAVE/FNSAVE lưu các thông tin FPU trong bộ nhớ để sử dụng xử lý ngoại lệ và sử dụng hệ thống khác hay ứng dụng. Câu lệnh FSTENV/FNSTENV lưu nội dung các thanh ghi trạng thái, thanh ghi điều khiển, thanh ghi thẻ, con trỏ lệnh FPU, con trỏ toán hạng FPU và thanh ghi opcode. Câu lệnh FSAVE/FNSAVE lưu thông tin và nội dung của thanh ghi dữ liệu. Lưu ý rằng câu lệnh FSAVE/FNSAVE khởi tạo FPU với các giá trị mặc định (giống như câu lệnh FINIT/FNINIT) sau khi hoàn thành lưu trữ trạng thái của FPU.

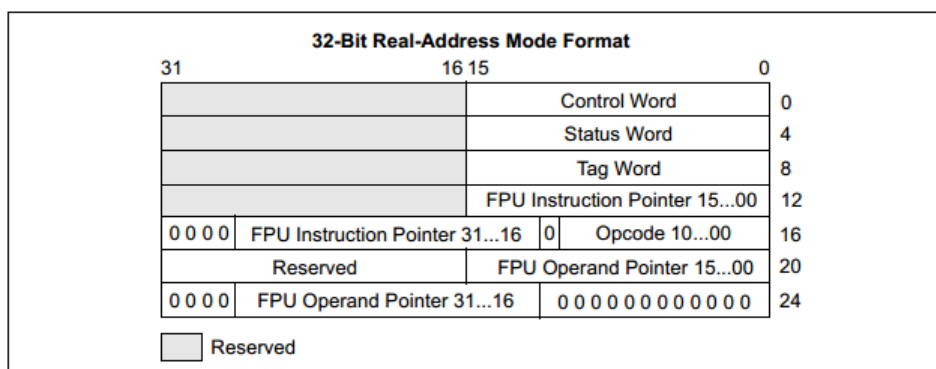


Hình 3.14: Nội dung thanh ghi opcode FPU

Cách mà thông tin được lưu trữ trong bộ nhớ phụ thuộc vào chế độ hoạt động của bộ xử lý (chế độ bảo vệ hoặc chế độ địa chỉ thực) và kích thước giá trị của toán hạng (32-bit hay 16-bit). Từ hình () đến hình (). Trong chế độ virtual-8086 hoặc SMM, địa chỉ thực thể hiện trong hình ().



Hình 3.15: Chế độ bảo vệ (protected) FPU trong bộ nhớ, định dạng 32-bit

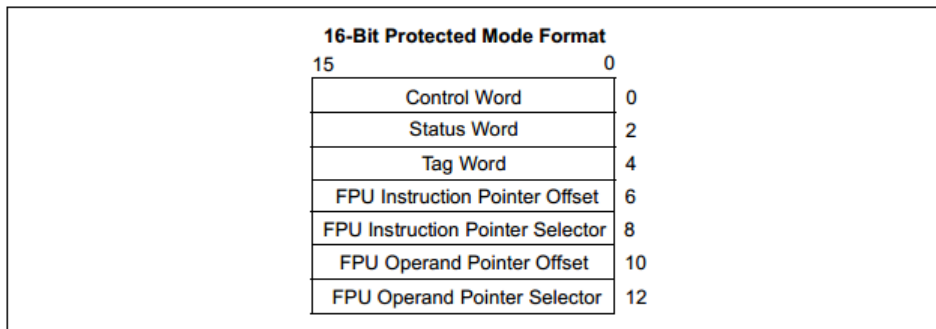


Hình 3.16: Chế độ địa chỉ thực FPU trong bộ nhớ , định dạng 32-bit

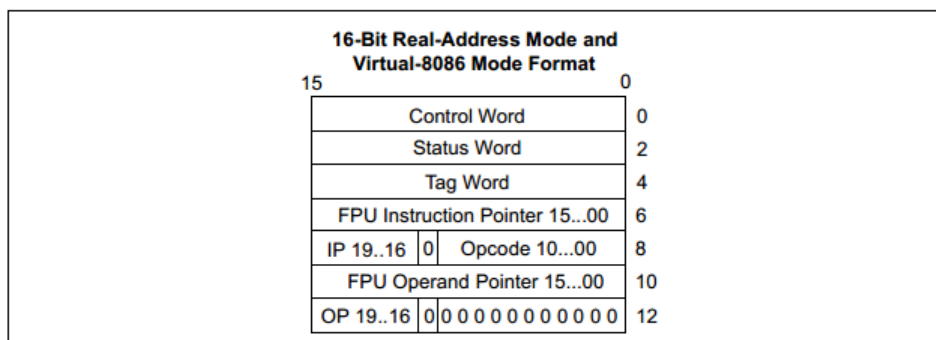
Các câu lệnh FLDENV và FRSTOR cho phép thông tin trạng thái FPU được tải từ bộ nhớ vào FPU. Câu lệnh FLDENV chỉ tải trạng thái, điều khiển, thẻ con trỏ lệnh FPU, con trỏ toán hạng và thanh ghi opcode. Câu lệnh FRSTOR tải tất cả thanh ghi FPU bao gồm cả thanh ghi dữ liệu.

### 3.2.2.3 Loại dữ liệu dấu chấm động và định dạng

Kiến trúc Intel FPU thao tác trên bảy loại dữ liệu, chia thành ba nhóm: số thực, số nguyên, và số nguyên gói BCD. Hình () cho thấy các định dạng dữ liệu cho mỗi loại dữ liệu FPU. Bảng () cho chiều dài, độ chính xác, và phạm vi bình thường gần đúng có thể được đại diện của từng loại dữ liệu FPU. Giá trị Denormal cũng được hỗ trợ trong mỗi loại thực sự, theo chuẩn của IEEE 854.



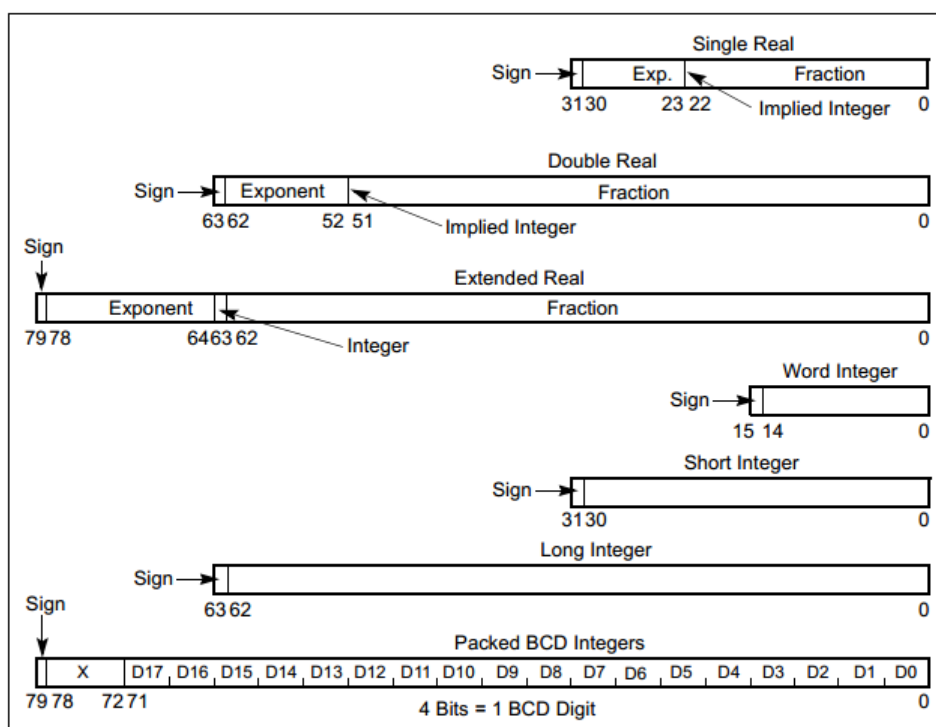
Hình 3.17: Chế độ bảo vệ (protected) FPU trong bộ nhớ, định dạng 16-bit



Hình 3.18: Chế độ địa chỉ thực FPU trong bộ nhớ, định dạng 16-bit

Với định dạng mở rộng 80-bit biểu diễn ngoại lệ, tất cả các loại dữ liệu chỉ tồn tại trong bộ nhớ. Khi chúng được nạp vào thanh ghi FPU dữ liệu, chúng được chuyển đổi sang định dạng mở rộng-thực và thao tác trên trong định dạng đó.

Khi lưu trữ trong bộ nhớ, byte thấp của một dữ liệu kiểu giá trị FPU được lưu trữ trong địa chỉ đầu tiên chỉ định giá trị đó. Các byte tiếp theo sau đó được lưu trữ trong các địa chỉ cao hơn trong bộ nhớ. Câu lệnh của dấu chấm động nạp và lưu bộ nhớ toán hạng chỉ sử dụng địa chỉ đầu của toán hạng.



Hình 3.19: Định dạng loại dữ liệu dấu chấm động

## Số thực

Ba loại dữ liệu số thực của FPU (single-real, double-real, và extended-real) tương ứng với chính xác đơn, đôi chính xác, và định chính xác mở rộng trong các định dạng chuẩn IEEE. Định dạng chính xác mở rộng là định dạng được sử dụng bởi các thanh ghi dữ liệu trong FPU. Bảng () đưa ra độ chính xác và phạm vi của các kiểu dữ liệu và hình () đưa ra các định dạng. Đối với các định dạng single-real và double-real, chỉ có một phần phân số của phần định trị được mã hóa. Số nguyên được giả định là 1 cho tất cả các số trừ 0 và số denormalized hữu hạn. Đối với các định dạng extended-real, các số nguyên được chứa trong bit 63. Ở đây, các số nguyên được thiết lập thành 1 cho số normalized, infinities, và NaNs, và 0 cho số không và số denormalized.

Kiểu dữ liệu	Độ dài	Chính xác (Bits)	Phạm vi giá trị	
			Hệ nhị phân	Cơ số 10
Binary Real				
Single real	32	24	$2^{-126}$ tới $2^{127}$	$1.18 \times 10^{-38}$ tới $3.40 \times 10^{38}$
Double real	64	53	$2^{-1022}$ tới $2^{1023}$	$2.23 \times 10^{-308}$ tới $1.79 \times 10^{308}$
Extended real	80	64	$2^{-16382}$ tới $2^{16383}$	$3.37 \times 10^{-4932}$ tới $1.18 \times 10^{4932}$
Binary Integer				
Word integer	16	15	$-2^{15}$ tới $2^{15} - 1$	$-32,768$ tới $32,767$
Short integer	32	31	$-2^{31}$ tới $2^{31} - 1$	$-2.14 \times 10^9$ tới $2.14 \times 10^9$
Long integer	64	63	$-2^{63}$ tới $2^{63} - 1$	$-9.22 \times 10^{18}$ tới $9.22 \times 10^{18}$

Bảng 3.16: Làm tròn số dương khi Underflow xảy ra

Giải thích Binary Real Single real Double real Extended real

Số mũ của từng loại dữ liệu thực được mã hóa ở định dạng cơ bản. Hằng số cơ bản là 127 cho các định dạng single-real, 1023 cho các định dạng doubler-real, và 16.383 cho các định dạng extended-real.

Bảng () cho thấy các mã hóa cho tất cả các lớp của các số thực (zero, denormalized-finite, normalized-finite, và vô cùng  $\infty$ ) và NaNs cho ba loại dữ liệu số thực. Bảng () cũng cung cấp cho các định dạng cho các số thực không xác định.

Khi lưu trữ các giá trị thực sự trong bộ nhớ, giá trị single-real được lưu trữ trong 4 byte liên tiếp trong bộ nhớ, giá trị double-real được lưu trữ trong 8 byte liên tiếp, và các giá trị extended-real được lưu trữ trong 10 byte liên tiếp.

Quy luật chung, giá trị nên được lưu trữ trong bộ nhớ ở định dạng double-real. Định dạng này cung cấp đầy đủ kích thước và độ chính xác để đưa ra kết quả chính xác tối thiểu mà lập trình viên quan tâm. Các định dạng single-real thích hợp cho các ứng dụng bị hạn chế bởi bộ nhớ. Tuy nhiên, single-real cung cấp độ chính xác ít hơn và có thể bị

tràn số. Các định dạng single-real hữu ích khi gỡ lỗi các thuật toán, vì làm tròn sẽ thực hiện một cách nhanh chóng ở định dạng này. Các định dạng extended-real thường dành riêng để giữ kết quả trung gian trong các thanh ghi FPU và hằng số. Chiều dài chính xác được thiết kế để đưa ra kết quả cuối cùng từ những phép toán bị ảnh hưởng của làm tròn số và tràn trên hoặc tràn dưới trong các phép tính trung gian. Tuy nhiên, khi một ứng dụng yêu cầu giới hạn chính xác và độ chính xác của các FPU (để lưu trữ dữ liệu, phép toán, và kết quả), các giá trị có thể được lưu trữ trong bộ nhớ ở định dạng mở rộng thực.

Các giá trị số thực không xác định là một QNaN được mã hóa và lưu giữ bởi câu lệnh dấu chấm động để đáp ứng với một dấu chấm động không hợp lệ tác ngoại lệ được đánh dấu (xem bảng () ).

Loại		Bit dấu	Bit số mũ	Phần định trị	
				Bit nguyên	Phần thập phân
Số dương	$+\infty$	0	11..11	1	00..00
	+Normals	0	11..10	1	11..11
		.	.	.	.
		.	.	.	.
	0	00..01	1	00..00	
	+Denormals	0	00..00	0	11..11
.		.	.	.	
.		.	.	.	
0		00..00	0	00..01	
+0	0	00..00	0	00..00	
Số âm	-0	1	00..00	0	00..00
	-Denormals	1	00..00	0	00..01
		.	.	.	.
		.	.	.	.
	1	00..00	0	11..11	
	-Normals	1	00..01	1	00..00
.		.	.	.	
.		.	.	.	
1		11..10	1	11..11	
$-\infty$	1	11..11	1	00..00	
NaN	SNaN	X	11..11	1	0X.. $XX_2$
	QNaN	X	11..11	1	1X..XX
	Không định nghĩa	1	11..11	1	10..00
Single-real			← 8 bits →		← 23 bits →
Double-real			← 11 bits →		← 53 bits →
Extended-real			← 15 bits →		← 63 bits →

Bảng 3.17: Mã hóa số thực và NaN



LƯU Ý:

- 1 Bit nguyên được ngụ ý và không được lưu trữ cho các định dạng single-real và double-real.
- 2 Phần phân số của SNaN mã hóa phải khác bit 0.

## Số nguyên được mã hóa nhị phân

Có ba loại số nguyên (word, short và long) có định dạng giống hệt nhau, ngoại trừ chiều dài. Bảng () cho độ chính xác và giới hạn của các kiểu dữ liệu và hình () đưa ra các định dạng. Bảng () đưa ra bảng mã của ba loại nguyên dạng nhị phân.

Lớp dữ liệu		Bit dấu	Độ lớn
Số dương	Lớn nhất	0	11..11
	.	.	.
	.	.	.
	Nhỏ nhất	0	00..01
Số 0		0	00..00
Số âm	Lớn nhất	1	11..11
	.	.	.
	.	.	.
	Nhỏ nhất	1	00..01
Số không định nghĩa		1	00..00
		Word Integer	← 15 bits →
		Short Integer	← 31 bits →
		Long Integer	← 63 bits →

Bảng 3.18: Mã hóa nhị phân số nguyên

Hầu hết các bit phần định trị của mỗi định dạng là bit dấu (0 là số dương và 1 số âm). Giá trị âm được biểu diễn trong chuẩn bù hai. Số 0 được biểu diễn với tất cả các

bit (kể cả bit dấu) thiết lập là bit 0. Lưu ý rằng kiểu dữ liệu word-integer của FPU là giống với kiểu dữ liệu word-integer được sử dụng trong bộ xử lý số nguyên và các định dạng short-integer là giống kiểu dữ liệu doubleword-integer.

Giá trị word-integer được lưu trữ trong 2 byte liên tiếp trong bộ nhớ; giá trị short-integer được lưu trữ trong 4 byte liên tiếp; và các giá trị long-integer được lưu trữ trong 8 byte liên tiếp. Khi được nạp vào thanh ghi dữ liệu của FPU, tất cả các số nguyên nhị phân là được biểu diễn theo định dạng mở rộng của số thực.

Việc mã hóa số nguyên nhị phân 100..00B biểu diễn cho một trong hai ý nghĩa, tùy thuộc vào hoàn cảnh của việc sử dụng nó:

- Các số âm lớn nhất được hỗ trợ bởi các định dạng ( $-2^{15}$ ,  $-2^{31}$ , hoặc  $-2^{63}$ ).
- Các nguyên giá trị không xác định.

Nếu mã hóa này được sử dụng như một toán hạng nguồn (nạp một số nguyên hoặc câu lệnh toán học sử dụng số nguyên), các FPU hiểu nó như là số âm lớn nhất có thể biểu diễn được trong các định dạng được sử dụng. Nếu các FPU kiểm tra một thao tác không hợp lệ khi lưu trữ một giá trị số nguyên trong bộ nhớ với câu lệnh FIST / FISTP và thao tác ngoại lệ được đánh dấu, các FPU lưu trữ số nguyên không xác định đã được mã hóa trong toán hạng đích như một đánh dấu để xử lý ngoại lệ. Trong tình huống mà giá trị gốc với kiểu mã hóa này có thể bị nhập nhầm, cờ toán hạng không hợp lệ có thể được kiểm tra để xem nếu giá trị kết quả đã được đánh dấu xử lý ngoại lệ.

Nếu số nguyên không xác định được lưu trữ trong bộ nhớ và sau đó được nạp lại vào một thanh ghi dữ liệu FPU, nó được hiểu là số âm lớn nhất theo định dạng được hỗ trợ.

## Số nguyên cơ số 10

Số nguyên cơ số 10 được lưu trữ 10-byte theo định dạng gói BCD. Bảng () cho biết độ chính xác và phạm vi của kiểu dữ liệu này và hình() hiển thị định dạng. Trong định dạng này, 9 byte đầu tiên giữ 18 số BCD, 2 số cho mỗi byte. Các chữ số phần định trị được chứa trong nửa-byte thấp từ byte 0 và chữ số nhất có ý nghĩa được chứa ở nửa byte

cao từ byte 9. Phần định trị của byte10 chứa bit dấu (0 số dương và 1 số âm). (Bits 0 đến 6 trong byte 10 không quan tâm bit.) Giá trị âm của số nguyên không được lưu trữ ở dạng bù hai; chúng được phân biệt với các số nguyên thập phân dương bởi các bit dấu.

Bảng () biểu diễn mã hóa giá trị có thể có trong kiểu dữ liệu số nguyên thập phân.

Các định dạng số nguyên thập phân chỉ tồn tại trong bộ nhớ. Khi một số nguyên thập phân được nạp trong một thanh ghi dữ liệu trong FPU, nó sẽ tự động chuyển đổi sang định dạng số thực mở rộng. Tất cả các số nguyên thập phân biểu diễn ở định dạng số thực mở rộng.

Việc đóng gói mã hóa số không xác định số được lưu trữ theo câu lệnh FBSTP một thao tác ngoại lệ sẽ được đánh dấu. Cố gắng nạp giá trị này với câu lệnh FBLD tạo ra một kết quả không xác định.

### Mã hóa theo định dạng Extended-real số không được hỗ trợ

Các định dạng extended-real cho phép mã hóa mà không giống với bất kỳ các cách thể hiện trong Bảng (). Bảng () cho thấy các mã hóa không được hỗ trợ. Một số các mã hóa được hỗ trợ bởi bộ xử lý toán học Intel 287. Tuy nhiên, hầu hết trong các bộ xử lý không được hỗ trợ bởi các bộ xử lý toán học Intel 387, hoặc các FPU trong bộ vi xử lý Intel486, Pentium, hay bộ xử lý Pentium Pro. Những mã hóa không còn được hỗ trợ do những thay đổi trong phiên bản cuối cùng của chuẩn IEEE 754 mà loại bỏ các bảng mã.

Các loại mã hóa trước đây gọi là pseudo-Nans, pseudo-infinities, và số un-normal không được hỗ trợ. Bộ xử lý toán học Intel 387 và FPU trong bộ vi xử lý Intel486, Pentium, và bộ xử lý Pentium Pro tạo ra các ngoại lệ và được đánh dấu khi đang gặp phải giống như xử lý toán hạng không hợp lệ.

Lớp dữ liệu	Bit dấu		Độ lớn					
			Số	Số	Số	Số	....	Số
Số dương lớn	0	0000000	1001	1001	1001	1001	...	1001
	.	.	.	.	.	.		
	.	.	.	.	.	.		
Số nhỏ	0	0000000	0000	0000	0000	0000	...	0001
Số +0	0	0000000	0000	0000	0000	0000	...	0000
Số -0	1	0000000	0000	0000	0000	0000	...	0000
Số nhỏ	1	0000000	0000	0000	0000	0000	...	0001
	.	.	.	.	.	.		
	.	.	.	.	.	.		
Số âm lớn	1	0000000	1001	1001	1001	1001	...	1001
Số không định nghĩa	1	1111111	1111	1111	UUUU*	UUUU	...	UUUU
	← 1 byte →		← 9 byte →					

Bảng 3.19: Mã hóa số nguyên đã được nén (BCD)

LƯU Ý: \* UUUU có nghĩa là giá trị bit không xác định và cũng có thể chứa giá trị bất kỳ.

### 3.2.2.4 Câu lệnh FPU

Các câu lệnh dấu chấm động được hỗ trợ trong kiến trúc Intel FPU có thể được nhóm lại thành sáu loại:

- Câu lệnh sao lưu dữ liệu
- Câu lệnh số học cơ bản
- Câu lệnh so sánh
- Câu lệnh sử dụng số siêu việt
- Câu lệnh nạp dữ liệu.

- Câu lệnh điều khiển FPU.

Số siêu việt là là số (thực hoặc phức) nhưng lại không là nghiệm của phương trình đại số nào. Ví dụ: số  $\pi$  và  $e$ .

Phần sau đây mô tả ngắn gọn các câu lệnh trong mỗi loại.

Lớp dữ liệu		Bit dấu	Bit mũ	Phần định trị	
				Bit nguyên	Phần thập phân
Số thực dương	Số vô cực	0	11..11	0	00..00
		1	11..11	1	00..00
	Số không bình thường	0	11..10	0	11..11
		1	11..10	1	11..11
	Số denormalized	0	00..01	0	00..00
		1	00..01	1	00..00
Số thực âm	Số denormalized	0	00..00	1	11..11
		1	00..00	1	11..11
	Số không bình thường	0	11..10	0	11..01
		1	11..10	1	11..01
	Số vô cực	0	11..11	0	00..00
		1	11..11	1	00..00
NaN	Signaling	0	11..11	0	01..11
		1	11..11	1	01..11
		0	11..11	0	00..01
	Quiet	0	11..11	0	11..11
		1	11..11	1	11..11
		0	11..11	0	10..00
			←15 bits→		← 63 bits →

Bảng 3.20: Mã hóa số không được hỗ trợ theo định dạng extended-real

## Câu lệnh thoát (ESC)

Tất cả các câu lệnh trong câu lệnh của FPU được thiết lập vào một nhóm các lệnh gọi là câu lệnh thoát (ESC). Tất cả các câu lệnh có một định dạng mã thông thường, nhưng một chút khác nhau từ các định dạng được sử dụng bởi các câu lệnh số nguyên và hệ điều hành.

## Câu lệnh FPU sử dụng toán hạng

Hầu hết các câu lệnh xử lý dấu chấm động cần một hoặc hai toán hạng, được lưu trữ trong các thanh ghi dữ liệu FPU hoặc trong bộ nhớ. (Không có câu lệnh xử lý dấu chấm động lấy toán hạng trực tiếp).

## Câu lệnh sao lưu dữ liệu

Các câu lệnh sao lưu dữ liệu (xem Bảng 3.21) các bước thực hiện sau đây:

- Nạp toán hạng là số thực, số nguyên, hoặc gói BCD từ bộ nhớ vào thanh ghi ST (0).
- Lưu trữ các giá trị trong thanh ghi ST (0) từ bộ nhớ theo định dạng số thực, số nguyên, hoặc gói BCD.
- Di chuyển các giá trị giữa các thanh ghi trong các thanh ghi dữ liệu FPU.

Số thực		Số thực		Gói số nguyên	
FLD	Nạp số thực	FILD	Nạp số nguyên	FBLD	Nạp gói số nguyên
FST	Lưu số thực	FIST	Lưu số nguyên		
FSTP	Lưu số thực và Pop	FISTP	Lưu số nguyên và Pop	FBSTP	Lưu gói số nguyên và Pop
FXCH	Hoán đổi				
FCMOVcc	Sao chép có điều kiện				

Bảng 3.21: Câu lệnh sao lưu dữ liệu

Toán hạng thường được lưu trong thanh ghi dữ liệu FPU ở định dạng extended-real (phụ thuộc vào cờ điều khiển chính xác (PC) ở thanh ghi điều khiển). Câu lệnh FLD ( nạp số thực) đẩy một toán hạng thực từ bộ nhớ vào đầu stack thanh ghi dữ liệu FPU. Nếu toán hạng là ở định dạng single-real hoặc double-real, FPU sẽ tự động chuyển đổi sang định dạng extended-real. Câu lệnh này cũng có thể được sử dụng để đẩy các giá trị của một thanh ghi dữ liệu FPU lên trên đỉnh của stack thanh ghi.

