University of Virginia Computer Science
CS216: Programma e rappresentazione dei dati, primavera 2006 08 marzo 2022

x86 Guida all'assemblaggio

Contenuto: Registri | Memoria e indirizzamento | Istruzioni | Convenzione di chiamata

Questa guida descrive le basi della programmazione in linguaggio assembly x86 a 32 bit, coprendo un piccolo ma utile sottoinsieme delle istruzioni disponibili e delle direttive assembler. Esistono diversi linguaggi assembly per la generazione di codice macchina x86. Quello che useremo in CS216 è l'assembler Microsoft Macro Assembler (MASM). MASM utilizza la sintassi Intel standard per la scrittura del codice assembly x86.

Il set di istruzioni x86 completo è ampio e complesso (i manuali del set di istruzioni x86 di Intel comprendono oltre 2900 pagine) e non lo copriamo tutto in questa guida. Ad esempio, esiste un sottoinsieme di 16 bit del set di istruzioni x86. L'utilizzo del modello di programmazione a 16 bit può essere piuttosto complesso. Ha un modello di memoria segmentato, maggiori restrizioni sull'utilizzo del registro e così via. In questa guida, limiteremo la nostra attenzione agli aspetti più moderni della programmazione x86 e approfondiremo il set di istruzioni solo in modo sufficientemente dettagliato per avere un'idea di base della programmazione x86.

Risorse

- Guida all'uso dell'assembly in Visual Studio : un'esercitazione sulla creazione e il debug del codice assembly in Visual Studio
- Riferimento al set di istruzioni Intel x86
- Manuali Pentium di Intel (tutti i dettagli cruenti)

Registri

I moderni processori x86 (cioè 386 e oltre) hanno otto registri per uso generico a 32 bit, come illustrato nella Figura 1. I nomi dei registri sono per lo più storici. Ad esempio, `EAX` era chiamato l'accumulatore poiché era utilizzato da una serie di operazioni aritmetiche, ed `ECX` era noto come contatore poiché era utilizzato per contenere un indice di loop. Mentre la maggior parte dei registri ha perso i propri scopi speciali nel moderno set di istruzioni, per convenzione, due sono riservati a scopi speciali: il puntatore dello stack (ESP) e il puntatore della base (`EBP`).

Per i registri `EAX` , `EBX` , `ECX` e `EDX` possono essere utilizzate le sottosezioni. Ad esempio, i 2 byte meno significativi di `EAX` possono essere trattati come un registro a 16 bit chiamato `AX` . Il byte meno significativo di `AX` può essere utilizzato come un singolo registro a 8 bit denominato `AL` , mentre il byte più significativo di `AX` può essere utilizzato come un singolo registro a 8 bit denominato `AH` . Questi nomi si riferiscono allo stesso registro fisico. Quando una quantità di due byte viene inserita in `DX` , l'aggiornamento influisce sul valore di `DH` , `DL` e `EDX`. Questi sottoregistri sono principalmente residui di versioni precedenti a 16 bit del set di istruzioni. Tuttavia, a volte sono convenienti quando si tratta di dati più piccoli di 32 bit (ad esempio caratteri ASCII a 1 byte).

Quando si fa riferimento ai registri in linguaggio assembly, i nomi non fanno distinzione tra maiuscole e minuscole. Ad esempio, i nomi `EAX` e `eax` si riferiscono allo stesso registro.

