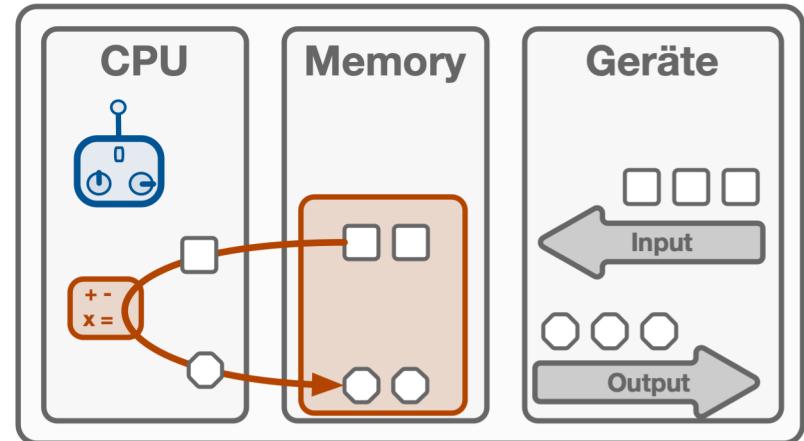

MIPS Befehlssatzarchitektur Review

[Adapted from Mary Jane Irwin for
Computer Organization and Design,
Patterson & Hennessy, © 2005, UCB]

(vonNeumann) Prozessor Gestaltung

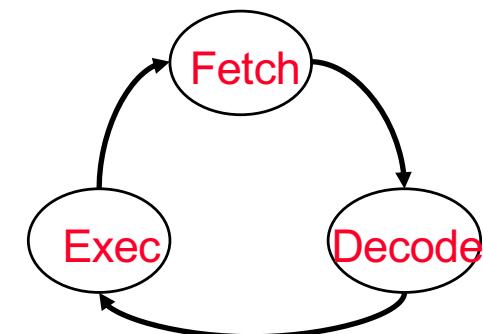
□ Steuerung

1. Anweisungen aus Memory laden
2. Erzeugen von Steuersignalen zur Kontrolle des Informationsflusses und Operationen auf den Komponenten des Datenpfads
3. Befehlsverarbeitungs-Zyklus

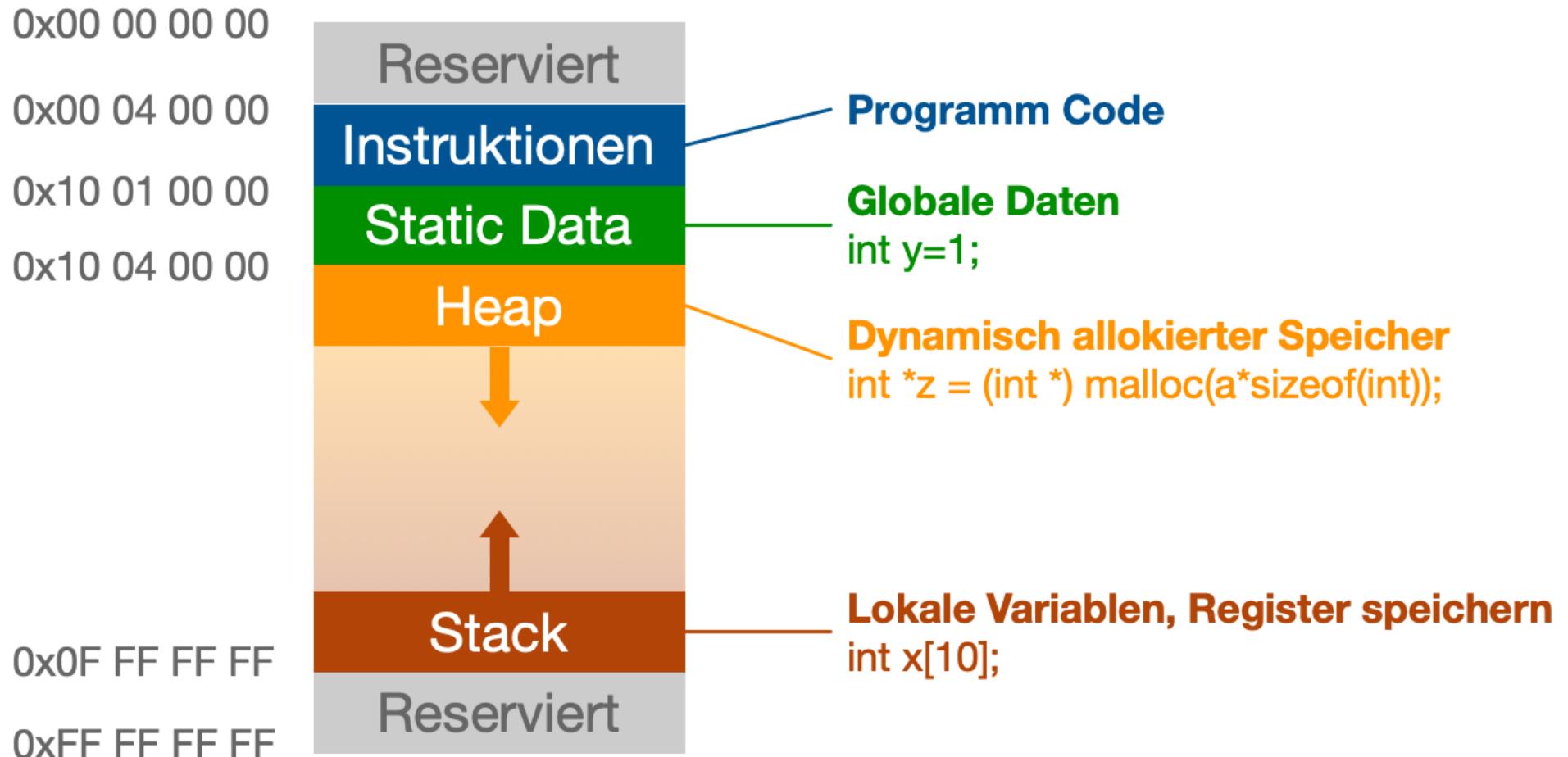


□ Datenpfad benötigt

- Komponenten – Funktionseinheiten und Speicher (z.B., Registerspeicher) zum ausführen der Befehle
- Verbindungen – Die Komponenten müssen so verbunden werden, dass die Befehle ausgeführt werden können und die Daten aus dem Memory geladen und wieder gespeichert werden können
- Aufgabe: Verarbeitung und Transport von Befehle und Daten



Einschub: Speicheraufteilung



RISC - Reduced Instruction Set Computer

- RISC Philosophie
 - Feste Länge der Befehle (Instruktionen)
 - Befehlssatz zum lesen/schreiben
 - Begrenzte Anzahl der Methoden zur Adressierung
 - Begrenzte Anzahl an Befehlen
- MIPS, Sun SPARC, HP PA-RISC, IBM PowerPC, Intel (Compaq) Alpha, ...
- Befehlssätze werden daran gemessen wie gut ein Compiler damit umgehen kann, nicht wie gut ein Mensch ihn benutzen kann.

