
Guida all'assemblaggio x86

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 e delle direttive assembler disponibili. Esistono diversi linguaggi assembly per la generazione di codice macchina x86. Quello che utilizzeremo in CS216 è l'assembler Microsoft Macro Assembler (MASM). MASM utilizza la sintassi standard Intel per la scrittura di codice assembly x86.

L'intero set di istruzioni x86 è ampio e complesso (i manuali del set di istruzioni x86 di Intel comprendono oltre 2900 pagine) e non lo tratteremo interamente in questa guida. Ad esempio, esiste un sottoinsieme a 16 bit del set di istruzioni x86. L'utilizzo del modello di programmazione a 16 bit può essere piuttosto complesso. Presenta un modello di memoria segmentato, maggiori restrizioni sull'utilizzo dei registri 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 da fornire un'idea di base della programmazione x86.

Risorse

- Guida all'utilizzo dell'assembly in Visual Studio : un tutorial 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 processori x86 moderni (ovvero 386 e successivi) hanno otto registri a 32 bit per uso generale, come illustrato nella Figura 1. I nomi dei registri sono per lo più storici. Ad esempio, EAX era chiamato accumulatore perché veniva utilizzato da numerose operazioni aritmetiche, mentre ECX era noto come contatore perché veniva utilizzato per contenere un indice di ciclo. Mentre la maggior parte dei registri ha perso le sue funzioni specifiche nel set di istruzioni

moderno, per convenzione, due sono riservati a scopi speciali: il puntatore allo stack (ESP) e il puntatore alla base (EBP).

Per i registri EAX , EBX , ECX ed EDX , è possibile utilizzare delle sottosezioni. Ad esempio, i 2 byte meno significativi di EAX possono essere trattati come un registro a 16 bit denominato 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 ed EDX . Questi sottoregistri sono principalmente residui di vecchie versioni a 16 bit del set di istruzioni. Tuttavia, a volte risultano utili quando si gestiscono dati di dimensioni inferiori a 32 bit (ad esempio, caratteri ASCII da 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 ed eax si riferiscono allo stesso registro.

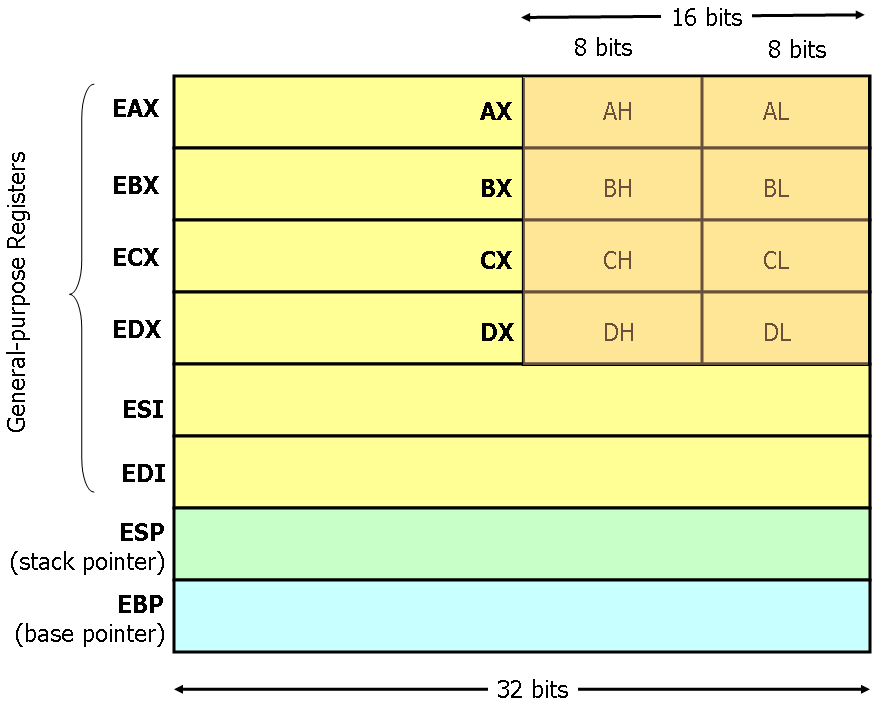


Figura 1. Registri x86

Modalità di memoria e indirizzamento

Dichiarazione di regioni dati statiche

È possibile dichiarare regioni dati statiche (analogamente alle variabili globali) in assembly x86 utilizzando apposite direttive assembler. Le dichiarazioni di dati

