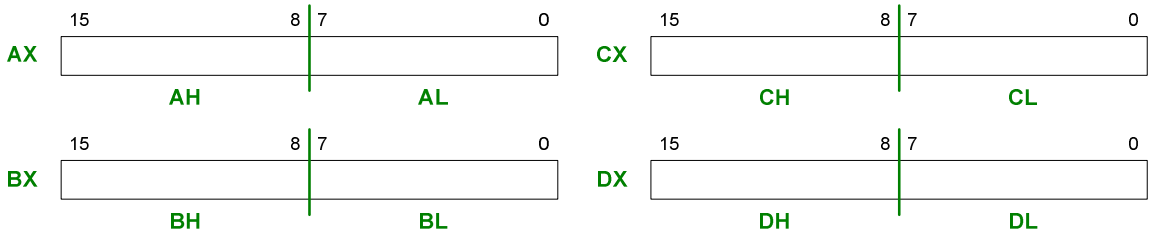
Bộ thanh ghi trong 8086

Các thanh ghi trong bộ vi xử lí 8086 đều là các thanh ghi 16 bit và được chia thành các nhóm như sau:

- Các thanh ghi công dụng chung

  AX (accumulator), BX (base), CX (counter), DX (data): có thể được truy xuất độc lập như 2 thanh ghi 8 bit : AH và AL, BH và BL, CH và CL, DH và DL.



- Các thanh ghi con trỏ và chỉ mục (xem chi tiết ở các phần sau)

  SP (Stack Pointer), BP (Base Pointer): con trỏ dùng khi làm việc với stack

  SI (Source Index), DI (Destination Index): chỉ số mảng khi xử lí mảng (chuỗi)

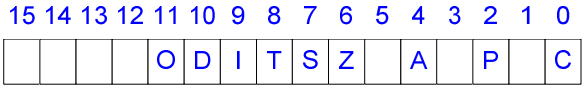
- Các thanh ghi phân đoạn

  CS (Code Segment), DS (Data Segment), ES (Extra data Segment), SS (Stack Segment): tương ứng lưu địa chỉ phân đoạn mã lệnh, phân đoạn dữ liệu, phân đoạn dữ liệu bổ sung, phân đoạn ngăn xếp. Địa chỉ phân đoạn này sẽ được kết hợp với địa chỉ offset để truy xuất ô nhớ. (xem chi tiết ở các phần sau)

- Các thanh ghi con trỏ lệnh và trạng thái

  IP (Intruction Pointer): thanh ghi chứa địa chỉ offset của lệnh kế tiếp cần thực hiện. Thanh ghi này không thể được truy xuất trực tiếp.

  FLAGS: thanh ghi cờ trạng thái, dùng để chứa các bit mô tả trạng thái của lệnh vừa được thực hiện, hoặc chứa các bit điều khiển cần thiết lập trước khi gọi lệnh. Bao gồm các bit cờ sau đây: (xem chi tiết ở các phần sau)



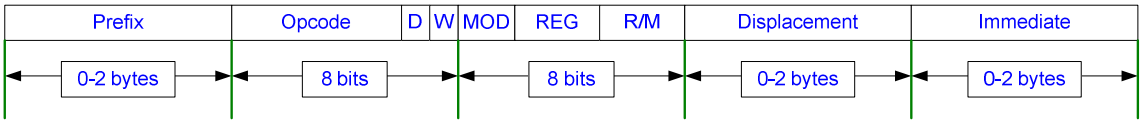
CF (Carry Flag):	bật khi phép tính vừa thực hiện có sử dụng bit nhớ
PF (Parrity Flag):	bật khi kết quả của phép tính vừa thực hiện có chẵn bit 1
AF (Auxilary Flag):	bật khi phép tính vừa thực hiện có sử dụng bit nhớ phụ
ZF (Zero Flag):	bật khi kết quả của phép tính vừa thực hiện là 0
SF (Sign Flag):	bật khi kết quả của phép tính vừa thực hiện có bit dấu bật
TF (Trace Flag):	bật để chuyển sang chế độ chạy từng bước
IF (Interrupt Flag):	bật để cho phép các ngắt xảy ra
DF (Direction Flag):	bật để chọn chế độ giảm chỉ số tự động khi làm việc với mảng
OF (Overflow Flag):	bật khi phép tính vừa thực hiện gây ra tràn số

Cấu trúc mã lệnh – Các kiểu định vị dữ liệu

Cấu trúc mã lệnh (Instruction format) trong 8086

Một lệnh (instruction) mà bộ vi xử lí có thể hiểu được thường rất đơn giản. Ví dụ như di chuyển dữ liệu từ một ô nhớ vào thanh ghi, cộng thanh ghi thứ hai vào thanh ghi thứ nhất,… Trong 8086 mỗi lệnh thường tác động đến 0,1 hoặc 2 đối tượng (operand, tạm gọi là toán hạng). Toán hạng có thể là một thanh ghi, một hằng số hoặc một ô nhớ.

Thông thường, một lệnh (instruction) có cấu trúc như sau:



Hình 1. Cấu trúc mã lệnh

Trường Prefix dùng để thay đổi thanh ghi phân đoạn mặc định hoặc chỉ định sự lặp lại của lệnh trong thao tác xử lí chuỗi.