Câu lệnh FILD ( nạp số nguyên) chuyển đổi một toán hạng số nguyên trong bộ nhớ sang định dạng extended-real và đẩy giá trị vào đầu stack thanh ghi. Câu lệnh FBLD ( nạp gói BCD số nguyên) thực hiện các hoạt động nạp cùng với một toán hạng gói BCD trong bộ nhớ.

Câu lệnh FST (lưu số thực) và FIST (lưu số nguyên) lưu trữ các giá trị trong thanh ghi ST(0) vào trong bộ nhớ trong theo định dạng của bộ nhớ (số thực hoặc số nguyên, tương ứng). Một lần nữa, việc chuyển đổi định dạng được thực hiện tự động.

Câu lệnh FSTP (lưu số thực và pop), FISTP (lưu số nguyên và pop), và FBSTP (lưu gói BCD và pop) lưu trữ các giá trị trong thanh ghi ST (0) vào bộ nhớ trong theo định dạng của bộ nhớ (số thực, số nguyên, hoặc gói BCD), sau đó thực hiện pop (lấy dữ liệu ra) khỏi stack thanh ghi. Khi thực hiện pop, thanh ghi ST (0) được đánh dấu trống và con trỏ stack (TOP) trong thanh ghi trạng thái FPU được tăng lên 1. Câu lệnh FSTP cũng có thể được sử dụng để sao chép giá trị và lưu vào trong thanh ghi ST (0) từ thanh ghi FPU [ST (i)].

Câu lệnh FXCH (thay đổi nội dung thanh ghi) thay đổi giá trị trong thanh ghi được chọn trong stack [ST (i)] với giá trị trong ST (0).

Câu lệnh FCMOVcc (sao chép có điều kiện) sao chép các giá trị trong thanh ghi được chọn trong stack [ST (i)] vào thanh ghi ST (0). Những câu lệnh sao chép các giá trị chỉ khi các điều kiện quy định với một mã điều kiện (cc) được thỏa mãn (xem bảng 3.22). Các điều kiện được thử nghiệm với các câu lệnh FCMOVcc phụ thuộc vào các cờ trạng thái trong thanh ghi EFLAGS.

Câu lệnh	Cờ trạng thái	Mô tả điều kiện
FCMOVB	CF=1	Nhỏ hơn
FCMOVNB	CF=0	Lớn hơn hoặc bằng
FCMOVE	ZF=1	Bằng
FCMOVNE	ZF=0	Không bằng
FCMOVBE	(CF or ZF)=1	Nhỏ hơn hoặc bằng
FCMOVNBE	(CF or ZF)=0	Lớn hơn
FCMOVU	PF=1	Không thứ tự
FCMOVNU	PF=0	Có thứ tự

Bảng 3.22: Câu lệnh sao chép có điều kiện

Giống như các câu lệnh CMOVcc, các câu lệnh FCMOVcc là hữu ích cho việc tối ưu hóa cờ IF. Các câu lệnh này cũng giúp loại bỏ các trường hợp tự động ngắt hay dừng đột ngột do bộ xử lý gây ra.

Lưu ý: Các câu lệnh FCMOVcc có thể không được hỗ trợ trên một số họ bộ vi xử lý Pro Pentium. Phần mềm có thể kiểm tra xem các câu lệnh FCMOVcc có được hỗ trợ bằng cách kiểm tra thông tin của bộ xử lý tính năng với sự câu lệnh CPUID.

## Câu lệnh nạp hằng số

Các câu lệnh sau đây thường được sử dụng để nạp hằng số vào đỉnh stack [ST(0)] của thanh ghi dữ liệu FPU [ST (0)]:

FLDZ Load +0.0

FLD1 Load +1.0

FLDPI Load  $\pi$



FLDL2T Load  $\log_2 10$

FLDL2E Load  $\log_2 e$

FLDLG2 Load  $\log_{10} 2$

FLDLN2 Load  $\log_e 2$

Các giá trị không đổi biểu diễn theo định dạng extended-real độ chính xác (64 bit) và chính xác đến gần 19 chữ số thập phân. Chúng được lưu trữ theo một định dạng chính xác hơn định dạng extended-real. Khi nạp hằng số, FPU làm tròn số theo cờ làm tròn RC (điều khiển làm tròn) từ thanh ghi điều khiển FPU.

## Câu lệnh toán học cơ bản

Các câu lệnh xử lý dấu chấm động sau đây thực hiện các phép tính cơ bản trên trường số thực. Khi áp dụng, các câu lệnh phù hợp với tiêu chuẩn IEEE 754:

ADD/FADDP Add real

FIADD Add integer to real

FSUB/FSUBP Subtract real

FISUB Subtract integer from real

FSUBR/FSUBRP Reverse subtract real

FISUBR Reverse subtract real from integer

FMUL/FMULP Multiply real

FIMUL Multiply integer by real

FDIV/FDIVP Divide real

FIDIV Divide real by integer

FDIVR/FDIVRP Reverse divide

FIDIVR Reverse divide integer by real

FABS Absolute value

FCHS Change sign

FSQRT Square root

FPREM Partial remainder

FPREM1 IEEE partial remainder

FRNDINT Round to integral value

EXTRACT Extract exponent and significand

Các câu lệnh cộng, trừ, nhân và chia thực hiện trên các loại toán hạng sau đây:

- Hai giá trị thanh ghi dữ liệu FPU.
- Một giá trị thanh ghi dữ liệu và giá trị thực hay số nguyên trong bộ nhớ.

Phép toán trong bộ nhớ có thể định dạng là single-real, double-real, short-integer, hoặc word-integer. FPU sẽ tự động chuyển đổi sang định dạng extended-real.

Câu lệnh phép trừ và phép chia được cung cấp để nâng cao hiệu quả mã hóa. Ví dụ, câu lệnh FSUB trừ giá trị trong một thanh ghi dữ liệu FPU được chọn [ST (i)] từ giá trị trong thanh ghi ST (0); trong khi đó, câu lệnh FSUBR trừ đi giá trị ST(0) từ giá trị trong ST (i). Các kết quả của cả hai hoạt động này được lưu trữ trong thanh ghi ST(0). Những câu lệnh này loại bỏ giá trị không cần thiết giữa đăng ký ST (0) và một thanh ghi FPU khác để thực hiện phép trừ hoặc phép chia. Các câu lệnh pop của phép cộng, trừ, nhân và chia pop stack thanh ghi dữ liệu FPU sau khi thực hiện các phép toán số học.

Câu lệnh FPREM tính số dư của hai toán hạng theo cách thức được sử dụng bởi bộ xử lý toán học Intel 8087 và Intel 287, câu lệnh FPREM1 tính số dư theo cách thức đặc tả trong IEEE.

Câu lệnh FSQRT tính căn bậc hai của toán hạng nguồn.

Câu lệnh FRNDINT làm tròn một giá trị số thực thành giá trị số nguyên gần nhất của nó, theo phương thức làm tròn theo cách thức làm tròn được quy định trong cờ RC từ thanh ghi điều khiển FPU. Câu lệnh này thực hiện chức năng tương tự như các câu lệnh FIST / FISTP, ngoại trừ kết quả được lưu trong định dạng số thực.

Câu lệnh FABS, FCHS, và FXTRACT thực hiện các phép tính thuận tiện. Câu lệnh FABS tạo ra giá trị tuyệt đối của toán hạng nguồn. Câu lệnh FCHS thay đổi dấu của toán hạng nguồn. Câu lệnh FXTRACT tách các toán hạng nguồn thành số mũ và phần phân số của toán hạng sau đó lưu trữ mỗi giá trị trong thanh ghi ở định dạng số thực.

## Câu lệnh so sánh và phân loại

Các câu lệnh sau đây so sánh và phân loại giá trị số thực:

FCOM/FCOMP/FCOMPP Compare real and set FPU condition code flags.

FUCOM/FUCOMP/FUCOMPP Unordered compare real and set FPU condition code flags.

FICOM/FICOMP Compare integer and set FPU condition code flags.

FCOMI/FCOMIP Compare real and set EFLAGS status flags.

FUCOMI/FUCOMIP Unordered compare real and set EFLAGS status flags.

FTST Test (compare real with 0.0).

FXAM Examine.

So sánh các giá trị số thực sự khác với so sánh các số nguyên bởi vì giá trị thực có bốn (chứ không phải là ba) các mối quan hệ lẫn nhau: nhỏ hơn, bằng, lớn hơn, và không có thứ tự.

Các mối quan hệ không có thứ tự là đúng khi có ít nhất một trong hai giá trị được so sánh là NaN hoặc định dạng không xác định. Thêm mối quan hệ này là cần thiết bởi vì, theo định nghĩa, NaNs không phải là số, vì vậy NaNs không thể nhỏ hơn, bằng hoặc lớn hơn trong các mối quan hệ với giá trị số thực khác.

Câu lệnh FCOM, FCOMP, và FCOMPP so sánh giá trị trong thanh ghi ST(0) với một toán hạng nguồn số thực và thiết lập các cờ điều kiện (C0, C2, và C3) trong thanh trạng thái FPU theo kết quả (xem Bảng 3.23). Nếu một điều kiện không có thứ tự được phát hiện (một hoặc cả hai giá trị là NaN hoặc định dạng không xác định), cờ ngoại lệ toán hạng không hợp lệ được bật.

Các câu lệnh pop stack thanh ghi dữ liệu FPU được pop một lần hoặc hai lần sau khi hoạt động so sánh thực hiện xong.

Câu lệnh FUCOM, FUCOMP, và FUCOMPP thực hiện giống như các câu lệnh FCOM, FCOMP, và FCOMPP. Sự khác biệt duy nhất với các câu lệnh FUCOM, FU-

COMP, và FUCOMPP là nếu một điều kiện không có thứ tự được phát hiện bởi một hoặc cả hai toán hạng là một QNaN, cờ ngoại lệ toán hạng không hợp lệ được bật.

Điều kiện	C3	C2	C0
$ST(0) > \text{Toán hạng nguồn}$	0	0	0
$ST(0) < \text{Toán hạng nguồn}$	0	0	1
$ST(0) = \text{Toán hạng nguồn}$	1	0	0
Không thứ tự	1	1	1

Bảng 3.23: Thiết lập cờ điều kiện FPU theo phép so sánh số thực

Câu lệnh FICOM và FICOMP cũng thực hiện giống như các hướng dẫn FCOM và FCOMP, ngoại trừ các toán hạng nguồn là một giá trị số nguyên trong bộ nhớ. Giá trị số nguyên được tự động chuyển đổi thành định dạng extended-real trước khi so sánh. Câu lệnh FICOMP thực hiện pop stack thanh ghi dữ liệu FPU sau khi thực hiện hoạt động so sánh xong.

Câu lệnh FTST thực hiện các hoạt động tương tự như câu lệnh FCOM, ngoại trừ các giá trị trong thanh ghi ST (0) luôn luôn được so sánh với giá trị 0.0.

Câu lệnh FCOMI và FCOMIP là câu lệnh mới trong bộ xử lý Intel Pro Pentium. Câu lệnh thực hiện phép so sánh tương tự như hướng dẫn FCOM và FCOMP, ngoại trừ việc câu lệnh thiết lập các cờ trạng thái (ZF, PF và CF) trong thanh ghi EFLAGS để chỉ ra kết quả của phép so sánh (xem Bảng 3.24) thay vì các cờ điều kiện FPU. Câu lệnh FCOMI và FCOMIP cho phép lệnh rẽ nhánh điều kiện (JCC) được thực hiện trực tiếp từ kết quả so sánh.

Phép so sánh	ZF	PF	CF
$ST(0) > ST(i)$	0	0	0
$ST(0) < ST(i)$	0	0	1
$ST(0) = ST(i)$	1	0	0
Không thứ tự	1	1	1

Bảng 3.24: Thiết lập thanh ghi trạng thái EFLAGS theo phép so sánh số thực

Câu lệnh FUCOMI và FUCOMIP thực hiện giống như các hướng dẫn FCOMI và FCOMIP, ngoại trừ việc họ không tạo ra một dấu chấm động không hợp lệ tác ngoại lệ nếu điều kiện không có thứ tự là kết quả của một hoặc cả hai toán hạng là một QNaN. Các hướng dẫn FCOMIP và FUCOMIP bật stack FPU đăng ký sau khi hoạt động so sánh.

Các hướng dẫn FXAM xác định việc phân loại các giá trị thực sự trong ST (0) đăng ký (có nghĩa là, cho dù giá trị là số không, một số denormal, một số hữu hạn bình thường,  $\infty$ , một NaN, hoặc một định dạng được hỗ trợ) hoặc đăng ký là trống rỗng. Nó đặt FPU mã điều kiện lá cờ để chỉ ra sự phân loại (xem "FXAM-Kiểm tra" trong Chương 3, Instruction Set tham khảo, trong tay các nhà phát triển của Intel Kiến trúc phần mềm, Volume 2). Nó cũng đặt cờ C1 để chỉ ra các dấu hiệu của giá trị.

### 3.2.3 Phân nhánh theo mã điều kiện FPU

Bộ xử lý không cung cấp bất kỳ câu lệnh nào để kiểm soát dòng chảy của chương trình mà các nhánh rẽ được thiết lập từ lá cờ mã điều kiện (C0, C2, và C3) trong thanh ghi trạng thái FPU. Để phân nhánh dựa vào các cờ trạng thái, thanh ghi trạng thái FPU trước tiên phải được sao lưu vào thanh ghi AX bộ xử lý số nguyên. Câu lệnh FSTSW AX (lưu trạng thái) được sử dụng cho mục đích này. Khi mã cờ đã được sao lưu trong thanh ghi AX, câu lệnh Test được sử dụng để kiểm soát rẽ nhánh có điều kiện như sau:

- 1 Kiểm tra kết quả không có thứ tự. Sử dụng câu lệnh TEST để so sánh nội dung của thanh ghi AX với hằng số 0400H (xem Bảng 3.25). Thao tác này sẽ xóa cờ ZF trong thanh ghi EFLAGS nếu mã cờ điều kiện đánh dấu kết quả là không có thứ tự; ngược lại cờ ZF sẽ được thiết lập. Câu lệnh JNZ sau đó được sử dụng để chuyển giao điều khiển (nếu cần thiết) để câu lệnh xử lý các toán hạng có thứ tự.
- 2 Câu lệnh kiểm tra so sánh kết quả. Sử dụng các hằng số được đưa ra trong Bảng 3.25 câu lệnh TEST để kiểm tra cho một số nhỏ hơn, bằng hoặc lớn hơn so với kết quả, sau đó sử dụng lệnh rẽ nhánh có điều kiện tương ứng để chuyển điều khiển chương trình tới đoạn mã thích hợp.

Điều kiện	Hằng số	Câu lệnh rẽ nhánh
$ST0 > \text{Toán hạng nguồn}$	4500H	JZ
$ST0 < \text{Toán hạng nguồn}$	0100H	JNZ
$ST0 = \text{Toán hạng nguồn}$	4000H	JNZ
Không thứ tự	0400H	JNZ

Bảng 3.25: Hằng số câu lệnh Test cho lệnh rẽ nhánh

Nếu một chương trình hay hàm đã được kiểm tra cho kết quả QNaN, thì bước tiếp theo không cần kiểm tra kết quả có thứ tự mỗi khi một so sánh được thực hiện.

Một số câu lệnh FPU không sử dụng phép so sánh mà cập nhật mã cờ điều kiện trong thanh ghi trạng thái FPU. Để đảm bảo rằng các từ trạng thái không bị thay đổi, lưu trữ giá trị mã cờ ngay khi thực hiện phép toán so sánh.

## Câu lệnh lượng giác

Các câu lệnh sau đây thực hiện bốn hàm lượng giác thường gặp:

FSIN Sine

FCOS Cosine

FSINCOS Sine and cosine

FPTAN Tangent

FPATAN Arctangent

Toán hạng của câu lệnh là một hoặc hai thanh ghi đầu của stack thanh ghi FPU và trả kết quả của mình vào stack. Các toán hạng nguồn phải định dạng góc là radian.

Câu lệnh FSINCOS trả kết quả sin và cosin từ giá trị toán hạng nguồn. Câu lệnh hoạt động nhanh hơn so với thực hiện hai câu lệnh FSIN và FCOS liên tiếp.

Câu lệnh FPATAN tính toán arctangent của  $ST(1)$  chia cho  $ST(0)$ . Câu lệnh rất hữu ích cho việc chuyển đổi tọa độ vuông góc với tọa độ cực.

## Hằng số pi

Khi đối số (toán hạng nguồn) của hàm số lượng giác là giới hạn, các đối số được tự động giảm để thích hợp với  $2\pi$  thông qua cơ chế giảm được sử dụng bởi các câu lệnh FPREM và FPREM1. Giá trị nội bộ của  $\pi$  mà kiến trúc Intel FPU sử dụng để giảm đối số và tính toán khác như sau:

$$\pi = 0.f * 2^2$$

Giá trị:

$$f = \text{C90FDAA2 2168C234 C}$$

(Giới hạn của phần định trị ở trên là 32 bit).

Giá trị  $\pi$  có 66-bit phần phân số, mà trong đó sẽ được thêm 2 bit vào trong phần định trị của một giá trị extended-real. (66 bits không phải là một số chia hết cho 4 để biểu diễn các chữ số thập lục phân, hai số 0 được thêm vào các giá trị để có thể được biểu diễn ở dạng thập lục phân. Như thế, con số ít nhất-đáng thập lục phân (C) là 1100B, khi hai bit cao nhất đại diện cho bit 67 và 68 của phần định trị.).

Giá trị  $\pi$  đã được lựa chọn để đảm bảo không mất đi phần định trị trong một toán hạng nguồn, cung cấp giá trị toán hạng trong giới hạn giá trị của câu từng câu lệnh.

Nếu kết quả của phép tính sử dụng giá trị  $\pi$  chính xác từ các câu lệnh FSIN, FCOS, FSINCOS, hoặc FPTAN, 66-bit phần phân số của  $\pi$  nên được sử dụng. Điều này đảm bảo rằng các kết quả phù hợp với các thuật toán. Sử dụng giá trị làm tròn của  $\pi$  có thể gây ra thiếu chính xác trong giá trị kết quả, mà nếu được sử dụng trong một số phép tính có thể dẫn đến kết quả là thiếu chính xác.

Một phương pháp phổ biến để biểu diễn 66-bit phần phân số của  $\pi$  là để tách các giá trị vào hai số (bit cao  $\pi$  và bit thấp  $\pi$ ) mà khi cộng lại với nhau đưa ra giá trị  $\pi$  được hiển thị giống trước đó tương ứng với 66-bit:

$$\pi = \text{bit cao } \pi + \text{bit thấp } \pi$$

Ví dụ, hai giá trị sau đây (được đưa ra trong ký hiệu khoa học với các phần trong thập lục phân và số mũ trong số thập phân) đại diện cho 33 bit cao và 33 bit thấp của phần phân số:

Bit cao  $\pi = 0.C90FDAA20 \times 2^2$

Bit thấ  $\pi = 0.42D184698 \times 2^{-31}$

Những giá trị này được mã hóa theo định dạng double-real chuẩn IEEE như sau:

Bit cao  $\pi = 400921FB\ 54400000$

Bit thấp  $\pi = 3DE0B461\ 1A600000$

(Lưu ý rằng trong các định dạng double-real IEEE, số mũ (1023) và các phần phân số đã được chuẩn hóa.)

Giá trị  $\pi$  cũng có thể được biểu diễn bằng định dạng extended-real.

Khi sử dụng giá trị  $\pi$  từ hai phần trong một thuật toán, việc tính toán được thực hiện trên từng phần, với các kết quả riêng. Khi tất cả các tính toán đầy đủ, hai kết quả được ghép với nhau để tạo thành kết quả cuối cùng.

Sai số của việc tính toán sử dụng số  $\pi$  có thể tránh được, bằng cách áp dụng các hàm lượng giác có đối số trong giới hạn, hoặc bằng cách thực hiện giảm đối số (xuống đến một mức độ ít hơn  $\frac{\pi}{4}$  ).

## Phép toán logarit, epsilon và mũ số

Các câu lệnh sau đây cung cấp phép toán khác nhau logarit, số mũ, và tỉ lệ:

FYL2X Compute  $\log(y \times \log_2 x)$

FYL2XP1 Compute  $\log_e(y \times \log_2(x + 1))$

F2XM1 Compute exponential  $(2^x - 1)$

FSCALE Scale

Câu lệnh FYL2X và FYL2XP1 thực hiện hai phép tính logarit khác nhau. Câu lệnh FYL2X tính  $\log$  của  $(y \times \log_2 x)$  Thao tác này cho phép tính toán sử dụng các phương trình sau đây:

$$\log_b x = (1/\log_2 b) \times \log_2 x$$



Câu lệnh FYL2XP1 tính toán epsilon log của  $(y \times \log_2(x + 1))$ . Câu lệnh này cung cấp độ chính xác tối ưu cho các giá trị của epsilon ( $\epsilon$ ) được gần 0.

Câu lệnh F2XM1 tính hàm mũ  $(2x - 1)$ . Câu lệnh này có giá trị nguồn trong phạm vi -1.0 đến +1.0.

Câu lệnh FSCALE mũ số toán hạng nguồn với số mũ là 2.

## Câu lệnh điều khiển FPU

Các câu lệnh sau đây kiểm soát trạng thái và phương thức hoạt động của FPU. Đồng thời cũng cho phép xem trạng thái của FPU được kiểm tra:

FINIT/FNINIT Initialize FPU

FLDCW Load FPU control word

FSTCW/FNSTCW Store FPU control word

FSTSW/FNSTSW Store FPU status word

FCLEX/FNCLEX Clear FPU exception flags

FLDENV Load FPU environment

FSTENV/FNSTENV Store FPU environment

FRSTOR Restore FPU state

FSAVE/FNSAVE Save FPU state

FINCSTP Increment FPU register stack pointer

FDECSTP giảm con trỏ stack trong stack thanh ghi FPU

FFREE làm rỗng các thanh ghi dữ liệu FPU

FNOP Không thao tác

WAIT / FWAIT kiểm tra và xử lý các trường hợp ngoại lệ khi được đánh dấu.

Câu lệnh FINIT / FNINIT khởi tạo các thanh ghi FPU giá trị mặc định.

Câu lệnh FLDCW nạp giá trị từ thanh ghi điều khiển FPU từ bộ nhớ. Câu lệnh FSTCW / FNSTCW và FSTSW / FNSTSW lưu giá trị của thanh ghi điều khiển, trạng thái FPU tương ứng vào trong bộ nhớ (hoặc câu lệnh FSTSW / FNSTSW lưu trong một thanh ghi trong bộ xử lý số nguyên).

Câu lệnh FSTENV / FNSTENV và FSAVE / FNSAVE lưu các giá trị môi trường FPU và trạng thái tương ứng vào trong bộ nhớ. Môi trường FPU bao gồm tất cả các thanh ghi điều khiển và trạng thái của FPU, thanh ghi dữ liệu FPU. (Câu lệnh FSAVE/FNSAVE cũng khởi tạo các FPU giá trị mặc định, như câu lệnh FINIT / FNINIT, sau lưu môi trường của FPU).

Câu lệnh FLDENV và FRSTOR nạp môi trường FPU và trạng thái tương ứng từ bộ nhớ vào các FPU. Những câu lệnh này thường được sử dụng khi chuyển đổi để thực hiện một thao tác riêng biệt hoặc phụ thuộc vào ngữ cảnh. Câu lệnh WAIT / FWAIT là câu lệnh đồng bộ hóa. Những câu lệnh kiểm tra tình trạng từ FPU đánh dấu ngoại lệ FPU. Nếu bất kỳ trường hợp ngoại lệ FPU đánh dấu được tìm thấy, chúng được xử lý trước khi thực hiện câu lệnh tiếp theo trong dòng lệnh. Câu lệnh WAIT / FWAIT được để bộ hóa các câu lệnh thực hiện giữa các FPU và bộ xử lý số nguyên.

### Câu lệnh không được hỗ trợ trong FPU

Trong Intel 8087 có câu lệnh FENI và FDISI và trong bộ xử lý toán học Intel 287 có câu lệnh FSETPM không thực hiện trong bộ xử lý toán học Intel 387, hoặc các bộ vi xử lý Intel486, Pentium, hoặc Pentium Pro. Nếu các opcodes được phát hiện trong các dòng lệnh, FPU thực hiện không thực thi cụ thể và có thanh ghi FPU nào bị ảnh hưởng.

#### 3.2.4 Xử lý ngoại lệ dấu chấm động

FPU phát hiện sáu loại ngoại lệ trong khi thực hiện những lệnh dấu phẩy động:

- Thao tác không hợp lệ (#I)
  - Stack bị tràn trên hoặc tràn dưới (#IS)
  - Thao tác số học không hợp lệ (#IA)
- Divide-by-zero (#Z)
- Toán hạng denormalized (#D)
- Tràn trên số học (#O)
- Tràn dưới số học (#U)
- Kết quả không chính xác (độ chính xác) (#P)

Các thuật ngữ của " # " biểu tượng theo một hoặc hai chữ cái (ví dụ, #IS) được sử dụng trong câu lệnh để chỉ ra ngoại lệ. Nó chỉ đơn thuần là một cú pháp viết tắt và không liên quan đến việc xử lý.