Entwurfsziele RISC: Geschwindigkeit, Kosten (Design, Produktion, Testing, Verpackung), Grösse, Zuverlässigkeit, Energieverbrauch (embedded systems)

MIPS R3000 Instruction Set Architecture (ISA)

❑ MIPS Befehlskategorien

- Arithmetische & logische Befehle
- Speicherbefehle
- Sprung (und Verzweigungsbefehle)
- Gleitkommazahlen
 - Gleitkommaeinheit, FPU
- Memory Management
- Special

Register

R0 - R31

PC

HI

LO

3 Befehlsformate: alle 32 Bits breit

OP	rs	rt	rd	sa	funct	R Format
OP	rs	rt	immediate			I Format
OP		jump target				J Format

MIPS Arithmetische Befehle

❑ MIPS Assembler Sprache: Arithmetische Befehle

add \$t0, \$s1, \$s2

sub \$t0, \$s1, \$s2

- ❑ Jeder arithmetische Befehl führt nur eine einzige Operation aus
 - ❑ Jeder arithmetische Befehl passt in 32 bit und spezifiziert genau drei Operanden
 - ❑ Die Reihenfolge der Operanden ist fest (Ziel zuerst)
 - ❑ Diese Operanden sind alle Teil des Datenpfad und des Registerspeicher (\$t0, \$s1, \$s2) – bezeichnet mit \$
-
- destination \leftarrow source1 op source2

MIPS Register Konvention

Name	Register Nummer	Verwendung	Sichern bei Funktionsaufruf
\$zero	0	constant 0 (hardware)	n.a.
\$at	1	reserved for assembler	n.a.
\$v0 - \$v1	2-3	returned values	nein
\$a0 - \$a3	4-7	arguments	nein
\$t0 - \$t7	8-15	temporaries	nein
\$s0 - \$s7	16-23	saved values	ja
\$t8 - \$t9	24-25	temporaries	nein
\$gp	28	global pointer	ja
\$sp	29	stack pointer	ja
\$fp	30	frame pointer	ja
\$ra	31	return addr (hardware)	ja

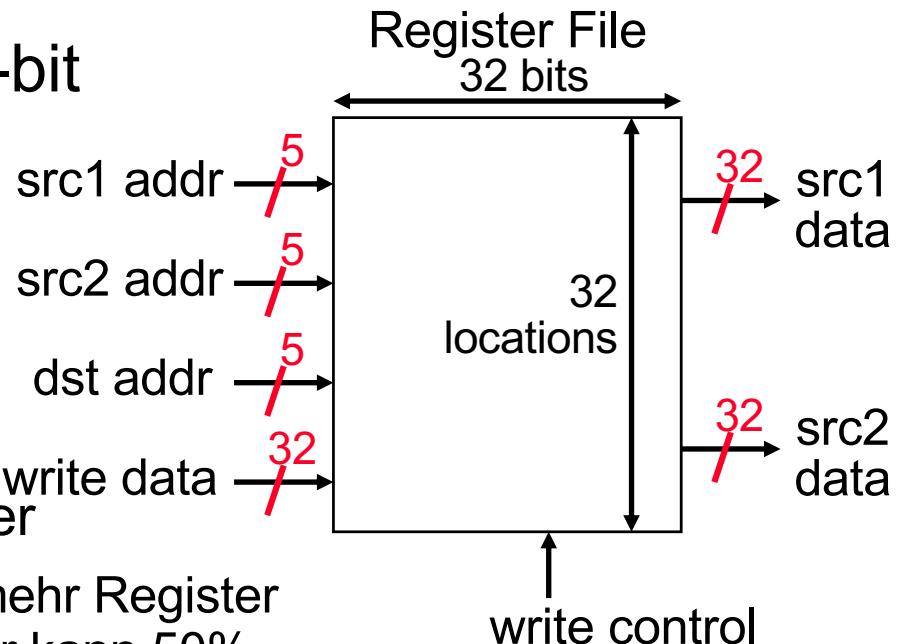
MIPS Registerspeicher

- Besitzt 32 Register mit je 32-bit

- Zwei lese ports
 - Ein schreib port

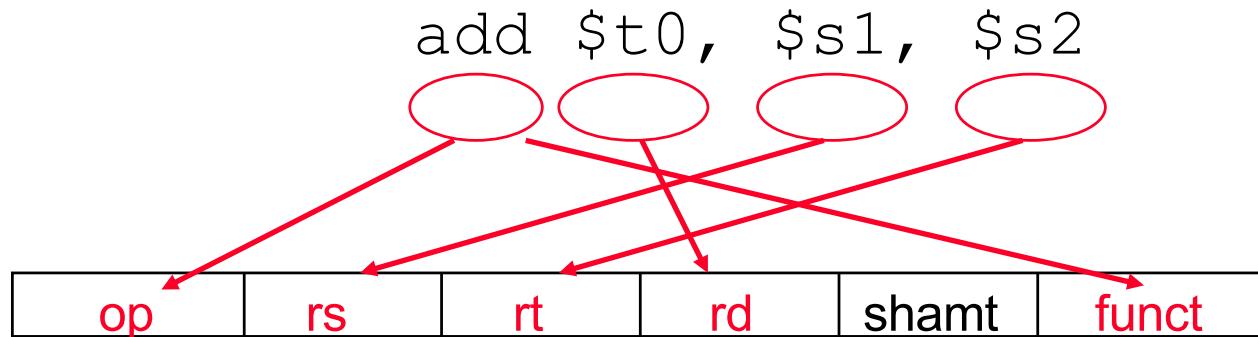
- Register sind

- Schneller als der Hauptspeicher
 - Aber, Registerspeicher mit mehr Register (z.B., Ein 64 Wörter-Speicher kann 50% langsamer sein als 32 Wörter-Speicher)
 - Lese/Schreib Port steigert die Geschwindigkeit quadratisch
 - Für Compiler einfacher zu benutzen
 - Z.B., $(A*B) - (C*D) - (E*F)$ kann in beliebiger Reihenfolge verarbeitet werden (im Gegensatz zu einem Stack)
 - Kann Variablen so speichern, dass
 - Codedichte sich verbessert (Register benötigen weniger Bits zur Adressierung als der Hauptspeicher)



Maschinensprache - Add Befehl

- ❑ Befehle, Registers und Datenwörter sind 32 Bits lang
- ❑ Arithmetisches Befehlsformat - (Im R Format):



op	6-bits	opcode der die gewünschte Operation spezifiziert
rs	5-bits	register-Adresse des ersten Operanden
rt	5-bits	register-Adresse des zweiten Operanden
rd	5-bits	register-Adresse des Resultats
shamt	5-bits	shift amount (für shift Befehle)
funct	6-bits	function Code der den opcode erweitert

MIPS Befehle für Zugriff auf Hauptspeicher

- ❑ MIPS hat zwei Grundarten von **data transfer** Befehlen für den Speicherzugriff

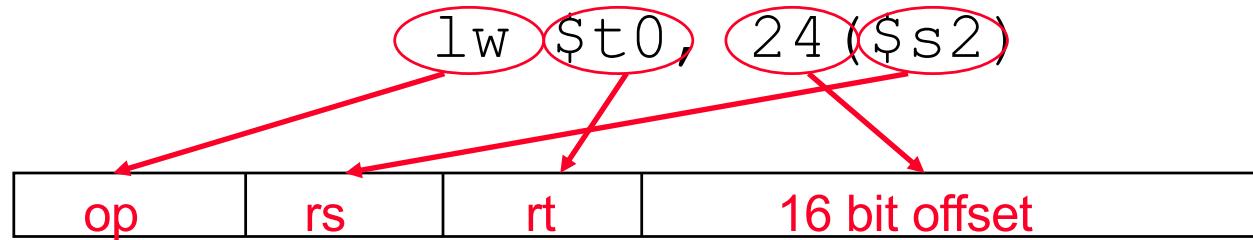
lw \$t0, 4(\$s3) #lese vom Hauptspeicher

sw \$t0, 8(\$s3) #schreib in den Hauptspeicher

- ❑ Daten werden entweder vom Hauptspeicher in den Registerspeicher geladen (lw) oder von den Register zurück in den Hauptspeicher geschrieben (sw)
- ❑ Die 32 Bit Adresse des Hauptspeichers wird aus der **Basisadresse** und einem **Offset** gebildet.
 - Ein 16-Bit Offset limitiert den Speicherbereich auf $\pm 2^{13} = 8,192$ **Wörter** (entspricht $\pm 2^{15} = 32,768$ **Bytes**) um die Basisadresse
 - Hinweis: Der Offset kann Positive oder Negativ sein