Trường Opcode là mã của thao tác, cho biết lệnh này làm gì.

Bit D (direction) cho biết hướng tác động của lệnh.

Bit W (width) cho biết kích thước của toán hạng.

Hai trường REG và R/M mô tả hai toán hạng chịu tác động của lệnh.

Trường REG chứa mã số của một thanh ghi.

Trường R/M có thể là mã số của một thanh ghi (trường hợp toán hạng là thanh ghi) hoặc là mô tả cách tính địa chỉ của một ô nhớ trong bộ nhớ (trường hợp toán hạng là ô nhớ).

Trường MOD cho biết trường R/M mô tả thanh ghi hay ô nhớ, cũng như cho biết có trường Displacement phía sau hay không.

Trường Displacement được sử dụng khi toán hạng là ô nhớ. Trường này được sử dụng kết hợp với các thông tin lưu trong trường R/M để tính địa chỉ của toán hạng.

Trường Immediate được sử dụng khi toán hạng là một hằng số.

Không phải lệnh nào cũng có đủ các trường được miêu tả ở trên.

Chi tiết về các trường như sau:

D = 1	REG là đích đến
D = 0	REG là nguồn

W = 1	Toán hạng là word
W = 0	Toán hạng là byte

REG	**W = 1**	**W = 0**	**Segment**
000	AX	AL	ES
001	CX	CL	CS
010	DX	DL	SS
011	BX	BL	DS
100	SP	AH	
101	BP	CH	
110	SI	DH	
111	DI	BH	

Bảng 1 Mã trường REG và các bit D, W

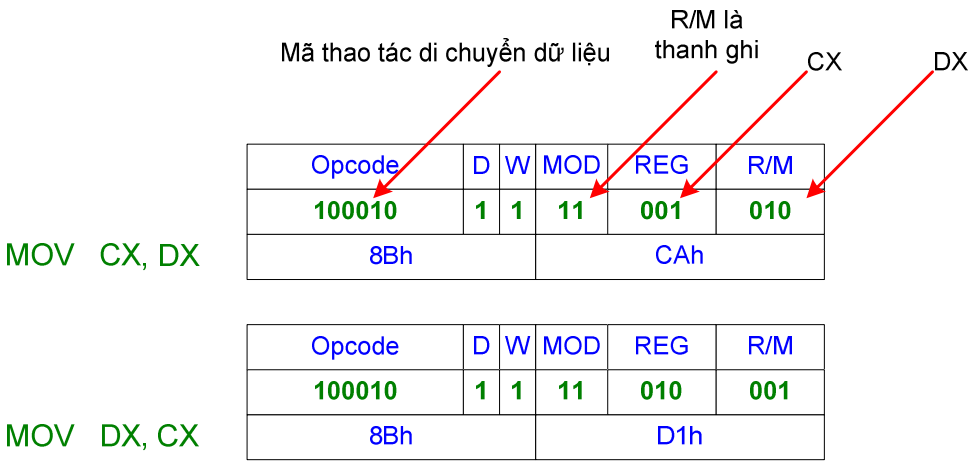
MOD	Ý nghĩa của R/M và Displacement
00	Nếu R/M = 110 thì EA = disp-high _ disp-low Nếu R/M ≠ 110 thì phần displacement không có.
01	DISP = disp-low, sẽ được signed extended
10	DISP = disp-high _ disp-low
11	R/M mô tả thanh ghi

Bảng 2. Mã trường MOD

R/M	Cách tính địa chỉ (Effective Address)
000	EA = BX + SI + DISP
001	EA = BX + DI + DISP
010	EA = BP + SI + DISP
011	EA = BP + DI + DISP
100	EA = SI + DISP
101	EA = DI + DISP
110	EA = BP + DISP (trừ trường hợp MOD = 00, xem ở trên)
111	EA = BX + DISP

Bảng 3. Mã trường R/M

Ví dụ: Lệnh chép nội dung thanh ghi DX vào CX có mã 8BCAh, lệnh chép nội dung thanh ghi CX vào DX có mã 8BD1h.



Hình 2. Mã lệnh MOV giữa hai thanh ghi

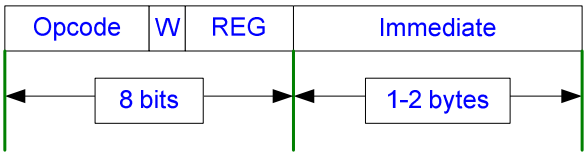
Ví dụ: Lệnh chép nội dung biến **var** (có địa chỉ DS:1234h) vào thanh ghi CX có mã 8B0E1234h. (Xem Hình 3). Lệnh chép ngược lại có mã 890E1234h. Hai mã này chỉ khác nhau ở bit D. Trường hợp thứ nhất, D = 1 vì thanh ghi CX là đích đến. Trường hợp thứ hai, D = 0 vì thanh ghi CX là nguồn. Trong cả hai trường hợp, địa chỉ (EA) của ô nhớ (biến **var**) được lấy trực tiếp từ 2 bytes của vùng Displacement (MOD = 00 và R/M = 110, xem ở bảng phía trên). Hai byte disp-high và disp-low của vùng Displacement chứa offset của biến **var** trong phân đoạn dữ liệu.