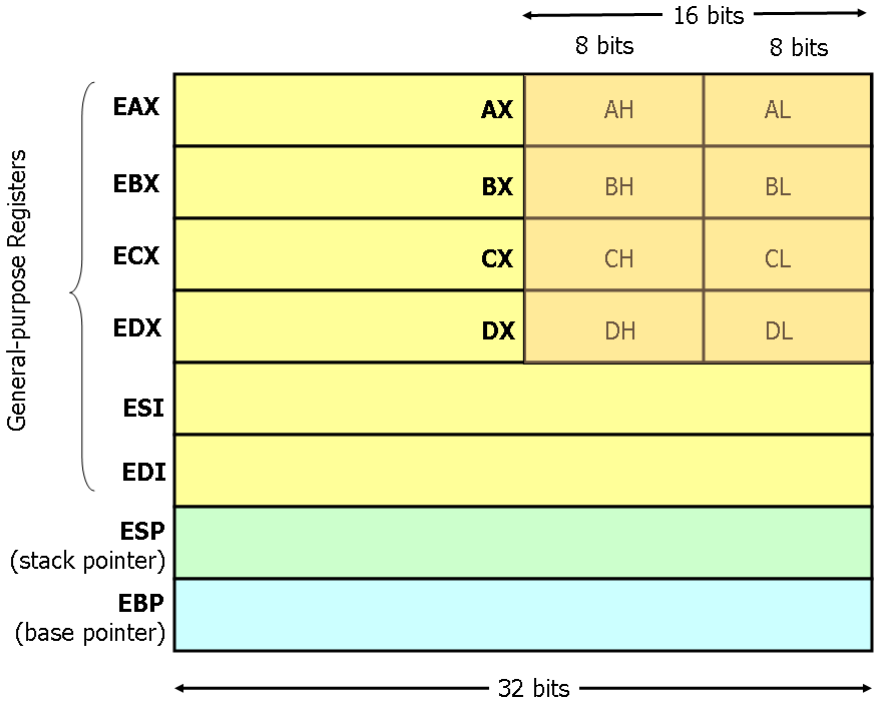


Figura 1. Registri x86

Memoria e modalità di indirizzamento

Dichiarazione di regioni dati statiche

È possibile dichiarare aree dati statiche (analoghe alle variabili globali) nell'assembly x86 utilizzando direttive assembler speciali per questo scopo. Le dichiarazioni di dati devono essere precedute dalla direttiva .DATA . Seguendo questa direttiva, le direttive DB , DW e DD possono essere utilizzate per dichiarare rispettivamente posizioni di dati a uno, due e quattro byte. Le posizioni dichiarate possono essere etichettate con nomi per riferimenti successivi: è simile alla dichiarazione di variabili per nome, ma rispetta alcune regole di livello inferiore. Ad esempio, le posizioni dichiarate in sequenza verranno posizionate in memoria l'una accanto all'altra.

Esempi di dichiarazioni:

```
.DATI
era     DB 64          ; Dichiara un byte, denominato location var , contenente il
                       valore 64.
var2    DB?            ; Dichiara un byte non inizializzato, denominato location var2 .
        DB 10          ; Dichiara un byte senza etichetta, contenente il valore 10. La
                       sua posizione è var2 + 1.
X       DW?            ; Dichiara un valore non inizializzato a 2 byte, denominato
                       posizione X .
Y       GG 30000       ; Dichiara un valore di 4 byte, indicato come posizione Y ,
                       inizializzato a 30000.
```

A differenza dei linguaggi di alto livello in cui gli array possono avere molte dimensioni e sono accessibili dagli indici, gli array nel linguaggio assembly x86 sono semplicemente un numero di celle situate in modo contiguo nella memoria. Un array può essere dichiarato semplicemente elencando i valori, come nel primo esempio di seguito. Altri due metodi comuni usati per dichiarare array di dati sono la direttiva DUP e l'uso di stringhe letterali. La direttiva DUP dice all'assembler di duplicare un'espressione un determinato numero di volte. Ad esempio, 4 DUP(2) equivale a 2, 2, 2, 2 .

Qualche esempio:

```
z       GG 1, 2, 3      ; Dichiara tre valori a 4 byte, inizializzati su 1, 2 e 3. Il
                          valore della posizione Z + 8 sarà 3.
byte    DB10 DUP(?)     ; Dichiara 10 byte non inizializzati a partire da location bytes
                          .
arr     DD 100          ; Dichiara 100 parole di 4 byte a partire dalla posizione arr ,
        DUP(0)            tutte inizializzate su 0
str     DB 'ciao',0     ; Dichiara 6 byte a partire dall'indirizzo str, inizializzato con
                          i valori dei caratteri ASCII per ciao e il byte null ( 0 ).
```

Memoria di indirizzamento

I moderni processori compatibili con x86 sono in grado di indirizzare fino a 2^{32} byte di memoria: gli indirizzi di memoria sono larghi 32 bit. Negli esempi precedenti, in cui abbiamo usato le etichette per fare riferimento alle regioni di memoria, queste etichette sono effettivamente sostituite dall'assembler con quantità a 32 bit che specificano gli indirizzi in memoria. Oltre a supportare il riferimento alle regioni di memoria tramite etichette (ovvero valori costanti), x86 fornisce uno schema flessibile per il calcolo e il riferimento agli indirizzi di memoria: è possibile sommare fino a due dei registri a 32 bit e una costante con segno a 32 bit per calcolare un indirizzo di memoria. Uno dei registri può essere facoltativamente premoltiplicato per 2, 4 o 8.

Le modalità di indirizzamento possono essere utilizzate con molte istruzioni x86 (le descriveremo nella prossima sezione). Di seguito illustriamo alcuni esempi utilizzando l' istruzione mov che sposta i dati tra i registri e la memoria. Questa istruzione ha due operandi: il primo è la destinazione e il secondo specifica la sorgente.

Alcuni esempi di istruzioni mov che utilizzano i calcoli degli indirizzi sono:

```
mov eax, [ebx]          ; Spostare i 4 byte in memoria all'indirizzo contenuto in EBX
                          in EAX
mov [var], ebx          ; Sposta il contenuto di EBX nei 4 byte all'indirizzo di
                          memoria var . (Nota, var è una costante a 32 bit).
mov eax,
[anteriore-4]           ; Sposta 4 byte all'indirizzo di memoria ESI + (-4) in EAX
mov [esi+eax], cl       ; Spostare il contenuto di CL nel byte all'indirizzo ESI+EAX
mov edx,
[esi+4*ebx]             ; Spostare i 4 byte di dati all'indirizzo ESI+4*EBX in EDX
```

Alcuni esempi di calcoli di indirizzi non validi includono:

```
mov eax, [ebx-ecx]          ; Può solo aggiungere valori di registro
mov [eax+esi+edi], ebx      ; Al massimo 2 registri nel calcolo degli indirizzi
```

Direttive sulle dimensioni

In generale, la dimensione prevista dell'elemento di dati in un dato indirizzo di memoria può essere dedotta dall'istruzione del codice assembly in cui viene fatto riferimento. Ad esempio, in tutte le istruzioni precedenti, la dimensione delle regioni di memoria potrebbe essere dedotta dalla dimensione dell'operando del registro. Quando stavamo caricando un registro a 32 bit, l'assembler poteva dedurre che la regione di memoria a cui ci riferivamo era larga 4 byte. Quando stavamo memorizzando il valore di un registro di un byte in memoria, l'assemblatore poteva dedurre che volevamo che l'indirizzo facesse riferimento a un singolo byte in memoria.