## Thao tác không hợp lệ

Thao tác xử lý dấu chấm động không hợp lệ ứng với hai loại ngoại lệ sau:

- Stack bị tràn trên hoặc tràn dưới (#IS)
- Thao tác số học không hợp lệ (#IA)

Cờ cho trường hợp ngoại lệ này (IE) là bit 0 của thanh ghi trạng thái FPU, và bit (IM) được đánh dấu là 0 từ thanh ghi điều khiển FPU. Cờ phát hiện lỗi (SF) của thanh ghi trạng thái FPU chỉ ra các loại hình hoạt động gây ra những ngoại lệ. Khi cờ SF được thiết lập là 1, một thao tác nào đó đã dẫn đến tràn trên hoặc tràn dưới; khi cờ SF được thiết lập là 0, một lệnh số học có toán hạng không hợp lệ. Lưu ý rằng FPU thiết lập cờ SF khi nó phát hiện một lỗi stack bị tràn trên hoặc tràn dưới, nhưng không rành mạch khi cờ SF phát hiện toán hạng không hợp lệ. Kết quả là, trạng thái của SF có thể là 1 sau một ngoại lệ thao tác số học không hợp lệ, nếu nó không được thiết lập lại từ lần cuối cùng của lỗi stack bị tràn trên hoặc tràn dưới xảy ra.

### Stack bị tràn trên hoặc tràn dưới (#IS)

Thanh ghi thể FPU theo dõi những nội dung của stack thanh ghi dữ liệu FPU đăng ký. Sau đó sử dụng thông tin này để phát hiện hai loại lỗi khác nhau stack:

- Stack bị tràn trên khi một câu lệnh cố gắng ghi một giá trị vào stack thanh ghi dữ liệu FPU khi hiện tại các thanh ghi đã có dữ liệu.
- Stack bị tràn dưới khi một lệnh cố gắng đọc một giá trị từ stack thanh ghi dữ liệu FPU mà các thanh ghi của FPU hiện tại không có dữ liệu.

Khi FPU phát hiện stack bị tràn trên hoặc tràn dưới, cờ IE (bit 0) và cờ SF (bit 6) trong thanh ghi trạng thái FPU được thiết lập là 1. Sau đó thiết lập mã điều kiện cờ C1 (bit 9) trong thanh ghi trạng thái FPU là 1 nếu stack bị tràn trên hoặc được thiết lập là

0 nếu stack bị tràn dưới.

Nếu thao tác không hợp lệ được đánh dấu, FPU sau đó trả về giá trị không xác định của số thực, số nguyên, hoặc BCD-integer tới toán hạng, tùy thuộc vào các câu lệnh được thực thi. Giá trị này sẽ ghi đè lên các ghi đích hoặc bộ nhớ mà câu lệnh được thực thi.

Nếu thao tác không hợp lệ không được đánh dấu, một phần mềm xử lý ngoại lệ được gọi để xử lý và top-of-stack pointer (TOP), toán hạng nguồn không thay đổi.

Stack bị tràn trên khi mà stack thanh ghi FPU đã chứa đủ tám giá trị và chen thêm giá trị tiếp theo đẩy vào stack. Stack bị tràn dưới ngược với stack bị tràn trên. Khi mà stack thanh ghi FPU không chứa giá trị nào và lấy thêm giá trị từ đỉnh stack trong khi đỉnh stack hiện tại không đang chứa giá trị.

### Thao tác số học không hợp lệ (#IA)

FPU có thể phát hiện một loạt các phép tính số học không hợp lệ mà có thể đã được mã hóa trong một chương trình. Các thao tác này thường chỉ ra lỗi lập trình, chẳng hạn như chia  $\infty$  cho  $\infty$ . Bảng 3.26 liệt kê các hoạt động số học không hợp lệ mà các FPU phát hiện. Các thao tác không hợp lệ quy định trong chuẩn IEEE 854.

Khi các FPU phát hiện một toán hạng số học không hợp lệ, cờ IE (bit 0) trong thanh ghi trạng thái FPU thiết lập là 1. Nếu thao tác không hợp lệ được đeo đánh dấu, các FPU sau đó trả về một giá trị không xác định được gán vào cho toán hạng đích hoặc gán mã điều kiện dấu chấm động, thể hiện trong Bảng 3.26. Nếu thao tác không hợp lệ không được đánh dấu, một phần mềm xử lý ngoại lệ sẽ được gọi và top-of-stack pointer (TOP), toán hạng nguồn vẫn không thay đổi.

Điều kiện	Kết quả trả về
Bất kỳ thao tác số học trên một toán hạng mà đó là một định dạng không được hỗ trợ.	Toán hạng đích được gán là giá trị không xác định.

Bất kỳ thao tác số học là SNaN.	Toán hạng đích được gán là QNaN.
Thao tác so sánh và kiểm tra: một hoặc cả hai toán hạng là NaNs.	Thiết lập các cờ điều kiện (C0, C2, và C3) trong thanh ghi trạng thái FPU là 111B (không so sánh).
Phép cộng: toán hạng là vô cùng ngược dấu. Phép trừ: toán hạng là vô cùng cùng dấu.	Toán hạng đích được gán là giá trị không xác định.
Phép nhân: $\infty$ nhân với 0; 0 nhân với $\infty$ .	Toán hạng đích được gán là giá trị không xác định.
Phép chia: $\infty$ chia cho $\infty$ ; 0 chia cho 0.	Toán hạng đích được gán là giá trị không xác định.
Phần dư của câu lệnh FPREM, FPREM1: số chia là 0 hoặc số bị chia là $\infty$ .	Toán hạng đích được gán là giá trị không xác định và cờ điều kiện C2 được thiết lập 0.
Câu lệnh FCOS, FPTAN, FSIN, FSIN-COS: toán nguồn là $\infty$ .	Toán hạng đích được gán là giá trị không xác định và cờ điều kiện C2 được thiết lập 0.
FSQRT: toán hạng là số âm (trừ FSQRT (-0) = -0); FYL2X: toán hạng là số âm (trừ FYL2X (-0) = $-\infty$ ); FYL2XP1: toán hạng nhỏ hơn -1.	Toán hạng đích được gán là giá trị không xác định.
FBSTP: thanh ghi rỗng, hoặc toán hạng là NaN, $\infty$ , hay một giá trị không thể được đại diện trong.	Toán hạng đích lưu BCD số nguyên là giá trị không xác định.
FXCH: một hoặc cả hai thanh ghi toán hạng rỗng.	Toán hạng đích được gán là giá trị không xác định, sau đó thực hiện hoán đổi.

Câu lệnh FIST / FISTP khi toán hạng đầu vào <> MAXINT kích thước của toán hạng đích.	Toán hạng đích được gán giá trị là MAXNEG.
--	--

Bảng 3.26: Thao tác số học không hợp lệ và kết quả trả về

**Chia một số cho số 0 (#Z)**

FPU xảy ra ngoại lệ khi một giá trị dấu chấm động chi cho số 0 khi bất cứ câu lệnh nào cố gắng chia một số hữu hạn khác không toán hạng cho 0. Cờ (ZE) cho trường hợp ngoại lệ này là bit 2 của thanh ghi trạng thái FPU, và ZM bit 2 của thanh ghi điều khiển FPU được thiết lập giá trị. Các câu lệnh FDIV, FDIVP, FDIVR, FDIVRP, FIDIV, và FIDIVR và các câu lệnh khác mà thực hiện phân chia nội bộ (FYL2X và FXTRACT) có thể được đánh dấu cho trường hợp ngoại lệ chia cho số 0.

Khi một ngoại lệ chia cho số 0 xảy ra và các cờ ngoại lệ được đánh dấu, FPU bật cờ ZE và trả về các giá trị thể hiện trong Bảng 3.27. Nếu ngoại lệ chia cho số 0 không được đánh dấu, cờ ZE được bật, một phần mềm xử lý ngoại lệ sẽ được gọi con trỏ topos-stack (TOP), toán hạng nguồn vẫn không thay đổi.

Điều kiện	Kết quả trả về
Phép chia hoặc phép chia đảo được chia cho 0.	Kết quả là $\infty$ với dấu là kết quả của phép toán OR của 2 toán hạng.
Câu lệnh FYL2X.	Kết quả là $\infty$ với dấu ngược với dấu của toán hạng khác 0 được lưu vào toán hạng đích.
Câu lệnh FXTRACT.	ST(1) được lưu là $-\infty$ , ST(0) được lưu là 0 với dấu cùng dấu với toán hạng nguồn.

Bảng 3.27: Điều kiện xảy ra ngoại lệ chia cho số 0 và kết quả trả về

**Toán hạng là số denormal (#D)**

FPU báo hiệu ngoại lệ toán hạng là số denormal dựa vào điều kiện sau đây:

- Nếu một lệnh số học cố gắng để thao tác trên một toán hạng denormal.
- Nếu một thao tác được thực hiện để nạp một giá trị denormal ở định dạng single-real hoặc double-real vào một thanh ghi FPU. (Nếu giá trị denormal được nạp ở định dạng extended-real, ngoại lệ toán hạng số học là denormal toán hạng không bị đánh dấu ngoại lệ này.)

Cờ (DE) cho trường hợp ngoại lệ là bit 1 của thanh ghi trạng thái FPU, và DM là bit 1 của thanh ghi điều khiển FPU được thiết lập giá trị.

Khi ngoại lệ toán hạng là số denormal xảy ra và các bit xử lý ngoại lệ này được đánh dấu, FPU bật cờ DE, sau đó thực hiện thao tác theo câu lệnh. Các toán hạng là số denormal ở định dạng single-real hoặc double-real tự động chuyển đổi sang định dạng extended-real để thực hiện phép toán. Thao tác với số denormal sẽ tạo ra kết quả có số rất nhỏ gần bằng 0. Trong thực tế, các thao tác tiếp theo sẽ thực hiện từ kết quả ở định dạng extended-real. Hầu hết các lập trình viên đánh dấu ngoại lệ này để phép tính toán có thể thực hiện, sau đó phân tích kết quả có thực sự chính xác khi đã được tính toán. Khi ngoại lệ toán hạng là số denormal xảy ra và các bit xử lý ngoại lệ này được đánh dấu, cờ DE được bật và phần mềm xử lý ngoại lệ được gọi top-of-stack pointer (TOP) và toán hạng nguồn vẫn không thay đổi. Khi toán hạng là số denormal đã làm giảm bit phần định trị do mất bit thấp, giá trị này không nên thao tác. Ngoại trừ toán hạng là số denormal là kết quả từ tính toán có thể dùng để thực hiện các phép toán tiếp theo và ngoại lệ này không được đánh dấu.

**Tràn trên số học (#O)**

FPU tràn trên số học (#O) xảy ra bất cứ khi nào kết quả làm tròn một số học vượt quá giá trị hữu hạn lớn nhất theo định dạng số thực của toán hạng đích của câu lệnh. Ví dụ, nếu định dạng của toán hạng đích là extended-real (80 bit), tràn xảy ra khi kết quả được làm tròn nằm ngoài giới hạn của định dạng từ  $-1.0 \times 2^{16.384}$  tới  $1,0 \times 2^{16.384}$ .

Tràn số có thể xảy ra trên các phép tính mà kết quả được lưu trữ trong một thanh ghi FPU. Nó cũng có thể xảy ra khi thao tác sao lưu giá trị (câu lệnh FST, FSTP), khi giá trị ở định dạng extended-real trong một thanh ghi dữ liệu được lưu trữ trong bộ nhớ ở định dạng single-real hoặc double-real. Giới hạn ngưỡng tràn cho các định dạng single-real là  $-1.0 \times 2^{128}$  tới  $1.0 \times 2^{128}$ ; giới hạn cho các định dạng double-real là  $-1.0 \times 2^{1024}$  tới  $1.0 \times 2^{1024}$ .

Ngoại lệ tràn số không thể xảy ra khi các giá trị ở định dạng số nguyên integer hoặc BCD. Thay vào đó, các ngoại lệ giá trị toán hạng không hợp lệ được đánh dấu.

Cờ (OE) cho các ngoại lệ số tràn trên ở bit 3 của thanh ghi trạng thái FPU, và cờ OM bit thứ 3 của thanh ghi điều khiển FPU.

Khi một ngoại lệ tràn trên số học xảy ra và đánh dấu các cờ xử lý ngoại lệ cho trường hợp này, các FPU bật OE và trả về các giá trị thể hiện trong Bảng 3.28. Giá trị trả lại phụ thuộc vào định dạng làm tròn hiện tại của FPU.

Phương thức làm tròn	Dấu của kết quả	Kết quả đã được làm tròn
Làm tròn tới số gần nhất	+	$+\infty$
	-	$-\infty$
Làm tròn tới $-\infty$	+	Số dương lớn nhất có thể làm tròn.
	-	$-\infty$
Làm tròn tới $+\infty$	+	$+\infty$ .
	-	Số âm lớn nhất có thể làm tròn
Làm tròn tới 0	+	Số dương lớn nhất có thể làm tròn.
	-	Số âm lớn nhất có thể làm tròn

Bảng 3.28: Kết quả khi tràn trên số học xảy ra



Các thao tác mà FPU khi xảy ra tràn trên số học và cờ xử lý tràn trên số học không đánh dấu, phụ thuộc vào các câu lệnh mà kết quả được lưu trong bộ nhớ hoặc trong các stack thanh ghi dữ liệu.

Nếu toán hạng đích là một bộ nhớ, cờ OE được bật và phần mềm xử lý ngoại lệ được gọi top-of-stack pointer (TOP), toán hạng nguồn, đích không thay đổi.

Nếu toán hạng đích là thanh ghi trong stack, số mũ của kết quả làm tròn được chia cho  $2^{24576}$  và kết quả được lưu giữ cùng với phần định trị trong toán hạng đích. Cờ điều kiện bit C1 trong thanh ghi trạng thái FPU (gọi là trong tình huống này "làm tròn lên") được thiết lập nếu phần định trị được làm tròn lên và xóa nếu kết quả đã làm tròn. Sau khi kết quả được lưu trữ, cờ OE được bật và phần mềm xử lý ngoại lệ được gọi.

Khi sử dụng câu lệnh FSCALE, tràn trên có thể xảy ra, mà kết quả là quá lớn để được biểu diễn, ngay cả với một số mũ được điều chỉnh. Ở đây, nếu tràn trên xảy ra một lần nữa, kết quả là một  $\infty$  (vô cùng) được lưu vào trong toán hạng đích.

## Tràn dưới số học (#U)

FPU xảy ra ngoại lệ tràn dưới số học (#U) bất cứ khi nào kết quả làm tròn của các câu lệnh là một số "rất nhỏ" (có nghĩa là số nhỏ so với bình thường, giá trị hữu hạn nhỏ nhất có thể mà sẽ phù hợp với các định dạng số thực của toán hạng đích). Ví dụ, nếu định dạng toán hạng đích là extended-real (80 bit), tràn dưới xảy ra khi kết quả được làm tròn xuống trong khoảng giới hạn  $-1.0 \times 2^{-16.382}$  tới  $1,0 \times 2^{16.382}$ . Giống như tràn trên số học, tràn dưới số học có thể xảy ra trên các phép tính mà kết quả được lưu trữ trong thanh ghi dữ liệu FPU. Ngoại lệ này cũng có thể xảy ra khi thực hiện thao tác lưu (câu lệnh FST, FSTP), một giá trị trong thanh ghi dữ liệu được lưu trữ vào trong bộ nhớ ở định dạng single-real hoặc double-real. Giới hạn tràn dưới của định dạng single-real là  $-1.0 \times 2^{-126}$  tới  $1,0 \times 2^{126}$ ; giới hạn tràn dưới của định dạng double-real là  $-1.0 \times 2^{-1022}$  tới  $1,0 \times 2^{1022}$ . (Ngoại lệ tràn dưới số học không có thể xảy ra khi lưu trữ các giá trị trong

một định dạng số nguyên hoặc BCD.)

Cờ (UE) xử lý ngoại lệ tràn dưới số học ở bit 4 của thanh ghi trạng thái FPU, và bit UM ở bit 4 của thanh ghi điều khiển FPU được thiết lập.

Khi một ngoại lệ tràn dưới số học xảy ra và các bit xử lý ngoại lệ này được đánh dấu, kết quả là số denormalizes. Nếu kết quả số denormalized là chính xác, FPU lưu kết quả trong toán hạng đích, mà không cần thiết lập các cờ UE. Nếu kết quả số denormal không chính xác, FPU bật UE, sau đó đi vào để xử lý các điều kiện ngoại lệ kết quả không chính xác.

Điều quan trọng cần lưu ý là nếu tràn dưới số học được đánh dấu, cờ ngoại lệ tràn dưới số học được bật kết quả số denormalized là không chính xác. Nếu kết quả số denormalized là chính xác, không có cờ nào được thiết lập và không có ngoại lệ xảy ra.

Các thao tác mà FPU khi xảy ra tràn dưới số học và cờ xử lý tràn dưới số học không được đánh dấu, phụ thuộc các câu lệnh để lưu trữ các kết quả trong bộ nhớ hoặc trong stack thanh ghi dữ liệu.

Nếu toán hạng đích là stack thanh ghi, số mũ kết quả làm tròn là  $2^{24576}$  và kết quả được lưu giữ cùng với phần định trị của toán hạng đích. Mã điều kiện bit C1 trong thanh ghi trạng thái FPU được thiết lập nếu phần định trị được làm tròn lên. Sau khi kết quả được lưu, cờ UE bật và một phần mềm xử lý ngoại lệ được gọi.

Khi sử dụng câu lệnh FSCALE, tràn dưới có thể xảy ra, mà kết quả là quá nhỏ để có thể biểu diễn, thậm chí với số mũ đã được điều chỉnh. Ở đây, nếu tràn dưới số học xảy ra một lần nữa, kết quả là 0 được lưu trữ trong toán hạng đích.

**Kết quả không chính xác (#P)**

Trường hợp ngoại lệ kết quả không chính xác (còn gọi là các ngoại lệ chính xác) xảy ra nếu kết quả của một thao tác được biểu diễn không chính xác theo định dạng toán hạng đích. Ví dụ, tỷ lệ  $\frac{1}{3}$  không thể được biểu diễn chính xác ở dạng nhị phân. Ngoại lệ này được hỗ trợ cho các ứng dụng mà cần phải thực hiện số học chính xác số học. Bởi vì kết quả làm tròn là thường phù hợp cho hầu hết các ứng dụng, ngoại lệ này thường được che dấu. Lưu ý rằng các câu lệnh có độ chính xác cao (FSIN, FCOS, FSIN-COS, FPTAN, FPATAN, F2XM1, FYL2X, và FYL2XP1) tạo ra kết quả không chính xác.

Ngoại lệ kết quả không chính xác sử dụng cờ (PE) là bit 5 của thanh ghi trạng thái FPU, và PM ở bit 5 của thanh ghi điều khiển FPU.

Nếu trường hợp ngoại lệ kết quả không chính xác xảy ra và lỗi tràn trên số học hoặc tràn dưới số học xảy ra đồng thời, FPU bật PE và lưu trữ các kết quả được làm tròn vào toán hạng đích. Chế độ làm tròn hiện tại xác định các phương thức được sử dụng để làm tròn kết quả. Bit C1 trong thanh ghi trạng thái FPU thể hiện kết quả không chính xác đã được làm tròn lên (C1 được bật) hoặc "không làm tròn lên" (C1 tắt). Trong trường hợp "không làm tròn lên", các bit ít thấp của các kết quả không chính xác được cắt ngắn vì vậy mà kết quả phù hợp theo định dạng toán hạng đích.

Nếu trường hợp ngoại lệ kết quả không chính xác, các bit xử lý ngoại lệ này không đánh dấu và tràn trên số học hoặc tràn dưới số học không xảy ra đồng thời, FPU thực hiện các thao tác gọi một hàm xử lý phần mềm ngoại lệ.

Nếu ngoại lệ kết quả không chính xác xảy ra kết hợp với tràn trên số học hoặc tràn dưới số học, một trong những hoạt động sau đây được thực hiện:

- Nếu ngoại lệ kết quả không chính xác xảy ra cùng với cờ xử lý tràn trên số học hoặc tràn dưới số học được đánh dấu, cờ OE hoặc UE cờ và cờ PE được bật và kết quả được lưu như mô tả cho các trường hợp ngoại lệ tràn trên hoặc tràn dưới.

- Nếu ngoại lệ kết quả không chính xác xảy ra cùng với cờ xử lý tràn trên số học hoặc tràn dưới số học không đánh dấu và cùng sử dụng một toán hạng đích, cờ OE hoặc cờ UE và cờ PE được bật, kết quả được lưu dưới dạng mô tả theo ngoại lệ tràn trên số học hoặc ngoại lệ tràn dưới số học, và gọi phần mềm xử lý ngoại lệ.
- Nếu ngoại lệ kết quả không chính xác xảy ra cùng với cờ xử lý tràn trên số học hoặc tràn dưới số học không đánh dấu và sử dụng chung toán hạng đích là bộ nhớ, ngoại lệ kết quả không chính xác bỏ qua.

### Thứ tự ưu tiên xử lý ngoại lệ

Các bộ xử lý xử lý các trường hợp ngoại lệ theo độ ưu tiên được xác định trước. Khi một câu lệnh tạo ra hai hoặc nhiều điều kiện ngoại lệ, các ưu tiên ngoại lệ xử lý kết quả trong các trường hợp ngoại lệ ưu tiên cao hơn được xử lý trước và các trường hợp ngoại lệ có mức ưu tiên thấp bị bỏ qua. Ví dụ, chia SNaN cho số 0 có khả năng có thể báo hiệu một ngoại lệ toán hạng không hợp lệ (do các toán hạng SNaN) và một ngoại lệ chia cho 0. Ở đây, nếu cả hai trường hợp ngoại lệ được đánh dấu, FPU xử lý các trường hợp ngoại lệ ưu tiên cao hơn (ngoại lệ toán hạng không hợp lệ) trả lại kết quả không xác định cho toán hạng đích. Ví dụ khác, ngoại lệ toán hạng là số denormal hoặc số không chính xác có thể xảy ra đồng thời với ngoại lệ tràn dưới số học hoặc tràn trên số học, với cả hai trường hợp ngoại lệ được xử lý.

Ưu tiên xử lý trường hợp ngoại lệ của dấu chấm động như sau:

1. Toán hạng không hợp lệ, chia như sau:
  - a. Stack tràn dưới.
  - b. Stack trên.
  - c. Toán hạng có định dạng không được hỗ trợ.
  - d. Toán hạng là SNaN.
2. Toán hạng là QNaN. Mặc dù đây không phải là một ngoại lệ, việc xử lý của một toán hạng QNaN được ưu tiên hơn. Ví dụ, toán hạng QNaN chia cho 0 có kết quả là QNaN, không phải là một ngoại lệ zero-chia.

3. Bất kỳ trường hợp toán hạng không hợp lệ không được đề cập ở trên xử lý trước ngoại lệ chia cho số 0.
4. Toán hạng là số denormal. Nếu ngoại lệ này được đánh dấu, sau đó thực hiện lệnh tiếp theo, và ngoại lệ ưu tiên thấp có thể xảy ra mà không bị ảnh hưởng.
5. Tràn trên số học và tràn dưới số học xử lý trước ngoại lệ kết quả không chính xác
6. Ngoại lệ kết quả không chính xác.

Toán hạng không hợp lệ, chia cho số 0, và trường hợp ngoại lệ toán hạng là số denormal được phát hiện trước khi bắt đầu thao tác câu lệnh xử lý dấu chấm động, trong khi tràn trên, tràn dưới, và kết quả không chính xác được phát hiện sau khi đã được tính toán và cho ra kết quả. Khi một ngoại lệ trước thao tác được phát hiện, stack thanh ghi dữ liệu FPU và bộ nhớ chưa được cập nhật và được đánh dấu trước khi thực hiện các câu lệnh, đồng thời câu lệnh đó không được thực hiện. Khi một ngoại lệ phát hiện sau khi thực hiện câu lệnh, stack thanh ghi dữ liệu FPU và bộ nhớ có thể được cập nhật với một kết quả (tùy thuộc vào loại ngoại lệ xảy ra).

### 3.3 Windows API

#### 3.3.1 Một vài bộ thư viện cần hỗ trợ

Ở những bước xây dựng đầu của đề tài, cần ưu tiên tiến hành cho những bộ thư viện phổ biến và được dùng rộng rãi trước tiên. Bảng 2 sau đây sẽ mô tả thông tin những bộ thư viện đã được hỗ trợ bởi BE-PUM:

Tên	Chức năng
Dịch vụ nền	Cho phép truy cập vào các nguồn tài nguyên cơ bản có sẵn trong hệ thống của Windows. Bao gồm những thứ như hệ thống tập tin, thiết bị, tiến trình (process), luồng (thread), xử lý lỗi (error handling). Các chức năng này nằm trong tập tin kernel32.dll trên hệ điều hành Windows 32-bit.
Dịch vụ nâng cao	Cho phép truy cập vào các chức năng bổ sung. Bao gồm những thứ như Windows registry, tắt/khởi động lại hệ thống (hoặc bãi bỏ - abort), bắt đầu/dừng/tạo ra một dịch vụ, quản lý tài khoản người dùng. Các chức năng này nằm trong tập tin advapi32.dll trên hệ điều hành Windows 32-bit.
Giao diện người dùng	Cung cấp các chức năng để tạo và quản lý cửa sổ màn hình và hầu hết những điều khiển cơ bản, chẳng hạn như nút và thanh cuộn, nhận chuột và bàn phím, cũng như các chức năng khác liên quan đến phần giao diện đồ họa người dùng của Windows.
Các chức năng này nằm trong tập tin user32.dll trên hệ điều hành Windows 32-bit. Windows shell	Cho phép các ứng dụng truy cập các chức năng được cung cấp bởi các shell của hệ điều hành, cũng như thay đổi nó. Windows shell ở đây được hiểu là những thành phần cấu tạo nên giao diện đồ họa người dùng của hệ điều hành Windows (bao gồm những thứ như: màn hình desktop, thanh làm việc taskbar và các thành phần con trên nó, ...). Các chức năng này nằm trong tập tin shell32.dll trên hệ điều hành Windows 32-bit.