	Opcode	D	W	MOD	REG	R/M	disp-high	disp-low
	100010	1	1	00	001	110	0001 0010	0011 0100
MOV CX, var	8Bh			0Eh			12h	34h

	Opcode	D	W	MOD	REG	R/M	disp-high	disp-low
	100010	0	1	00	001	110	0001 0010	0011 0100
MOV var, CX	89h			0Eh			12h	34h

Hình 3. Mã lệnh MOV giữa thanh ghi và bộ nhớ

Một số lệnh có cấu trúc đặc biệt, ví dụ, ghép chung trường opcode với trường reg, nhằm giúp rút ngắn độ dài lệnh. Ví dụ: lệnh gán AX bằng 4567h có mã B84567h.



a) Cấu trúc chung

	Opcode	W	REG	Immediate
	1011	1	000	0100 0101 0110 0111
MOV AX, 4567h	B8h			4567h

b) Ví dụ: MOV AX, 4567h

Hình 4. Mã lệnh MOV giữa Accumulator và hằng số

Khi toán hạng là một thanh ghi phân đoạn, chỉ cần 2 bit để chỉ định một trong bốn thanh ghi phân đoạn. Nghĩa là trường REG luôn có dạng 0xx. Bit xx được định nghĩa như trong Bảng 1. Ví dụ, lệnh chép nội dung thanh ghi AX vào thanh ghi phân đoạn DS có mã là 8ED8h.

	Opcode	D	0	MOD	0xx	R/M
	100011	1	0	11	011	000
MOV DS, AX	8Eh			D8h		

R/M là thanh ghi — DS — AX

Hình 5. Mã lệnh MOV giữa thanh ghi thường và thanh ghi phân đoạn

Khi có chỉ định thanh ghi phân đoạn dùng để truy xuất bộ nhớ khác với thanh ghi phân đoạn mặc định thì trong mã lệnh xuất hiện thêm Prefix. Ví dụ: lệnh chép nội dung ô nhớ ES:2345h vào thanh ghi DS có mã 268E1E2345h, trong đó 26h là prefix.

	001 xx 110	Opcode	D	0	MOD	REG	R/M	disp-high disp-low
	001 00 110	100011	1	0	00	011	110	0010 0011 0100 0101
MOV DS,ES:2345h	26h	8Eh			1Eh			2345h

ES

Hình 6. Mã lệnh MOV giữa thanh ghi phân đoạn và ô nhớ, có sử dụng segment override prefix

Stack và ứng dụng trong việc gọi chương trình con, gọi ngắt

Khái niệm stack

Stack là một vùng bộ nhớ mà ở đó, ngoài việc truy xuất trực tiếp các ô nhớ bằng địa chỉ, người ta định nghĩa thêm hai thao tác là PUSH (bỏ vào) và POP (lấy ra). Stack thường được dùng làm nơi lưu trữ tạm thời các giá trị trung gian hoặc dùng trong việc gọi chương trình con.

Nếu *chỉ sử dụng hai thao tác PUSH & POP* để truy xuất dữ liệu trong stack thì stack giống như một cái thùng đựng tài liệu. Những gì bỏ vào sau sẽ nằm trên những gì bỏ vào trước, do đó khi lấy ra thì bao giờ cũng phải lấy cái bỏ vào sau cùng. Người ta gọi cấu trúc như vậy là LIFO (last in first out).

Để ghi lại địa chỉ offset nơi bỏ dữ liệu vào sau cùng, người ta dùng thanh ghi SP, còn địa chỉ segment thì được lưu trong thanh ghi SS. Ví dụ: (Xem Hình 7, Hình 8)

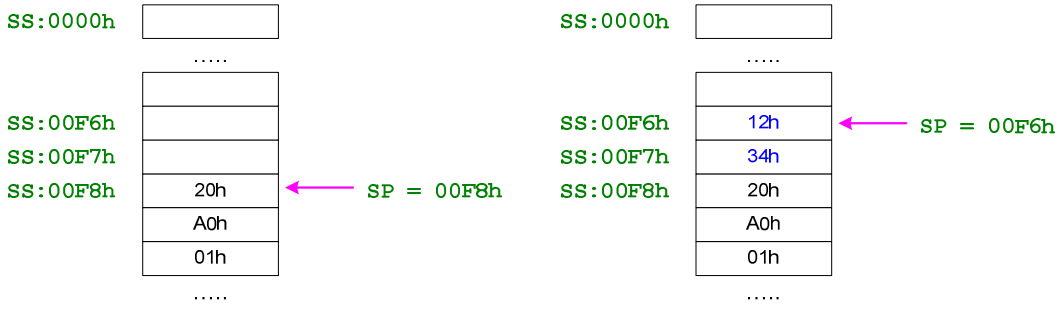
Thao tác *PUSH AX* bao gồm: giảm SP đi 2, đưa giá trị của AX vào ô nhớ có địa chỉ SS:SP.