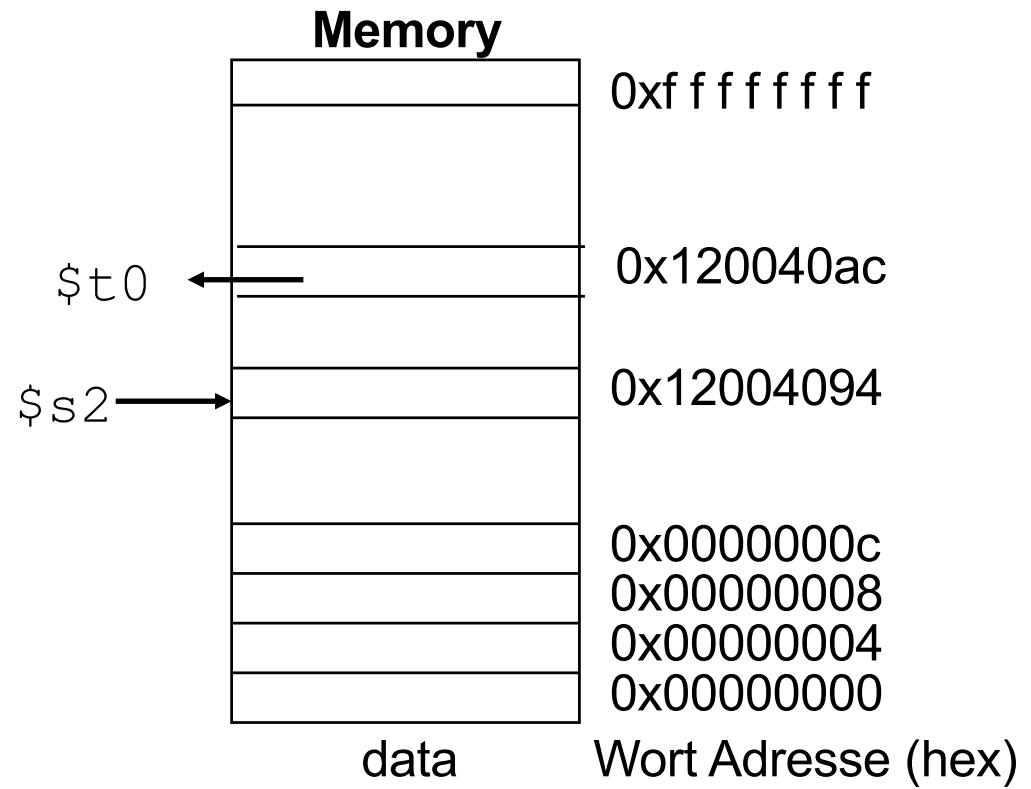
Machinensprache - Ladebefehl

- ❑ Lade/Speicher Befehlsformat (I Format):



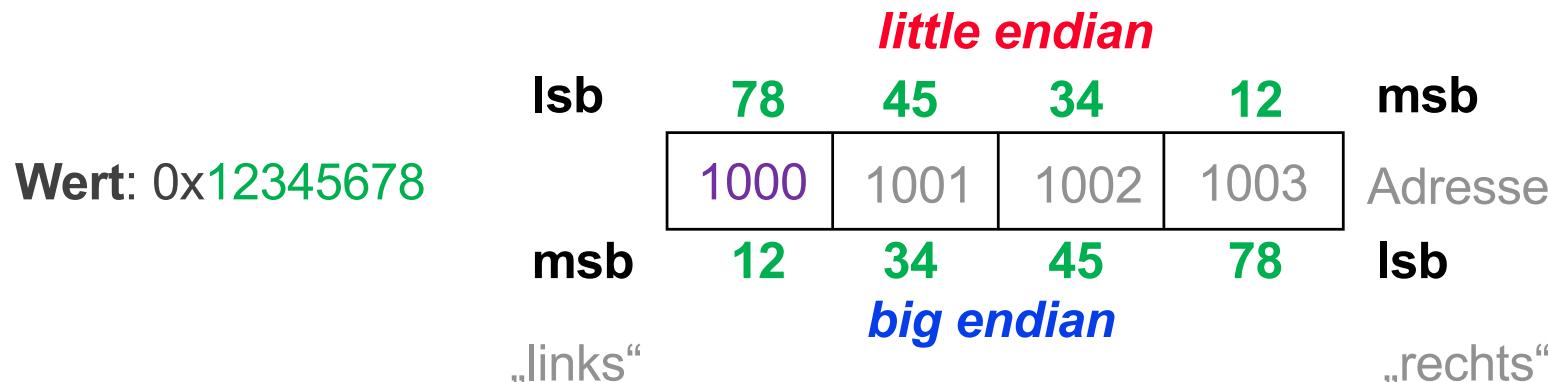
$$24_{10} + \$s2 =$$

$$\begin{array}{r} \dots 0001\ 1000 \\ + \underline{\dots\ 1001\ 0100} \\ \hline \dots 1010\ 1100 = \\ 0x120040ac \end{array}$$



Byte Addresses

- Da 8-Bit Bytes äussert nützlich sind, adressieren die meisten Architekturen einzelne **Bytes** im Hauptspeicher
 - Die Speicheradresse eines Wort muss ein Vielfaches von 4 sein (**alignment restriction**)
- **Little Endian:** Das niedrigwertigste Byte ist **Wortadresse**
Intel 80x86, DEC Vax, DEC Alpha
- **Big Endian:** Das hochwertigste Byte ist **Wortadresse**
IBM 360/370, Motorola 68k, **MIPS**, Sparc, HP PA



Test

```
#define LITTLE_ENDIAN 0  
  
#define BIG_ENDIAN 1  
  
int endian() {  
  
    int i = 1;  
  
    char *p = (char *)&i;  
  
    if (p[0] == 1)  
        return LITTLE_ENDIAN;  
  
    else  
        return BIG_ENDIAN;  
  
}
```

Laden und Speichern von Bytes

- ❑ MIPS bietet spezielle Befehle für einzelne Bytes

lbu \$t0, 1(\$s3) #Lade Byte vom Hauptspeicher

sb \$t0, 6(\$s3) #Schreib Byte in Hauptspeicher



- ❑ Welche 8 Bit werden geladen/gespeichert?

- Lädt ein Byte aus dem Speicher und platziert es in die rechts gelegenen (niederwertigen) Bits eines Registers.
 - Was passiert mit den anderen 24 Bits im Register?
- Speichert die niederwertigen acht Bits eines Registers in die bezeichnete Speicherposition
 - Was passiert mit den anderen 24 Bits des Wort im Hauptspeicher?

