



MMIX-Unterprogramme

- Unterprogramme: Aufruf mit GO
- Parameterübergabe durch Register
- Speichern der Rücksprungadresse auf dem Stack
- Implementierung eines Stack über Stackpointer im Hauptspeicher
- Parameterübergabe auf dem Stack
- Übergabe der Parameter auf dem Stack

Heute

- MMIX-Registerstack
- lokale/globale Register
- PUSHJ, POP
- Exkurs: Stack overflow



Der Registerstack

Parameterübergabe auf dem Stack vs. in Registern

- Übergabe auf dem Stack ist insbesondere bei großen Programmen einfacher zu handhaben
- Übergabe in Registern ist schneller (kein Zugriff auf den Hauptspeicher)
- Verfahren haben beide Nachteile

Neues Konzept: Registerstack

- Verbindet Vorteile beider Übergabeformen
- In dieser oder ähnlicher Form in allen heute gängigen Mikroprozessoren realisiert

Beobachtung:

- Im Stack wird immer nur der oberste Teil benutzt
- Betrachte die Register \$0,\$1,\$2, ... als Stapel von Registern
- Bestimmte Register sind in Gebrauch, andere (höhere) noch frei
- Nehme beim Push auf den Stack stets weiter oben liegende Register
- die ganz weit unten befindlichen Einträge kann man zu gegebener Zeit in einem Rutsch in den Hauptspeicher auslagern, um Platz für neue Register zu schaffen



Globale und lokale Register

MMIX hat 256 Register: \$0,\$1, ..., \$255

- Obere Teil der Register sind die **globalen Register**
- Spezialregister r_G zeigt an, ab welcher Nummer Register global sind
- r_G kann zwischen 32 und 255 liegen
- globale Register sind **nicht** Teil des Registerstacks!

Untere Teil der Register sind die lokalen Register:

- Beispiele: \$0, \$1, usw.
- Spezialregister r_L gibt die Anzahl der gerade in Gebrauch befindlichen Register an
- Beispiel: $r_L = 5$ bedeutet: \$0,\$1,\$2,\$3,\$4 sind gerade als lokale Register in Verwendung

Register mit den Nummern r_L bis r_G-1 sind unbenutzt: marginale Register

- beim Start des Programms gibt es nur marginale Register
- wird ein marginales Register gelesen, erhält man den Wert 0
- wird ein marginales Register geschrieben, so wird es automatisch zu einem lokalen Register
- Wert von r_L wird automatisch angepasst



Registerstack

Die Register $\$0, \$1, \dots, \$(rL-1)$ bilden den natürlichen Stackframe des gerade laufenden Unterprogramms

- Normalerweise sind die Parameter in den unteren Registern $\$0, \1 usw. sowie die lokalen Variablen in den oberen Registern $\dots, \$(rL-1)$

Ideales Vorgehen bei Unterprogrammaufruf:

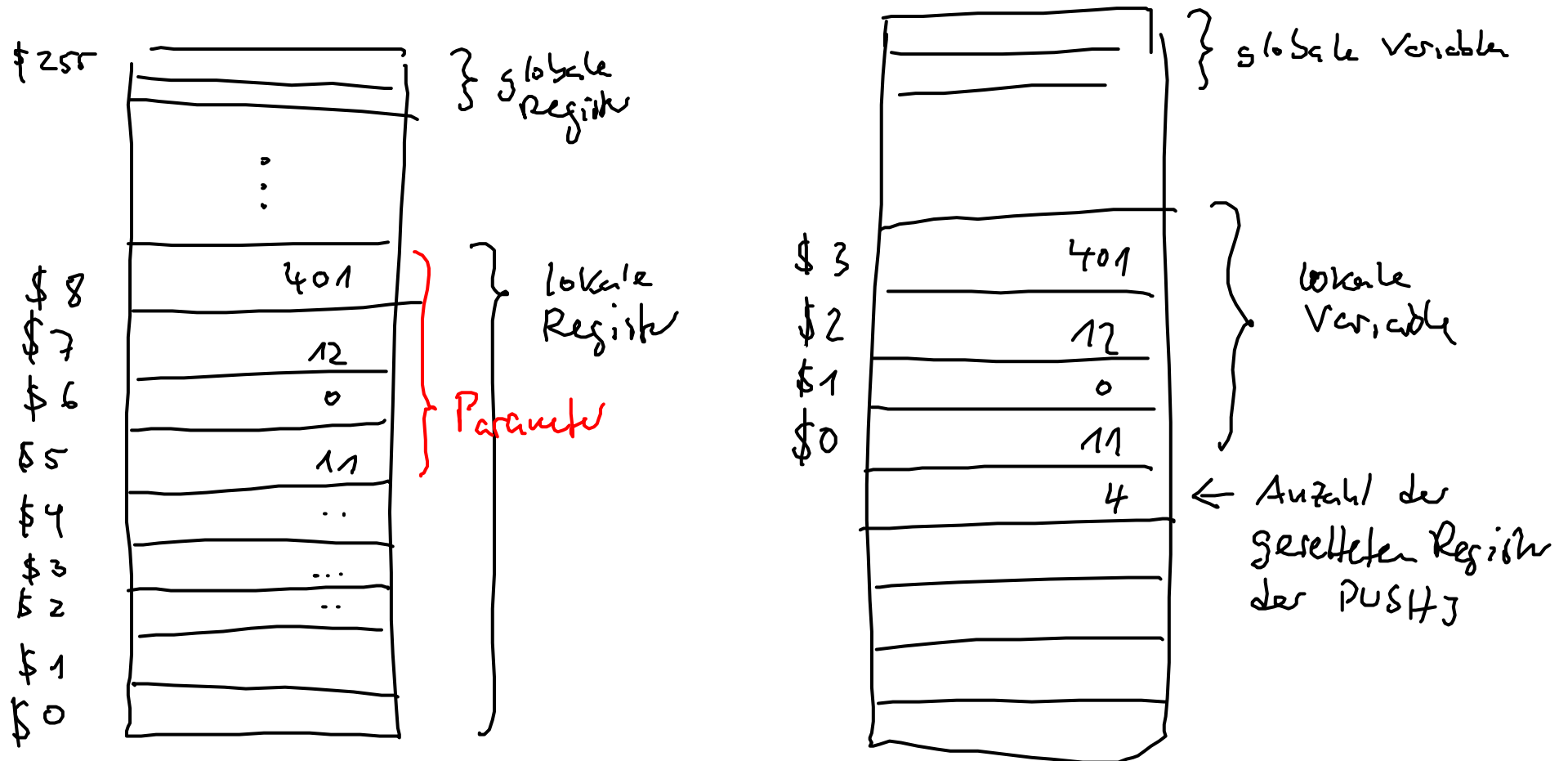
- Lokale Register kommen auf den Stack und werden im Unterprogramm "frei" verwendbar
- Genau das macht der Registerstack. Er wird unterstützt durch spezielle Operationen PUSHJ und POP

PUSHJ: "schnelles Umnummerieren der Register": PUSHJ $\$X, YZ$ bewirkt

- Register $\$0, \dots, \$X-1$ werden auf den Registerstack gerettet und verschwinden vorübergehend
- Register $\$X+1, \$X+2 \dots$ (also die Parameter) werden umnummeriert und sind nun die Register $\$0, \$1, \dots$
- An der Stelle $\$X$ entsteht ein Loch: dort wird die Anzahl der Register (ohne Loch) im darunter liegenden Stackframe gespeichert
- die Rücksprungadresse kommt in Spezialregister rJ, YZ ist eine Sprungmarke



Beispiel: Effekt von PUSHJ \$4,XY





Die Operation POP

POP macht gerade die Veränderungen der letzten PUSHJ-Operation rückgängig

POP X,YZ hat den folgenden Effekt:

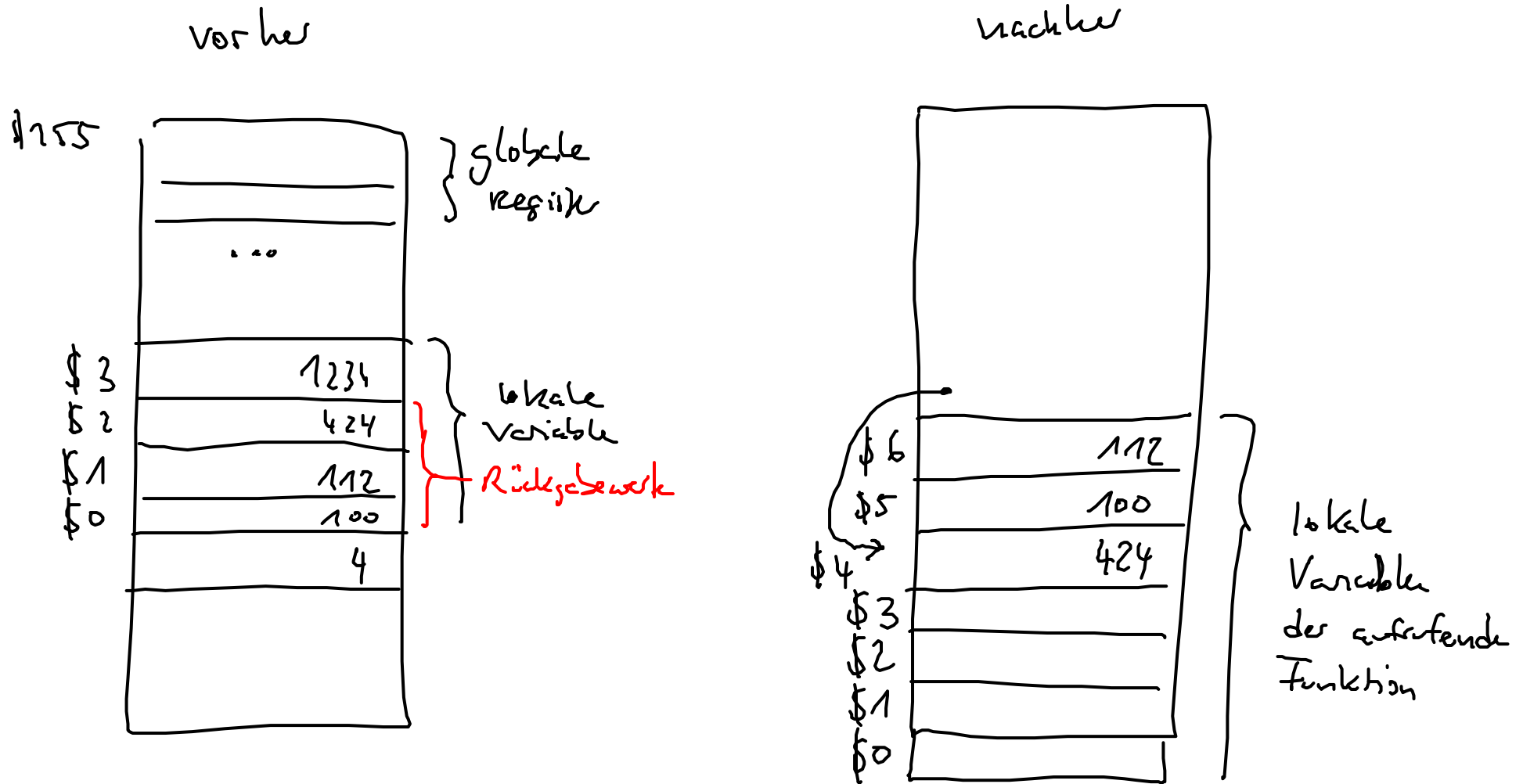
- Umnummerierung der Register des letzten PUSHJ-Befehls wird rückgängig gemacht
- Es gibt X neue lokale Register für Rückgabewerte
- Diese erhalten der Reihe nach die Werte $\$(X-1)$, $\$0$, $\$1$, ..., $\$(X-2)$
- Es erfolgt ein Rücksprung zur Adresse $rJ+4YZ$ (in der Regel hat YZ den Wert 0)

Anordnung der Rückgabewerte macht Sinn:

- Hauptergebnis liegt vorher ganz oben, nachher ganz unten (und füllt das Loch im Registerstack)
- Register werden optimal genutzt



Schematische Darstellung von POP 3,0





Ausblick

- Für rekursive Prozeduren muss man selbst die Rücksprungadresse sichern (wohin?) → auf den Stack!
- Registerstack wird durch einen Registerring implementiert
- Unterer Teil des Stacks wird ggf. in den Hauptspeicher ausgelagert (Spezialregister rS ist dafür der Stackpointer)
- Hierfür ist das Stacksegment des Hauptspeichers vorgesehen (#6000 0000 0000 0000 - #7FFF FFFF FFFF FFFF)
- Man muss aufpassen, dass jeweils genug lokale Register zur Verfügung stehen (Pseudobefehl LOCAL)

- Jetzt ein kleiner Exkurs über sichere Programmierung...