Tuttavia, in alcuni casi la dimensione di una regione di memoria a cui si fa riferimento è ambigua. Consideriamo l'istruzione mov [ebx], 2 . Questa istruzione dovrebbe spostare il valore 2 nel singolo byte all'indirizzo EBX ? Forse dovrebbe spostare la rappresentazione

intera a 32 bit di 2 nei 4 byte a partire dall'indirizzo EBX . Poiché una delle due è una possibile interpretazione valida, l'assemblatore deve essere esplicitamente indirizzato su quale sia corretta. Le direttive di dimensione BYTE PTR , WORD PTR e DWORD PTR servono a questo scopo, indicando dimensioni rispettivamente di 1, 2 e 4 byte.

Per esempio:

```
mov BYTE PTR [ebx], 2     ; Spostare 2 nel singolo byte all'indirizzo memorizzato in EBX.
mov PAROLA PTR [ebx], 2   ; Sposta la rappresentazione intera a 16 bit di 2 nei 2 byte a partire dall'indirizzo in EBX.
mov DWORD PTR [ebx], 2    ; Sposta la rappresentazione intera a 32 bit di 2 nei 4 byte a partire dall'indirizzo in EBX.
```

Istruzioni

Le istruzioni macchina generalmente rientrano in tre categorie: movimento dati, aritmetica/logica e flusso di controllo. In questa sezione, esamineremo importanti esempi di istruzioni x86 di ciascuna categoria. Questa sezione non deve essere considerata un elenco esaustivo di istruzioni x86, ma piuttosto un utile sottoinsieme. Per un elenco completo, vedere il riferimento al set di istruzioni di Intel.

Usiamo la seguente notazione:

<reg32>	Qualsiasi registro a 32 bit (EAX , EBX , ECX , EDX , ESI , EDI , ESP o EBP)
<reg16>	Qualsiasi registro a 16 bit (AX , BX , CX o DX)
<reg8>	Qualsiasi registro a 8 bit (AH , BH , CH , DH , AL , BL , CL o DL)
<reg>	Qualsiasi registro
<mem>	Un indirizzo di memoria (ad esempio, [eax] , [var + 4] o dword ptr [eax+ebx])
<con32>	Qualsiasi costante a 32 bit
<con16>	Qualsiasi costante a 16 bit
<con8>	Qualsiasi costante a 8 bit
<con>	Qualsiasi costante a 8, 16 o 32 bit

Istruzioni per il movimento dei dati

mov - Sposta (codici operativi: 88, 89, 8A, 8B, 8C, 8E, ...)

L' istruzione mov copia l'elemento di dati a cui fa riferimento il suo secondo operando (cioè il contenuto del registro, il contenuto della memoria o un valore costante) nella posizione a cui fa riferimento il suo primo operando (cioè un registro o una memoria). Mentre gli spostamenti da registro a registro sono possibili, gli spostamenti diretti da memoria a memoria non lo sono. Nei casi in cui si desiderano trasferimenti di memoria, il contenuto della memoria di origine deve essere prima caricato in un registro, quindi può essere memorizzato nell'indirizzo di memoria di destinazione.

Sintassi
mov <reg>,<reg>
mov <reg>,<mem>
mov <mem>,<reg>
mov <reg>,<const>
mov <mem>,<const>

Esempi
mov eax, ebx — copia il valore in ebx in eax
mov byte ptr [var], 5 — memorizza il valore 5 nel byte nella posizione var

push — Push stack (Codici operativi: FF, 89, 8A, 8B, 8C, 8E, ...)

L' istruzione push posiziona il suo operando in cima allo stack supportato dall'hardware in memoria. Nello specifico, push prima decrementa ESP di 4, quindi inserisce il suo operando nel contenuto della posizione a 32 bit all'indirizzo [ESP]. ESP (il puntatore dello stack) viene decrementato da push poiché lo stack x86 cresce verso il basso, ovvero lo stack cresce da indirizzi alti a indirizzi inferiori.

Sintassi
push <reg32>
push <mem>
push <con32>

Esempi
push eax — inserisci eax nello stack
push [var] — inserisci i 4 byte all'indirizzo *var* nello stack

pop — Pop stack

L' istruzione pop rimuove l'elemento di dati a 4 byte dalla parte superiore dello stack supportato dall'hardware nell'operando specificato (ovvero registro o posizione di memoria). Prima sposta i 4 byte situati nella posizione di memoria [SP] nel registro o nella posizione di memoria specificata, quindi incrementa SP di 4.

Sintassi
pop <reg32>
pop <mem>

Esempi
pop edi : inserisci l'elemento in cima allo stack in EDI.
pop [ebx] — inserisce l'elemento superiore dello stack in memoria nei quattro byte a partire dalla posizione EBX.

lea — Carica l'indirizzo effettivo

L' istruzione lea inserisce l' *indirizzo* specificato dal suo secondo operando nel registro specificato dal suo primo operando. Si noti che il *contenuto* della posizione di memoria non viene caricato, solo l'indirizzo effettivo viene calcolato e inserito nel registro. Questo è utile per ottenere un puntatore in una regione di memoria.