Unser erstes Beispiel

- C Code in Assembler kompiliert (32Bit)
 - Argumente werden in \$a0-\$a3 übergeben
 - Rücksprungadresse \$ra
- Was macht dieser Code?

```
swap(int v[], int k);  
{ int temp;  
    temp = v[k];  
    v[k] = v[k+1];  
    v[k+1] = temp;  
}
```

```
swap: muli $t0, $a1, 4  
        add $t0, $a0, $t0  
        lw $t1, 0($t0)  
        lw $t2, 4($t0)  
        sw $t2, 0($t0)  
        sw $t1, 4($t0)  
        jr $ra
```

MIPS Kontrollfluss Anweisungen

- MIPS **bedingte Sprungbefehle** (conditional branch instruction):

bne \$s0, \$s1, Lbl #spring zu Lbl if $\$s0 \neq \$s1$

beq \$s0, \$s1, Lbl #spring zu Lbl if $\$s0 = \$s1$

- Beispiel: if ($i == j$) $h = i + j;$

bne \$s0, \$s1, Lbl1

add \$s3, \$s0, \$s1

Lbl1: ...

- Befehls Format (**I** Format):

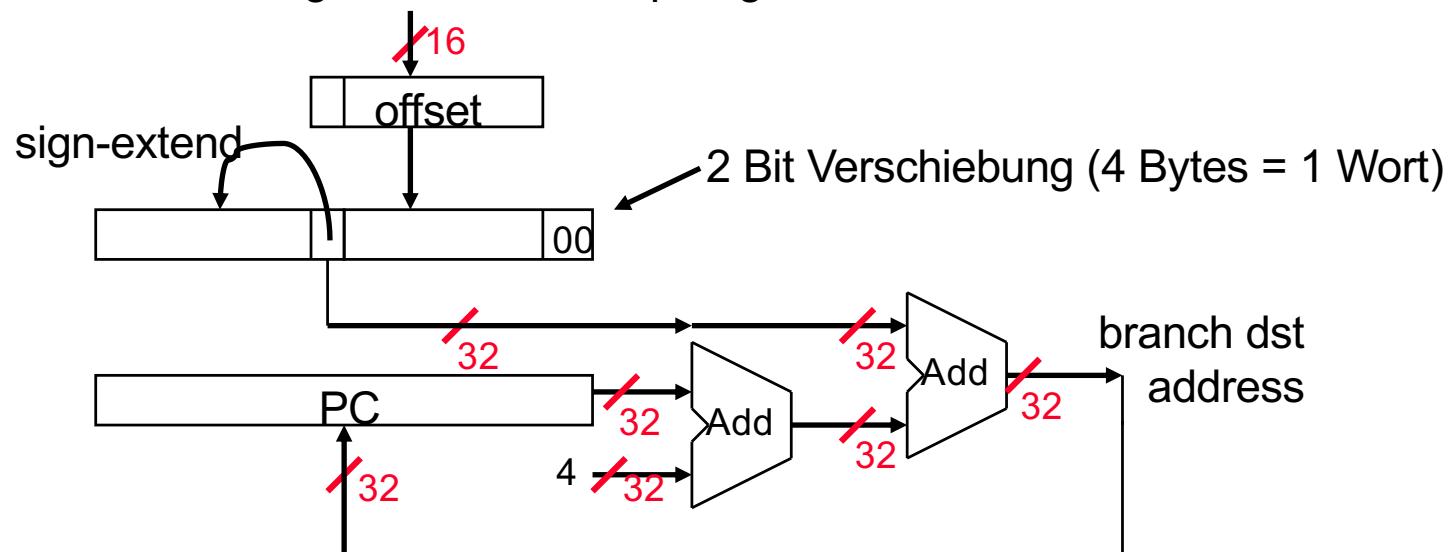
op	rs	rt	16 bit offset
----	----	----	---------------

- Wie wird die **32-Bit** Sprungadresse festgelegt?

Festlegen von Sprungzielen

- Addiere ein Register (wie in lw and sw) zum 16-bit Offset
 - Welchen Register? Befehlszähler/Programmzähler (PC)
 - Die Nutzung des PC wird automatisch vorgenommen.
 - Der PC wird während des **fetch** Zyklus aktualisiert ($PC+4$)
Dadurch zeigt er jeweils auf den nächsten Befehl.
 - Limitiert die Sprungdistanz auf -2^{15} to $+2^{15}-1$ Befehle um den (Befehl nach dem) Sprungbefehl. Die meisten Sprünge sind lokal.

Die niederwertigen 16 Bits des Sprungbefehls



Mehr Sprungbefehle

- Wir haben nun beq, bne, aber was ist mit anderen Sprüngen wie z.B “*spring wenn ist kleiner als*“?
Dafür benötigen wir einen zusätzlichen Befehl, slt
- slt (*set on less than*) Befehl:

```
slt $t0, $s0, $s1      # if $s0 < $s1      then  
                        # $t0 = 1 (True)   else  
                        # $t0 = 0 (False)
```

- Befehlsformat (**R** Format):

op	rs	rt	rd		funct
----	----	----	----	--	-------

Mehr Sprungbefehle - Fortsetzung

- Nutzen von slt, beq, bne, und dem fixen Wert 0 im Register \$zero um andere Bedingungen zu erzeugen

- Kleiner als

blt \$s1, \$s2, Label

slt \$at, \$s1, \$s2 #\$at set to 1 if

bne \$at, \$zero, Label # \$s1 < \$s2

- Kleiner als oder gleich

ble \$s1, \$s2, Label

- Grösser als

bgt \$s1, \$s2, Label

- Grösser als oder gleich

bge \$s1, \$s2, Label

- Solche Sprünge sind im Befehlssatz als Pseudobefehle vorhanden - erkannt (und erweitert) vom Assembler

- Deswegen benötigt der Assembler ein reserviertes Register (\$at)

Weitere Sprungsanweisungen

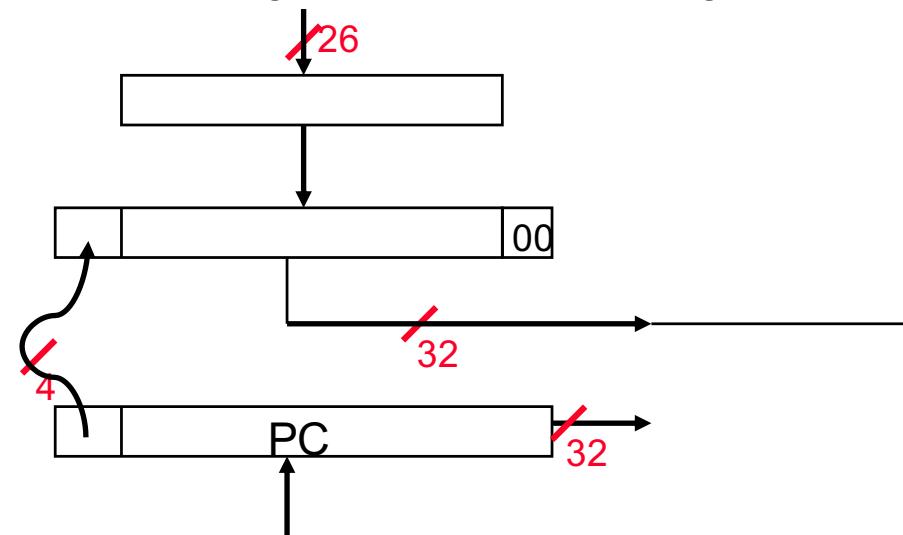
- ❑ MIPS hat auch unbedingte Sprungbefehle (**jump** instruction):

j label #go to label

- ❑ Befehlsformat (**J** Format):