Bảng 3.29: Những bộ thư viện được hỗ trợ bởi BE-PUM

### 3.3.2 Trình tự các bước để ứng dụng JNA vào BE-PUM

Để tiến hành áp dụng những khả năng mang lại từ JNA và hệ thống BE-PUM, ta cần trải qua những bước sau đây:

1. Ánh xạ tương ứng tên thư viện sẽ được gọi
2. Ánh xạ tên hàm của API và những kiểu dữ liệu sẽ được dùng (bao gồm kiểu dữ liệu trả về và kiểu dữ liệu của các thông số đầu vào) từ ngôn ngữ lập trình C sang Java
3. Tiến hành lấy giá trị bộ nhớ của các thông số đầu vào có trong BE-PUM
4. Truyền các giá trị đó vào những kiểu tương ứng đã được ánh xạ trong Java, gọi API và lấy kết quả trả về.
5. Lưu giá trị nhận được đó về lại bộ nhớ của BE-PUM

### 3.3.3 Những kiểu dữ liệu và ánh xạ của chúng vào JNA

Việc ánh xạ kiểu dữ liệu từ ngôn ngữ lập trình C sang Java được hỗ trợ sẵn bởi JNA cho một số kiểu cơ bản thường dùng như sau:

STT	Kiểu dữ liệu trong C	Kiểu dữ liệu trong Java
1	char	byte
2	wchar_t	char
3	short	short
4	int	int
5	int	boolean
6	enum	int (thông thường)
7	long long, __int64	long
8	float	float
9	double	double
10	Con trỏ (VD: void*)	Buffer Pointer

11	Con trỏ (VD: void*, char*) Mảng	<P>[] (mảng của những kiểu nguyên thủy)
12	long	NativeLong
13	const char*	String
14	const wchar_t*	WString
15	char**	String[]
16	wchar_t**	WString[]
17	void**	Pointer[]
18	struct* struct	Structure
19	union	Union
20	struct[]	Structure[]
21	void (*FP)()	Callback
22	pointer (<T> *)	PointerType

Bảng 3.30: Một số ánh xạ kiểu dữ liệu cơ bản

Cần bản việc ánh xạ kiểu dữ liệu trong JNA được đáp ứng bằng việc kích thước dữ liệu được sử dụng giữa hai ngôn ngữ C và Java có kích thước bằng nhau. JNA hỗ trợ cho việc nạp một mảng các kí tự trong C bằng cách cho vào một đối tượng kiểu String.

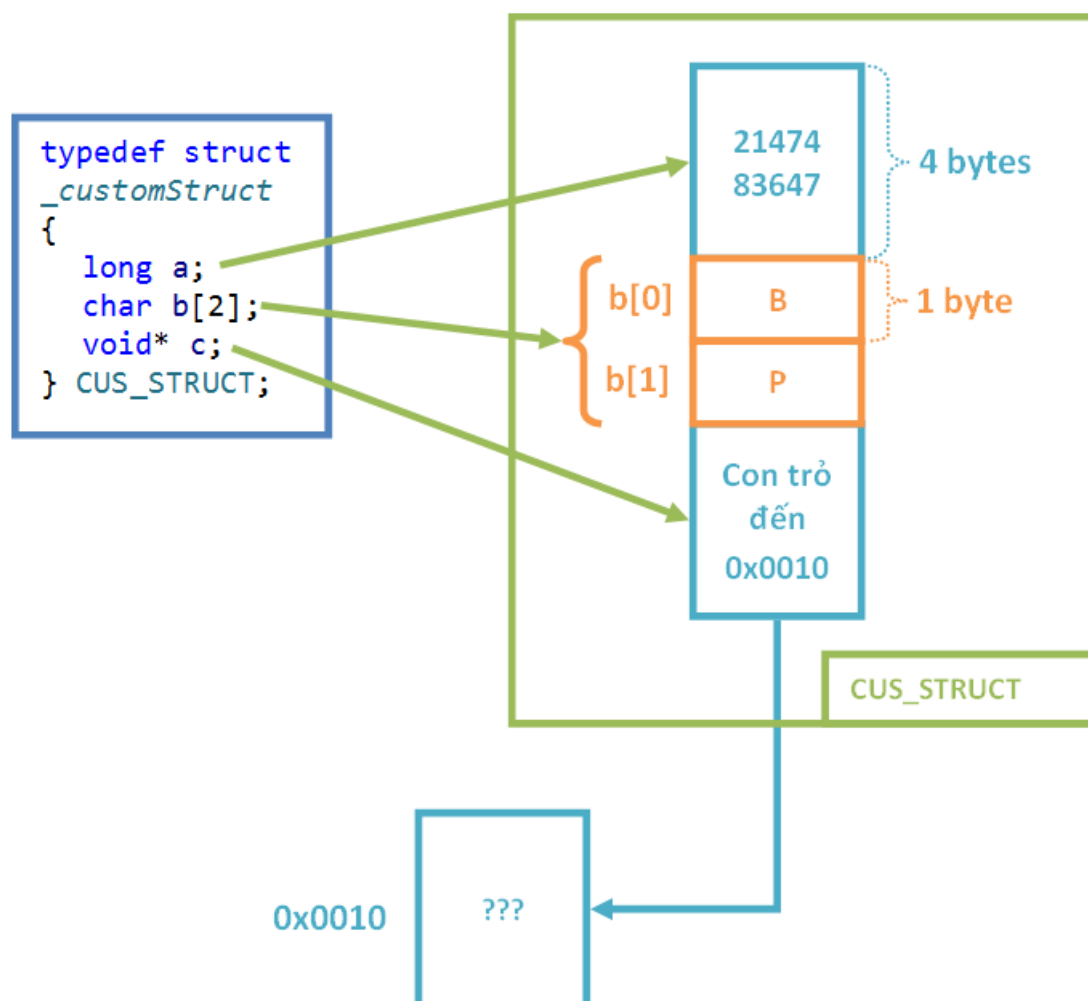
Điểm hỗ trợ mạnh trong JNA đó là sử dụng và truy cập con trỏ của C thông qua đối tượng kiểu Pointer, dù rằng đang làm việc trong ngôn ngữ Java. Và nếu muốn lấy giá trị của một vùng nhớ thay vì chỉ một giá trị ô nhớ như Pointer, ta có thể sử dụng đối tượng kiểu Buffer để JNA đưa giá trị vùng nhớ đó vào trong đối tượng.



### 3.3.4 Dữ liệu kiểu cấu trúc (structure)

Một trong những lợi thế to lớn nhất mà JNA mang lại đó là nạp dữ liệu kiểu cấu trúc (structure) vào trong C mà ngôn ngữ Java không có. Điều này được hỗ trợ qua việc tạo ra một lớp mới, kế thừa từ lớp Structure của JNA.

Để tạo ra chính xác kiểu cấu trúc để nạp vào JNA, ta cũng phải tìm hiểu và nắm rõ các quy tắc xây dựng và làm việc với kiểu cấu trúc trong ngôn ngữ lập trình C.



Hình 3.20: Cách sắp xếp và lưu trữ bộ nhớ của structure

Một struct trong ngôn ngữ lập trình C là một khai báo dữ liệu kiểu phức hợp, trong đó nó định nghĩa một danh sách các biến được đặt bên trong một khối bộ nhớ. Nó cho

phép các biến khác nhau được truy xuất thông qua một con trỏ duy nhất (con trỏ của structure). Một structure có khả năng chứa nhiều kiểu dữ liệu khác nhau từ kiểu dữ liệu nguyên thủy đến kiểu dữ liệu phức tạp khác (enum, structure).

Định nghĩa struct khá giống với định nghĩa lớp trong Java, nhưng struct không hề có khả năng khai báo phương thức, cũng như các từ khóa public, private,... như Java. Thêm vào đó, cơ chế cấp phát bộ nhớ của struct chỉ đơn thuần là sự sắp xếp liên tục của các vùng nhớ chứa giá trị của các biến đã được khai báo bên trong. Cách sắp xếp này hoàn toàn tương ứng với ngôn ngữ assembly; do đó ta cần đảm bảo việc lấy ra và gán trở lại bộ nhớ BE-PUM một cách chính xác theo trình tự như vậy.

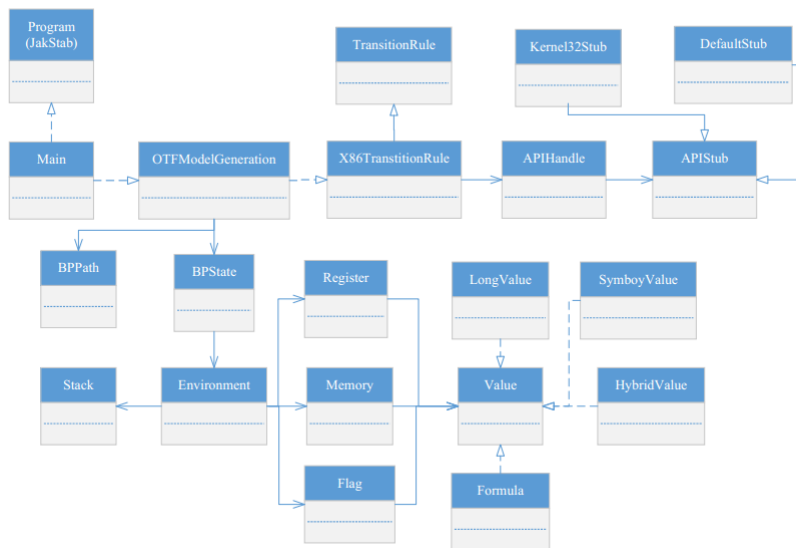
## Chương 4

# Thiết Kế Và Xây Dựng

### 4.1 Các câu lệnh hợp ngữ

#### 4.1.1 Sơ đồ chương trình BE-PUM

BE-PUM được xây dựng chủ yếu trên mã nguồn của JakStab do đó hình 4.1 thể hiện phần BE-PUM đang được phát triển. Dưới đây là sơ đồ class chương trình:



Hình 4.1: Sơ đồ class chương trình BE-PUM

Sơ đồ thể hiện mối liên hệ giữa các class với nhau. Trong sơ đồ không thể hiện các biến của class, các hàm của class, mục đích của sơ đồ thể hiện mối liên kết giữa các class

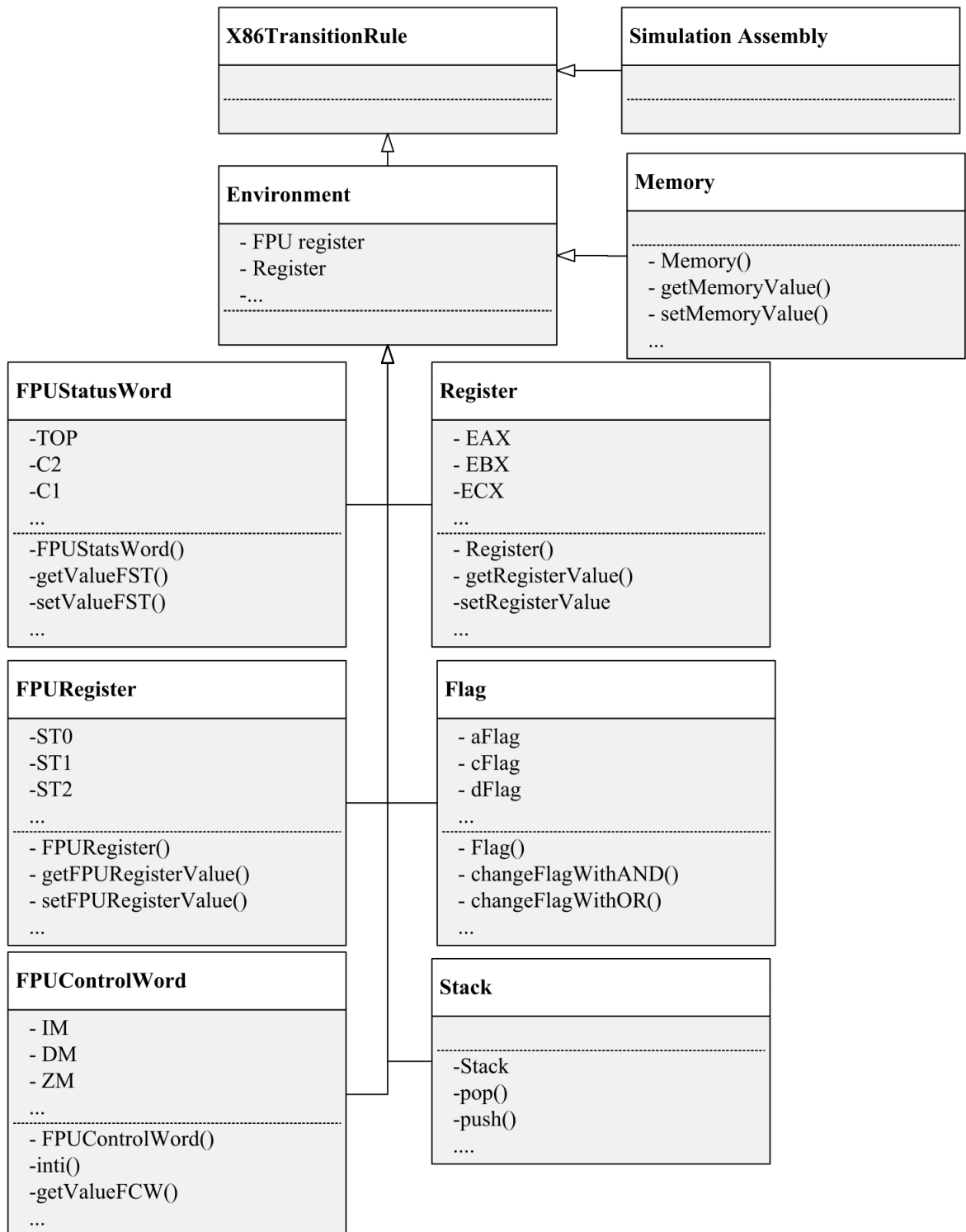
với nhau. Class *Program* là một class được phát triển từ dự án JakStab, BE-PUM kế thừa và phát triển lên. Class chính của BE-PUM là class *Main*, nơi gọi chương trình cần phân tích. Class *Main* sẽ gọi class *OTFModelGeneration* để tạo ra sơ đồ CFG(control flow graph: sơ đồ điều khiển) với mỗi đỉnh là một câu lệnh assembly được dịch từ mã nhị phân của chương trình cần phân tích. Mỗi câu lệnh assembly có các biến môi trường khác nhau. Class *Environment* có nhiệm vụ thể hiện môi trường làm việc của từng câu lệnh. Biến môi trường có các thành phần khác nhau như *Register*, *Flag*, *Stack*, *Memory* cũng được hiện thực bằng các class tương ứng.

Class *X86TransitionRule* có nhiệm vụ hiện thực mô phỏng các câu lệnh assembly. Để hiện thực mô phỏng các câu lệnh khác nhau, class *X86TransitionRule* được hỗ trợ thêm các class khác sẽ được trình bày trong phần "phân tích class mô phỏng câu lệnh Assembly". Class *X86TransitionRule* cũng đảm nhiệm việc tiến hành phân tích từng câu lệnh assembly, trong quá trình phân tích sẽ có API (Microsoft Windows application programming interface) tham gia, vấn đề về API sẽ được đề cập đến phần "Window API".

### 4.1.2 Phân tích class mô phỏng câu lệnh Assembly

Hình 4.2 mô tả lược đồ lớp (class) chính dùng để mô phỏng câu lệnh assembly. Class quan trọng *X86TransitionRule* và *Simulation Assembly* dùng để mô phỏng thao tác thực hiện của các câu lệnh assembly. Ngoài ra còn có class *Environment* mô phỏng các biến sẽ bị thay đổi khi câu lệnh assembly được thực thi. Các class *FPUStatusWord*, *FPURegister*, *FPUControlWord* mô phỏng lại các biến môi trường mà các câu lệnh assembly xử lý FPU (Floating-Point Unit) sử dụng. Class *Register*, *Flag*, *Stack*, *Memory* mô phỏng lại các câu lệnh assembly thao tác trên số nguyên.

Trong phần này sẽ phân tích môi trường cần để câu lệnh assembly xử lý số nguyên thực hiện và môi trường FPU xử lý số thực. Câu lệnh xử lý số nguyên và số thực tuy cùng mục đích thực hiện nhưng khi thao tác và thực hiện câu lệnh, các biến môi trường được sử dụng khác nhau. Nên việc thao tác, phân tách các biến môi trường sao cho phù hợp với hai nhóm xử lý số nguyên.



Hình 4.2: Sơ đồ class mô phỏng câu lệnh Assembly

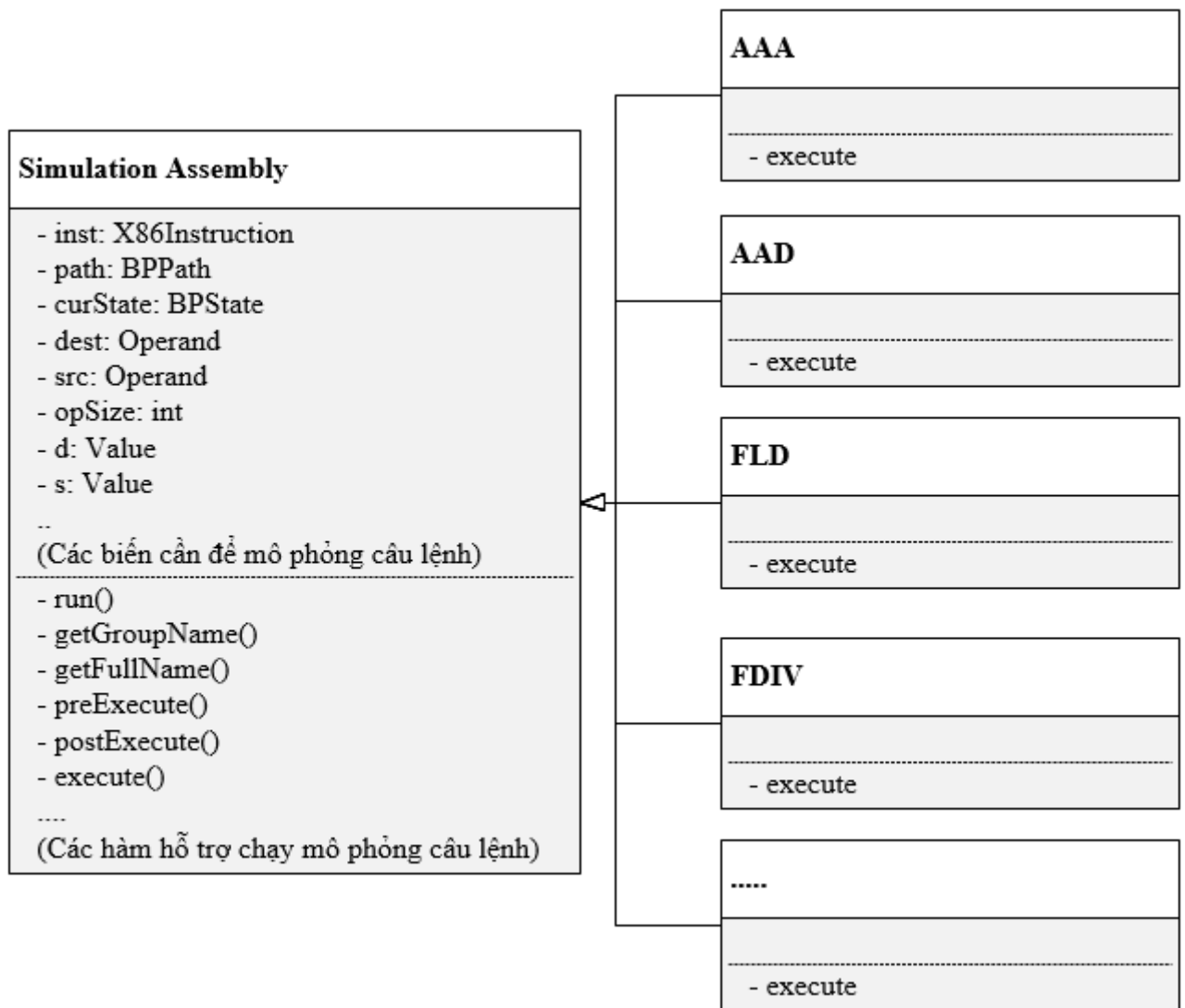
Để thể hiện các biến môi trường các class Register, Flag, Stack, FPUStatusWord, FPURregister, FPUControlWord, Be-Pum xây dựng các class Environment để hỗ trợ trong việc truy xuất, thao tác.



Hình 4.3: Class Environment

Class Environmet có nhiệm vụ liên kết các class Stack, Register, Memory, Flag, FPUS-tatusWord, FPURregister, FPUControlWord, để thao tác truy xuất trực tiếp lên các biến môi trường này. Class Enviroment hiện thực các hàm để hỗ trợ thao tác truy xuất thuận tiện hơn. Ngoài ra, hiện thực hàm equals() để so sánh biến môi trường của câu lệnh assembly này với biến môi trường của câu lệnh assembly khác. Từ đó hỗ trợ trong việc đánh giá, phân tích câu lệnh assembly.

Class Simulation Assembly là một class abstract dùng để gọi các class với tên câu lệnh assembly tương ứng

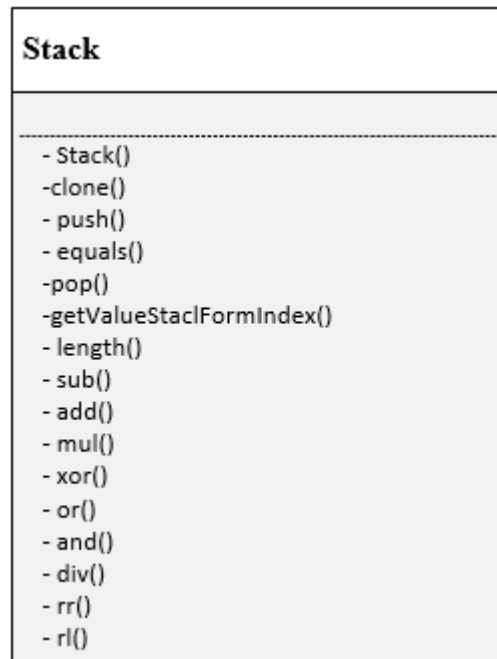


Hình 4.4: Class

Class Simulation Assembly có nhiệm vụ hash tìm câu lệnh tương ứng với class để thực thi. Hàm execute() dùng để gọi lệnh thực thi tương ứng với câu lệnh đầu vào. Ngoài ra còn có các hàm getGroupName() hỗ trợ trong quá trình tìm kiếm được tối ưu. Hàm getFullName() hỗ trợ tìm kiếm với tên câu lệnh đầy đủ, không bị thiếu ký tự trong quá trình tìm kiếm class tương ứng. Ngoài ra có các biến dest, src lần lượt là toán hạng tương ứng với từng câu lệnh assembly được thực thi.

## Môi trường xử lý số nguyên

Class Environmet liên kết với class Stack, hình () thể hiện class Stack.



Hình 4.5: Class Stack

Class Stack gồm các hàm hỗ trợ thao tác trên stack của chương trình. Hàm push() và pop() là hai hàm quan trọng có nhiệm vụ thao tác cơ bản trên stack là đưa vào ngăn xếp và lấy ngăn xếp ra. Ngoài ra còn có các hàm length() thể hiện kích thước stack, equals() để so sánh hai stack với nhau, việc này hỗ trợ rất nhiều trong quá trình phân tích hay so trùng được sử dụng trong các kỹ thuật phân tích virus. Ngoài ra còn có các hàm để hỗ trợ thao tác toán học như add() để cộng thêm một số, sub() để nhân với một số,...



Class Register được hiện thực như các thanh ghi Register trong hợp ngữ assembly. Hình () thể hiện class Register:

Register
<ul style="list-style-type: none"><li>- eax</li><li>- ebx</li><li>- ecx</li><li>- edx</li><li>- esi</li><li>- ax</li><li>- ah</li><li>- al</li><li>- sp</li><li>...</li></ul> (Các biến thanh ghi)
<ul style="list-style-type: none"><li>- Register()</li><li>- getRegisterValue()</li><li>- setRegisterValue()</li><li>- add()</li><li>- and()</li><li>- mul()</li><li>- mov()</li><li>- or()</li><li>...</li></ul> (Các hàm hỗ trợ thao tác thanh ghi)

Hình 4.6: Class Register

Class Register gồm các biến được thể hiện là các thanh ghi Register được sử dụng trong hợp ngữ assembly. Các biến này sẽ lưu giá trị của thanh ghi trong quá trình thực thi chương trình cần phân tích. Mỗi giá trị của biến có thể bị thay đổi khi thực hiện phân tích từng câu lệnh assembly của chương trình cần phân tích. Class Register xây dựng các hàm hỗ trợ thao tác trên thanh ghi như getRegisterValue() có nhiệm vụ lấy giá trị của thanh ghi nào đó, setRegisterValue() có nhiệm vụ gán lại giá trị của một thanh ghi. Ngoài ra, class Register còn hiện thực một số hàm hỗ trợ thao tác toán học trên thanh ghi như: add() có nhiệm vụ cộng thêm một giá trị nào đó vào thanh ghi, and() có nhiệm vụ như thực hiện phép and (đại số Boole),... những hàm này giúp hỗ trợ nhanh trong việc hiện thực câu lệnh sau này.