Thao tác *POP AX* bao gồm: đưa giá trị từ ô nhớ có địa chỉ SS:SP vào AX, tăng SP lên 2.

Như thế, stack được sử dụng (còn gọi là "nở ra") theo chiều giảm của địa chỉ, khác với các vùng nhớ thông thường được sử dụng theo chiều tăng của địa chỉ.

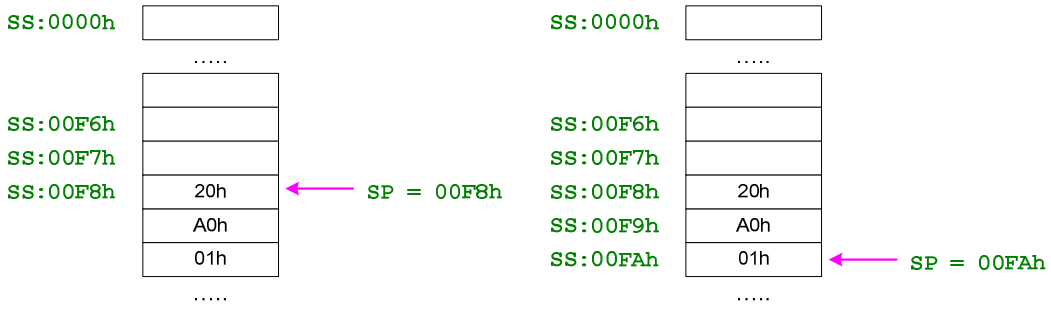
Nếu không khai báo phân đoạn stack, theo mặc định, đoạn stack và code sẽ chung nhau, và khi chương trình bắt đầu thực hiện, SP được khởi động giá trị 0. Vì vậy, thao tác *PUSH xx* lần đầu tiên sẽ đặt giá trị *xx* tại địa chỉ SS:FFFEh (vì 0 – 2 = FFFEh), thao tác *PUSH yy* liền tiếp theo sẽ đặt giá trị *yy* tại địa chỉ SS:FFFCh. (Xem Hình 9)

Nếu có khai báo phân đoạn stack, ví dụ: ".stack 200h", đoạn stack sẽ được cấp phát riêng, và khi chương trình bắt đầu thực hiện, SP được khởi động giá trị 200h. Vì vậy, thao tác *PUSH xx* lần đầu tiên sẽ đặt giá trị *xx* tại địa chỉ SS:01FEh, thao tác *PUSH yy* liền tiếp theo sẽ đặt giá trị *yy* tại địa chỉ SS:01FCh. (Xem Hình 10)



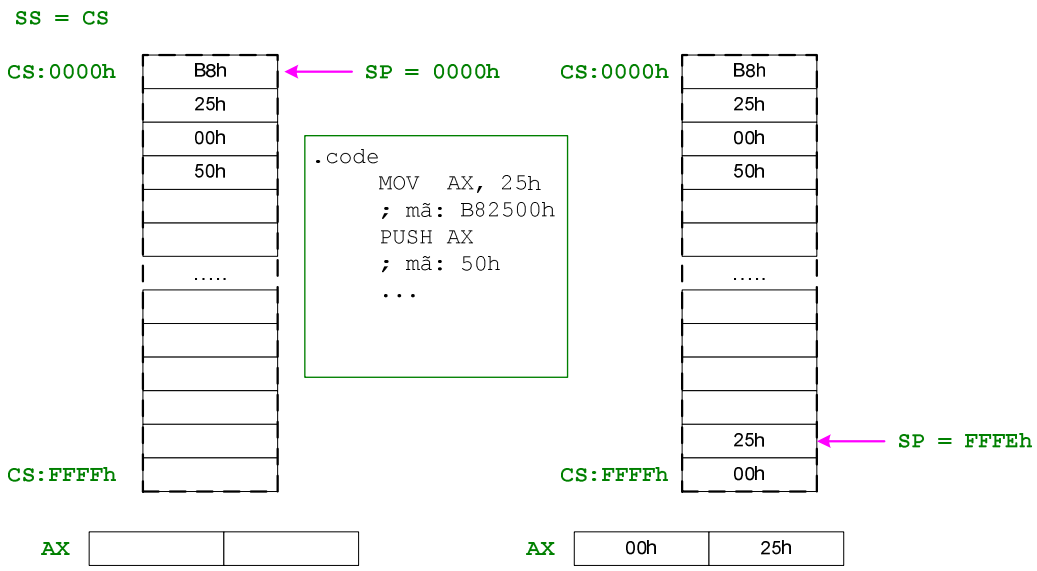
a) AX có giá trị 3412h, trước thao tác PUSH AX b) Sau thao tác PUSH AX