Die niederwertigen 26 Bits des Sprungbefehls



Ein Sprung weit weg

- ❑ Was wenn ein Sprungziel eines Sprungbefehls im I-Format weiter weg ist, als mit 16 Bit beschreibbar?
- ❑ Dann fügt der Assembler einen unbedingten Sprung zum Sprungziel ein und invertiert die Bedingung

beq \$s0, \$s1, L1

wird zu:

bne \$s0, \$s1, L2
j L1

L2:

Befehle für Funktionsaufrufe/Prozedurauftrufe

- MIPS Funktionen (Prozeduren/Subroutinen)

jal ProcedureAddress #jump and link

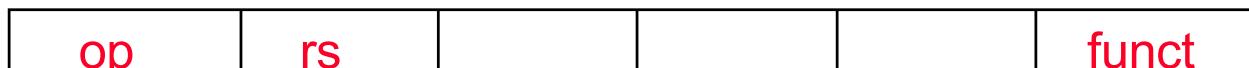
- Speichert PC+4 im Register **\$ra**. Dort steht der nächste Befehl welcher nach dem **return** ausgeführt werden muss
- Befehlsformat (**J** Format):



- Ausführen der Prozedur und **return** mit einem

jr \$ra #return

- Befehlsformat (**R** Format):



Beispiel: Funktion/Prozedur

```
int leaf_ex  
    (int g, int h, int i, int j)  
{  
    int f;  
    f = (g+h) - (i+j);  
    return f;  
}
```

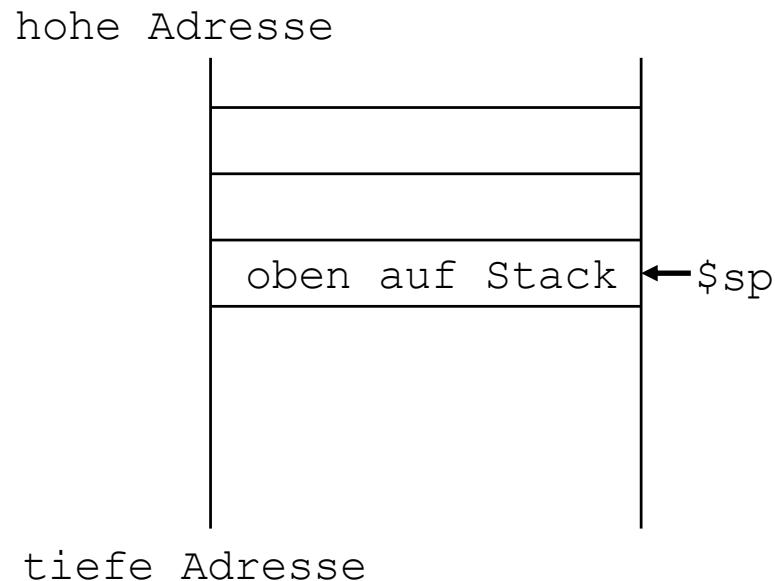
Hier werden alle Register gerettet.
Konvention ist, dass nur die \$s0-\$s7
gerettet werden müssen.

Die Argumente werden in \$a0-\$a3
übergeben,
Resultate in \$v0,\$v1 zurückgegeben.

```
leaf_ex:  
addi $sp,$sp,-12  
sw $t1, 8($sp)  
sw $t0, 4($sp)  
sw $s0, 0($sp)  
add $t0,$a0,$a1  
add $t1,$a2,$a3  
sub $s0,$t0,$t1  
add $v0,$s0,$zero  
lw $s0, 0($sp)  
lw $t0, 4($sp)  
lw $t1, 8($sp)  
addi $sp,$sp,12  
jr $ra
```

Auslagern von Register (Spilling Registers)

- Was wenn eine aufgerufene Funktion mehr Register benötigt oder eine Funktion rekursiv ist?
 - Den **Stack** – (**last-in-first-out Queue**) – im Hauptspeicher zur Übergabe zusätzlicher Werte oder speichern von (rekursiven) return Adressen benutzen.



- Nutzen des allgemeinen Register, \$sp, ("wächst" von hohen zu tiefen Adressen)
 - Daten auf den Stack legen – **push**
 $\$sp = \$sp - 4$
Daten auf Stack bei neuen \$sp
 - Daten vom Stack entnehmen – **pop**
Daten vom Stack bei \$sp
 $\$sp = \$sp + 4$

Beispiel: Rekursive Funktion

```
int fact( int n )
{
    if (n < 1)
        return (1);
    else
        return ( n * fact(n-1));
}
```

\$ra und \$a0 müssen auf den Stack gerettet werden, damit sie nach dem Funktionsaufruf wieder verfügbar sind.

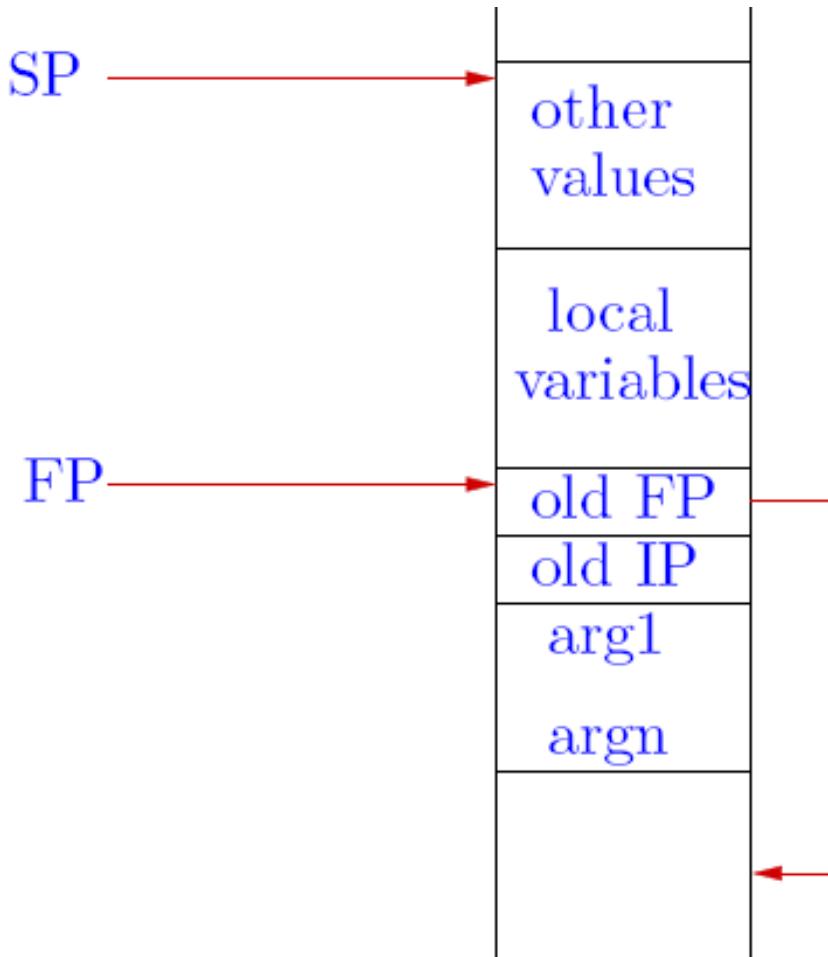
```
fact: addi $sp,$sp,-8
      sw $ra, 4($sp)
      sw $a0, 0($sp)
      slti $t0, $a0, 1
      beq $t0,$zero,L1
      addi $v0,$zero,1
      addi $sp,$sp,8
      jr $ra
L1:   addi $a0,$a0,-1
      ← jal fact
      → lw $a0,0($sp)
      lw $ra, 4($sp)
      addi $sp,$sp,8
      mul $v0,$a0,$v0
      jr $ra
```

Funktionsaufrufe und Stack-Frame

- ❑ Jede Funktion hat die Möglichkeit für lokale Variablen Speicher auf dem Stack zu allozieren.
 - Ein solcher Datenblock, wird „Stack-Frame“ genannt.
- ❑ Der Frame Pointer (FP) zeigt die Position des aktuellen Frame. Das ermöglicht einen einfachen Zugriff auf die lokalen Variablen während der Laufzeit.
- ❑ Nach dem return aus einer Funktion, muss der Befehl nach dem Funktionsaufruf ausgeführt werden.
- ❑ Speichern des alten Befehlszähler (PC) im Stack-Frame.

