Vorlesung: Rechnerstrukturen, Teil 2 (Modul IP7)

J. Zhang
zhang@informatik.uni-hamburg.de

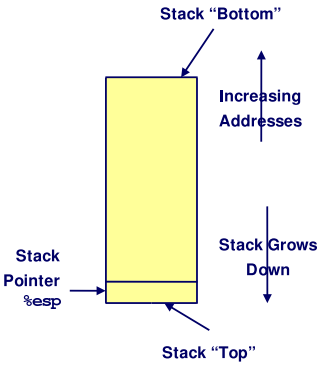
Universität Hamburg
Fachbereich Informatik
AB Technische Aspekte Multimodaler Systeme

T|A M|S Arbeitsbereich TAMS
Fachbereich Informatik
Vorlesung: Rechnerstrukturen, Teil 2 (Modul IP7)

Inhaltsverzeichnis

IA32 Kellerspeicher

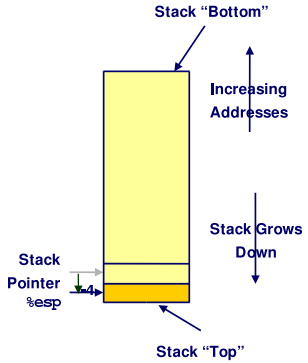
- Speicherregion, die mit Stack-operationen verwaltet wird
- Wächst in Richtung niederigerer Adressen
- Register % esp gibt die aktuelle Stack-Adresse an
  - Adresse des obersten Elements



IA32 Stack

"Pushing"

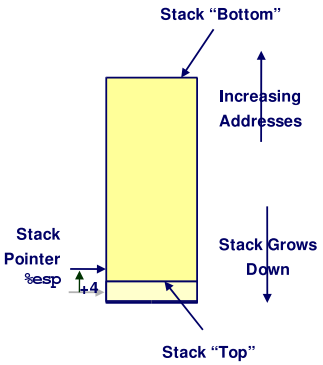
- pushl Src
- Holt den Operanden bei Src
- Dekrement %esp bei 4
- Speichert den Operanden bei der von %esp vorgegebenen Adresse



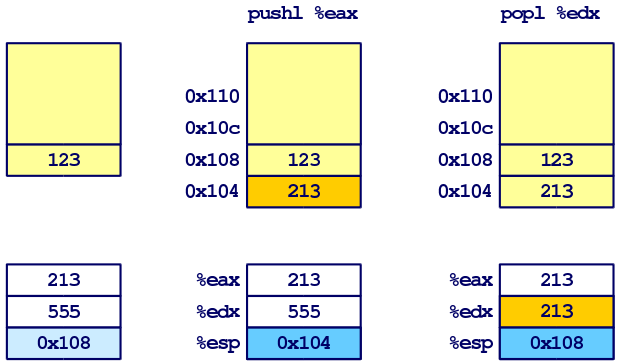
IA32 Stack "Popping"

"Popping"

- popl Dest
- Liest den Operanden bei der von %esp vorgegebenen Adresse
- Inkrement %esp bei 4
- Schreibt nach *Dest*



Beispiele für Stack-Operation



Prozedur der Ablaufsteuerung

Prozedur-Aufruf ("Call")

- Benutzt den Stack zur Unterstützung von "Call" und "Return"
  call label "Push" "Return" Adresse in Stack,
  "Jump" to label

Wert der "Return" Adresse

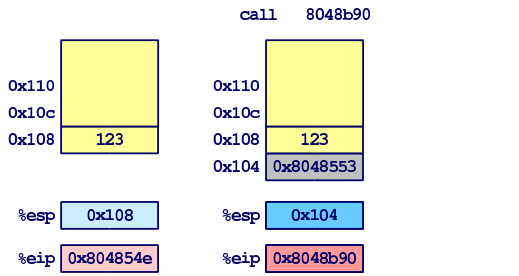
- Adresse der Anweisung über den "call" hinaus
- Disassembler Beispiel:
  804854e: e8 3d 06 00 00 call 8048b90
  $< main >$
  8048553: 50 pushl %eax
- Zurückkommende ("Return") Adresse = 0x8048553