Hình 7. Thực hiện PUSH AX



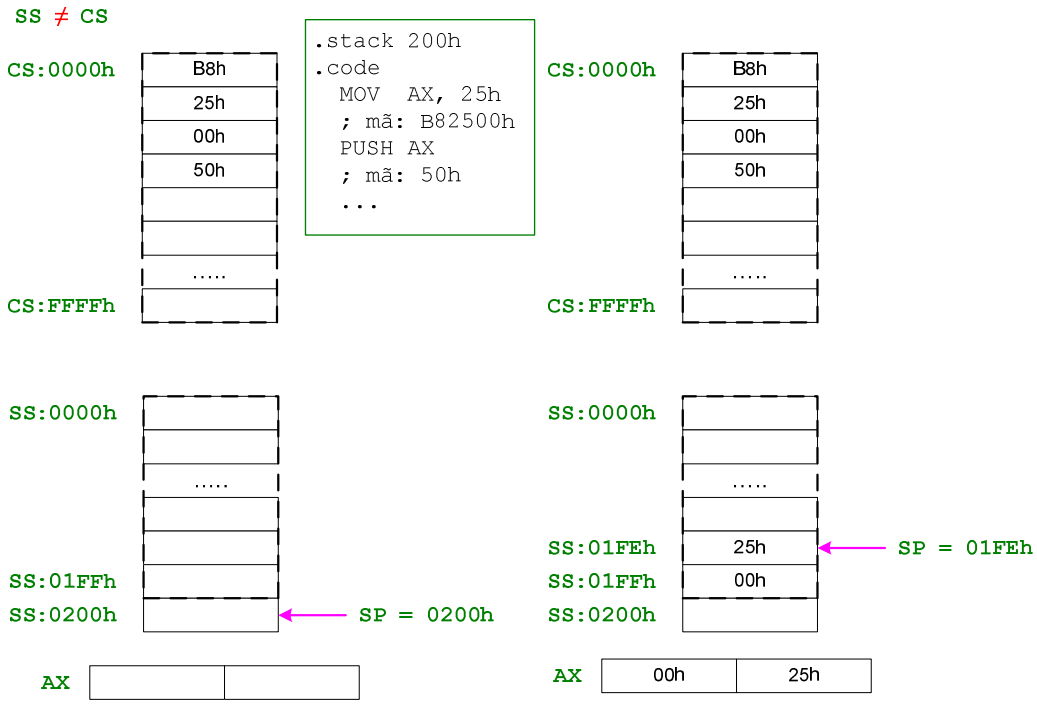
a) Trước thao tác POP AX b) Sau thao tác POP AX, thanh ghi AX có giá trị A020h

Hình 8. Thực hiện POP AX



a) Trước khi thực hiện PUSH b) Sau khi thực hiện PUSH

Hình 9. Trường hợp không khai báo stack segment



a) Trước khi thực hiện PUSH b) Sau khi thực hiện PUSH

Hình 10. Trường hợp có khai báo stack segment

Chương trình con

Chương trình con là một nhóm các lệnh thực hiện một công việc nhất định, có thể cần được làm lặp đi lặp lại nhiều lần ở nhiều thời điểm khác nhau. Mỗi khi cần thực hiện công việc đó, người ta nói rằng cần phải *"gọi chương trình con"* tương ứng.

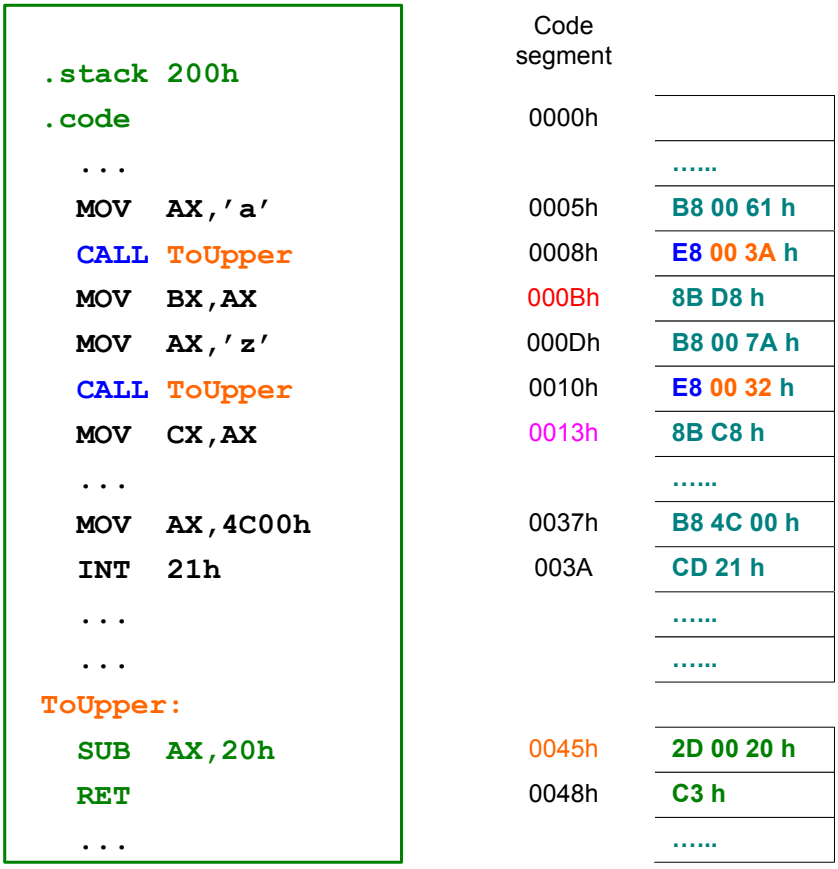
Nếu nhìn một chương trình máy tính như là một dãy liên tiếp các lệnh, thì việc "gọi một chương trình con" chỉ là việc *thay đổi trật tự thực hiện lệnh*, hay nói khác đi là thay vì thực hiện lệnh tiếp theo liền sau lệnh vừa thực hiện, CPU "nhảy" đến một chỗ khác để thực hiện các lệnh ở đó, sau đó quay lại chỗ cũ và thực hiện tiếp các lệnh đang bỏ dở. Như vậy, để gọi một chương trình con, ta cần hai thao tác là CALL (gọi) và RET (trở về). Đây chính là hai lệnh thuộc nhóm các lệnh chuyển điều khiển. Lệnh CALL dùng trong chương trình chính để gọi một chương trình con, lệnh RET dùng ở cuối chương trình con để quay trở về chương trình chính.