devono essere precedute dalla direttiva .DATA . Dopo questa direttiva, è possibile utilizzare le direttive DB , DW e DD per dichiarare rispettivamente posizioni dati a uno, due e quattro byte. Le posizioni dichiarate possono essere etichettate con nomi per riferimento futuro: questo è simile alla dichiarazione di variabili per nome, ma rispetta alcune regole di livello inferiore. Ad esempio, le posizioni dichiarate in sequenza saranno posizionate in memoria una accanto all'altra.

Dichiarazioni di esempio:

```
.DATI
varietà     DB 64       ; Dichiara un byte, denominato location var , contenente il valore 64.
var2        DB ?        ; Dichiara un byte non inizializzato, denominato posizione 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 di 2 byte, denominato posizione X.
E           GG 30000    ; Dichiara un valore di 4 byte, denominato posizione Y , inizializzato a 30000.
```

A differenza dei linguaggi di alto livello, dove gli array possono avere molte dimensioni e sono accessibili tramite indici, gli array nel linguaggio assembly x86 sono semplicemente un numero di celle contigue nella memoria. Un array può essere dichiarato semplicemente elencandone i valori, come nel primo esempio seguente. Altri due metodi comuni utilizzati per dichiarare array di dati

sono la direttiva DUP e l'uso di stringhe letterali. La direttiva DUP indica all'assembler di duplicare un'espressione un dato numero di volte. Ad esempio, 4 DUP(2) equivale a 2, 2, 2, 2 .

Alcuni esempi:

```
Z       DD 1, 2, 3          ; Dichiarare tre valori da 4 byte, inizializzati a 1, 2 e 3. Il valore della posizione Z + 8 sarà 3.
byte    DB 10 DUP(?)        ; Dichiara 10 byte non inizializzati a partire dalla posizione byte .
arr     DD 100 DUP(0)       ; Dichiara 100 parole da 4 byte a partire dalla posizione arr , tutte inizializzate a 0
str     DB 'ciao',0         ; Dichiara 6 byte a partire dall'indirizzo str, inizializzati ai valori dei caratteri ASCII per hello e al byte null ( 0 ).
```

Indirizzamento della memoria

I moderni processori compatibili con x86 sono in grado di indirizzare fino a ^{232} byte di memoria: gli indirizzi di memoria sono ampi 32 bit. Negli esempi precedenti, dove abbiamo utilizzato le etichette per fare riferimento alle regioni di memoria, queste etichette vengono in realtà 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: fino a due registri a 32 bit e una costante con segno a 32 bit possono essere sommati per calcolare un indirizzo di memoria. Uno dei registri può essere opzionalmente pre-moltiplicato per 2, 4 o 8.

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

Ecco alcuni esempi di istruzioni mov che utilizzano calcoli di indirizzi:

```
mov eax, [ebx]          ; Sposta 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, [esi-4]        ; Sposta 4 byte all'indirizzo di memoria ESI + (-4) in EAX

mov [esi+eax], cl       ; Sposta il contenuto di CL nel byte all'indirizzo ESI+EAX

mov edx, [esi+4*ebx]    ; Sposta i 4 byte di dati all'indirizzo ESI+4*EBX in EDX
```

Ecco alcuni esempi di calcoli di indirizzi non validi:

```
mov eax, [ebx-ecx]      ; È possibile aggiungere solo valori di registro

mov [eax+esi+edi], ebx  ; Al massimo 2 registri nel calcolo degli indirizzi
```

Direttive sulle dimensioni

In generale, la dimensione desiderata dell'elemento dati a un dato indirizzo di memoria può essere dedotta dall'istruzione del codice assembly in cui viene referenziato. Ad esempio, in tutte le istruzioni precedenti, la dimensione delle regioni di memoria potrebbe essere dedotta dalla dimensione dell'operando registro. Quando caricavamo un registro a 32 bit, l'assembler poteva dedurre che la regione di memoria a cui facevamo riferimento era larga 4 byte. Quando memorizzavamo il valore di un registro di un byte in memoria, l'assembler poteva dedurre che volevamo che l'indirizzo si riferisse a un singolo byte in memoria.