Sintassi
lea <reg32>,<mem>

Esempi
lea edi, [ebx+4*esi] — la quantità EBX+4*ESI viene inserita in EDI.
lea eax, [var] — il valore in *var* viene inserito in EAX.
lea eax, [val] — il valore *val* viene inserito in EAX.

Istruzioni aritmetiche e logiche

add — Addizione di interi

L' istruzione add somma i suoi due operandi, memorizzando il risultato nel suo primo operando. Si noti che mentre entrambi gli operandi possono essere registri, al massimo un operando può essere una locazione di memoria.

Sintassi
add <reg>,<reg>
add <reg>,<mem>
add <mem>,<reg>
add <reg>,<con>
add <mem>,<con>

Esempi
add eax, 10 — EAX ← EAX + 10
add BYTE PTR [var], 10 — aggiunge 10 al singolo byte memorizzato all'indirizzo di memoria var

sub — Sottrazione di numeri interi

L' istruzione sub memorizza nel valore del suo primo operando il risultato della sottrazione del valore del suo secondo operando dal valore del suo primo operando. Come con aggiungi

Sintassi
sub <reg>,<reg>
sub <reg>,<mem>
sub <mem>,<reg>
sub <reg>,<con>
sub <mem>,<con>

Esempi
sub al, ah — AL ← AL - AH
sub eax, 216 — sottrarre 216 dal valore memorizzato in EAX

inc, dec — Incremento, Decremento

L' istruzione inc incrementa di uno il contenuto del suo operando. L' istruzione dec decrementa di uno il contenuto del suo operando.

Sintassi
inc <reg>
inc <mem>
dec <reg>
dec <mem>

Esempi
dec eax — sottrai uno dal contenuto di EAX.
inc DWORD PTR [var] — aggiunge uno al numero intero a 32 bit memorizzato nella posizione *var*

imul - Moltiplicazione di interi

L' istruzione imul ha due formati di base: due operandi (i primi due elenchi di sintassi sopra) e tre operandi (gli ultimi due elenchi di sintassi sopra).

La forma a due operandi moltiplica i suoi due operandi e memorizza il risultato nel primo operando. Il risultato (cioè il primo) operando deve essere un registro.

La forma a tre operandi moltiplica insieme il secondo e il terzo operando e memorizza il risultato nel primo operando. Di nuovo, l'operando risultato deve

essere un registro. Inoltre, il terzo operando è limitato ad essere un valore costante.

Sintassi
imul <reg32>,<reg32>
imul <reg32>,<mem>
imul <reg32>,<reg32>,<con>
imul <reg32>,<mem>,<con>

Esempi

imul eax, [var] — moltiplica il contenuto di EAX per il contenuto a 32 bit della posizione di memoria *var* . Memorizza il risultato in EAX.
imul esi, edi, 25 — ESI → EDI * 25

idiv — Divisione intera

L' istruzione idiv divide il contenuto dell'intero a 64 bit EDX:EAX (costruito visualizzando EDX come i quattro byte più significativi ed EAX come i quattro byte meno significativi) per il valore dell'operando specificato. Il risultato del quoziente della divisione viene memorizzato in EAX, mentre il resto viene inserito in EDX.

Sintassi
idiv <reg32>
idiv <mem>

Esempi

idiv ebx — divide il contenuto di EDX:EAX per il contenuto di EBX. Metti il quoziente in EAX e il resto in EDX.
idiv DWORD PTR [var] — divide il contenuto di EDX:EAX per il valore a 32 bit archiviato nella posizione di memoria *var* . Metti il quoziente in EAX e il resto in EDX.

and, or, xor — Bitwise logico e, or ed esclusivo or

Queste istruzioni eseguono l'operazione logica specificata (rispettivamente logico bit per bit e or e esclusivo or) sui relativi operandi, posizionando il risultato nella posizione del primo operando.

Sintassi
e <reg>,<reg>
e <reg>,<mem>
e <mem>,<reg>
e <reg>,<con>
e <mem>,<con>

o <reg>,<reg>
o <reg>,<mem>
o <mem>,<reg>
o <reg>,<con>
o <mem>,<con>

xor <reg>,<reg>
xor <reg>,<mem>
xor <mem>,<reg>
xor <reg>,<con>
xor <mem>,<con>

Esempi
e eax, 0fH — cancella tutto tranne gli ultimi 4 bit di EAX.
xor edx, edx — imposta il contenuto di EDX su zero.

not — Bitwise Logical Not

Nega logicamente il contenuto dell'operando (ovvero, capovolge tutti i valori di bit nell'operando).

Sintassi
not <reg>
not <mem>

Esempio
not BYTE PTR [var] — nega tutti i bit nel byte nella posizione di memoria *var* .

neg — negare

Esegue la negazione in complemento a due del contenuto dell'operando.

Sintassi
neg <reg>
neg <mem>

Esempio
neg eax — EAX → - EAX

shl, shr — Maiusc a sinistra, Maiusc a destra

Queste istruzioni spostano i bit nel contenuto del loro primo operando a sinistra ea destra, riempiendo di zeri le posizioni di bit vuote risultanti. L'operando spostato può essere spostato fino a 31 posizioni. Il numero di bit da spostare è specificato dal secondo operando, che può essere una costante a 8 bit o il registro CL. In entrambi i casi, vengono eseguiti conteggi di turni maggiori di 31 modulo 32.