Thao tác CALL:

- *sử dụng stack* để lưu trữ (PUSH) địa chỉ của *lệnh tiếp ngay sau lệnh CALL* (nơi cần quay lại)
- ghi vào thanh ghi con trỏ lệnh *IP* địa chỉ của *lệnh đầu tiên của chương trình con*.

Thao tác RET lấy (POP) giá trị từ stack và ghi vào thanh ghi con trỏ lệnh IP, làm cho lệnh tiếp theo được thực hiện chính là lệnh ngay sau lệnh CALL.

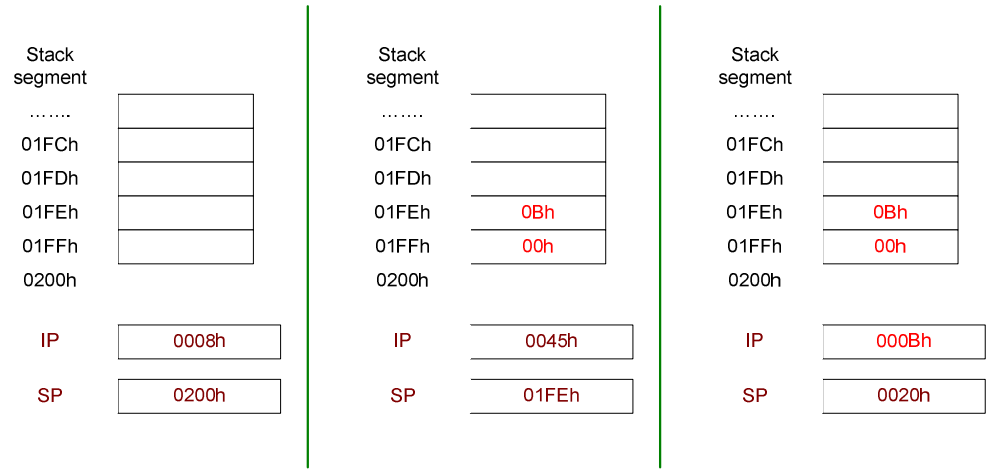
Như đã biết, *địa chỉ có thể là địa chỉ gần hoặc địa chỉ xa*. Nếu chương trình con và chương trình chính nằm cùng một segment thì lệnh CALL chỉ cần PUSH vào stack 2 byte (địa chỉ gần). Trong trường hợp chương trình con và chương trình chính nằm ở hai segment khác nhau lệnh CALL phải PUSH 4 byte vào stack (địa chỉ xa). Tương ứng, lệnh RET sẽ lấy ra 2 hoặc 4 byte tùy trường hợp. Xem ví dụ đoạn chương trình và mã lệnh tương ứng như sau:



Hình 11. Ví dụ sử dụng CALL

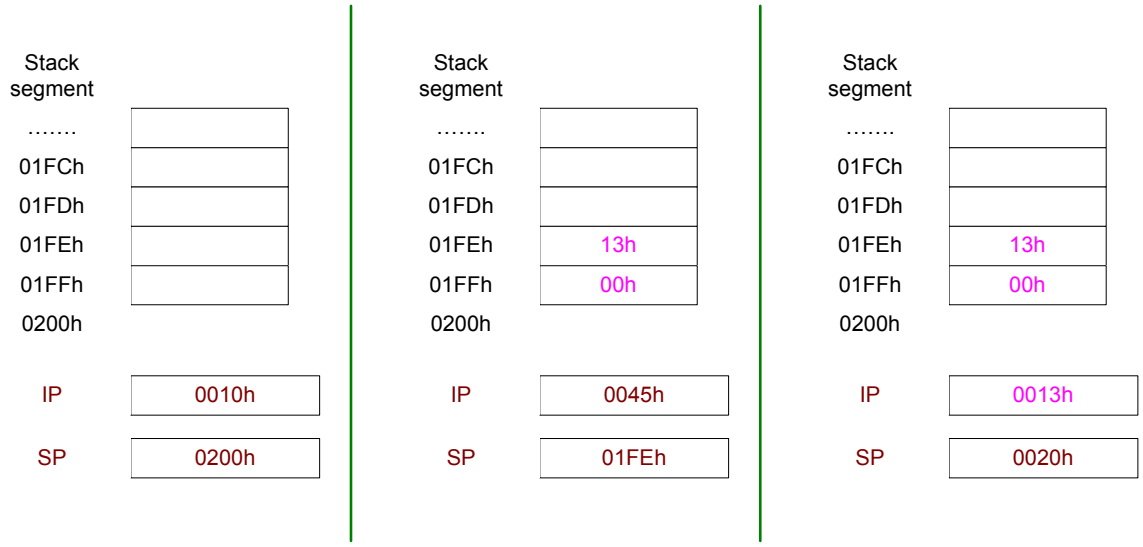
Trong ví dụ, hàm (chương trình con) ToUpper được gọi 2 lần. (Lưu ý rằng nhãn ToUpper không tạo ra mã lệnh nên không chiếm chỗ trong code segment). Lần thứ nhất khi thực hiện lệnh CALL ở địa chỉ CS:0008h (Xem Hình 12)