Tuttavia, in alcuni casi la dimensione di una regione di memoria a cui si fa riferimento è ambigua. Si consideri 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 entrambe le interpretazioni sono valide, l'assembler deve essere esplicitamente indicato su quale sia corretta. Le direttive di dimensione BYTE PTR , WORD PTR e DWORD PTR servono a questo scopo, indicando rispettivamente dimensioni di 1, 2 e 4 byte.

Per esempio:

```
mov BYTE PTR [ebx], 2   ; Sposta 2 nel singolo byte all'indirizzo memorizzato in EBX.
```

mov WORD PTR [ebx], 2	; Sposta la rappresentazione intera a 16 bit di 2 nei 2 byte che iniziano 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 rientrano generalmente in tre categorie: spostamento dati, aritmetica/logica e controllo del flusso. In questa sezione, esamineremo importanti esempi di istruzioni x86 per ciascuna categoria. Questa sezione non deve essere considerata un elenco esaustivo delle istruzioni x86, ma piuttosto un utile sottoinsieme. Per un elenco completo, consultare il riferimento ai set di istruzioni Intel.
Utilizziamo 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 lo spostamento dei dati

mov — Sposta (Opcode: 88, 89, 8A, 8B, 8C, 8E, ...)
L' istruzione mov copia l'elemento di dati a cui fa riferimento il suo secondo operando (ad esempio, il contenuto di un registro, il contenuto di una memoria o

un valore costante) nella posizione a cui fa riferimento il suo primo operando (ad esempio, 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 siano richiesti trasferimenti di memoria, il contenuto della memoria sorgente deve prima essere 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 (Opcode: 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 posiziona 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 dagli indirizzi più alti a quelli più bassi.
Sintassi
push <reg32>
push <mem>
push <con32>

Esempi
push eax — spinge eax sullo stack
push [var] — spinge i 4 byte all'indirizzo *var* sullo stack

pop — Pila di pop
L' istruzione pop rimuove l'elemento dati di 4 byte dalla cima dello stack supportato dall'hardware nell'operando specificato (ovvero registro o posizione di memoria). Innanzitutto sposta i 4 byte situati nella posizione di memoria [SP] nel registro o nella posizione di memoria specificati, quindi incrementa SP di 4.

Sintassi
pop <reg32>
pop <mem>

Esempi

pop edi — inserisce l'elemento in cima allo stack in EDI.
pop [ebx] — inserisce l'elemento in cima allo stack nella memoria a partire dai quattro byte a partire dalla posizione EBX.
lea — Carica 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 a 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

aggiungi — Addizione di numeri interi
L' istruzione add somma i suoi due operandi, memorizzando il risultato nel primo operando. Nota: sebbene entrambi gli operandi possano essere registri, al massimo uno può essere una locazione di memoria.
Sintassi
aggiungi <reg>,<reg>
aggiungi <reg>,<mem>
aggiungi <mem>,<reg>
aggiungi <reg>,<con>
aggiungi <mem>,<con>
Esempi
aggiungi eax, 10 — EAX ← EAX + 10
aggiungi BYTE PTR [var], 10 — aggiungi 10 al singolo byte memorizzato nell'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 add
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 — sottrae 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 — sottrae uno dal contenuto di EAX.
inc DWORD PTR [var] — aggiunge uno all'intero a 32 bit memorizzato nella posizione *var*

imul — Moltiplicazione di numeri interi
L' istruzione imul ha due formati di base: a due operandi (i primi due elenchi di sintassi sopra) e a tre operandi (gli ultimi due elenchi di sintassi sopra).
La forma a due operandi moltiplica i suoi due operandi tra loro e memorizza il risultato nel primo operando. Il risultato (ovvero il primo) operando deve essere un registro.
La forma a tre operandi moltiplica tra loro il secondo e il terzo operando e memorizza il risultato nel primo operando. Anche in questo caso, l'operando risultato deve essere un registro. Inoltre, il terzo operando può essere solo 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 considerando EDX come i quattro byte più significativi ed EAX come i quattro byte meno significativi) per il valore dell'operando specificato. Il risultato della divisione viene memorizzato in EAX, mentre il resto viene inserito in EDX.