Sintassi
shl <reg>,<con8>
shl <mem>,<con8>
shl <reg>,<cl>
shl <mem>,<cl>

shr <reg>,<con8>
shr <mem>,<con8>
shr <reg>,<cl>
shr <mem>,<cl>

Esempi

shl eax, 1 — Moltiplica il valore di EAX per 2 (se il bit più significativo è 0)
shr ebx, cl — Memorizza in EBX il floor del risultato della divisione del valore di EBX per 2^n dove *n* è il valore in CL.

Istruzioni di flusso di controllo

Il processore x86 mantiene un registro IP (Instruction Pointer) che è un valore a 32 bit che indica la posizione in memoria in cui inizia l'istruzione corrente. Normalmente, si incrementa per puntare all'istruzione successiva in memoria che inizia dopo l'esecuzione di un'istruzione.

Il registro IP non può essere manipolato direttamente, ma viene aggiornato implicitamente dalle istruzioni del flusso di controllo fornite.

Usiamo la notazione <label> per fare riferimento a posizioni etichettate nel testo del programma. Le etichette possono essere inserite ovunque nel testo del codice assembly x86 immettendo un nome di etichetta seguito da due punti. Per esempio,

```
       mov esi, [ebp+8]
inizio: xor ecx, ecx
       mov eax, [esi]
```

La seconda istruzione in questo frammento di codice è etichettata begin . In altre parti del codice, possiamo fare riferimento alla posizione di memoria in cui si trova questa istruzione utilizzando il nome simbolico più conveniente begin . Questa etichetta è solo un modo conveniente per esprimere la posizione invece del suo valore a 32 bit.

jmp — Salta

Trasferisce il flusso di controllo del programma all'istruzione nella posizione di memoria indicata dall'operando.

Sintassi
jmp <etichetta>

Esempio
jmp begin — Salta all'istruzione etichettata begin .

***condizione* j** — Salto condizionato

Queste istruzioni sono salti condizionali basati sullo stato di un insieme di codici condizionali memorizzati in un registro speciale chiamato *parola di stato macchina* . Il contenuto della parola di stato della macchina include informazioni sull'ultima operazione aritmetica eseguita. Ad esempio, un bit di questa parola indica se l'ultimo risultato era zero. Un altro indica se l'ultimo risultato è stato negativo. Sulla base di questi codici di condizione, è possibile eseguire una serie di salti condizionati. Ad esempio, l' istruzione jz esegue un salto all'etichetta dell'operando specificato se il risultato dell'ultima operazione aritmetica era zero. In caso contrario, il controllo procede all'istruzione successiva in sequenza.

A un certo numero di rami condizionali vengono dati nomi che sono intuitivamente basati sull'ultima operazione eseguita essendo un'istruzione di confronto speciale, cmp (vedi sotto). Ad esempio, i rami condizionali come jle e jne si basano sull'esecuzione di un'operazione cmp sugli operandi desiderati.

Sintassi
je <label> (salta quando è uguale)
jne <label> (salta quando non è uguale)
jz <label> (salta quando l'ultimo risultato era zero)
jg <label> (salta quando è maggiore di)
jge <label> (salta quando è maggiore di o uguale a)
jl <label> (salta quando è minore di)
jle <label> (salta quando è minore o uguale a)

Esempio
cmp eax, ebx
jle done

Se il contenuto di EAX è minore o uguale al contenuto di EBX, passa all'etichetta *done* . In caso contrario, passare all'istruzione successiva.

cmp — Confronta

Confrontare i valori dei due operandi specificati, impostando opportunamente i codici di condizione nella parola di stato della macchina. Questa istruzione è equivalente all'istruzione sub , tranne per il fatto che il risultato della sottrazione viene scartato invece di sostituire il primo operando.

Sintassi
cmp <reg>,<reg>
cmp <reg>,<mem>
cmp <mem>,<reg>
cmp <reg>,<con>

Esempio
cmp DWORD PTR [var],
ciclo 10 jeq

Se i 4 byte memorizzati nella posizione *var* sono uguali alla costante intera a 4 byte 10, passa alla posizione etichettata *loop* .

call , **ret** — Subroutine chiamata e ritorno

Queste istruzioni implementano una subroutine chiamata e ritorno. L' istruzione di chiamata inserisce prima la posizione del codice corrente nello stack supportato dall'hardware in memoria (vedere l' istruzione push per i dettagli), quindi esegue un salto incondizionato alla posizione del codice indicata dall'operando dell'etichetta. A differenza delle semplici istruzioni di salto, l' istruzione di chiamata salva la posizione a cui tornare al termine della subroutine.

L' istruzione ret implementa un meccanismo di ritorno di subroutine. Questa istruzione estrae prima una posizione di codice dallo stack in memoria supportato dall'hardware (vedere l' istruzione pop per i dettagli). Quindi esegue un salto incondizionato alla posizione del codice recuperato.