Prozedur "Return":

- ret Pop-Adresse vom Stack; "Jump" zur Adresse

Beispiel Prozeduraufruf

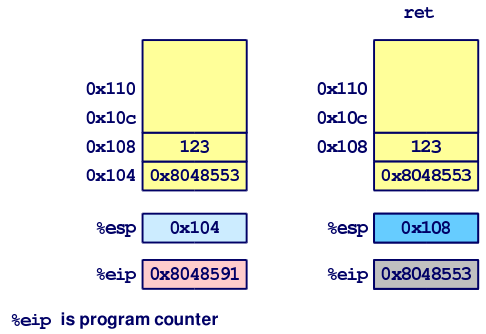
```
804854e:  e8 3d 06 00 00       call   8048b90 <main>
8048553:  50                   pushl  %eax
```



`%eip` is program counter

Beispiel Prozedur "Return"

```
8048591:   c3                  ret
```



`%eip` is program counter

Stack-basierende Sprachen

Sprachen, die Rekursion unterstützen

- z.B., C, Pascal, Java
- Code muss "Reentrant" sein
  - Multiple simultane Instantiierungen einer einzelnen Prozedur
- Braucht einen Ort um den Zustand jeder Instantiierung zu speichern
  - Argumente
  - Lokale Variable
  - Rücksprungadresse

Stack-basierende Sprachen

Stack-Disziplin

- Zustand einer gegebene Prozedur, die einen beschränkten Zeitraum benötigt wird
  - von "when called" zu "when return"
- Der "Callee" kommt vor dem "Caller" zurück

Stack "Frames"

- Zustand für eine einzelne Prozedur-Instantiierung

Stack "Frames"

Inhalt

- Lokale Variablen
- "Return" Adresse
- Temporäre Daten

Verwaltung

- Der Speicherbereich, der bei "Enter" Prozedur zugeteilt wird
  - "Set-up" Code
- Freigegeben bei "Return"
  - "Finish" Code

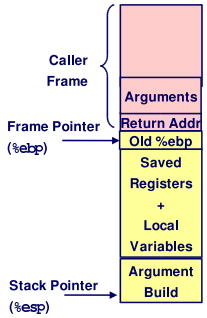
Adressenverweise ("Pointer")

- Stack Adressenverweis %esp gibt das obere Ende des Stacks an
- "Frame" Adressenverweis %ebp gibt den Anfang des aktuellen "Frame" an

IA32/Linux Stack "Frame"

Aktueller Stack "Frame" (Von oben (Top) nach unten (Bottom))

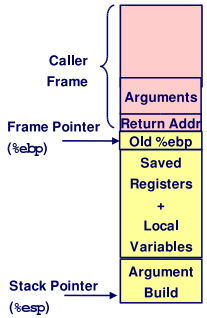
- Parameter für eine weitere Funktion, die gleich aufgerufen wird ("call")
  - Anstieg der Argumente
- Lokale Variablen
  - wenn sie nicht in Registern behalten werden können
- Gespeicherter Register Kontext
- Alter "Frame" Adressenverweis



IA32/Linux Stack "Frame"

"Caller" Stack "Frame"

- "Return" Adresse
  - wird von "Call"-Anweisung "gepusht"
- Argumente für diesen "Call"



Register Speicherungs-Konventionen

Wenn who von Prozedur yoo aufgerufen wird

- ist yoo der "Caller", who der "Callee"

Kann das Register für vorübergehende Speicherung benutzt werden?

- Der Inhalt des Registers %edx wird von who überschrieben

Register Speicherungs-Konventionen

Wenn who von Prozedur yoo aufgerufen wird

- ist yoo der "Caller", who ist der "Callee"

Kann das Register für vorübergehende Speicherung benutzt werden?