Sintassi
idiv <reg32>
idiv <mem>

Esempi

idiv ebx — dividere il contenuto di EDX:EAX per il contenuto di EBX. Inserire 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 memorizzato nella posizione di memoria *var* . Inserisci il quoziente in EAX e il resto in EDX.
e, o, xor — And logico bit a bit e or esclusivo
Queste istruzioni eseguono l'operazione logica specificata (rispettivamente and, or logici bit a bit e or esclusivi) sui propri operandi, posizionando il risultato nella prima posizione dell'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 tutti i bit di EAX tranne gli ultimi 4.
xor edx, edx — imposta il contenuto di EDX a zero.

non — Non logico bit a bit
Nega logicamente il contenuto dell'operando (ovvero inverte tutti i valori dei bit nell'operando).

Sintassi
non <reg>
non <mem>

Esempio
non 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 — Sposta a sinistra, Sposta a destra
Queste istruzioni spostano i bit nel contenuto del primo operando a sinistra e a destra, riempiendo le posizioni vuote dei bit con degli zeri. 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, gli spostamenti maggiori di 31 vengono eseguiti 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 valore minimo 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 puntatore di istruzione (IP), ovvero un valore a 32 bit che indica la posizione in memoria in cui inizia l'istruzione corrente. Normalmente, il registro viene incrementato 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 di controllo del flusso fornite.

Utilizziamo la notazione <etichetta> per fare riferimento alle posizioni etichettate nel testo del programma. Le etichette possono essere inserite ovunque nel testo del codice assembly x86 inserendo il nome dell'etichetta seguito da due punti. Ad 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 pratico begin . Questa etichetta è semplicemente un modo pratico per esprimere la posizione anziché il 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 — Passa all'istruzione etichettata begin .

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

Ad alcuni rami condizionali vengono assegnati nomi intuitivamente basati sul fatto che l'ultima operazione eseguita è un'istruzione di confronto speciale, cmp (vedi sotto). Ad esempio, rami condizionali come jle e jne si basano sull'esecuzione di un'operazione cmp sugli operandi desiderati.

Sintassi
je <etichetta> (salta quando uguale)
jne <etichetta> (salta quando diverso da)

jz <etichetta> (salta quando l'ultimo risultato era zero)
jg <etichetta> (salta quando maggiore di)
jge <etichetta> (salta quando maggiore o uguale a)
jl <etichetta> (salta quando minore di)
jle <etichetta> (salta quando minore o uguale a)

Esempio
cmp eax, ebx
jle done

Se il contenuto di EAX è minore o uguale al contenuto di EBX, salta all'etichetta *done* . Altrimenti, continua con l'istruzione successiva.
cmp — Confronta
Confronta 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 , con la differenza che il risultato della sottrazione viene scartato anziché 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