Sintassi
chiamata <etichetta>
ret

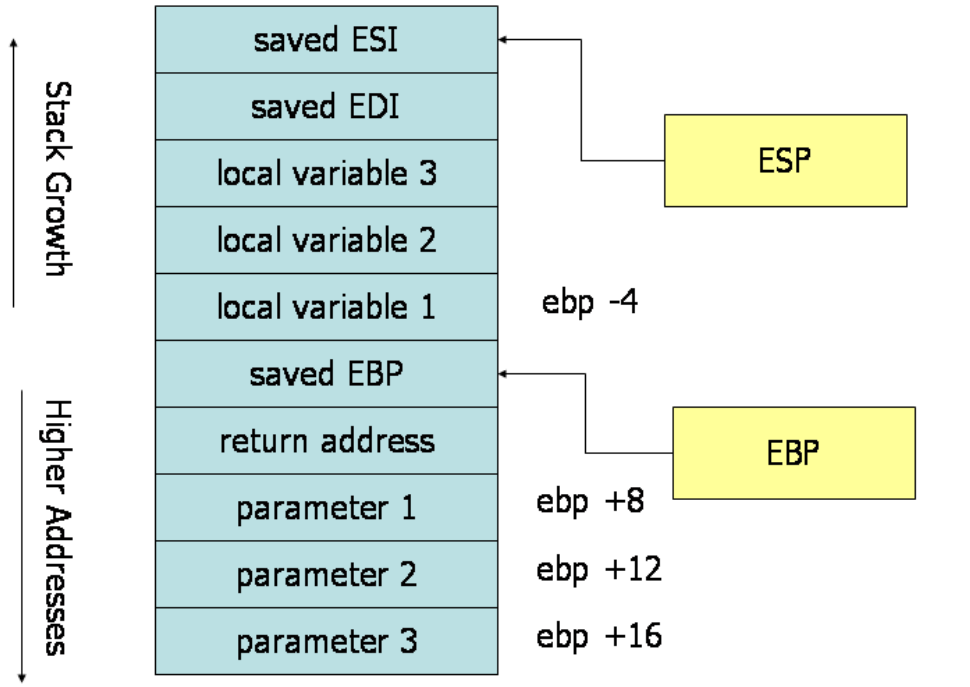
Convenzione di chiamata

Per consentire a programmatori separati di condividere il codice e sviluppare librerie utilizzabili da molti programmi e per semplificare l'uso delle subroutine in generale, i programmatori adottano tipicamente una *convenzione di chiamata* comune . La convenzione di chiamata è un protocollo su come chiamare e restituire dalle routine. Ad esempio, dato un insieme di regole di convenzione di chiamata, un programmatore non ha bisogno di esaminare la definizione di una subroutine per determinare come devono essere passati i parametri a quella subroutine. Inoltre, dato un insieme di regole di convenzione di chiamata, è possibile fare in modo che i compilatori di linguaggi di alto livello seguano le regole, consentendo così alle routine di linguaggio assembly codificate a mano e alle routine di linguaggio di alto livello di chiamarsi a vicenda.

In pratica, sono possibili molte convenzioni di chiamata. Useremo la convenzione di chiamata del linguaggio C ampiamente usata. Seguire questa convenzione ti consentirà di scrivere subroutine in linguaggio assembly che possono essere richiamate in modo sicuro dal codice C (e C++) e ti consentirà anche di chiamare le funzioni della libreria C dal tuo codice in linguaggio assembly.

La convenzione di chiamata C si basa fortemente sull'uso dello stack supportato dall'hardware. Si basa sulle istruzioni push , pop , call e ret . I parametri della subroutine vengono passati allo stack. I registri vengono salvati nello stack e le variabili locali utilizzate dalle subroutine vengono poste in memoria nello stack. La stragrande maggioranza dei linguaggi procedurali di alto livello implementati sulla maggior parte dei processori ha utilizzato convenzioni di chiamata simili.

La convenzione di chiamata è suddivisa in due serie di regole. Il primo insieme di regole è impiegato dal chiamante della subroutine, e il secondo insieme di regole è osservato dall'autore della subroutine (il chiamato). Va sottolineato che gli errori nell'osservanza di queste regole si traducono rapidamente in errori fatali del programma poiché lo stack verrà lasciato in uno stato incoerente; pertanto è necessario prestare particolare attenzione quando si implementa la convenzione di chiamata nelle proprie subroutine.



Stack durante Subroutine Call

[Grazie a Maxence Faldor per aver fornito una figura corretta e a James Peterson per aver trovato e corretto il bug nella versione originale di questa figura!]

Un buon modo per visualizzare il funzionamento della convenzione di chiamata consiste nel disegnare il contenuto della regione vicina dello stack durante l'esecuzione della subroutine. L'immagine in alto mostra il contenuto dello stack durante l'esecuzione di una subroutine con tre parametri e tre variabili locali. Le celle rappresentate nello stack sono posizioni di memoria larghe 32 bit, quindi gli indirizzi di memoria delle celle sono distanti 4 byte. Il primo parametro risiede a un offset di 8 byte dal puntatore di base. Sopra i parametri nello stack (e sotto il puntatore di base), l' istruzione di chiamata ha posizionato l'indirizzo di ritorno, portando così a 4 byte extra di offset dal puntatore di base al primo parametro. Quando il retviene utilizzata per tornare dalla subroutine, salterà all'indirizzo di ritorno memorizzato nello stack.