Beispiel: Stack

- ❑ Beim Rücksprung aus einer Funktion, wird das aktuelle Stack-Frame entfernt und die Ausführung im letzten Stack-Frame fortgesetzt
- ❑ Speicher alten FP auf dem Stack.



Ein einfaches Beispiel eines Funktionsaufruf

```
/* function.c */
void f (int x, int y){
    int a,b,c;
}

int main(){
    int x = 10;
    int y = 20;
    f(x,y);
}
```

Schauen wir uns den kompilierten Code an

Der Aufrufer (caller)

```
(gdb) disassemble main
...
0x804832f <main+19>: push $0x14      ← push auf den Stack
0x8048331 <main+21>: push $0xa        ←
0x8048333 <main+23>: call 0x8048314 <f>   Sprung zur Funktion
0x8048337 <main+27>: mov $0x0,%eax
...
```

- Hinweis: Kein MIPS sondern INTEL x86 (CISC)
 - Output mit dem GNU Debugger *gdb* erzeugt.
- Die Argumente werden auf den Stack gepusht
 - 0x14 = 20
 - 0xA = 10
- Anschliessend wird die Funktion aufgerufen
 - Rücksprungadresse auf Stack gepusht

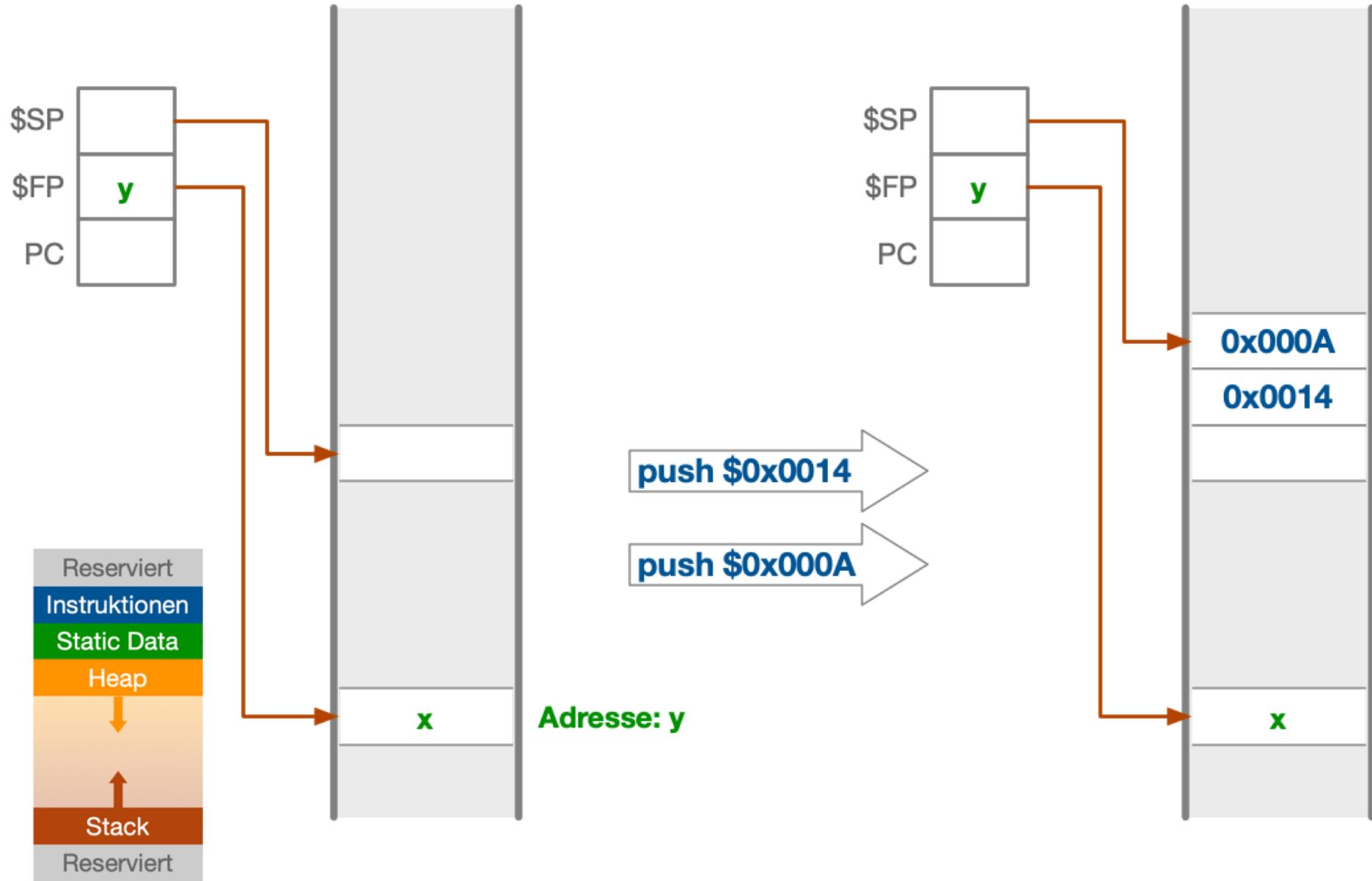
Und der Aufgerufene (callee)

```
(gdb) disassemble f
0x8048314 <f+0>: push %ebp
0x8048315 <f+1>: mov $esp,%ebp
0x8048317 <f+3>: sub $0xc,%esp
0x804831a <f+6>: leave
0x804831b <f+7>: ret
```