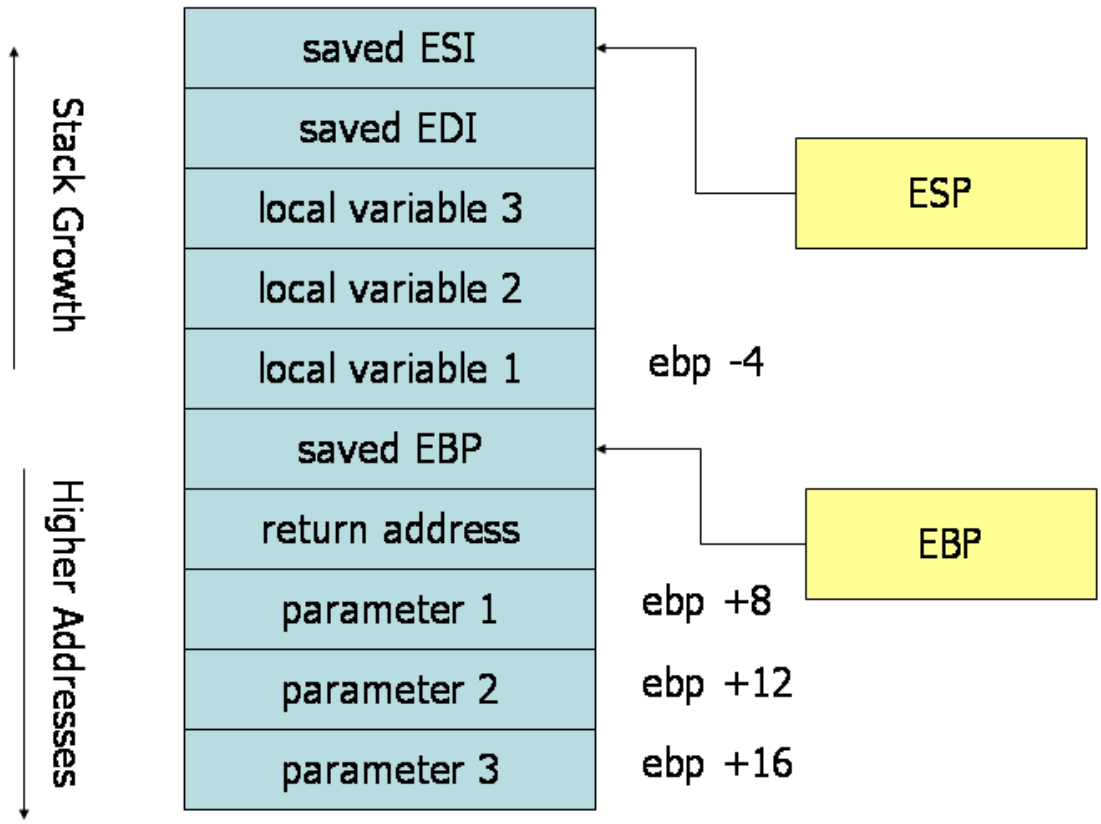
Se i 4 byte memorizzati nella posizione *var* sono uguali alla costante intera di 4 byte 10, salta alla posizione etichettata *loop* .
call , **ret** — Chiamata e ritorno della subroutine
Queste istruzioni implementano una chiamata e un ritorno a una subroutine. L' istruzione di chiamata prima inserisce la posizione corrente del codice nello stack di memoria supportato dall'hardware (vedere l' istruzione di push per i dettagli), quindi esegue un salto incondizionato alla posizione del codice indicata dall'operando 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 innanzitutto 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 di codice recuperata.

Chiamata sintattica
<etichetta>
ret

Convenzione di chiamata

Per consentire a programmatori diversi di condividere codice e sviluppare librerie utilizzabili da più programmi, e per semplificare l'uso delle subroutine in generale, i programmatori adottano in genere una *convenzione di chiamata* comune . La convenzione di chiamata è un protocollo che definisce come chiamare e restituire le routine. Ad esempio, dato un insieme di regole per la convenzione di chiamata, un programmatore non ha bisogno di esaminare la definizione di una subroutine per determinare come i parametri debbano essere passati a tale subroutine. Inoltre, dato un insieme di regole per la convenzione di chiamata, i compilatori di linguaggi di alto livello possono essere impostati in modo da seguire tali regole, consentendo così alle routine in linguaggio assembly scritte a mano e alle routine in linguaggio di alto livello di chiamarsi a vicenda. In pratica, sono possibili numerose convenzioni di chiamata. Useremo la convenzione di chiamata del linguaggio C, ampiamente utilizzata. Seguire questa convenzione vi permetterà di scrivere subroutine in linguaggio assembly che siano richiamabili in modo sicuro da codice C (e C++), e vi permetterà anche di chiamare funzioni della libreria C dal vostro codice in linguaggio assembly.
La convenzione di chiamata del C si basa in larga parte sull'uso dello stack supportato dall'hardware. Si basa sulle istruzioni push , pop , call e ret . I parametri delle subroutine vengono passati sullo stack. I registri vengono salvati sullo stack e le variabili locali utilizzate dalle subroutine vengono memorizzate sullo 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 insiemi di regole. Il primo insieme di regole è utilizzato dal chiamante della subroutine, mentre il secondo insieme di regole è rispettato da chi scrive la subroutine (il chiamato). È importante sottolineare che errori nell'osservanza di queste regole si traducono rapidamente in errori fatali nel programma, poiché lo stack rimarrà in uno stato incoerente; pertanto, è necessario prestare la massima attenzione quando si implementa la convenzione di chiamata nelle proprie subroutine.