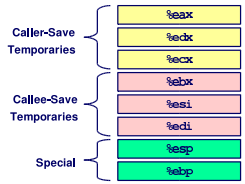
Konventionen

- "Caller" speichert ("Caller Save")
  - Der "Caller" speichert vor dem "Calling" vorübergehend in seinem Frame
- "Callee" speichert ("Callee Save")
  - Der "Callee" speichert vor Verwendung vorübergehend in seinem Frame

IA32/Linux Register Verwendung

Ganzzahlige Register

- Zwei werden speziell verwendet
  %ebp, %esp
- Drei werden als "Callee save" verwaltet
  %ebx, %esi, %edi
  - Alte Werte werden vor Verwendung auf dem Stack gesichert
- Drei werden als "Caller-save" verwaltet
  %eax, %edx, %ecx
  - Tu was Du willst, aber erwarte, dass jeder "Callee" ebenso handelt
- Register %eax speichert auch den zurückgelieferten Wert



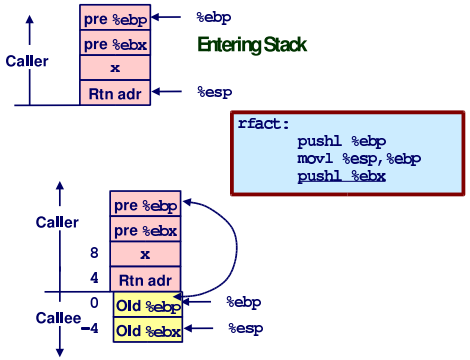
Rekursive Fakultät

- %eax gebraucht ohne vorheriges Speichern
- %ebx gebraucht, aber am Anfang gespeichert und am Ende zurückgeschrieben

```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

```
.globl rfact
        .type
rfact,@function
rfact:
        pushl %ebp
        movl %esp,%ebp
        pushl %ebx
        movl 8(%ebp),%ebx
        cmpl $1,%ebx
        jle .L78
        leal -1(%ebx),%eax
        pushl %eax
        call rfact
        imull %ebx,%eax
        jmp .L79
        .align 4
.L78:
        movl $1,%eax
.L79:
        movl -4(%ebp),%ebx
        movl %ebp,%esp
        popl %ebp
        ret
```

Rfact Stack-Aufbau



Rfact Stack-Körper

Recursion

```
  movl 8(%ebp),%ebx# ebx = x
  cmpl $1,%ebx        # Compare x : 1
  jle .L78            # If <= goto Term
  leal -1(%ebx),%eax  # eax = x-1
  pushl %eax          # Push x-1
  call rfact          # rfact(x-1)
  imull %ebx,%eax     # rval * x
  jmp .L79            # Goto done
.L78:               # Term:
  movl $1,%eax        # return val = 1
.L79:               # Done:
```