Class Memory được hiện thực như là các thao tác trên một bộ nhớ được sử dụng trong hợp ngữ assembly. Hình () thể hiện class Memory:

Memory
<ul style="list-style-type: none"><li>- equals()</li><li>- Memory()</li><li>- getMemoryValue()</li><li>- setMemoryValue()</li><li>- getByteMemoryValue()</li><li>- setByteMemoryValue()</li><li>- getWordMemoryValue()</li><li>- setWordMemoryValue()</li><li>- addMemoryValue()</li><li>- mulMemoryValue()</li><li>- orMemoryValue()</li><li>- divMemoryValue()</li><li>- mulMemoryValue()</li><li>- subMemoryValue()</li><li>- changeValue()</li><li>- resetValue()</li><li>...</li></ul> <p>(Các hàm hỗ trợ thao tác trên bộ nhớ)</p>

Hình 4.7: Class Memory

Class Memory gồm các hàm được hiện thực để hỗ trợ trong việc thao tác với địa chỉ bộ nhớ. Mỗi hàm được hiện thực có nhiệm vụ khác nhau nhưng cùng một mục đích là phân tích câu lệnh assembly như hàm equals() dùng để so sánh bộ nhớ, getMemoryValue() dùng để lấy giá trị của địa chỉ bộ nhớ đó, setMemoryValue() dùng để gán lại giá trị của bộ nhớ, getByteMemoryValue() dùng để lấy giá trị Byte của địa chỉ bộ nhớ, ngoài ra còn có các hàm hỗ trợ thao tác toán học được sử dụng trên bộ nhớ như hàm addMemoryValue() dùng để cộng thêm một số vào giá trị bộ nhớ, mulMemoryValue() dùng để nhân thêm một số vào giá trị bộ nhớ, changeValue() với địa chỉ xác định dùng để thay đổi giá trị của địa chỉ bộ nhớ đó, việc thay đổi giá trị tại địa chỉ bộ nhớ giúp hỗ trợ trong quá trình phân tích, đọc hiểu mã assembly.

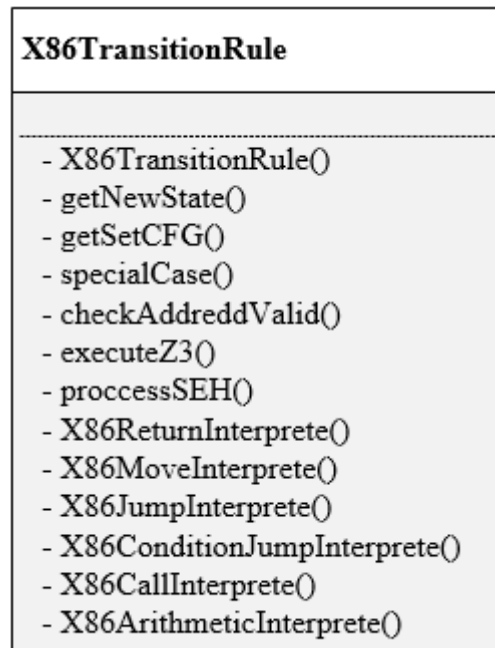
Class Flag được hiện thực như các Flag trong hợp ngữ assembly. Hình () thể hiện class Flag của chương trình BE-PUM:

Flag
<ul style="list-style-type: none"><li>- aFlag</li><li>- cFlag</li><li>- dFlag</li><li>- iFlag</li><li>- oFlag</li><li>- pFlag</li><li>- sFlag</li><li>- zFlag</li></ul>
<ul style="list-style-type: none"><li>- getAFlag()</li><li>- setAFlag()</li><li>- getCFlag()</li><li>- setCFlag()</li><li>- getDFlag()</li><li>- getDFlag()</li><li>- changeFlagWithINC()</li><li>- changeFlagWithDEC()</li><li>- changeFlagWithSUB()</li><li>- changeFlagWithADD()</li><li>...</li></ul> <p>(Các hàm hỗ trợ thao tác trên Flag)</p>

Hình 4.8: Class Flag

Class Flag gồm các biến được phỏng theo các biến Flag được sử dụng trong hợp ngữ assembly, mỗi tên biến tương với mỗi giá trị Flag trong quá trình phân tích. Các giá trị của Flag sẽ thay đổi theo từng câu lệnh assembly, dựa vào sự thay đổi này để hỗ trợ trong quá trình phân tích chương trình. Ngoài ra, class Flag còn hiện thực thêm một số hàm để hỗ trợ trong quá trình phân tích như getAFlag() để lấy giá trị của aFlag, setAFlag() để thay đổi giá trị của aFlag, tương tự như vậy đối với các biến đã được mô phỏng. Thêm vào đó là các hàm thay đổi Flag theo các câu lệnh hỗ trợ toán học như hàm changeFlagWithINC() có nhiệm vụ thay đổi các giá trị của Flag tương ứng khi thực hiện câu lệnh INC (tăng giá trị), changeFlagWithSUB() thay đổi Flag khi thực hiện phép toán công (SUB),... Tương tự như vậy, class tiếp tục xây dựng các hàm hỗ trợ thay đổi nhanh khi thực hiện các phép toán.

Class X86TransitionRule là class đặc biệt quan trọng, là nơi xử lý các câu lệnh assembly, class X86TransitionRule được hỗ trợ bởi nhiều class con khác để thực hiện việc phân tích mã assembly. Hình () biểu diễn class X86TransitionRule:



Hình 4.9: Class X86TransitionRule

Class X86TransitionRule được hiện thực các hàm để hỗ trợ trong quá trình phân tích chương trình assembly. Mỗi đỉnh trong đồ thị phân tích là một câu lệnh, mỗi câu lệnh có nhiệm vụ, chức năng khác nhau. Do đó, cần phân nhóm các câu lệnh để hiện thực. Trong class X86TransitionRule, các hàm được chia theo nhóm câu lệnh như X86IntrustionInterprete() hiện thực các câu lệnh cơ bản thường dùng như STD (gán giá trị dFlag = 1) cmovcc (các điều kiện gán)... X86RetrunInterprete() hiện thực các câu lệnh trả kết quả về như RET (kết quả trả về khi kết thúc một hàm), X86MoveInterprete() hiện thực các câu liên quan đến gán giá trị như MOV, MOVZ,... tương tự như vậy các nhóm câu lệnh nhảy, biểu thức toán học, các câu lệnh gọi hàm được hiện thực. Ngoài ra còn có các hàm hỗ trợ trong việc phân tích và tính toán như hàm executeZ3() có nhiệm vụ gọi thư viện Z3 để tính toán. Đây là class rất quan trọng vì class này đảm nhận nhiệm vụ phân tích chương trình assmebly từ đó đưa ra kết quả.

## Môi trường xử lý số thực

Class `FPUControlWord` mô phỏng thanh ghi Control Word FPU. Hình () biểu diễn các biến và hàm quan trọng trong để mô phỏng thanh ghi điều khiển FPU.

<b>FPUControlWord</b>
<ul style="list-style-type: none"> <li>- IM</li> <li>- DM</li> <li>- ZM</li> <li>- OM</li> <li>- UM</li> <li>- PM</li> <li>- PC</li> <li>- RC</li> <li>- X</li> </ul>
<hr/> <ul style="list-style-type: none"> <li>-FPUControlWord</li> <li>- inti()</li> <li>- reset()</li> <li>- getIM()</li> <li>- setIM()</li> <li>- getDM()</li> <li>- setDM()</li> <li>- getZM()</li> <li>- setZM()</li> <li>- getValueFCW()</li> <li>...</li> <li>(Các hàm thao tác trên thanh ghi điều khiển)</li> </ul>

Hình 4.10: Class `FPUControlWord`

Class `FPUControlWord` chứa các biến IM, DM, ZM, ... tương ứng với các bit cờ trong thanh ghi điều khiển FPU. Mỗi biến biểu diễn một bit trong thanh ghi. Hàm `getValueFCW()` trả giá trị hex, việc trả giá trị hex giúp so sánh, kiểm tra kết quả nhanh hơn. Ngoài ra còn các hàm hỗ trợ truy xuất, thiết lập các biến IM, DM, ZM... giúp thuận tiện trong quá trình hiện thực các câu lệnh, thay đổi giá trị trong thanh ghi điều khiển FPU. Hàm `inti()` thiết lập các thông số ban đầu của thanh ghi điều khiển FPU.

Class FPUStatusWord mô phỏng thanh ghi Status Word FPU. Hình () biểu diễn các biến và hàm quan trọng trong để mô phỏng thanh ghi trạng thái FPU.

<b>FPUStatusWord</b>
<ul style="list-style-type: none"><li>- B</li><li>- C3</li><li>- TOP</li><li>- C2</li><li>- C1</li><li>- C0</li><li>- ES</li><li>- SF</li><li>...</li></ul> <p>(Các biến thanh ghi trạng thái FPU)</p>
<ul style="list-style-type: none"><li>- FPUStatusWord()</li><li>- inti()</li><li>- reset()</li><li>- getB()</li><li>- setB()</li><li>- getC3()</li><li>- setC3()</li><li>- getTOP()</li><li>- setTOP()</li><li>- changeFCLEX()</li><li>- getValueFSW</li><li>...</li></ul> <p>(Các hàm hỗ trợ thao tác trên thanh ghi trạng thái)</p>

Hình 4.11: Class FPUStatusWord

Class FPUStatusWord mô phỏng các bit được sử dụng trong thanh ghi trạng thái FPU tương ứng với các biến B, C3, TOP, C2, .... Mỗi biến biểu diễn một bit trong thanh ghi, riêng biến TOP tương ứng với giá trị đỉnh của stack được biểu diễn bằng số nguyên. Class FPUStatusWord có các hàm hỗ trợ trong việc truy xuất và thiết lập các biến getB(), setB(),... Hàm getValueFSW() trả về giá trị hex, việc trả giá trị hex giúp kiểm tra và so sánh kết quả của từng câu lệnh assembly sau khi thực hiện xong. Hàm inti() thiết lập các thông số ban đầu cho thanh ghi trạng thái FPU.

Class FPURegister mô phỏng stack thanh ghi dữ liệu FPU. Hình () biểu diễn các biến và hàm quan trọng trong để mô phỏng stack thanh ghi dữ liệu FPU.

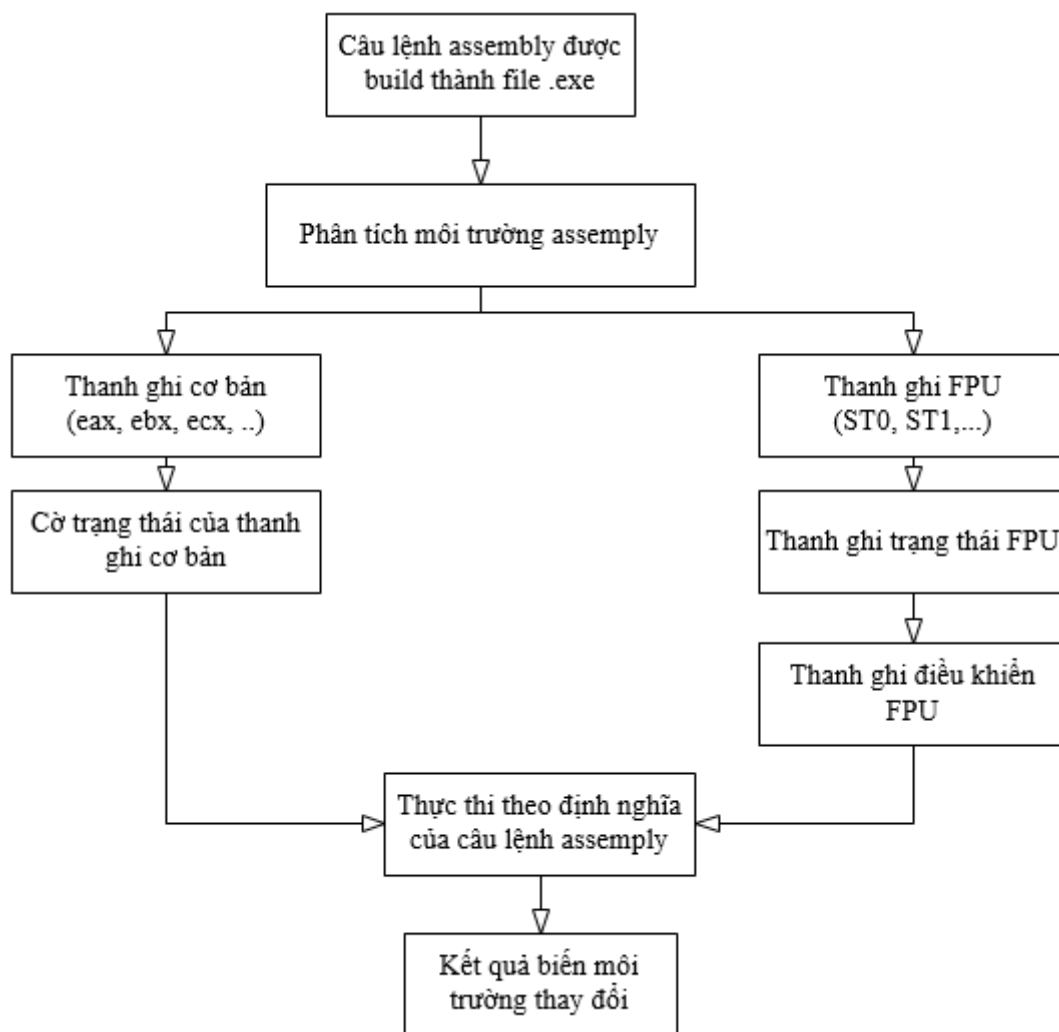
<b>FPURegister</b>
<ul style="list-style-type: none"><li>- st0</li><li>- st1</li><li>- st2</li><li>- st3</li><li>...</li><li>- tag0</li><li>- tag1</li><li>- tag2</li><li>- tag3</li><li>...</li></ul> <p>(Các biến biểu diễn stack thanh ghi dữ liệu và TagWord)</p>
<ul style="list-style-type: none"><li>- FPURegister()</li><li>- reset()</li><li>- FPUTagWord()</li><li>- POP()</li><li>- FLD()</li><li>- FDIV()</li><li>- FADD()</li><li>- FSUB()</li><li>- FMUL()</li><li>...</li></ul> <p>(Các hàm hỗ trợ thao tác trên stack thanh ghi dữ liệu)</p>

Hình 4.12: Class

Class FPURegister có các biến ST0, ST1, ST2,... lần lượt tương ứng với các stack thanh ghi dữ liệu được dùng để thao tác trong xử lý dấu chấm động. Các biến tag0, tag1, tag2,... tương ứng với các biến trong thanh ghi Tag Word được mô phỏng lại. Các biến tag dùng để chỉ trạng thái thanh ghi dữ liệu đang lưu trữ số có hợp lệ hay không. Class có các hàm hỗ trợ trong việc thao tác trên stack thanh ghi dữ liệu FPU. Hàm pop() lấy thanh ghi đỉnh của stack ra ngoài. Hàm FLD() dùng để nạp giá trị vào stack thanh ghi dữ liệu. Các hàm FDIV(), FADD(), FSUB(), FMUL() hỗ trợ trong việc thao tác toán học trên stack thanh ghi dữ liệu FPU.

### 4.1.3 Quy trình thực hiện

Với những biến môi trường được mô phỏng lại để hiện thực và kiểm tra kết quả chính xác của từng câu lệnh. Mỗi câu lệnh có những thao tác riêng trên những biến môi trường khác nhau vì vậy để đảm bảo tính chính xác khi hiện thực, trong quá trình hiện thực cần phải xây dựng các test-case để kiểm tra. Hình 4.13 thể hiện quy trình hiện thực một câu lệnh assembly.



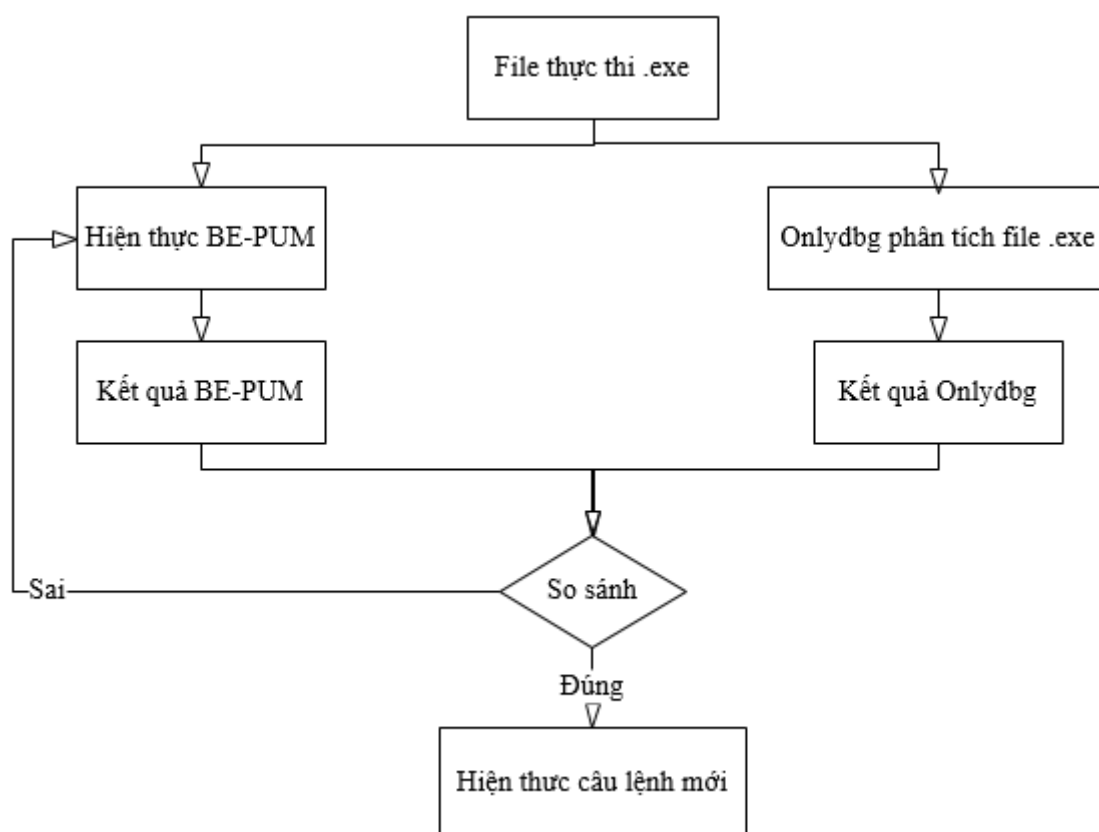
Hình 4.13: Quy trình hiện thực một câu lệnh assembly

Mỗi câu lệnh được xây dựng và build thành một file thực thực có đuôi file là .exe. Mỗi câu lệnh thao tác trên các môi trường khác nhau có thể là môi trường số nguyên hoặc môi trường số thực. Sau khi phân loại để xác định môi trường thay đổi của câu lệnh assembly



cần, bước tiếp theo xây dựng mô phỏng quá trình thực thi của câu lệnh đó. Mỗi câu lệnh có thao tác khác nhau trên môi trường thực hiện do đó kết quả cũng khác nhau. Kết quả thu được là biến môi trường sẽ bị thay đổi.

Để kiểm tra kết quả đó đúng hay sai cần sử dụng thêm công cụ Onlydbg<sup>1</sup>. Hình 4.14 thể hiện quy trình kiểm tra kết quả của một câu lệnh assembly.



Hình 4.14: So sánh kết quả với Onlydbg

Khi một file thực thi .exe tương ứng với một câu lệnh assembly là đầu vào cho công cụ Onlydbg. Kết quả Onlydbg cho biết sự thay đổi môi trường mà câu lệnh assembly đó thực hiện. Giả sử kết quả Onlydbg hoàn toàn chính xác và lấy kết quả đó làm chuẩn cho kết quả mà BE-PUM mô phỏng lại. Nếu kết quả BE-PUM trả về sai so với kết quả Onlydbg, sẽ hiện thực lại câu lệnh đó trong BE-PUM. Ngược lại, nếu đúng thì thực hiện tiếp các câu lệnh assembly khác

<sup>1</sup> Xem trong phần phụ lục

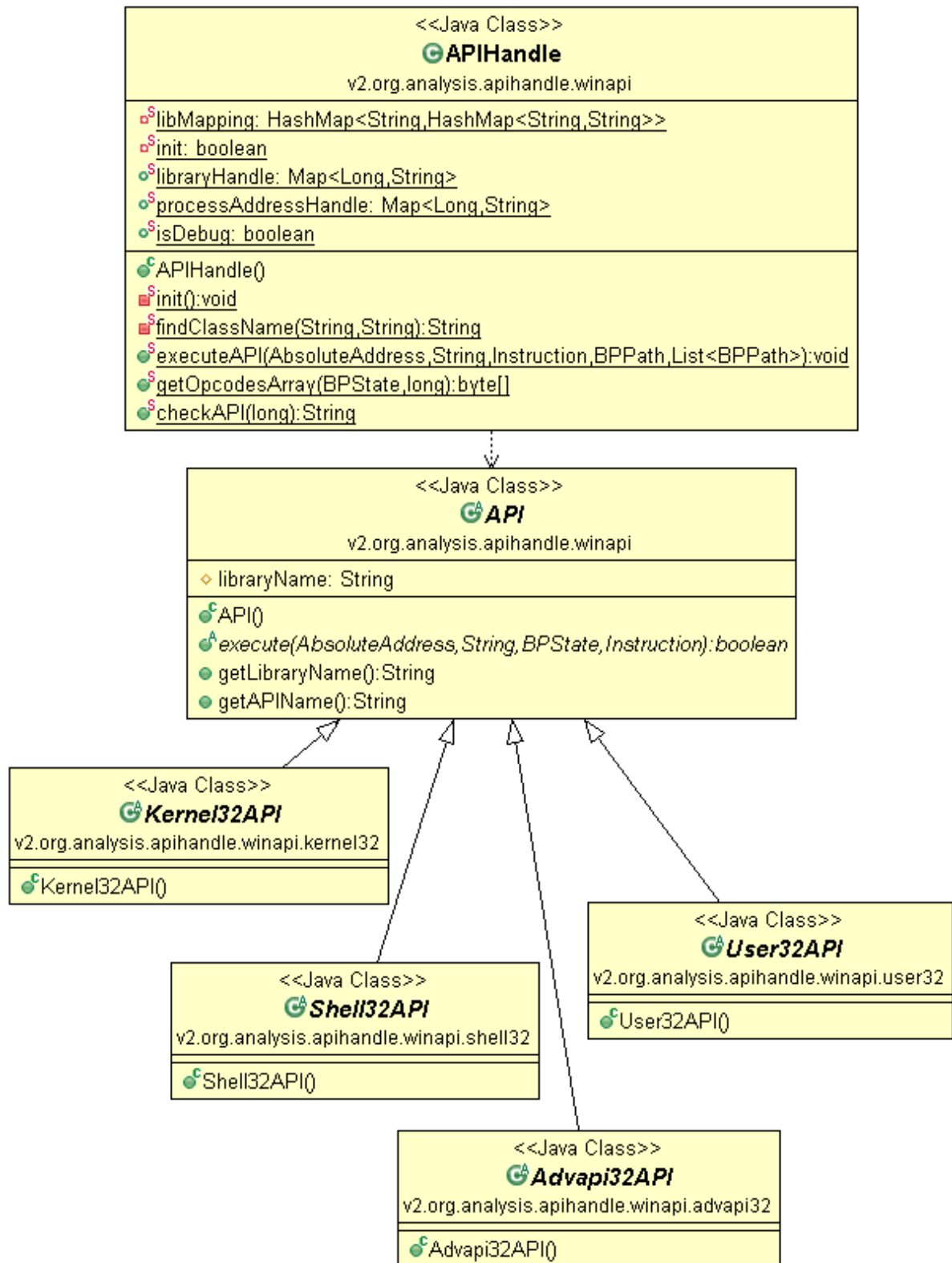
## 4.2 Windows API

### 4.2.1 Các thành phần chính của bộ xử lý Windows API

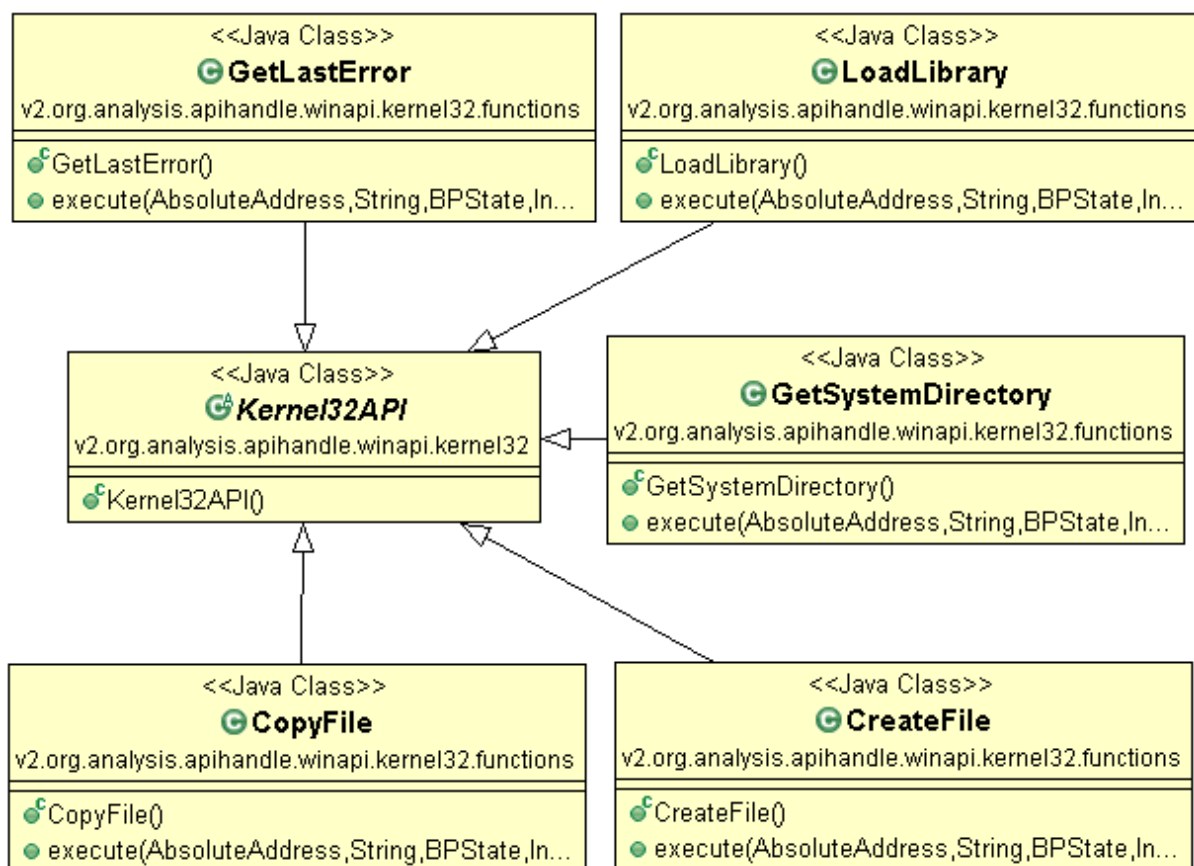
Hình 4.15 mô tả lược đồ lớp (class) chính được xây dựng cho bộ xử lý Windows API. Trong đó có một class nắm giữ toàn bộ giao tiếp và làm việc giữa hệ thống BE-PUM và các xử lý chính cho từng API đó là class *APIHandle*. Mỗi khi hệ thống BE-PUM cần xử lý một API bất kỳ nào đó, phương thức *executeAPI* của *APIHandle* sẽ được gọi để làm việc.