≥

Stack durante la chiamata alla subroutine

[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 è disegnare il contenuto della regione vicina dello stack durante l'esecuzione di una subroutine. L'immagine sopra mostra il contenuto dello stack durante l'esecuzione di una subroutine con tre parametri e tre variabili locali. Le celle rappresentate nello stack sono locazioni di memoria a 32 bit, quindi gli indirizzi di memoria delle celle sono distanti 4 byte. Il primo parametro si trova a un offset di 8 byte dal puntatore base. Sopra i parametri sullo stack (e sotto il puntatore base), l' istruzione di chiamata ha posizionato l'indirizzo di ritorno, determinando così un offset aggiuntivo di 4 byte dal puntatore base al primo parametro. Quando l' istruzione ret viene utilizzata per tornare dalla subroutine, salta 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 deve salvare il contenuto di determinati registri denominati *"caller-saved"* . I registri "caller-saved" sono EAX, ECX, EDX. Poiché la subroutine chiamata può modificare

questi registri, se il chiamante si affida ai loro valori dopo il ritorno della subroutine, deve caricare 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, è necessario inserirli nello stack prima della chiamata. I parametri devono essere inseriti in ordine inverso (ovvero, dall'ultimo parametro per primo). Poiché lo stack cresce verso il basso, il primo parametro verrà memorizzato all'indirizzo più basso (questa inversione dei parametri è stata storicamente utilizzata per consentire alle funzioni di passare un numero variabile di parametri).
3. Per chiamare la subroutine, utilizzare l' istruzione call . Questa istruzione posiziona l'indirizzo di ritorno in cima ai parametri nello stack e salta al codice della subroutine. Questo richiama la subroutine, che deve seguire le regole per il destinatario della chiamata riportate 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. Rimuove i parametri dallo stack. In questo modo lo stack viene ripristinato 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 sta chiamando 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* .

```
push [var] ; Invia prima l'ultimo parametro
push 216 ; Spingi il secondo parametro
push eax ; Invio primo parametro ultimo

call _myFunc ; Chiama la funzione (si assume la denominazione C)

aggiungi esp, 12
```

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

Il risultato prodotto da *_myFunc* è ora disponibile per l'utilizzo nel registro EAX. I valori dei registri salvati dal chiamante (ECX ed EDX) potrebbero essere stati

modificati. Se il chiamante li utilizzasse dopo la chiamata, avrebbe dovuto salvarli nello stack prima della chiamata e ripristinarli dopo.

Regole per i chiamati

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

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

```
spingere ebp
mov ebp, esp
```

1. Questa azione iniziale mantiene il *puntatore base* , EBP. Il puntatore base viene utilizzato per convenzione come punto di riferimento per trovare parametri e variabili locali sullo stack. Quando una subroutine è in esecuzione, il puntatore base contiene una copia del valore del puntatore dello stack da quando la subroutine ha iniziato l'esecuzione. Parametri e variabili locali saranno sempre posizionati a offset noti e costanti rispetto al valore del puntatore base. Inseriamo il vecchio valore del puntatore base all'inizio della subroutine in modo da poter ripristinare in seguito il valore appropriato del puntatore base per il chiamante al termine della subroutine. Ricordate, il chiamante non si aspetta che la subroutine modifichi il valore del puntatore base. Quindi spostiamo il puntatore dello stack in EBP per ottenere il nostro punto di riferimento per l'accesso a parametri e variabili locali.
2. Successivamente, allocare le variabili locali liberando spazio sullo stack. Ricordate, lo stack cresce verso il basso, quindi per liberare spazio in cima allo stack, il puntatore dello stack deve essere decrementato. L'entità del decremento del puntatore dello stack dipende dal numero e dalla dimensione delle variabili locali necessarie. Ad esempio, se fossero necessari 3 interi locali (4 byte ciascuno), il puntatore dello stack dovrebbe essere decrementato di 12 per liberare spazio per queste variabili locali (ovvero, sub esp, 12). Come per i parametri, le variabili locali saranno posizionate a offset noti dal puntatore base.
3. Successivamente, salva i valori dei registri *salvati dal chiamante* che verranno utilizzati dalla funzione. Per salvare i registri, inseriscili nello stack. I registri salvati dal chiamante sono EBX, EDI ed ESI (anche ESP ed EBP saranno preservati dalla convenzione di chiamata, ma non è necessario inserirli nello stack durante questa fase).