Regole del chiamante

Per effettuare una chiamata di subrouting, il chiamante deve:

1. Prima di chiamare una subroutine, il chiamante dovrebbe salvare il contenuto di alcuni registri designati come *salvato dal chiamante* . I registri salvati dal chiamante sono EAX, ECX, EDX. Poiché la subroutine chiamata può modificare questi registri, se il chiamante fa affidamento sui loro valori dopo il ritorno della subroutine, il chiamante

deve inserire i valori in questi registri nello stack (in modo che possano essere ripristinati dopo il ritorno della subroutine.
2. Per passare i parametri alla subroutine, inseriscili nello stack prima della chiamata. I parametri devono essere inseriti in ordine inverso (cioè l'ultimo parametro per primo). Poiché lo stack si riduce, il primo parametro verrà archiviato all'indirizzo più basso (questa inversione di parametri è stata storicamente utilizzata per consentire alle funzioni di passare un numero variabile di parametri).
3. Per chiamare la subroutine, utilizzare l' istruzione di chiamata . Questa istruzione posiziona l'indirizzo di ritorno in cima ai parametri nello stack e passa al codice della subroutine. Questo richiama la subroutine, che dovrebbe seguire le regole del chiamato di seguito.

Dopo il ritorno della subroutine (immediatamente dopo l' istruzione di chiamata), il chiamante può aspettarsi di trovare il valore di ritorno della subroutine nel registro EAX. Per ripristinare lo stato della macchina, il chiamante deve:

1. Rimuovi i parametri dallo stack. Ciò ripristina lo stack allo stato precedente all'esecuzione della chiamata.
2. Ripristina il contenuto dei registri salvati dal chiamante (EAX, ECX, EDX) estraendoli dallo stack. Il chiamante può presumere che nessun altro registro sia stato modificato dalla subroutine.

Esempio
Il codice seguente mostra una chiamata di funzione che segue le regole del chiamante. Il chiamante chiama una funzione *_myFunc* che accetta tre parametri interi. Il primo parametro è in EAX, il secondo parametro è la costante 216; il terzo parametro è nella posizione di memoria *var* .

```
spingere [var] ; Premi l'ultimo parametro prima
premi 216 ; Premi il secondo parametro
push eax ; Push primo parametro ultima

chiamata _myFunc ; Chiama la funzione (assumendo la denominazione C)

add esp, 12
```

Si noti che dopo il ritorno della chiamata, il chiamante ripulisce lo stack utilizzando l' istruzione add . Abbiamo 12 byte (3 parametri * 4 byte ciascuno) nello stack e lo stack si riduce. Pertanto, per eliminare i parametri, possiamo semplicemente aggiungere 12 al puntatore dello stack.

Il risultato prodotto da *_myFunc* è ora disponibile per l'uso nel registro EAX. I valori dei registri salvati dal chiamante (ECX e EDX) potrebbero essere stati modificati. Se il chiamante li utilizza dopo la chiamata, avrebbe dovuto salvarli nello stack prima della chiamata e ripristinarli dopo.

Regole del chiamato

La definizione della subroutine dovrebbe rispettare le seguenti regole all'inizio della subroutine:

1. Inserisci il valore di EBP nello stack, quindi copia il valore di ESP in EBP utilizzando le seguenti istruzioni:

   ```
   push ebp
   mov ebp, esp
   ```

   Questa azione iniziale mantiene il *puntatore di base* , EBP. Il puntatore di base viene utilizzato per convenzione come punto di riferimento per la ricerca di parametri e variabili locali nello stack. Quando una subroutine è in esecuzione, il puntatore di base

contiene una copia del valore del puntatore dello stack da quando la subroutine ha iniziato l'esecuzione. I parametri e le variabili locali si troveranno sempre a offset noti e costanti rispetto al valore del puntatore di base. Inseriamo il vecchio valore del puntatore di base all'inizio della subroutine in modo da poter successivamente ripristinare il valore del puntatore di base appropriato per il chiamante quando la subroutine ritorna. Ricorda, il chiamante non si aspetta che la subroutine modifichi il valore del puntatore di base. Quindi spostiamo il puntatore dello stack in EBP per ottenere il nostro punto di riferimento per l'accesso ai parametri e alle variabili locali.
2. Successivamente, alloca le variabili locali facendo spazio nello stack. Ricorda, lo stack cresce verso il basso, quindi per fare spazio in cima allo stack, il puntatore dello stack dovrebbe essere decrementato. La quantità di cui viene decrementato il puntatore dello stack dipende dal numero e dalla dimensione delle variabili locali necessarie. Ad esempio, se fossero richiesti 3 interi locali (4 byte ciascuno), il puntatore dello stack dovrebbe essere decrementato di 12 per fare spazio a queste variabili locali (cioè, sub esp, 12). Come con i parametri, le variabili locali saranno posizionate a offset noti dal puntatore base.

3. Successivamente, salva i valori dei registri *salvati dal chiamato* che verranno utilizzati dalla funzione. Per salvare i registri, inseriscili nello stack. I registri salvati dal chiamato sono EBX, EDI ed ESI (anche ESP ed EBP verranno preservati dalla convenzione di chiamata, ma non è necessario inserirli nello stack durante questo passaggio).