Kiến trúc chính của đề tài được xây dựng dựa trên dạng thức thiết kế adapter hay còn gọi là dạng thiết kế điều hợp. Với adapter là lớp trừu tượng – abstract class: lớp *API*, mỗi bộ thư viện Windows API sẽ được trừu tượng hóa thành từng abstract class mà sẽ kế thừa từ lớp *API* (ví dụ như các lớp *Kernel32API*, *User32API*,...). Rồi sau đó, mỗi Windows API sẽ được trừu tượng hóa thành một class, được đặt tên trùng với tên của chính Windows API đó, kèm theo sẽ là việc kế thừa và hiện thực đúng abstract class của bộ thư viện mà Windows API thuộc về.

Hình 4.16 ngay sau đây sẽ mô tả ngắn gọn cho việc hiện thực từng Windows API của bộ thư viện dịch vụ nền được chứa trong tập tin *kernel32.dll*.






Hình 4.15: Kiến trúc chính của bộ xử lý Windows API dành cho BE-PUM



Hình 4.16: Mô hình hiện thực một số Windows API trong *kernel32.dll*

## 4.2.2 Những khai báo ánh xạ của bộ xử lý Windows API

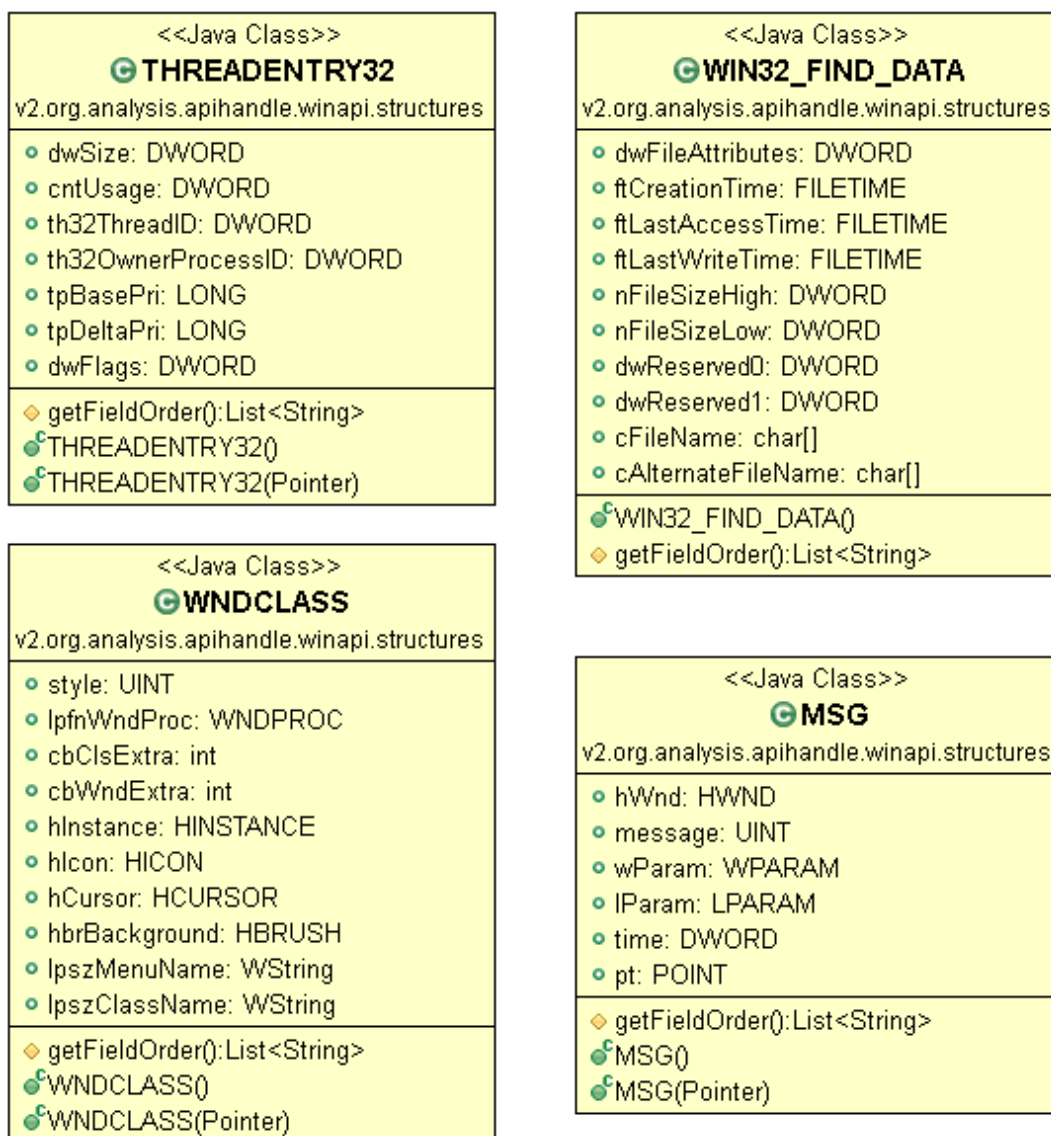
Ngoài những thành phần chính nêu trên, bộ xử lý Windows API còn có những lớp khác để nắm thông tin ánh xạ giữa hai ngôn ngữ lập trình Java và C thông qua JNA.

<<Java Interface>>	
 <b>User32DLL</b>	
v2.org.analysis.apihandle.winapi.user32	
 INSTANCE: User32DLL	
 SYNC INSTANCE: User32DLL	
<ul style="list-style-type: none"> <li>● PostMessage(HWND,int,WPARAM,LPARAM): BOOL</li> <li>● SendMessage(HWND,int,WPARAM,LPARAM): LRESULT</li> <li>● MessageBox(HWND,String,String,UINT): int</li> <li>● CharLower(char[]): Pointer</li> <li>● CharNext(char[]): long</li> <li>● CharPrev(WString,WString): WString</li> <li>● CharUpperBuff(char[],DWORD): DWORD</li> <li>● CreateDialogParamW(HINSTANCE,String(HWND,int,LPARAM): HWND</li> <li>● DestroyMenu(HMENU): BOOL</li> <li>● GetClassInfoEx(HINSTANCE,WString,WNDCLASSEX): BOOL</li> <li>● GetFocus(): HWND</li> <li>● GetKeyboardType(int): int</li> <li>● GetMessage(MSG,HWND,UINT,UINT): BOOL</li> <li>● GetSysColor(int): DWORD</li> <li>● GetSysColorBrush(int): HBRUSH</li> <li>● GetSystemMenu(HWND,BOOL): HMENU</li> <li>● GetWindowThreadProcessId(HWND,DWORDByReference): DWORD</li> <li>● InsertMenuItem(HMENU,UINT,BOOL,MENUITEMINFO): BOOL</li> <li>● IsDialogMessage(HWND,MSG): BOOL</li> <li>● LoadBitmap(HINSTANCE,WString): HBITMAP</li> <li>● LoadCursor(HINSTANCE,WString): HCURSOR</li> <li>● LoadIcon(HINSTANCE,WString): HICON</li> <li>● LoadMenu(HINSTANCE,WString): HMENU</li> <li>● LoadString(HINSTANCE,UINT,char[],int): int</li> <li>● OpenClipboard(HWND): BOOL</li> <li>● RegisterClass(WNDCLASS): ATOM</li> <li>● SetCursor(HCURSOR): HCURSOR</li> <li>● ShowWindow(HWND,int): BOOL</li> <li>● TranslateMessage(MSG): BOOL</li> <li>● UnregisterClass(String,HINSTANCE): BOOL</li> <li>● GetTopWindow(HWND): HWND</li> <li>● BlockInput(BOOL): BOOL</li> <li>● GetWindowLong(HWND,int): LONG</li> <li>● SetWindowLong(HWND,int,LONG): LONG</li> </ul>	

Hình 4.17: Lớp chứa thông tin ánh xạ những Windows API trong *user32.dll*

Trong Hình 4.17 là một mô tả về ánh xạ tên hàm và kiểu dữ liệu đầu vào của bộ thư viện giao diện người dùng được chứa trong tập tin *user32.dll*. Ngoài ra, bộ thư viện JNA có khai báo ánh xạ sẵn một số hàm Windows API cùng những kiểu dữ liệu chính mà Windows API thường dùng, nhờ đó mà những kiểu dữ liệu thông dụng như *HWND*, *HINSTANCE*, *LPARAM*, *DWORD*,... không đòi hỏi lập trình viên phải khai báo và ánh xạ lại.

Nhưng không phải tất cả kiểu dữ liệu đều có sẵn, đặc biệt là những kiểu dữ liệu cấu trúc, cần có những trường hợp phải khai báo thêm để ánh xạ và phục vụ cho đề tài này.

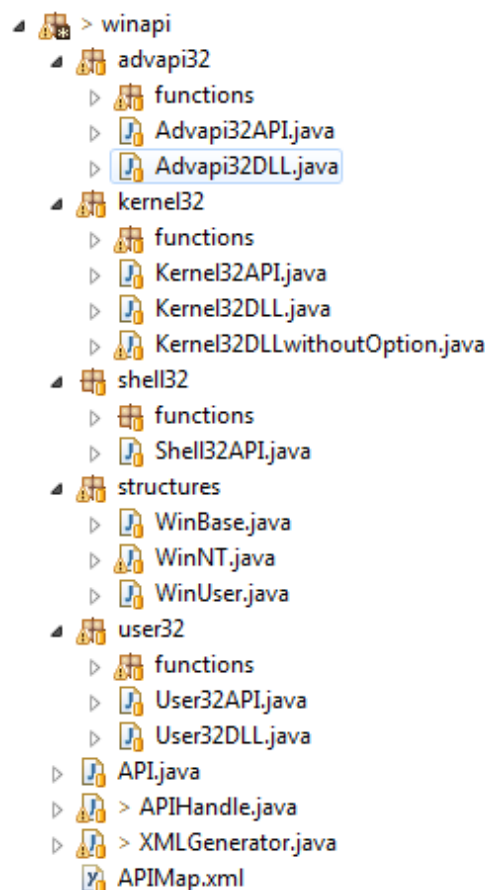


Hình 4.18: Một số dữ liệu kiểu cấu trúc được xây dựng thêm

Nội dung của Hình 4.18 thể hiện khai báo thêm của một số dữ liệu kiểu cấu trúc dành cho đề tài này. Như đã nói ở Mục 3.3.4, việc xây dựng này đòi hỏi phải chính xác về kích thước và trình tự các biến được khai báo bên trong struct, để dữ liệu được nạp ra vào chính xác.

### 4.2.3 Kiến trúc xây dựng và làm việc

Từ những thành phần đã trình bày ở Mục 4.2.1 và Mục 4.2.2, kiến trúc của bộ xử lý Windows API sẽ được tổ chức lại cho thống nhất và dễ dàng mở rộng về sau:



Hình 4.19: Cây cấu trúc mô tả những thành phần trong bộ xử lý Windows API

Gói ngoài cùng của bộ xử lý (*winapi*) sẽ nắm:

- Khai báo lớp trừu tượng *API*: đặc tả phương thức thực thi dùng chung cho mọi Windows API;

- Lớp *APIHandle*: giao tiếp chính giữa bộ xử lý Windows API và hệ thống BE-PUM;
- Những gói được đặt tên ứng với từng tên tập tin (không bao gồm phần mở rộng) mà bộ thư viện đó thuộc về. Mỗi gói đó sẽ chứa những thành phần như sau:
  - Lớp khai báo trừu tượng cho từng Windows API của bộ thư viện đó (ví dụ như lớp *Kernel32API*), trong đó sẽ nạp vào tên của bộ thư viện hiện tại;
  - Lớp khai báo ánh xạ thư viện và tên hàm mà hiện bộ thư viện JNA chưa có sẵn (chỉ tồn tại trong trường hợp những Windows API mà đề tài hiện thực chưa có trong bộ thư viện JNA)
  - Một gói được đặt tên *functions*: dùng để chứa mọi lớp xử lý cho từng Windows API của bộ thư viện hiện tại (ví dụ như hiện tại gói bên ngoài là *kernel32*, những hàm Windows API được xây dựng bên trong sẽ là: *CreateFile*, *CopyFile*, *GetLastError*...).
- Gói *structures*: chứa những lớp khai báo ánh xạ dữ liệu kiểu cấu trúc mà bộ thư viện JNA chưa có sẵn.
- Lớp *XMLGenerator*: có chức năng sinh ra tự động tập tin cấu trúc XML từ tổ chức như trên của bộ xử lý Windows API, mô tả ánh xạ giữa tên của hàm Windows API và tên đầy đủ của lớp sẽ đảm nhận việc xử lý hàm Windows API đó, kèm theo là nó thuộc tập tin thư viện nào. Ví dụ như sau:

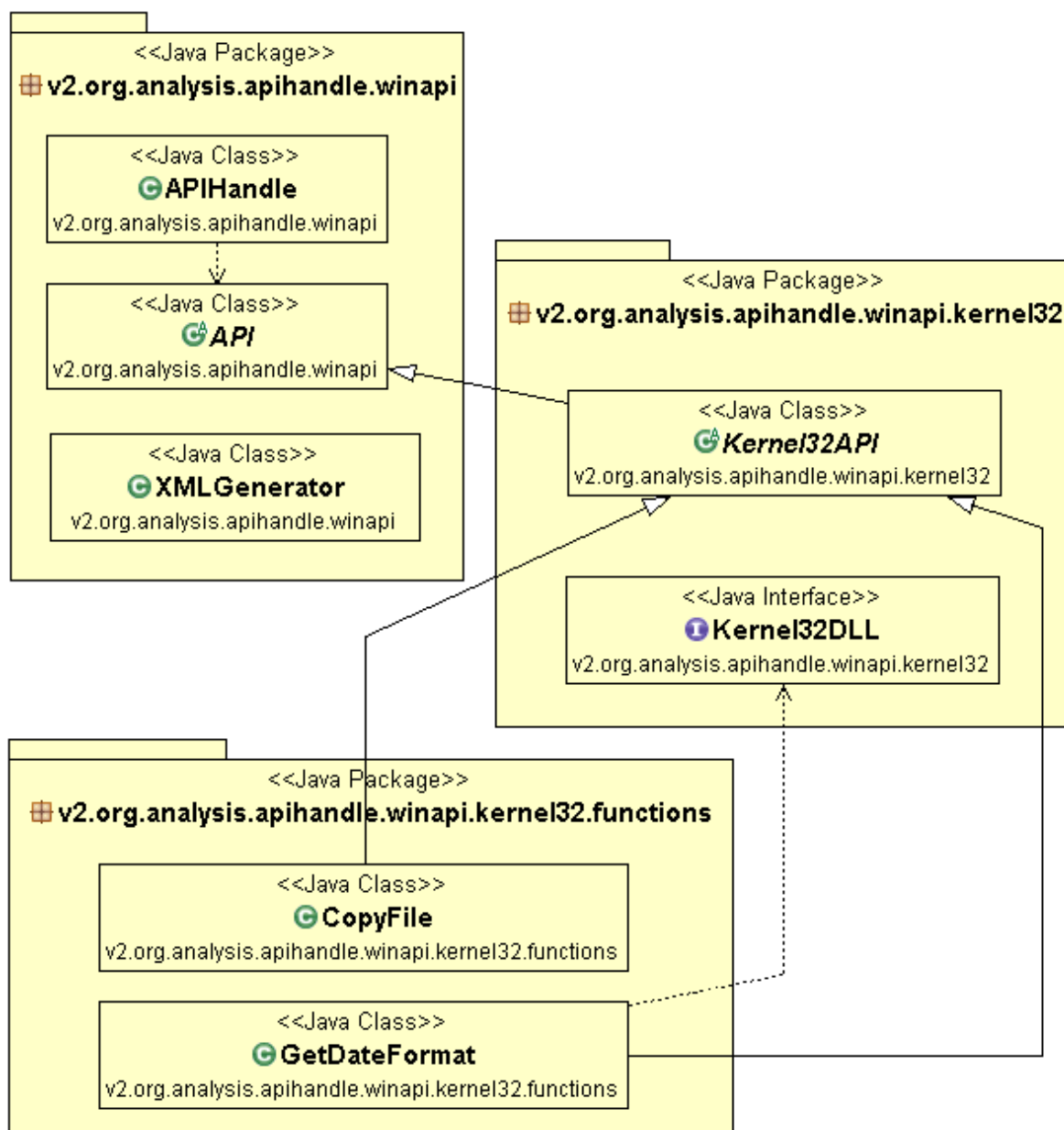
```
<APIMap>
  <DLL name="advapi32">
    <API funcName="cryptdecrypt"
className="advapi32.functions.CryptDecrypt"/>
  </DLL>
  <DLL name="kernel32">
    <API funcName="closehandle"
className="winapi.kernel32.functions.CloseHandle"/>
  </DLL>
```



```
</APIMap>
```

- Tập tin *APIMap.xml*: là nội dung do lớp *XMLGenerator* sinh ra.

Nội dung của tập tin *APIMap.xml* được sinh ra dùng để làm chỉ mục cho lớp trung gian *APIHandle* biết được rằng: với thông tin đầu vào là tên của Windows API mà hệ thống BE-PUM yêu cầu, thì lớp nào sẽ xử lý được yêu cầu đó. Sau khi có được tên đầy đủ của lớp xử lý, *APIHandle* sẽ khởi tạo lớp đó với kiểu đối tượng là *API* (do tất cả lớp xử lý đều hiện thực trên lớp này), rồi sau đó gọi phương thức *execute* để đối tượng này xử lý toàn bộ.



Hình 4.20: Giải đồ thể hiện mối liên hệ giữa các thành phần

Với việc tổ chức theo cấu trúc này và sinh ra tập tin đánh chỉ mục thì sẽ có được những ưu điểm sau đây:

- Dễ dàng quản lý mã nguồn: do Windows API có rất rất nhiều hàm, việc tạo ra mỗi lớp độc lập để xử lý cho từng hàm Windows API sẽ giúp cho mã nguồn được rõ ràng và độc lập cũng như việc sửa chữa sau này được tiện lợi;
- Dễ dàng bổ sung: nếu muốn hiện thực thêm một Windows API bất kỳ, ta chỉ việc tạo ra một lớp theo đúng quy định cấu trúc như trên, lớp XMLGenerator

sẽ giúp sinh ra nội dung của tập tin `APIMap.xml` và chỉ đơn giản như vậy là đã tích hợp được một Windows API mới.

Nội dung của Hình 4.20 trình bày ví dụ về mối liên hệ mà cấu trúc vừa nêu. Khi hệ thống BE-PUM yêu cầu lớp `APIHandle` xử lý hàm `CopyFile`, nó sẽ dùng `APIMap.xml` để tìm được tên đầy đủ của lớp xử lý được hàm này. Tên ấy được dùng để khai báo khởi tạo động một đối tượng bất kỳ và sau đó ép kiểu về kiểu đối tượng API. Do lớp xử lý được hàm này được hiện thực dựa trên lớp trừu tượng API. Còn với trường hợp nếu là hàm `GetDateFormat`, thì trong lớp này còn có sử dụng lớp `Kernel32DLL` do bộ thư viện JNA chưa khai báo sẵn hàm này.

### 4.2.4 Quản lý môi trường tương tác vật lý

Do một số Windows API có khả năng làm việc với hệ thống tập tin lưu trữ, dẫn đến khả năng làm ảnh hưởng xấu đến hệ thống do chúng ta đang tập trung vào việc phân tích những phần mềm độc hại cho máy vi tính. Vì vậy ta cần quản lý và ngăn chặn việc đó.

Để hiện thực việc đó, một lớp mang tên *Storage* được viết ra để ánh xạ toàn bộ địa chỉ của hệ thống thật vào một vùng lưu trữ đã được quy định. Và rồi mỗi khi một Windows API nào có sử dụng đường dẫn để làm việc, thì cần phải kiểm tra rằng liệu Windows API đó có khả năng ảnh hưởng đến hệ thống hay không, nếu có ta cần ánh xạ sang vùng lưu trữ đã được quản lý để tránh những nguy hại.

## Chương 5

# Kết Quả

### 5.1 Thí nghiệm và đánh giá kết quả

Example	Size	JakStab			IDA Pro			BE-PUM		
	KB	Nodes	Edges	Time	Nodes	Edges	Time	Nodes	Edges	Time
Virus.Artelad.2173	23	134	154	10	159	162	1133	1610	1611	236468
Email-Worm.LoveLetter.b	60	1027	1026	297	984	1011	10	7558	7602	1073984
Virus.Pulkfer.a	129	907	924	10	805	823	20	8347	8353	44672
Email-Worm.Klez.h	137	192	178	20	50	56	1	5652	5651	46344
Email-Worm.Coronex.a	12	26	27	500	148	157	204	308	339	1000
Trojan-PSW.QQRob.16.d	25	89	100	766	17	15	382	91	105	953
Virus.Aztec	8	104	111	1973	223	215	495	300	313	44384
Virus.Belial.a	4	41	42	407	118	116	198	128	134	985
Virus.Benny.3219.a	8	138	153	890	599	603	415	149	164	2438
Virus.Benny.3223	12	42	47	328	770	781	135	149	164	2218
Virus.Bogus.4096	38	87	98	546	88	86	269	88	98	656
Virus.Brof.a	8	17	17	343	98	102	167	137	147	1484
Virus.Cerebrus.1482	8	6	5	156	164	165	70	179	198	735
Virus.Compan.a	8	25	26	360	83	81	176	91	98	484
Virus.Cornad	4	21	20	141	68	72	67	94	100	344
Virus.Eva.a	8	14	13	329	381	392	145	249	277	13438
Virus.Htrip.a	8	10	10	359	145	143	172	148	157	2187
Virus.Htrip.d	8	10	10	265	164	162	124	165	173	2296
Virus.Seppuku.1606	8	131	136	1968	381	390	965	339	364	8372
Virus.Wit.a	4	54	60	360	153	151	172	185	203	2641

Email-Worm.Bagle.af	21	123	143	937	142	151	461	140	166	2157
Email-Worm.Bagle.ag	17	127	147	828	13	12	413	127	147	1047
Virus.Cabanas.a	8	3	2	156	1	1	78	68	72	1532
Virus.Cabanas.b	8	3	2	140	9	7	70	63	66	1781
Virus.Canabas.2999	8	2	1	656	7	6	85	358	401	8703
Virus.Seppuku.1638	8	139	144	2266	414	412	112	689	712	13000
Virus.Seppuku.3291	8	26	25	187	556	554	66	253	270	12156
Virus.Seppuku.3426	8	27	27	188	30	28	61	299	317	13484
<i>non-malware binary</i>										
hostname.exe	8	329	360	2412	343	389	33	326	357	235610
winver.exe	6	162	166	422	310	345	24	232	240	122484
systray.exe	4	110	136	532	115	138	14	123	139	16125
regedt32.exe	3	52	54	266	56	61	11	61	69	22844
actmovie.exe	4	164	179	281	187	215	51	180	196	243469
nddeapir.exe	4	164	179	500	187	215	24	180	196	223297

Bảng 5.1: Một số kết quả của thí nghiệm sinh mô hình

## 5.2 Các câu lệnh hợp ngữ đã được hỗ trợ

Trong quá trình mô phỏng các câu lệnh assembly có sự chọn lọc và ưu tiên các câu lệnh được sử dụng phổ biến trước. Với rất nhiều câu lệnh assembly trong ngôn ngữ cấp thấp được phân ra thành hai nhóm dựa trên thao tác xử lý của câu lệnh bao gồm bộ xử lý số nguyên và bộ xử lý số thực.