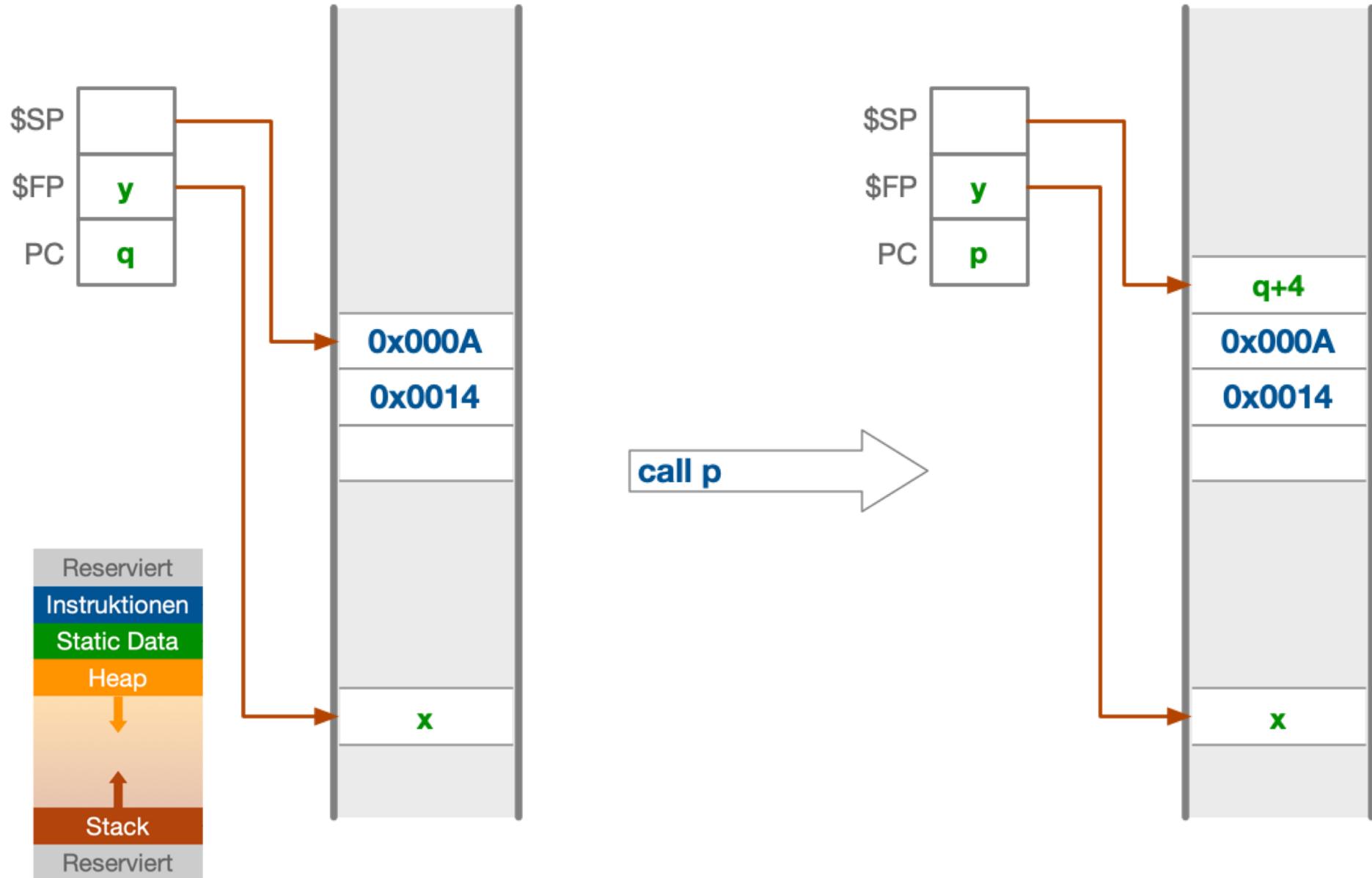
Push alten \$FP auf Stack
Update \$FP (Start Stack-Frame)
Allociere Speicher für 3 x int

- ❑ Speichere alten FP, update FP
 - *mov*: Stack Pointer (%esp) → Frame Pointer (%ebp) Register
 - *sub*: Allociere Speicher für lokale Variablen (0xC = 12 Bytes für 3 * int) also PC wiederherstellen, PC vom Stack entfernen
- ❑ Berechnungen ausführen (In obigen Beispiel ausgelassen)
- ❑ leave, ret
 - *leave*: FP wiederherstellen und FP vom Stack entfernen
 - *ret*: Return, also PC wiederherstellen, PC vom Stack entfernen