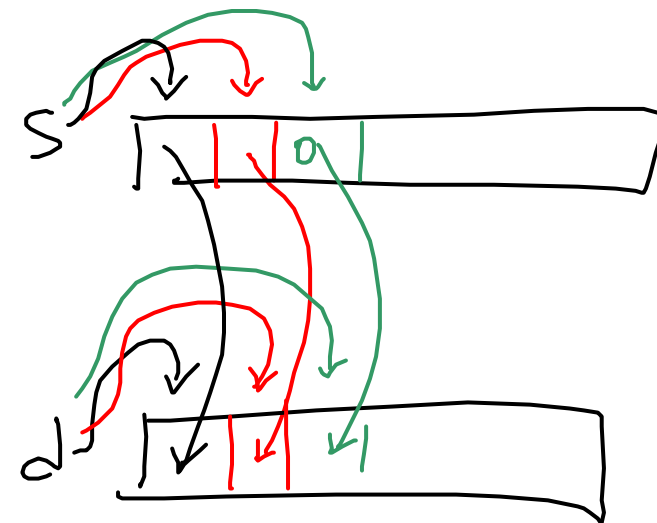
C-Funktion strcpy

Zeichenketten sind in C als Byte-Arrays definiert, die mit einem Nullzeichen terminiert werden (wie in MMIX)

Beliebte C-Funktion: strcpy (string copy)

- kopiert einen Quellstring s in einen Zielstring d
- es wird so lange kopiert, bis in s das Nullzeichen erreicht wird
- implizite Annahme: d ist gross genug für s

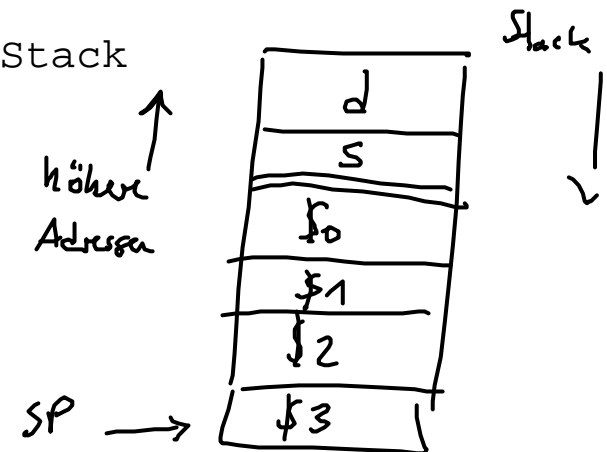
```
#include <stdlib.h>
char *
strcpy (char *d, char *s)
{
    char *d1 = d;
    while ((*d1++ = *s++));
    return d;
}
```





strcpy in MMIX

strcpy	SUB	SP, SP, 4*8	
	STO	\$0, SP, 3*8	Rücksprungadresse auf den Stack retten
	STO	\$1, SP, 2*8	Retten der in strcpy verwendeten Register (es werden \$1, \$2 und \$3 verwendet)
	STO	\$2, SP, 8	
	STO	\$3, SP, 0	
	LDO	\$3, SP, 4*8	Holen der Parameter vom Stack (zuerst s, darunter d)
	LDO	\$2, SP, 5*8	
L	LDB	\$1, \$3 0	Kopierschleife
	STB	\$1, \$2, 0	
	INCL	\$3, 1	
	INCL	\$2, 1	
	BNZ	\$1, L	
	LDO	\$3, SP, 0	Laden der geretteten Registerwerte
	LDO	\$2, SP, 8	
	LDO	\$1, SP, 2*8	
	LDO	\$0, SP, 3*8	Laden der Rücksprungadresse
	ADD	SP, SP, 6*8	Korrektur des Stacks (4 Werte + 2 Parameter)
	GO	\$0, \$0, 0	Rücksprung