- Tại thời điểm *trước khi thực hiện* lệnh CALL này, thanh ghi IP có giá trị 0008h, còn thanh ghi SP có giá trị 0200h (trỏ xuống đáy stack).
- Khi lệnh CALL *được nạp* từ bộ nhớ vào CPU để thực hiện, thanh ghi IP được tự động tăng lên một lượng bằng kích thước mã lệnh CALL, do đó sẽ có giá trị 000Bh và trỏ đến lệnh tiếp theo là lệnh MOV.
- Khi lệnh CALL *được thực hiện*, giá trị của thanh ghi IP (chính là địa chỉ của lệnh MOV) được push vào stack (SP giảm xuống còn 01FEh), sau đó thanh ghi IP được cộng thêm một lượng bằng displacement lưu trong mã của lệnh CALL (003Ah). Kết quả là IP có giá trị 0045h (=000Bh+003Ah), chính là địa chỉ của ToUpper.
- Lệnh tiếp theo được nạp vào CPU để thực hiện sẽ là lệnh SUB ở địa chỉ 0045h.
- Sau đó, lệnh RET được thực hiện, làm cho giá trị trong stack được POP ra thanh ghi IP. Kết quả là IP có giá trị 000Bh, trỏ đến lệnh MOV, còn SP tăng lên 0200h.
- Tiếp theo, lệnh MOV ở địa chỉ 000Bh được thực hiện.



a) Trước CALL b) Sau CALL c) Sau RET

Hình 12. Quá trình gọi chương trình con



a) Trước CALL b) Sau CALL c) Sau RET

Hình 13. Gọi lần 2

Lần thứ 2, hàm ToUpper được gọi tại địa chỉ 0010h, quá trình diễn ra tương tự. (Xem Hình 13). Lưu ý rằng hai lời gọi CALL ToUpper được dịch thành 2 mã lệnh khác nhau (E8003Ah và E80032h), bởi vì hai lời gọi nằm ở hai vị trí khác nhau, có displacement khác nhau.

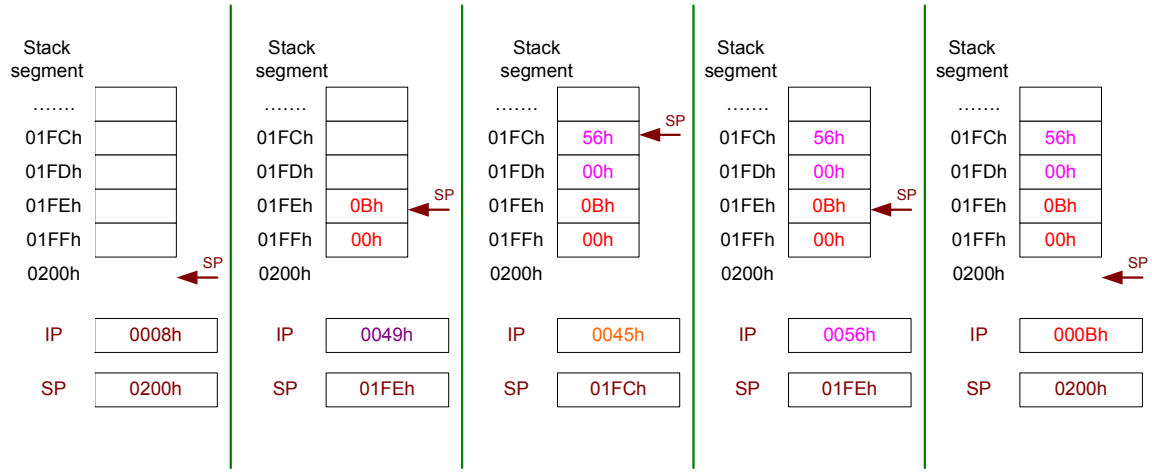
Nhưng *vì sao cần dùng stack* để lưu địa chỉ trở về ? Nguyên nhân là do các lời gọi có thể lồng nhau, nghĩa là trong chương trình con này có thể xuất hiện lời gọi chương trình con khác, hoặc là gọi chính nó. Khi đó, thao tác RET đầu tiên cần lấy địa chỉ trở về được cất bởi CALL sau cùng. Điều này phù hợp với *nguyên tắc LIFO* của stack. (Xem Hình 14)

```
.stack 200h
.code
   ...
   MOV   AX,'V'
   CALL  Upcase
   MOV   BX,AX
   MOV   AX,'n'
   CALL  Upcase
   MOV   CX,AX
   ...
   MOV   AX,4C00h
   INT   21h
   ...
   ...
ToUpper:
   SUB   AX,20h
   RET
Upcase:
   CMP   AX,'a'
   JB    Notaz
   CMP   AX,'z'
   JA    Notaz
   CALL  ToUpper
Notaz:
   RET
   ...
```