Tính tới thời điểm của bài báo cáo này, đã có khoảng 200 câu lệnh xử lý số nguyên và khoảng 50 câu lệnh xử lý số thực được mô phỏng trong BE-BUM. Bảng 5.2 thống kê danh sách câu lệnh assembly đã mô phỏng được.

Bộ xử lý		Câu lệnh assembly
Bộ xử lý số nguyên	Lệnh toán học	ADD, AND, SUB, OR, XOR, IMUL, ROR, DIV, SBB, CLC, NOT, IDIV, XADD, ADC, DEC, SHR, SAR, SHL, SAL, DEC, INC, SHR, ROR, REP, MUL, SAR, RCR, ROL, RCL, SBB
	Lệnh gọi	CALL

Lệnh điều kiện nhảy	JE, JNZ, JC, JC, JLE, JGE, JGE, JLNLE, JS, JNO, JMP, LOOP, JZ, JB, JNB, JNG, JA, JNL, JO, JNS, JNO, JPE, LOOPE, LOOPZ, JNE, JNAE, JAEM JNAEM JL, JNBE, JG, LOOP, JP, JECXZM LOOPNE, LOOPNZ
Lệnh nhảy	JUMP
Lệnh sao chép	MOV, XCHG, MOVZ, MOVSB, MOVSW, MOSX, MOVZB, MOVZW, CMOVA, CMOVB, CMOVBE, CMOVC, CMOVE, CMOVP, CMOVPE, CMOVG, CMOVL, CMOVNA, CMOVNAE, CMOVNBE, CMOVNE, CMOVNG, CMOVNGE, CMOVNL, CMOVNLE, CMOVNO, CMOVNS, CMOVNZ, CMOVPO, CMOVZ
Lệnh trả giá trị	RET
Lệnh điều khiển	CMP, INT, AAA, AAD, AAM, ASS, BSF, BSWAP, BT, BTC, BRT, BTS, CBW, CDQ, CLC, CLD, CLI, CLTD, CMC, OUT, POP, POPA, POPF, PUSH, PUSHA, PUSHF, RDTSC, SAHF, SCAS, SETA, SETAE, SETB, SETBE, SETC, SETE, SETG, SETGE, SETL, SETNA, SETNAE, SETNB, SETNBE, SETNC, SETNE, SETNG, SETNGE, SETNL, SETNLE, SETNO, SETNP, SETNS, SETO, SETP, SETPE, SETPO, SETS, SETZ, LODS, MOVS, NEG, NOP, SHLD, SHRD, STC, STOS, TEST, XLAT, CBW, CWDE, CMPS, CMPXCHG, CMPXCHG8B, CPUID, CWD, CWDE, CWT, DAA, DAS, ENTER, IN INT1, INT3, LAHF, LEA, LEAVE

Bộ xử lý số thực	Lệnh toán học	FADD, FADDP, FIADD, FSUB, FSUBP, FSUBR, FSUBRP, FISUB, FISUBR, FMUL, FMULP, FIMUL, FDIV, FDIVP, FIDIV, FDIVR, FIDIVR, FABS, FCHS, FSQRT, FPREM, FPREM1, FRNDINT, FEXTRACT, FSIN, FCOS, FSINCOS, FPTAN, FPATAN, FYL2X, FYL2XP1, F2XM1, FSCALE
	Lệnh so sánh	FCOM, FCOMP, FCOMPP, FUCOM, FUCOMP, FUCOMPP, FICOM, FICOMP, FCONI, FCOMIP, FUCOMI, FUCOMIP, FTST, FXAM
	Lệnh nạp giá trị	FLDZ, FLD1, FLDPI, FLDL2T, FLDL2E, FLDLG2, FLDLN2
	Lệnh sao lưu	FLD, FST, FSTP, FXCH, FCMOVcc, FCMOVB, FCMOVNB, FCMOVE, FCMOVNE, FCMOVBE, FCMOVNBE, FCMOVU, FCMOVNU.
	Lệnh điều khiển	FINIT, FNINIT, FLDCW, FSTCW, FNSTCW, FSTSW, FNSTSW, FCLEX, FNCLEX, FLDENV, FNSTENV, FRSTOR, FSAVE, FNSAVE, FINCSTP

Bảng 5.2: Thống kê các câu lệnh assembly đã được hiện thực

### 5.3 Các Windows API đã được hỗ trợ

Quá trình xây dựng có sự chọn lọc và ưu tiên cho các Windows API được dùng phổ biến trước tiên. Với rất nhiều Windows API của dịch vụ nền được các phần mềm độc hại cho máy tính sử dụng (được chứa bên trong tập tin kernel32.dll), đây là bộ thư viện đang được ưu tiên hỗ trợ hàng đầu cho hệ thống BE-PUM. Kế tiếp đó là các bộ thư viện giao diện người dùng (được chứa bên trong tập tin user32.dll), dịch vụ nâng cao (được chứa bên trong tập tin advapi32.dll) và cuối cùng là Windows Shell (được chứa bên trong tập

tin shell32.dll).

Tính đến thời điểm của bài báo cáo này, đã có tất cả khoảng 400 Windows API được xây dựng và hỗ trợ cho hệ thống BE-PUM.

Tên tập tin thư viện	Các Windows API đã được hiện thực
advapi32.dll	CryptAcquireContext, CryptCreateHash, CryptDecrypt, CryptDeriveKey, CryptDestroyHash, CryptDestroyKey, CryptHashData, CryptReleaseContext, IsValidAcl, OpenProcessToken, OpenThreadToken, RegCloseKey, RegCreateKeyEx, RegOpenKey, RegOpenKeyEx, RegQueryValueEx, RegSetValueEx
comctl32.dll	InitCommonControls
gdi32.dll	AddFontResource, AnimatePalette, CreateCompatibleDC, CreatePalette, DeleteObject, DPtoLP, GdiFlush, GdiGetBatchLimit, GetBkColor, GetDeviceCaps, GetStockObject, SetBkColor, SetTextAlign, StrokeAndFillPath, StrokePath



	<p>AreFileApisANSI, CloseHandle, CompareFileTime, CompareString, CopyFile, CreateDirectory, CreateEvent, CreateFile, CreateFileMapping, CreateMutex, CreateProcess, CreateRemoteThread, CreateThread, CreateToolhelp32Snapshot, DecodePointer, DeleteCriticalSection, DeleteFile, DeviceIoControl, DisableThreadLibraryCalls, DuplicateHandle, EncodePointer, EnterCriticalSection, EnumDateFormats, ExitProcess, ExitThread, FileTimeToSystemTime, FindAtom, FindClose, FindFirstFile, FindNextFile, FindResource, FlsAlloc, FlsGetValue, FlsSetValue, FlushInstructionCache, FormatMessage, FreeEnvironmentStrings, FreeLibrary, GetAclInformation, GetACP, GetAtomName, GetCommandLine, GetComputerName, GetConsoleCP, GetConsoleMode, GetCPInfo, GetCPInfoEx, GetCurrentDirectory, GetCurrentProcess, GetCurrentProcessId, GetCurrentThread, GetCurrentThreadId, GetDateFormat, GetDiskFreeSpace, GetDiskFreeSpaceEx, GetDriveType, GetEnvironmentStrings, GetEnvironmentVariable, GetExitCodeProcess, GetFileAttributes, GetFileSize, GetFileTime, GetFileType, GetFullPathName, GetLargestConsoleWindowSize, GetLastError, GetLocaleInfo, GetLocalTime, GetLogicalDrives, GetLogicalDriveStrings, GetModuleFileName, GetModuleHandle, GetModuleHandleEx, GetNativeSystemInfo, GetOEMCP, GetPriorityClass, GetProcAddress, GetProcessHeap, GetProcessId, GetProcessVersion, GetShortPathName, GetStartupInfo, GetStdHandle, GetStringTypeA, GetStringTypeW, GetSystemDefaultLCID, GetSystemDefaultUILanguage, GetSystemDirectory, GetSystemInfo,</p>
	<p>GetSystemTime,<sup>123</sup> GetSystemTimeAsFileTime, GetTempFileName, GetTempPath, GetThreadLocale, GetThreadPriority, GetTickCount, GetUserDefaultLangID,</p>

mscoree.dll	_CorExeMain
msvcrt.dll	atexit, calloc, fclose, fopen, fprintf, fread, free, fwrite, isalpha, is_wctype, malloc, memcpy, memmove, memset, remove, srand, strncat, strncmp, strrchr, strstr, time, _cexit, _chkesp, _controlfp, _except_handler3, _fpreset, _initterm, _itoa, _itow, _ltoa, _mbsrchr, _setmode, _strcmpi, __getmainargs, __p__commode, __p__environ, __p__fmode, __p____initenv, __set_app_type
ntdll.dll	NtCreateSection, NtProtectVirtualMemory, NtQueryInformationProcess, NtWriteVirtualMemory
ole32.dll	CoCreateGuid, CoFileTimeNow, OleInitialize, OleUninitialize
shell32.dll	ShellAbout, ShellExecute, SHGetFileInfo, SHGetSpecialFolderPath, StrChrI
shlwapi.dll	PathFindFileName, PathUndecorate

user32.dll	AnyPopup, BlockInput, CharLower, CharNext, CharPrev, CharUpperBuff, ClientToScreen, CloseWindow, CreateCaret, CreateDialogParam, CreateWindowEx, DefFrameProc, DestroyCaret, DestroyMenu, DestroyWindow, DialogBoxIndirectParam, DialogBoxParam, DispatchMessage, EqualRect, FindWindow, GetActiveWindow, GetAsyncKeyState, GetCaretPos, GetClassInfoEx, GetClassName, GetCursorPos, GetDC, GetDCEX, GetFocus, GetForegroundWindow, GetGUIThreadInfo, GetKeyboardState, GetKeyboardType, GetLastActivePopup, GetMessage, GetParent, GetProcessWindowStation, GetSysColor, GetSysColorBrush, GetSystemMenu, GetSystemMetrics, GetTitleBarInfo, GetTopWindow, GetUserObjectInformation, GetWindowLong, GetWindowRect, GetWindowText, GetWindowTextLength, GetWindowThreadProcessId, InflateRect, InsertMenuItem, IsChild, IsDialogMessage, IsIconic, IsWindowUnicode, IsWindowVisible, IsZoomed, LoadBitmap, LoadCursor, LoadIcon, LoadMenu, LoadString, MessageBox, OffsetRect, OpenClipboard, PeekMessage, PostMessage, RegisterClass, RegisterClassEx, RemoveProp, ScrollDC, SendMessage, SendNotifyMessage, SetCaretBlinkTime, SetCaretPos, SetClassLong, SetCursor, SetFocus, SetForegroundWindow, SetLastErrorEx, SetWindowLong, ShowCaret, ShowWindow, TranslateMessage, UnregisterClass, UpdateLayeredWindow, UpdateWindow
wininet.dll	InternetSetOption

winspool.dll	EnumPrinters
ws2_32.dll	WSACleanup, WSAStartup

Bảng 5.3: Thống kê các Windows API đã được xây dựng

## Chương 6

# Hướng Phát Triển Trong Tương Lai

### 6.1 Tăng số lượng các câu lệnh hợp ngữ được hỗ trợ

Ở thời điểm hiện tại, số lượng các câu lệnh assembly đã được mô phỏng trong BE-PUM vẫn còn khiêm tốn. Bởi thực tế vẫn còn khá nhiều câu lệnh assembly vẫn chưa được mô phỏng. Theo một thống kê, số lượng assembly được sử dụng khoảng 386 câu lệnh được sử dụng trong hệ điều hành Windows và có xu hướng mở rộng trên các nền tảng khác.

Điều đó cho thấy vẫn còn khá nhiều câu lệnh assembly chưa được mô phỏng trong BE-PUM. Ngoài ra còn những câu lệnh assembly ít được sử dụng. Trong quá trình chạy thí nghiệm, việc sinh đỉnh biểu diễn địa chỉ câu lệnh và nội dung câu lệnh, BE-PUM vẫn chưa thể xử lý bao quát hết toàn bộ các câu lệnh assembly mà bộ malware yêu cầu.

Mỗi khi chạy các malware từ bộ thí nghiệm, BE-PUM ghi nhận được thêm các câu lệnh assembly chưa được mô phỏng. Số lượng câu lệnh assembly chưa được hỗ trợ ngày càng tăng lên theo số lượng malware. Do đó việc mô phỏng các câu lệnh assembly vẫn phải được cập nhật và hiện thực để cung cấp thêm sức mạnh cho BE-PUM phân tích.

## 6.2 Tăng số lượng các Windows API được hỗ trợ

Ở thời điểm hiện tại, số lượng các Windows API đã được hỗ trợ cho BE-PUM vẫn còn rất ít ỏi. Bởi thực tế, con số Windows API mà hệ điều hành Windows đang cung cấp vượt xa con số đó. Theo một thống kê nhỏ từ những bộ thư viện thường dùng của Windows, số lượng Windows API hiện đang được cung cấp lên đến khoảng 4000 hàm.

Điều dễ mở ra cho việc còn rất nhiều Windows API cần được hiện thực thêm cho BE-PUM. Chưa nói tới những hàm Windows API được ít người biết và dùng đến; trong quá trình chạy thí nghiệm, phân tích tự động một số lượng lớn các malware được tập hợp từ những phòng nghiên cứu trên thế giới, BE-PUM vẫn chưa thể xử lý bao quát hết mọi lời gọi hàm Windows API mà bộ malware đó yêu cầu.

Vì thế, mỗi khi chạy hàng loạt malware từ bộ thí nghiệm, chúng tôi lại ghi nhận được thêm một danh sách dài các Windows API cần được hỗ trợ ngay cho BE-PUM. Danh sách đó được cập nhật thường xuyên và vẫn đã, đang luôn được giải quyết qua từng ngày để cung cấp thêm sức mạnh cho BE-PUM.

## 6.3 Hiện thực hóa việc tự động sinh mã cho công tác hỗ trợ Windows API

Với thiết kế và xây dựng một cách minh bạch và dễ dàng mở rộng như đã trình bày ở Mục 4.2, đi kèm với số lượng Windows API đang cần được hỗ trợ còn quá lớn; điều đó dẫn đến công việc cần thiết cho tương lai, đó là bộ sinh mã tự động cho bộ xử lý Windows API trong BE-PUM.

Trong quá trình 40 ngày thực tập tại đại học JAIST (Japan Advanced Institute of Science and Technology) Nhật Bản. Dưới sự hướng dẫn của giáo sư Mizuhito Ogawa, cùng với sự hợp tác của nghiên cứu sinh Lê Vinh đã giúp lên ý tưởng và hiện thực bước đầu cho dự án sinh mã tự động này. Dự án sẽ nhận đầu vào là thông tin tên Windows API

cần hiện thực, sau đó sẽ tự động tìm kiếm tài liệu về Windows API đó dưới dạng ngôn ngữ tự nhiên, trích xuất thông tin cần thiết và sinh ra bộ mã theo yêu cầu.

Hiện tại dự án vẫn chỉ khởi động bước đầu và cần thêm sự hợp tác lâu dài để có thể biến mọi thứ thành hiện thực. Và đây là một trong những bước đi đầy thách thức và quan trọng trong thời gian sắp tới cho sự phát triển kế tiếp của BE-PUM.

# PHỤ LỤC 1

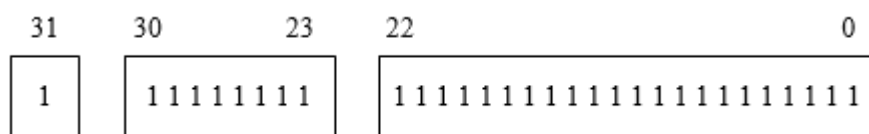
## *Biểu diễn số thực dạng nhị phân*

Có bốn định dạng chuẩn IEEE biểu diễn số dấu chấm động ở dạng nhị phân được sử dụng trong bộ xử lý Intel:

Kiểu	Bit dấu (sign)	Phần mũ (exponent)	Phần định trị (mantissa)	Tổng số bit	Số cơ sở
Half-real	1	5	10	16	15
Short-real	1	8	23	32	127
Double-real	1	11	52	64	1023
Extended-real	1	15	112	128	16383

Bảng 6.1: Các kiểu định dạng chuẩn IEEE

Cả hai định dạng đều dùng một phương thức toán để biểu diễn số dấu chấm động ở dạng nhị phân vì vậy lấy định dạng IEEE short real 32-bits làm ví dụ. Các bit ở định dạng short real được hình 6.1 với bit cao nhất nằm bên trái.



Hình 6.1: Biểu diễn số thực 32 bits

### Bit dấu

Bit dấu của số dấu chấm động được biểu diễn bằng một bit. Nếu bit dấu bằng 1 số được biểu diễn là số âm, nếu là 0 số được biểu diễn là số dương.



## Phần Mantissa

Phần mantissa rất hữu ích cho việc biểu diễn số thực dấu chấm động. Lấy ví dụ  $-3.154 \times 10^5$ , bit sign là số âm (1), phần mantissa là 3.154 và expoment là 5. Phần thập phân của mantissa là tổng mỗi chữ số nhân với cơ số 10 số mũ của 10 theo vị trí lần lượt từ trái qua phải, đằng sau dấu phẩy:

$$.154 = \frac{1}{10} + \frac{5}{100} + \frac{4}{1000}$$

Số thực dấu chấm động ở dạng nhị phân rất đơn giản. Ví dụ, trong số  $+11.1011 \times 2^3$ , bit sign là số dương (0), phần mantissa 11.1011, và expoment là 3. Phần thập phân của mantissa là tổng mỗi chữ số nhân với cơ số 2 số mũ của 2 lần lượt là vị trí của từng chữ số sau dấu phẩy:

$$.1011 = \frac{1}{2} + \frac{0}{4} + \frac{1}{16}$$

Có thể tính phần thập phân của mantissa 1011 bằng cách chia cho  $2^4$ . Giá trị có được là  $11/16 = 0.6875$ . Kết hợp với phần nguyên của ví dụ là 11. tương ứng là 3, cơ số 10 của ví dụ này là 3.6875. Dưới đây là một số ví dụ khác:

Nhị phân dấu chấm động	Cơ số 10 của phần thập phân	Cơ số 10
11.11	$3 \frac{3}{4}$	3.75
0.000000000000000000000001	$\frac{1}{8388608}$	0.000001192092895578125

Bảng 6.3 trình bày một số ví dụ đơn giản cách biểu diễn phần thập phân của số thực dấu chấm động, giá trị cơ số 10 của mỗi giá trị:

Nhị phân	Phần thập phân	Giá trị cơ số 10
.1	$\frac{1}{2}$	.5
.01	$\frac{1}{4}$	.25
.001	$\frac{1}{8}$	.125
.0001	$\frac{1}{16}$	.0625
.00001	$\frac{1}{32}$	.03125

Bảng 6.3: Ví dụ biểu diễn số dấu chấm động

## Phần Expoment

Phần expoment theo định dạng IEEE short real lưu trữ 8-bit là một số nguyên không dấu có số cơ sở là 127. Lấy lại ví dụ  $1.101 \times 2^5$  có expoment là 5, số 5 này được cộng với số cơ sở 127 được tổng là 132 biểu diễn ở dạng nhị phân số nguyên là 1000100. Bảng 6.4 đưa ra một số ví dụ về điều chỉnh phần expoment và được biểu diễn sang số nhị phân:

Expoment	Điều chỉnh	Số nhị phân
+10	137	10001001
0	127	01111111
-5	122	01111010
+128	255	11111111
-127	0	00000000

Bảng 6.4: Ví dụ về expoment

Giá trị expoment sau khi điều chỉnh không được âm. Giá trị nhất của số expoment là 128 khi được cộng thêm số mũ cơ sở (127) là 255. Đây là số lớn nhất mà 8-bit có thể biểu diễn được. Phạm vi của số mũ trong định dạng short real là  $1.0 \times 2^{-127}$  tới  $1.0 \times 2^{+128}$

## Biểu diễn Mantissa về định dạng chuẩn

Trước khi số thực dấu chấm động được mã hóa nhị phân lưu chính xác giá trị của số đó, phần mantissa cần phải đưa về dạng chuẩn. Việc xử lý này dựa trên thao tác toán học số thực dấu chấm động. Ví dụ, để biểu diễn số 1234.567 sang dạng chuẩn là  $1.234567 \times 10^3$  bằng cách dịch chuyển dấu phẩy sang trái đồng thời nhân với cơ số 10 số mũ tăng lên theo vị trí mỗi số dịch qua. Số mũ tăng lên khi được dịch qua trái, và giảm xuống khi dịch qua phải.

Tương tự như vậy khi biểu diễn số thực sang số nhị phân. Lấy ví dụ 1101.101 đưa về dạng chuẩn sẽ là  $1.101101 \times 2^3$  bằng cách dịch dấu chấm sang trái ba số đồng thời nhân với  $2^3$ . Bảng 6.5 đưa ra một số ví dụ:

Giá trị nhị phân	Dạng chuẩn	Số Expoment
1111.001	1.111001	3
.0000101	1.01	-5
1.1010	1.1010	0
100011000.0	1.000110000	8

Bảng 6.5: Ví dụ biểu diễn số mantissa

### Biểu diễn nhị phân theo chuẩn IEEE

Khi đã biểu diễn được bit sign, phần expoment và phần mantissa theo dạng chuẩn. Biểu diễn các bit theo thứ tự như hình 6.1. Ví dụ lấy giá trị nhị phân là  $1.101 \times 2^0$  có bit dấu là 0 (số dương), số mũ hiện là 0 nhưng được điều chỉnh bằng cách cộng với 127 (số cơ sở theo định dạng short real) là 127 biểu diễn ở mã nhị phân là 01111111. Số "1" ở phía bên trái cao nhất của phần mantissa (1.101) được loại bỏ lấy phần sau dấu phẩy là 101. Theo thứ tự bit định dạng IEEE short real là 1 01111111 1010000000000000000000. Bảng 6.6 trình bày một số ví dụ:

Giá trị nhị phân	Expoment	Sign, Expoment, Mantissa
-1.101	127	1 01111111 101000000000000000000000
+1101.1110	130	0 10000010 111000000000000000000000
-0.000101	124	1 01111100 010000000000000000000000
+1111110001.0	136	1 10001000 111111000100000000000000
+0.0000001101011	120	1 01111000 101011000000000000000000

Bảng 6.6: Ví dụ chuẩn IEEE

## PHỤ LỤC 2

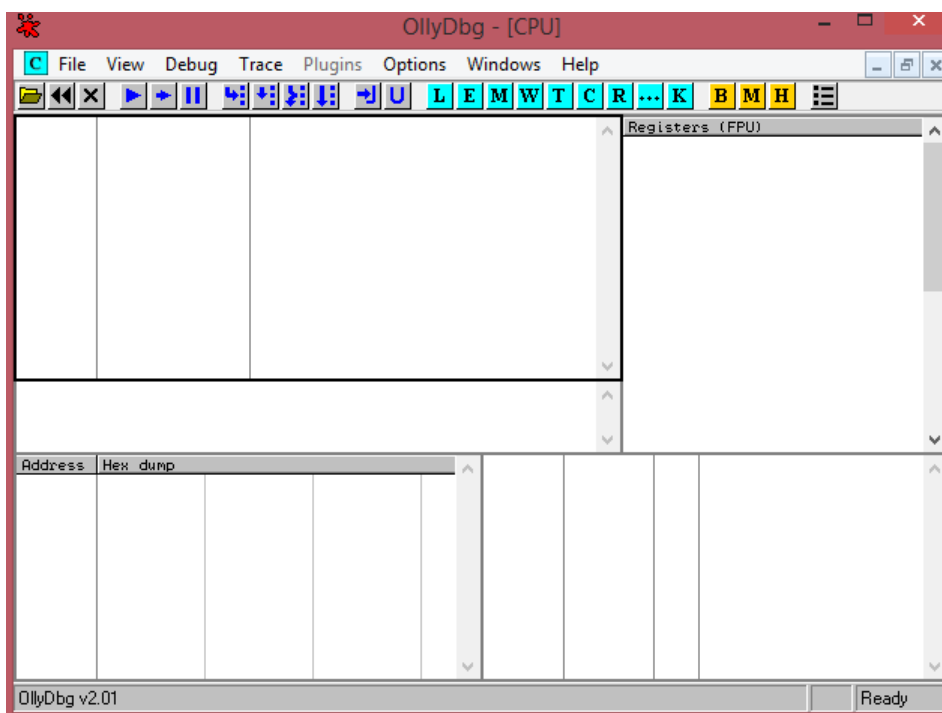
### *Công cụ Ollydbg*

#### Giới thiệu

OllyDbg là một công cụ sử dụng hợp ngữ trên nền Windows 32-bit chú trọng đến việc phân tích mã nhị phân. Công cụ dò xét các thanh ghi, nhận diện các thao tác với câu lệnh assembly, các lời gọi hàm API, hằng số, chuỗi, từ khóa chuyển câu lệnh, cũng như chỉ ra địa chỉ thao tác từ tập tin và các thư viện. Công cụ ollydbg hoàn toàn miễn phí và đầy đủ chức năng, không có giới hạn thời gian sử dụng, nhưng có thông tin đăng ký với tác giả như ở các dạng phần mềm dùng thử. Các phiên bản hiện tại của OllyDbg không thể thao tác được các tập tin biên dịch cho các bộ vi xử lý 64-bit.