```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1) ;
  return rval * x;
}
```

Registers

%ebx **Stored value of x**

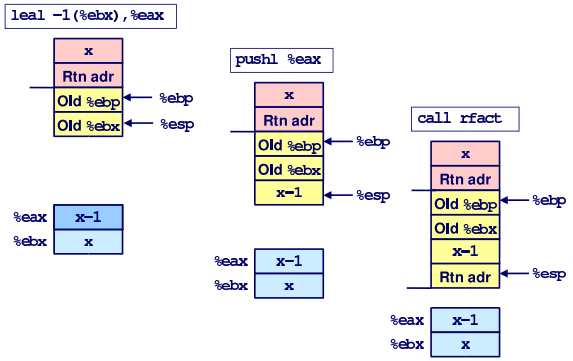
%eax

Temporary value of x-1

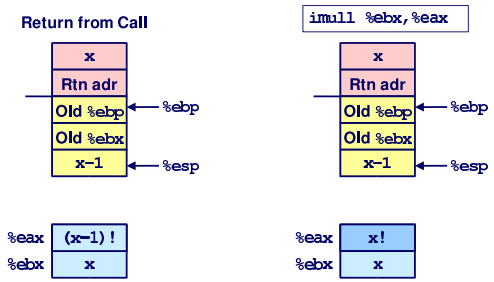
Returned value from rfact(x-1)

Returned value from this call

Rfact Rekursion

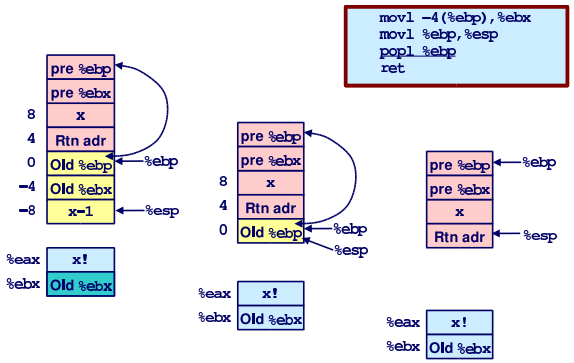


Rfact Ergebnis



Assume that `rfact(x-1)` returns `(x-1)!` in register `%eax`

Rfact Durchführung



Adressenverweis

- Adressenverweis übergeben, um Wert einer Variablen zurückschreiben zu können

Recursive Procedure

```
void s_helper
  (int x, int *accum)
{
  if (x <= 1)
    return;
  else {
    int z = *accum * x;
    *accum = z;
    s_helper (x-1,accum);
  }
}
```

Top-Level Call

```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

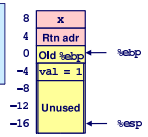
Adressenverweis schaffen und initialisieren

Benutzen des Stacks für Lokale Variable

- Variable val muss auf dem Stack gespeichert werden
  - Ein Adressenverweis dorthin muss geschaffen werden
- Computer Adressenverweis als -4(%ebp)
- In den Stack "pushen" als zweites Argument

Initial part of `sfact`

```
_sfact:
        pushl %ebp      # Save %ebp
        movl %esp,%ebp          # Set %ebp
        subl $16,%esp # Add 16 bytes
        movl 8(%ebp),%edx       # edx = x
        movl $1,-4(%ebp)        # val = 1
```




```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```


Zusammenfassung

Der Stack ermöglicht die Rekursion

- Private Speicherung für jeden Prozedur-"Call"
  - Instantiierungen kommen sich nicht ins Gehege
  - Adressierung von lokalen Variablen und Argumenten kann relativ zu den Positionen des Stacks sein
- Kann von Stack-Disziplin verwaltet werden
  - Prozeduren kommen in umgekehrter Reihenfolge der "Calls" zurück

IA32 Prozeduren sind Kombination von Anweisungen + Konventionen

- "Call" / "Return" Anweisungen
- Register Verwendungs-Konventionen
  - "Caller/Callee save"
  - %ebp und %esp
- Organisations-Konventionen des Stack "Frames"

Grundlegende Datentypen

Ganzzahl (Integer)

- Wird in allgemeinen Registern gespeichert und angewandt
- "Mit Vorzeichen" vs. "ohne Vorzeichen": abhängig von den verwendeten Anweisungen

Intel	GAS	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int

Gleitkomma (Floating Point)

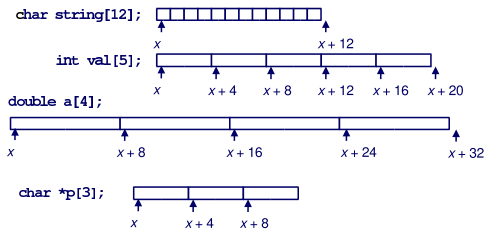
- Wird in Gleitkomma- Registern gespeichert und angewandt

Intel	GAS	Bytes	C
Single	s	4	float
Double	l	8	double
Extended	t	10/12	long double

"Array" Zuordnung

Grundlegendes Prinzip
$T\ A[L];$

- "Array" mit Daten des Typs T und der Länge L
- Fortlaufend zugeteilte Region von $L^* \ sizeof(T)$ Bytes

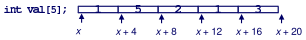


"Array" Zugang

Grundlegendes Prinzip

$T\ A[L];$

- "Array" mit Daten des Typs T und der Länge L
- Bezeichner A kann als Adressenverweis verwendet werden, um Element 0 anzuordnen ("to array")

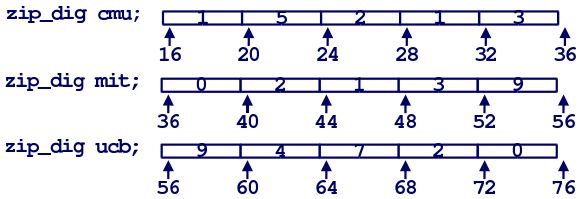


Reference	Type	Value
val[4]	int	3
val	int *	x
val+1	int *	x + 4
&val[2]	int *	x + 8
val[5]	int	??
*(val+1)	int	5
val + i	int *	x + 4 i

"Array" Beispiel