Code segment	
0000h	
	.....
0005h	B8 00 56 h
0008h	E8 00 3E h
000Bh	8B D8 h
000Dh	B8 00 6E h
0010h	E8 00 36 h
0013h	8B C8 h
	.....
0037h	B8 4C 00 h
003A	CD 21 h
	.....
	.....
0045h	2D 00 20 h
0048h	C3 h
0049h	3D 00 61 h
004Ch	72 08 h
004Eh	3D 00 7A h
0051h	77 03 h
0053h	E8 FF EF h
0056h	C3 h

Hình 14. CALL lồng nhau

Quá trình PUSH và POP các địa chỉ trở về vào stack diễn ra như sau. Xem Hình 15.



a) Trước CALL Upcase | b) Sau CALL Upcase | c) Sau CALL ToUpper | d) Sau RET của ToUpper | e) Sau RET của Upcase

Hình 15. Sự thay đổi của stack khi CALL lồng nhau

Interrupt (ngắt)

Ngắt là một cơ chế cho phép CPU nhận biết về những sự kiện xảy ra bên ngoài (hardware interrupt, ngắt cứng) cũng như bên trong CPU (software interrupt, ngắt mềm) và có một đáp ứng thích hợp.

Ngắt cứng được dùng để *tránh việc CPU phải chờ đợi* những thiết bị ngoại vi khác có *tốc độ xử lí chậm* hơn. Trong thời gian các thiết bị này còn đang xử lí, thì CPU có thể làm những công việc khác. Sau khi hoàn thành công việc của mình, thiết bị sẽ *chủ động gây ra một sự thay đổi tín hiệu trên một dây dẫn* nhằm mục đích báo cho CPU biết về tình trạng của mình. Khi CPU nhận được sự thay đổi tín hiệu này (xem như một *sự kiện*), CPU sẽ *ngưng công việc hiện tại* để thực hiện một đoạn chương trình con (interrupt handler, trình xử lí ngắt) làm những thao tác cần thiết (*đáp ứng*), sau đó quay trở lại *tiếp tục công việc*.

Ngắt mềm được dùng khi chương trình chủ động *gọi* một đoạn chương trình con hệ thống (interrupt handler, trình xử lí ngắt). Các chương trình con hệ thống là một phần của hệ điều hành hoặc của BIOS. Việc gọi này thực hiện không phải bằng lệnh CALL mà bằng lệnh INT. Có thể xem như lệnh INT đã tạo ra một *sự kiện* đòi hỏi *đáp ứng* của CPU.

Như vậy, cả hai loại ngắt cứng và ngắt mềm đều có liên quan đến việc *gọi một chương trình con khi có một sự kiện xảy ra*, chỉ khác nhau ở cách hình thành sự kiện này. Đối với ngắt cứng, một *tín hiệu* trên phần cứng thay đổi gây ra sự kiện. Đối với ngắt mềm, *lệnh INT* gây ra sự kiện.

Địa chỉ bắt đầu của các chương trình con (trình xử lí ngắt) này được lưu trong một bảng, gọi là bảng vector ngắt (interrupt vector table). Mỗi ngắt có một số hiệu để phân biệt. Với mỗi số hiệu ngắt, trong bảng lưu giữ địa chỉ xa trỏ đến *lệnh đầu tiên của trình xử lí ngắt* tương ứng.

Nhưng không giống như việc gọi chương trình con bằng CALL thông thường, ở đây giá trị của 3 thanh ghi được PUSH vào stack. Đầu tiên là thanh ghi cờ, sau đó là CS và cuối cùng là IP. Tiếp theo, dựa vào số hiệu của ngắt (được cung cấp bởi phần cứng hoặc bởi đối số của lệnh INT), địa chỉ xa của trình xử lí ngắt được lấy từ bảng vector ngắt và đặt vào CS, IP.

Để có thể trở về chương trình chính, ở cuối trình xử lí ngắt có lệnh IRET. Lệnh này theo thứ tự ngược lại, POP giá trị từ stack vào IP, CS và thanh ghi cờ.

Ngoài ra, còn một loại ngắt nữa, đó là ngắt nội bộ bên trong CPU (internal interrupt), xảy ra khi có một lỗi đặc biệt, ví dụ, thực hiện phép chia cho 0, hoặc thực hiện một mã lệnh không tồn tại. Cơ chế gọi và trở về từ ngắt hoàn toàn giống như hai loại trên.

Một số ngắt có thể bị che, nghĩa là không cho phép chúng xảy ra, bằng cách thay đổi một số bit trong một thanh ghi điều khiển. Các ngắt như vậy gọi là maskable. Các ngắt không cho phép che gọi là non-maskable.