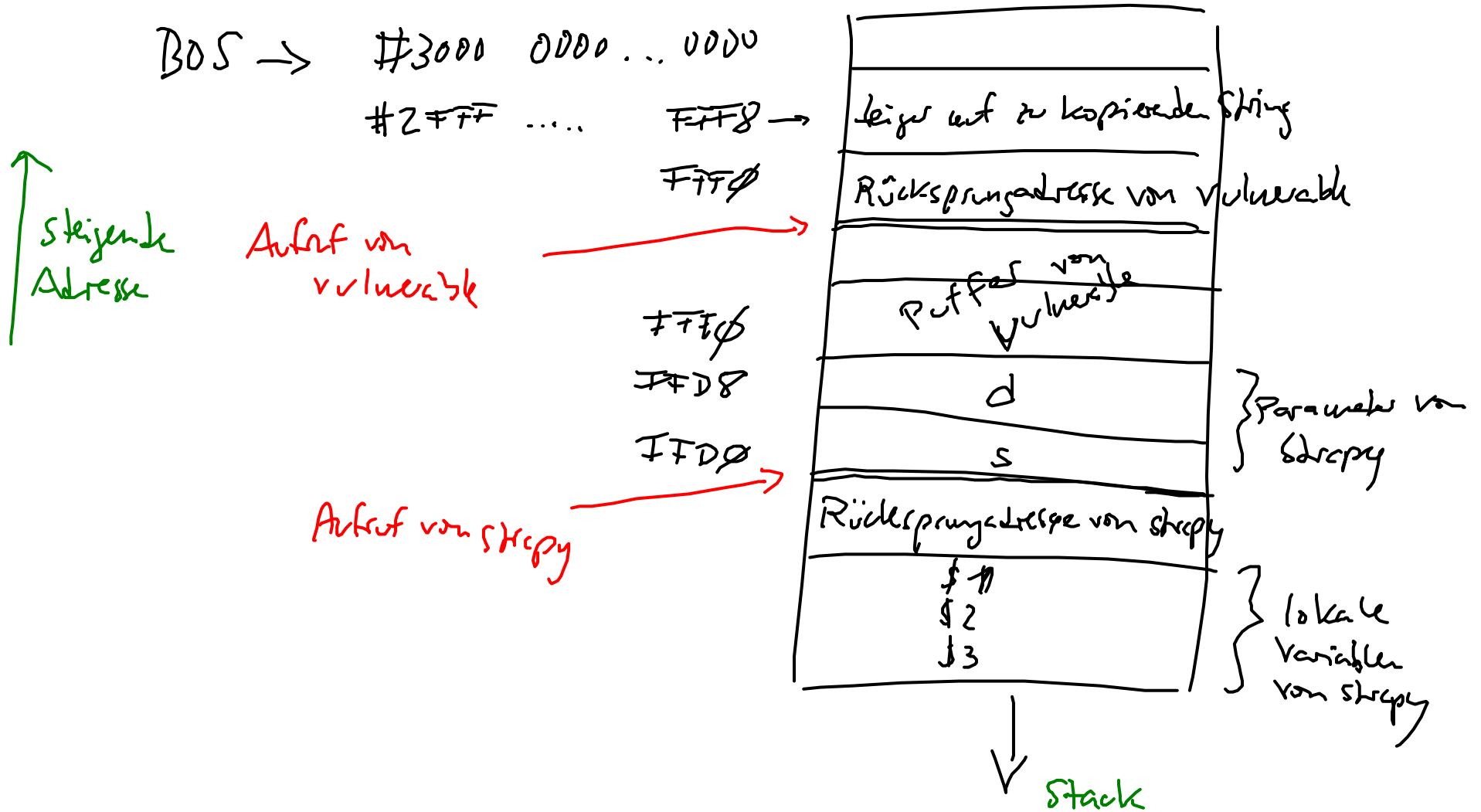
Eine schlecht programmierte Funktion

- * Unterprozedur vulnble(s) ("vulnerable")
- * erwartet auf dem Stack einen Zeiger auf einen String, kopiert diesen
- * String in einen Puffer auf dem Stack mit strcpy und gibt ihn aus

```
vulnble SUB    SP,SP,8
          STO   $0,SP,0      Rücksprungadresse sichern
          SUB   SP,SP,2*8    interner Puffer auf dem Stack
          SUB   SP,SP,16     Parameterübergabe an strcpy
          LDA   $0,SP,16     d: Anfangsadresse des internen Puffers
          STO   $0,SP,8      auf den Stack
          LDO   $0,SP,5*8    Adresse von s laden
          STO   $0,SP,0      und auf den Stack
          GO    $0,strcpy     Aufruf von strcpy
          LDA   $255,SP,0    Anfangsadresse des internen Puffers laden
          TRAP  0,Fputs,StdOut Pufferinhalt ausgeben
          LDO   $0,SP,16     Rücksprungadresse laden
          ADD   SP,SP,3*8
          ADD   SP,SP,8      Parameter auf dem Stack löschen
          GO    $0,$0,0      Rücksprung
```