Dopo che queste tre azioni sono state eseguite, il corpo della subroutine può procedere. Quando la subroutine viene restituita, deve seguire questi passaggi:

1. Lasciare il valore restituito in EAX.

2. Ripristina i vecchi valori di tutti i registri salvati dal chiamato (EDI ed ESI) che sono stati modificati. I contenuti del registro vengono ripristinati estraendoli dallo stack. I registri dovrebbero essere estratti nell'ordine inverso rispetto a quello in cui sono stati premuti.
3. Dealloca le variabili locali. Il modo ovvio per farlo potrebbe essere quello di aggiungere il valore appropriato al puntatore dello stack (poiché lo spazio è stato allocato sottraendo la quantità necessaria dal puntatore dello stack). In pratica, un modo meno soggetto a errori per deallocare le variabili è spostare il valore nel puntatore base nel puntatore stack: mov esp, ebp . Funziona perché il puntatore di base contiene sempre il valore contenuto dal puntatore dello stack immediatamente prima dell'allocazione delle variabili locali.
4. Immediatamente prima di tornare, ripristina il valore del puntatore di base del chiamante estraendo EBP dallo stack. Ricordiamo che la prima cosa che abbiamo fatto entrando nella subroutine è stata spingere il puntatore base per salvare il suo vecchio valore.
5. Infine, torna al chiamante eseguendo un'istruzione ret . Questa istruzione troverà e rimuoverà l'indirizzo di ritorno appropriato dallo stack.

Si noti che le regole del chiamato si suddividono nettamente in due metà che sono fondamentalmente immagini speculari l'una dell'altra. La prima metà delle regole si applica all'inizio della funzione e si dice comunemente che definisca il *prologo* della funzione. La seconda metà delle regole si applica alla fine della funzione, e quindi si dice comunemente che definisca l' *epilogo* della funzione.

Esempio
Ecco un esempio di definizione di funzione che segue le regole del chiamato:

```
.486
.MODEL FLAT
.CODE
PUBLIC _myFunc
```

```
_myFunc PROC
  ; Subroutine Prologue
  push ebp     ; Save the old base pointer value.
  mov ebp, esp ; Set the new base pointer value.
  sub esp, 4   ; Make room for one 4-byte local variable.
  push edi     ; Save the values of registers that the function
  push esi     ; will modify. This function uses EDI and ESI.
  ; (no need to save EBX, EBP, or ESP)

  ; Subroutine Body
  mov eax, [ebp+8]   ; Move value of parameter 1 into EAX
  mov esi, [ebp+12]  ; Move value of parameter 2 into ESI
  mov edi, [ebp+16]  ; Move value of parameter 3 into EDI

  mov [ebp-4], edi ; Sposta EDI nella variabile locale
  add [ebp-4], esi ; Aggiungere ESI nella variabile locale
  add eax, [ebp-4] ; Aggiungi il contenuto della variabile locale
                   ; in EAX (risultato finale)

  ; Subroutine Epilogo
  pop esi ; Recupera valori registro
  pop edi
  mov esp, ebp ; Dealloca le variabili locali
  pop ebp ; Ripristina il valore del puntatore base del chiamante
  ret
_myFunc ENDP
END
```

Il prologo della subroutine esegue le azioni standard di salvare un'istantanea del puntatore dello stack in EBP (il puntatore di base), allocare variabili locali decrementando il puntatore dello stack e salvare i valori del registro nello stack.

Nel corpo della subroutine possiamo vedere l'uso del puntatore base. Sia i parametri che le variabili locali si trovano a offset costanti dal puntatore base per la durata dell'esecuzione delle subroutine. In particolare, notiamo che poiché i parametri sono stati posizionati nello stack prima che la subroutine fosse chiamata, essi si trovano sempre sotto il puntatore base (cioè agli indirizzi più alti) nello stack. Il primo parametro della subroutine si trova sempre nella locazione di memoria EBP + 8, il secondo in EBP + 12, il terzo in EBP + 16. Allo stesso modo, poiché le variabili locali sono allocate dopo che il puntatore base è stato impostato, risiedono sempre sopra il puntatore di base (cioè agli indirizzi più bassi) sullo stack. In particolare, la prima variabile locale si trova sempre a EBP - 4, la seconda a EBP - 8 e così via.

L'epilogo della funzione è fondamentalmente un'immagine speculare del prologo della funzione. I valori del registro del chiamante vengono recuperati dallo stack, le variabili locali vengono deallocate reimpostando il puntatore dello stack, il valore del puntatore base del chiamante viene recuperato e l'istruzione ret viene utilizzata per tornare alla posizione di codice appropriata nel chiamante.

Utilizzo di questi materiali

Questi materiali sono rilasciati sotto una licenza Creative Commons Attribuzione-Non commerciale-Condividi allo stesso modo 3.0 Stati Uniti . Siamo lieti quando le persone desiderano utilizzare o adattare i materiali del corso che abbiamo sviluppato e sei libero di riutilizzare e adattare questi materiali per scopi non commerciali (se desideri utilizzarli per scopi commerciali, contatta David Evans per maggiori informazioni). Se adatti o usi questi materiali, includi un credito come *"Adattato da materiali sviluppati per l'Università della Virginia cs216 da David Evans. Questa guida è stata rivista per cs216 da David Evans, basata su materiali originariamente creati da Adam Ferrari molti anni fa, e successivamente aggiornato da Alan Batson, Mike Lack e Anita Jones."*e un collegamento a questa pagina.

CS216: Programma e rappresentazione dei dati
Università della Virginia

David Evans
evans@cs.virginia.edu
Uso di questi materiali