```
typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



Zugriff auf "Array"

Referenzierung

- Der Code führt keine Bereichsüberprüfung ("bounds checking") durch
- Verhalten außerhalb des Indexbereichs ist Implementierungsabhängig
  - Keine garantierte relative Zuteilung verschiedener "Arrays"

Umsetzung durch GCC

- Eliminert "Loop"-Variable i
- Konvertiert "Array"-Code zu Adressenverweis-Code
- Wird in "do-while" Form ausgedrückt
  - Test bei Eintritt unnötig

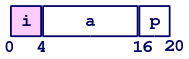
Strukturen

Konzept

- Fortlaufend zugeteilte Speicherregion
- Bezieht sich mit Namen auf Teile in der Struktur
- Teile können verschiedenen Typs sein

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

Memory Layout




```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

Assembly

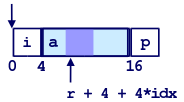
```
# %eax = val
# %edx = r
movl %eax,(%edx)    # Mem[r] = val
```

Generieren eines Adressenverweises auf ein Strukturteil

Generieren eines Adressenverweises auf ein "Array"-Element

- "Offset" jedes Strukturteils wird zum Compilerzeitpunkt festgelegt

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

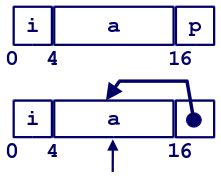



```
int *
find_a
 (struct rec *r, int idx)
{
  return &r->a[idx];
}
```

```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax    # 4*idx
leal 4(%eax,%edx),%eax  # r+4*idx+4
```


Strukturreferenzierung

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```




```
void
set_p(struct rec *r)
{
  r->p =
   &r->a[r->i];
}
```

```
# %edx = r
      movl (%edx),%ecx     # r->i
      leal 0(,%ecx,4),%eax # 4*(r->i)
      leal 4(%edx,%eax),%eax      # r+4+4*(r->i)
      movl %eax,16(%edx)   # Update r->p
```

Zusammenfassung

"Arrays" in C

- Fortlaufend zugeteilter Speicher
- Adressverweis auf das erste Element
- Keine Bereichsüberprüfung ("Bounds Checking")

Compiler Optimierungen

- Compiler wandelt Array Code oft in Adressverweise ("pointer code") um
- Verwendet Adressierungsmodi um "Array"-Indizes zu skalieren
- Viele Tricks, um die "Array"-Indizierung in Schleifen zu verbessern

Strukturen

- Bytes werden in der ausgewiesenen Reihenfolge zugeteilt
- In der Mitte und am Ende polstern, um die richtige Ausrichtung zu erreichen

Ergänzende Literatur

Zur Rechnerarchitektur (Teil 2, 4. Termin):

Literatur

[1] Randal E. Bryant and David O'Hallaron. Computer systems. pages 1000–2000. Pearson Education, Inc., New Jersey, 2003.