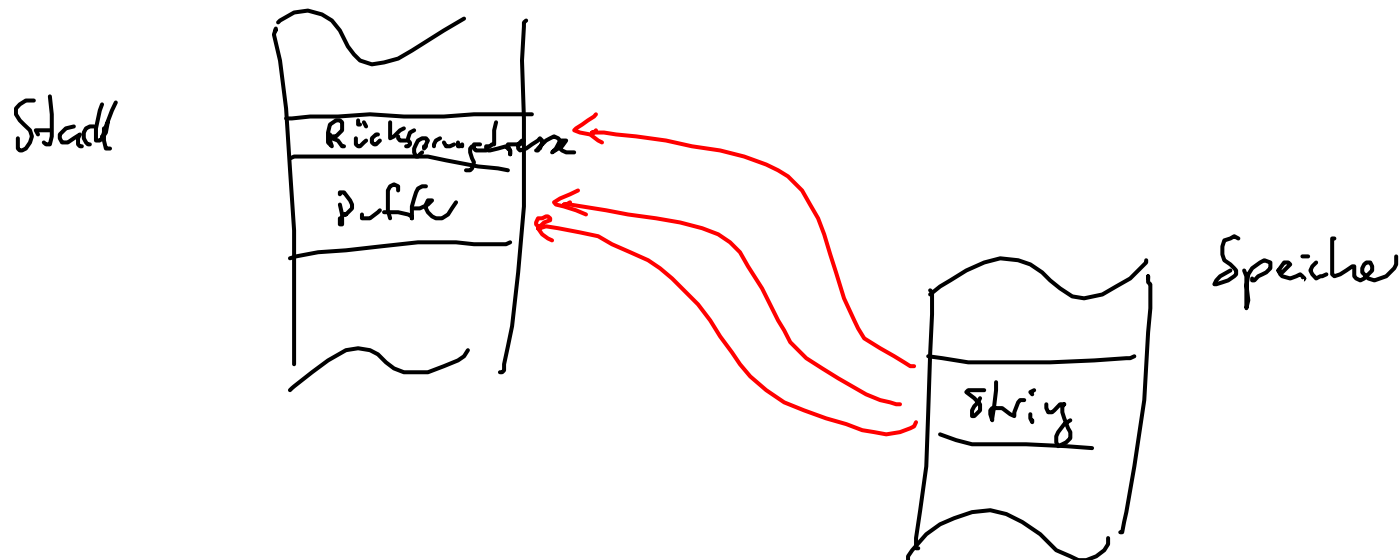
Struktur des Stacks bei Aufruf von strcpy





Gefahr durch fehlende Bereichsüberprüfung

- Der an die Funktion `vulnerable()` übergebene String kann durchaus länger als 16 Byte sein
- `strcpy` kopiert so lange, bis das erste Nullzeichen auftaucht
- Man kann durch einen speziell konstruierten String böse Sachen machen
 - das Programm zum Absturz bringen
 - das Programm an eine bestimmte Stelle verzweigen lassen
- Idee: Überschreibe die Rücksprungadresse mit dem "richtigen" Wert





Stack Smashing

- Ausnutzen eines Pufferüberlaufs und das Überschreiben der Rücksprungadresse auf dem Stack nennt man `stack smashing`
- `Stack smashing` ist eine beliebte Methode von Hackern, eigenen Code auf fremden Rechnern auszuführen
 - nicht erreichbare Funktion ist dann `shellcode`: startet eine Shell mit den Rechten des schlecht programmierten Programmes
- Es gibt auch noch andere Arten von Pufferüberläufen und Methoden, um sie auszunutzen
 - Heap overflows
 - Formatstring-Angriffe
- Pufferüberläufe sind für den Großteil der Schwachstellen in heutiger Software ursächlich (einfach schlecht programmiert)

Lehren:

- Beim Programmieren immer Bereichsgrenzen überprüfen
- Funktionen wie `strcpy` meiden (es gibt bessere Varianten, z.B. `strncpy`)
- ... mehr dazu in den Vorlesungen des Hauptstudiums