Dopo aver eseguito queste tre azioni, il corpo della subroutine può procedere. Quando la subroutine viene restituita, deve seguire questi passaggi:

1. Lasciare il valore di ritorno in EAX.

2. Ripristinare i vecchi valori di tutti i registri salvati dal chiamante (EDI ed ESI) che sono stati modificati. Il contenuto dei registri viene ripristinato estraendoli dallo stack. I registri devono essere estratti nell'ordine inverso rispetto a quello di push.
3. Dealock delle variabili locali. Il modo più 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 del puntatore base nello stack: mov esp, ebp . Questo funziona perché il puntatore base contiene sempre il valore che il puntatore dello stack conteneva immediatamente prima dell'allocazione delle variabili locali.
4. Immediatamente prima di tornare, ripristina il valore del puntatore base del chiamante estraendo EBP dallo stack. Ricorda che la prima cosa che abbiamo fatto all'ingresso nella subroutine è stata eseguire il push del puntatore base per salvare il suo vecchio valore.
5. Infine, si ritorna 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 dividono nettamente in due metà, che sono sostanzialmente speculari l'una all'altra. La prima metà delle regole si applica all'inizio della funzione e si dice comunemente che ne definisca il *prologo* . La seconda metà delle regole si applica alla fine della funzione e si dice quindi comunemente che ne definisca l' *epilogo* .

Esempio

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

```
.486
.MODELLO APPARTAMENTO
.CODICE
PUBBLICO _myFunc
_myFunc PROC
  ; Prologo della subroutine
  push ebp ; Salva il vecchio valore del puntatore base.
  mov ebp, esp ; Imposta il nuovo valore del puntatore base.
  sub esp, 4 ; Lascia spazio per una variabile locale da 4 byte.
  push edi ; Salva i valori dei registri che la funzione
  push esi ; modificherà. Questa funzione utilizza EDI ed ESI.
  ; (non è necessario salvare EBX, EBP o ESP)

  ; Corpo della subroutine
  mov eax, [ebp+8] ; Sposta il valore del parametro 1 in EAX
  mov esi, [ebp+12] ; Sposta il valore del parametro 2 in ESI
  mov edi, [ebp+16] ; Sposta il valore del parametro 3 in EDI
```

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

  ; Epilogo della subroutine
  pop esi ; Recupera i valori del registro
  pop edi
  mov esp, ebp ; Dealloca le variabili locali
  pop ebp ; Ripristina il valore del puntatore base del chiamante
  ritirarsi
_myFunc ENDP
FINE
```

Il prologo della subroutine esegue le azioni standard di salvataggio di un'istantanea del puntatore dello stack in EBP (il puntatore di base), allocazione di variabili locali decrementando il puntatore dello stack e salvataggio dei valori del registro sullo stack.

Nel corpo della subroutine possiamo osservare l'utilizzo del puntatore base. Sia i parametri che le variabili locali si trovano a offset costanti rispetto al puntatore base per tutta la durata dell'esecuzione della subroutine. In particolare, notiamo che, poiché i parametri sono stati inseriti nello stack prima della chiamata della subroutine, si trovano sempre sotto il puntatore base (ovvero a 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. Analogamente, poiché le variabili locali vengono allocate dopo l'impostazione del puntatore base, si trovano sempre sopra il puntatore base (ovvero a indirizzi più bassi) nello stack. In particolare, la prima variabile locale si trova sempre in EBP - 4, la seconda in EBP - 8 e così via. Questo utilizzo convenzionale del puntatore base ci consente di identificare rapidamente l'utilizzo di variabili e parametri locali all'interno del corpo di una funzione.

L'epilogo della funzione è sostanzialmente un'immagine speculare del prologo della funzione. I valori dei registri 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

riutilizzarli e adattarli per qualsiasi scopo non commerciale (se desiderate utilizzarli per scopi commerciali, contattate David Evans per maggiori informazioni). Se adattate o utilizzate questi materiali, vi preghiamo di includere una dicitura 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 aggiornata da Alan Batson, Mike Lack e Anita Jones."* e un link a questa pagina.

CS216: Programma e rappresentazione dei dati
Università della Virginia

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