#### Thao tác

Hình 6.2 thể hiện giao diện chính của Ollydbg



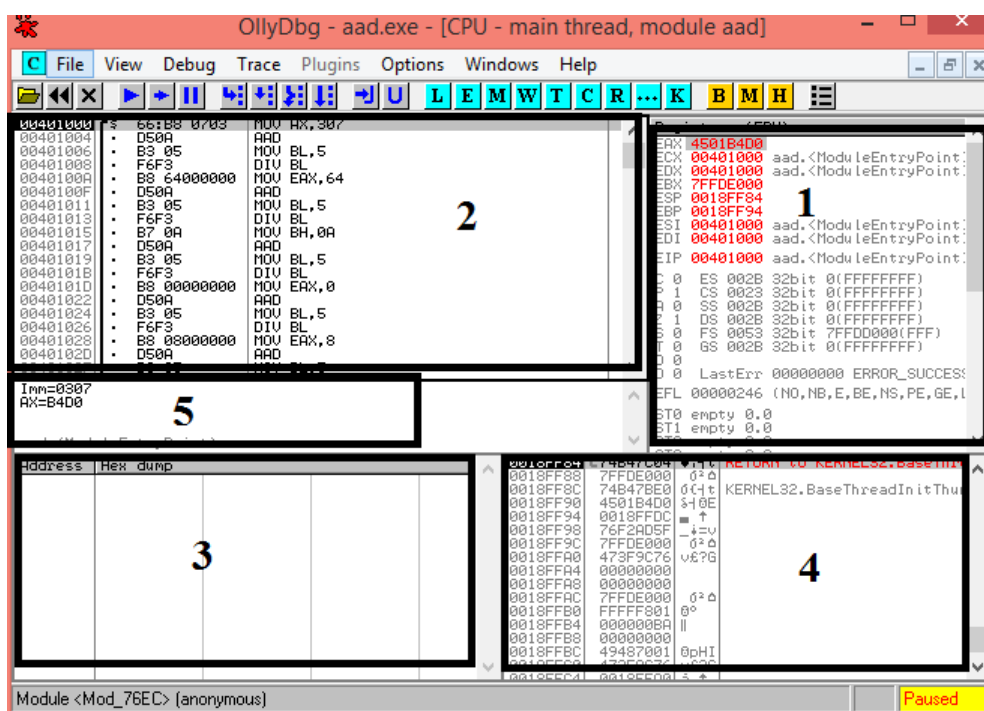
Hình 6.2: Giao diện công cụ Ollydbg

File đầu vào của công cụ Ollydbg là một file thực thi có đuôi file là ".exe". Để mở file, theo đường dẫn File->Open để chọn nơi lưu file sau đó Click Open. Ngoài ra có thể thao tác click vào biểu tượng thư mục như hình 6.3 để mở file.



Hình 6.3: Mở file trong Ollydbg

Khi mở một file thực thi sẽ có giao diện như hình 6.4



Hình 6.4: Ví dụ Ollydbg

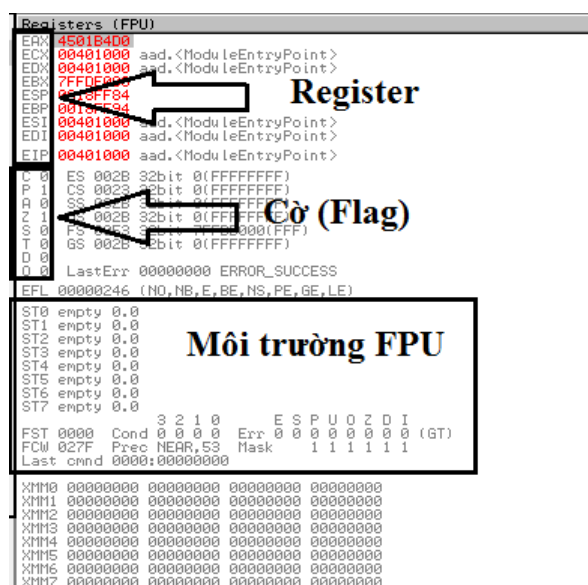
Giao diện khi mở một file thực thi có 5 cửa sổ

- 1 Cửa sổ thanh ghi (Registers): đây là cửa sổ chứa thông tin chi tiết về các thanh ghi như eax, ebx, ecx v...v...v. Các cờ trạng thái cũng được quản lý tại cửa sổ này.
- 2 Cửa sổ giả mã assembly (Disassembler): cửa sổ này cho thấy các đoạn code của chương trình ở dạng ngôn ngữ assembly, và đồng thời tại cửa sổ này các bạn cũng có thể chú thích cho từng từng dòng mã assembly.

- 3 Cửa sổ giải mã giá trị (Dump): cửa sổ cho thấy bộ nhớ hiện tại của chương trình theo 2 dạng là hex và Ascii đồng thời cho phép chỉnh sửa bộ nhớ.
- 4 Cửa sổ stack: mọi câu lệnh trước khi được thực hiện phải được nạp vào Stack.
- 5 Cửa sổ thông báo toán hạng (Tip): Khi bạn đang ở tại một dòng code nào đó trong quá trình debug , Olly sẽ cho bạn thấy thông tin chi tiết về dòng code đó. Ví dụ : nếu đang debug ở dòng lệnh “ *mov eax , dword ptr [123]*” . Thì cửa sổ này sẽ cho biết được giá trị hay con số nào đang được lưu giữ tại [123].

## Cửa sổ Register

Hình 6.5 thể hiện cửa sổ Register chứa các biến môi trường được dùng để thao tác với mỗi câu lệnh.

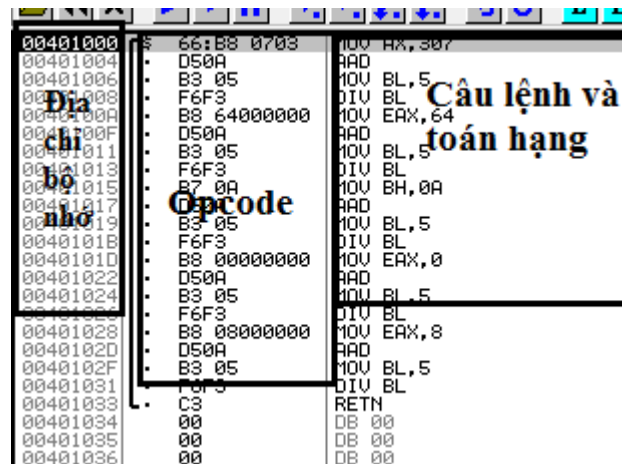


Hình 6.5: Cửa sổ Register trong Ollydbg

Cửa sổ này sẽ cung cấp rất nhiều thông tin trong quá trình chúng ta làm việc với Ollydbg. Các biến EAX, EBX, ECX, EDX, ... thể hiện các thanh ghi xử lý số nguyên. Các biến cờ C, P, A, Z... thay đổi trong quá trình thao tác với các câu lệnh số nguyên. Các biến ST0, ST1, ST2, ... FCW, FSW biểu diễn stack thanh ghi dữ liệu FPU, thanh ghi điều khiển FPU, thanh ghi trạng thái FPU.

## Cửa sổ Disassembler

Hình 6.6 thể hiện cửa sổ Disassembler, biểu diễn các câu lệnh assembly của chương trình đầu vào..

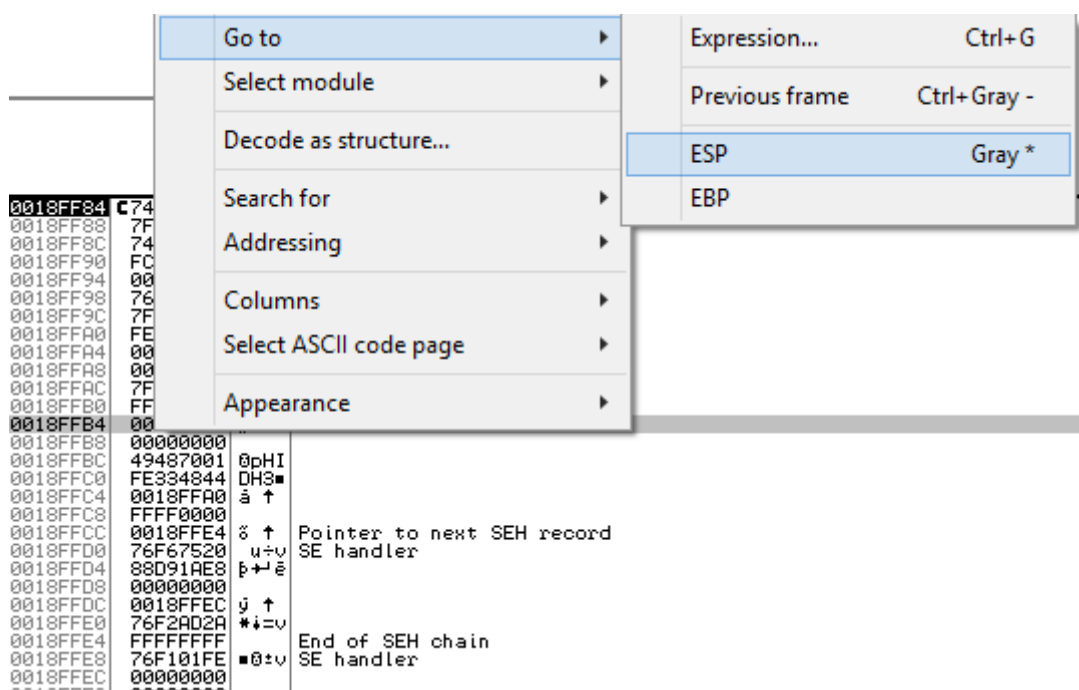


Hình 6.6: Cửa sổ Disassembler trong Ollydbg

Khi muốn debug một chương trình, cần phải load file thực thi của chương trình đó vào trong Ollydbg. Các chương trình mà đã được load vào Ollydbg là những chương trình có thể được code bằng những ngôn ngữ khác nhau như : VB, VC++, Borland Delphi hay MASM nhưng tại cửa sổ này toàn bộ code của chương trình sẽ được list ra dưới dạng các mã ASM. Ollydbg tiến hành phân tích chương trình và thể hiện chương trình dưới dạng assembly ở cửa sổ này đồng thời cung cấp tên câu lệnh assembly cùng với các toán hạng. Cho biết địa chỉ câu lệnh đang ở đâu trong bộ nhớ cùng mã opcode của câu lệnh assembly đang hiện hành.

## Cửa sổ stack

Trước tiên sẽ đi tìm hiểu sơ qua về Stack. Đây là nơi lưu trữ tạm thời các dữ liệu và địa chỉ, nó là một cấu trúc dữ liệu một chiều. Các phần tử được cất vào và lấy ra từ một đầu của cấu trúc này, tức là nó được xử lý theo phương thức “vào trước, ra sau” (LIFO : Last In First Out). Phần tử được cất vào cuối cùng gọi là đỉnh của Stack. Có thể hình dung Stack như là một chồng đĩa, chiếc đĩa được đặt lên cuối cùng sẽ nằm trên đỉnh và chỉ có nó mới có thể được lấy ra đầu tiên. Hai thanh ghi chính làm việc với Stack là ESP và EBP. Theo mặc định trong Olly, Stack được biểu diễn theo thanh ghi ESP tuy nhiên ta có thể luân chuyển qua lại giữa ESP và EBP bằng cách nhấn chuột phải và chọn như hình 6.7

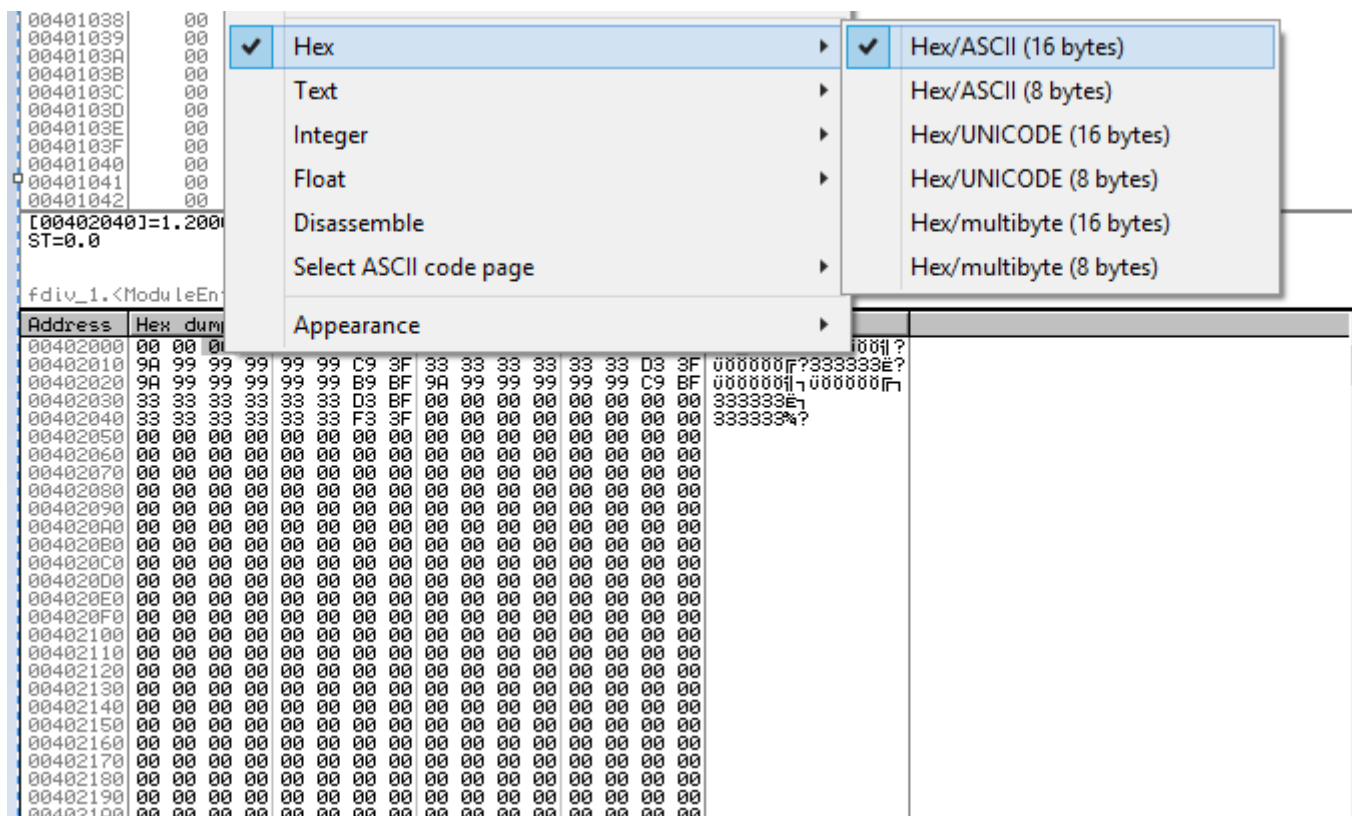


Hình 6.7: Cửa sổ Stack trong Ollydbg



## Cửa sổ Dump

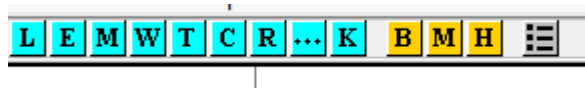
Đây là cửa sổ hiện thị nội dung của bộ nhớ hoặc file. Ta có thể chọn nhiều định dạng khác nhau để biểu diễn nội dung của memory trong cửa sổ này : byte, text, integer, float, address, disassembly hoặc PE Header. Cửa sổ này cho phép chúng ta tìm kiếm cũng như thực hiện các chức năng chỉnh sửa, thiết lập các Break points v.v... được thể hiện trong hình 6.8.



Hình 6.8: Cửa sổ Dump trong Ollydbg

## Chức năng

Tiếp theo sẽ giới thiệu qua chức của từng nút trên thanh công cụ Ollydbg(hình 6.9 )



Hình 6.9: Chức năng trong Ollydbg

- Nút L dùng để mở cửa sổ Log của Olly, cửa sổ thể hiện những thông tin mà Olly ghi lại. Theo mặc định thì cửa sổ này sẽ lưu các thông tin về các module, import library hoặc các Plugins được load cùng chương trình tại thời điểm đầu tiên khi ta load chương trình vào Olly. Bên cạnh đó cửa sổ này cũng ghi lại các thông tin về các Break points mà chúng ta đặt trong chương trình.
- Nút E dùng để mở cửa sổ Executables, cửa sổ này thể hiện danh sách những file có khả năng thực thi được chương trình sử dụng như file exe, dlls, ocxs , v.v..
- Nút M dùng để mở cửa sổ Memory, cửa sổ này sẽ cho chúng ta thông tin về bộ nhớ đang được sử dụng bởi chương trình.
- Nút T dùng để mở cửa sổ Threads, cửa sổ này liệt kê các Threads của chương trình.
- Nút W dùng để mở cửa sổ Windows.
- Nút H dùng để mở cửa sổ Handles.
- Nút ... để mở cửa sổ Patches, cửa sổ này sẽ cho chúng ta các thông tin về những thông tin đã edit trong chương trình.
- Nút K để mở cửa sổ Call Stack, hiển thị một danh sách các lệnh call mà chương trình đã thực hiện khi Run bằng F9 và dùng F12 để tạm dừng chương trình.
- Nút B để mở cửa sổ Break Points, cửa sổ này sẽ hiển thị tất cả các BPs mà đã đặt trong chương trình. Tuy nhiên nó chỉ hiển thị các BPs được set bằng cách nhấn F2, còn các dạng BPs khác như : hardware breakpoint hoặc memory breakpoints thì không được liệt kê ra ở phụ lục này.
- Nút R để mở cửa sổ References, cửa sổ này là kết quả khi thực hiện chức năng Search trong Ollydbg.

**Phím tắt**

**F7 :** Khi nhấn F7 sẽ thực thi từng dòng lệnh 1. Nếu trong quá trình thực thi mà gặp lệnh Call thì sẽ đi vào trong lòng của lệnh Call đó và thực thi từng câu lệnh trong lệnh Call này cho đến khi gặp lệnh Retn để trở lại chương trình chính, tức là câu lệnh tiếp theo sau lệnh Call.

**F8 :** Cũng tương tự như F7 nhưng có 1 điểm khác biệt trong quá trình thực thi từng câu lệnh, nếu như gặp lệnh Call nó bỏ qua không cần quan tâm các lệnh bên trong lệnh Call mà thực thi luôn lệnh Call đó và dừng lại tại câu lệnh tiếp theo dưới lệnh Call.

**F2 :** Đặt một Break point trong chương trình. Vậy Break point là gì , đơn giản nó chỉ là việc chúng ta tạo 1 điểm ngắt trong chương trình theo một điều kiện nào đó để khi thực thi chương trình, nếu thỏa điều kiện mà chúng ta đặt ra thì chương trình sẽ dừng lại tại vị trí mà chúng ta đã đặt BP.

**F9 :** Cho phép thực thi chương trình trong chế độ Debug, tương tự như việc chúng ta nhấp đúp chuột vào chương trình để thực thi nó. Tuy nhiên khác với việc nhấp đúp chuột, nếu chúng ta nhấn F9 thì Olly sẽ tìm xem có BP nào được Set hay không, chương trình có hiện thị ra các Exception gì không, hay nếu chương trình có cơ chế chống Debug thì nó sẽ ngắt ngay lập tức. Nếu như không có bất kì cản trở nào thì chương trình sẽ Run hoàn toàn và trên status bar của Olly sẽ báo cho chúng ta biết điều này.

**F12 :** Tạm dừng chương trình lại.

## PHỤ LỤC 3

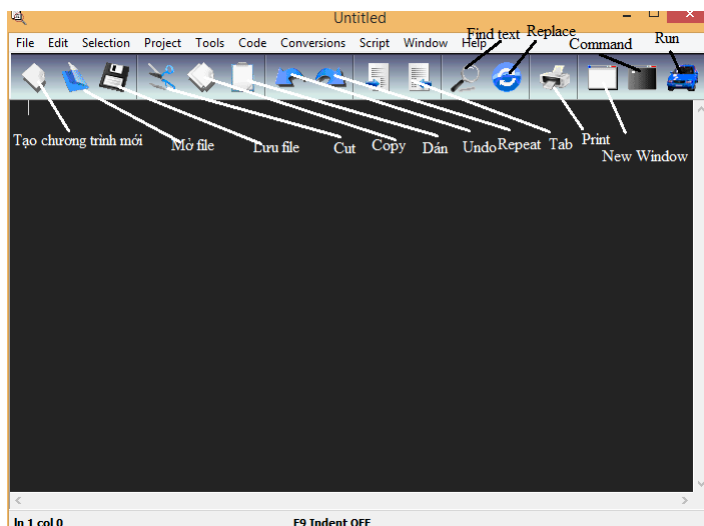
### *Công cụ MASM32*

#### Giới thiệu

Có hai loại trình biên dịch được sử dụng để biên dịch chương trình hợp ngữ (từ tập lệnh hợp ngữ của các vi xử lý họ Intel) sang chương trình thực thi: Trình biên dịch hợp ngữ 16 bit, MASM (Macro Assembler), được sử dụng để dịch thành các chương trình chạy trên nền hệ điều hành 16 bit MS-DOS; Trình biên dịch hợp ngữ 32 bit, MASM32 (Macro Assembler 32 bit), được sử dụng để dịch thành các chương trình chạy trên nền hệ điều hành 32 bit MS-Windows. Trong quá trình làm luận văn, công cụ MASM 32 bit được sử dụng để tạo các test-case, hiện thực các câu lệnh.

#### Thao tác

Để viết một chương trình bằng MASM ta sử dụng QEDITOR.exe trong thư mục MASM có giao diện như hình 6.10



Hình 6.10: Giao diện công cụ MASM32

Trên thanh status bar có các chức năng: Tạo chương trình mới, mở file, lưu file, cut, copy, dán, undo, repeat, tab, find text, replace, print, command, new window, run.

Trước khi biên dịch chương trình bạn cần phải lưu chương trình trước, và trong quá trình làm việc nếu có thay đổi bạn phải lưu trước khi biên dịch vì MASM32 không có cơ chế tự lưu những thay đổi như VC hay VB, và một điều nữa cần chú ý là chức năng Undo trong Masm chỉ cho phép undo 1 hành động vì vậy khi có nhiều thay đổi mà bạn nghĩ có thể phải undo thì nên save trước, nếu cần phục hồi lại thì exit và không save thì nó sẽ ở trạng thái ở lần save cuối cùng. Ví dụ nếu đã có mã code và bây giờ cần biên dịch vào Menu item: Project, trong menu Project có các mục sau:

- Compile Resource File: biên dịch file resource, file resource có phần mở rộng \*.rc file này chứa các tài nguyên như Icon, DialogBox, Bitmap... mà bạn sử dụng.
- Assemble Asm file: Tạo file \*.Obj từ file .asm.
- Link Obj: từ file Obj link tới các tài nguyên cần thiết để tạo file exe.
- Assemble & Link: thực hiện cả hai bước trên, việc này sẽ tạo sự thuận tiện cho người lập trình, không phải tốn công thực hiện qua hai bước mới tạo nên file .exe
- Build all: Chức năng này có tác dụng biên dịch cả file resource, và tạo file .exe. Chức năng này được sử dụng khi có thay đổi những tài nguyên ở file resource. Còn nếu chỉ thay đổi về code trong chương trình thì nên sử dụng Assemble & link, nó sẽ rút ngắn thời gian biên dịch.
- Run Makeit.bat: nếu muốn có một file Makeit.bat và muốn sử dụng nó để biên dịch thay vì xài những tùy chọn biên dịch mặc định của MASM. Cũng với những chức năng trên nhưng có thêm console thì khi chạy chương trình còn kèm theo một cửa sổ dòng lệnh, nếu bạn có ý định tạo chương trình chạy trong Windows thì không nên sử dụng những chức năng này.
- Run Program: để chạy thử chương trình sau khi biên dịch.

## Tài liệu tham khảo

1. Frances E. Allen. *Control flow analysis*. SIGPLAN Notices 5 (7): 1–19. July 1970.
- [ 1 ] Nguyen Minh Hai, Mizuhito Ogawa and Quan Thanh Tho, “Pushdown Model Generation of Malware”, June 24th 2006.
- [ 2 ] Quan Thanh Tho, with Nguyen Minh Hai, in Collaboration with Mizuhito Ogawa, “BE – PUM: Binary Emulation for Pushdown Model Generation, a tool for under approximated model generation”, March 2015
- [ 3 ] Nguyen Minh Hai, with Quan Thanh Tho, in Collaboration with Mizuhito Ogawa, “Pushdown Model Generation for Malware Deobfuscation”, March 2015
- [ 4 ] Irvine, Kip R. *Assembly Language for x86 Processors*. Prentice Hall, 2011.
- [ 5 ]. Hyde, Randall. *The art of assembly language*. No Starch Press, 2010.
- [ 6 ] Corportation, Intel. "IA-32 Intel Architecture software developer's manual." Intel Corportation (2001).
- [ 7 ] Tất cả thông tin về các câu lệnh x86, <http://x86.renejeschke.de/html>
- [ 8 ] Link tham khảo online: <http://www.felixcloutier.com/x86/>
- [ 9 ] Phép toán chuyển đổi <http://www.exploringbinary.com/floating-point-converter/>