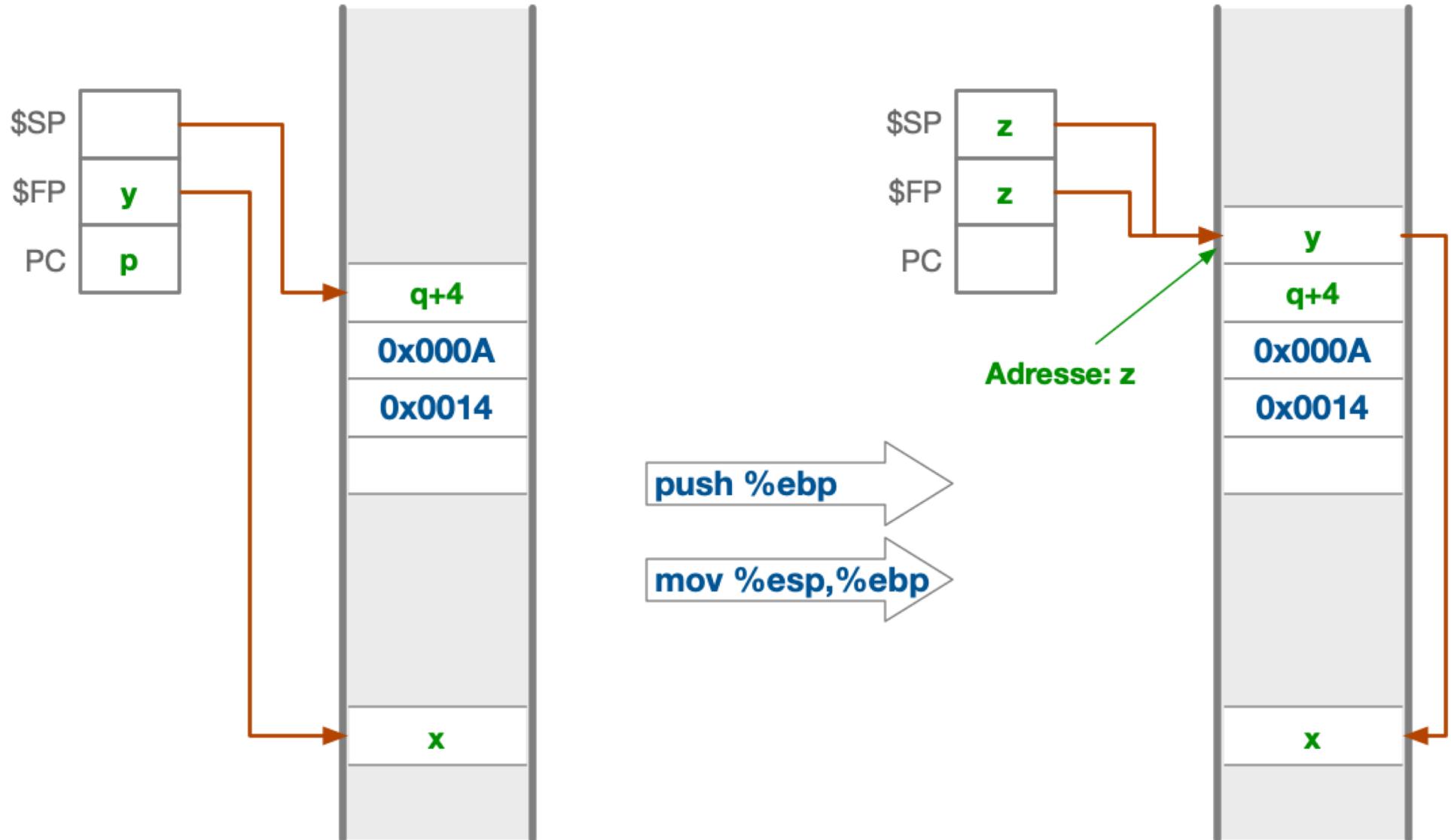
Zur Laufzeit: Argumente pushen



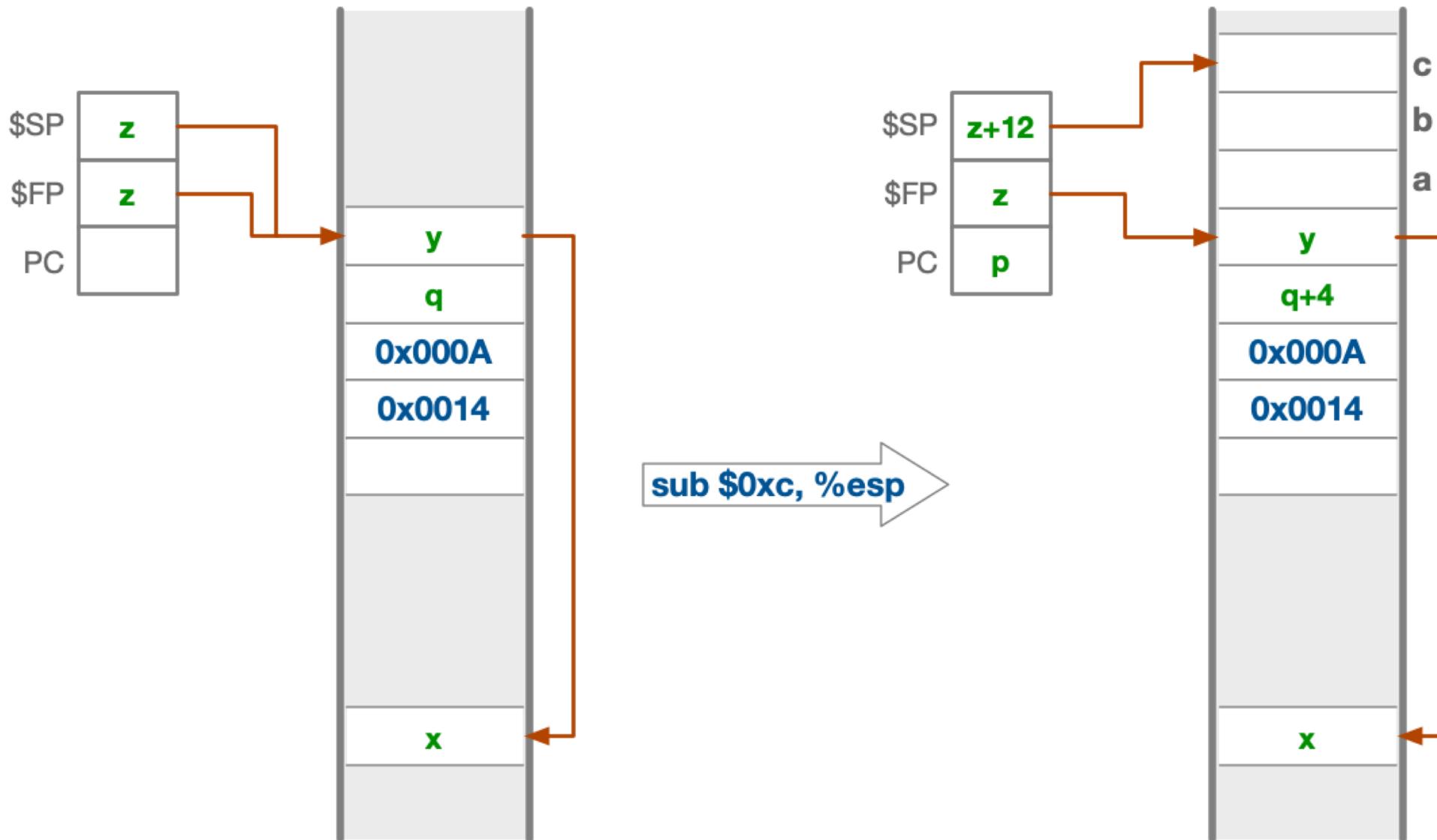
Funktionsaufruf: Speicher PC und update PC



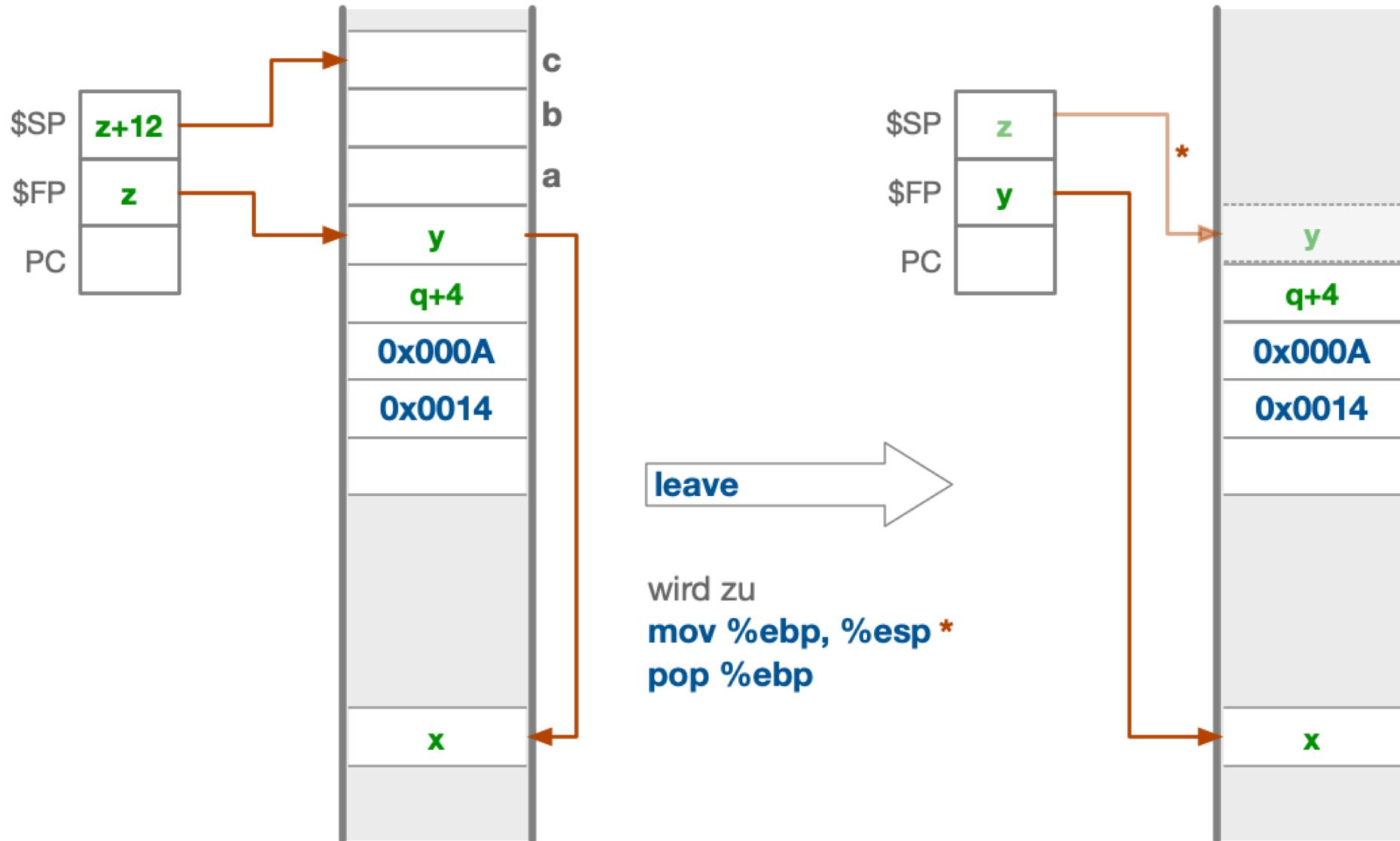
In der Funktion: speicher FP und update FP



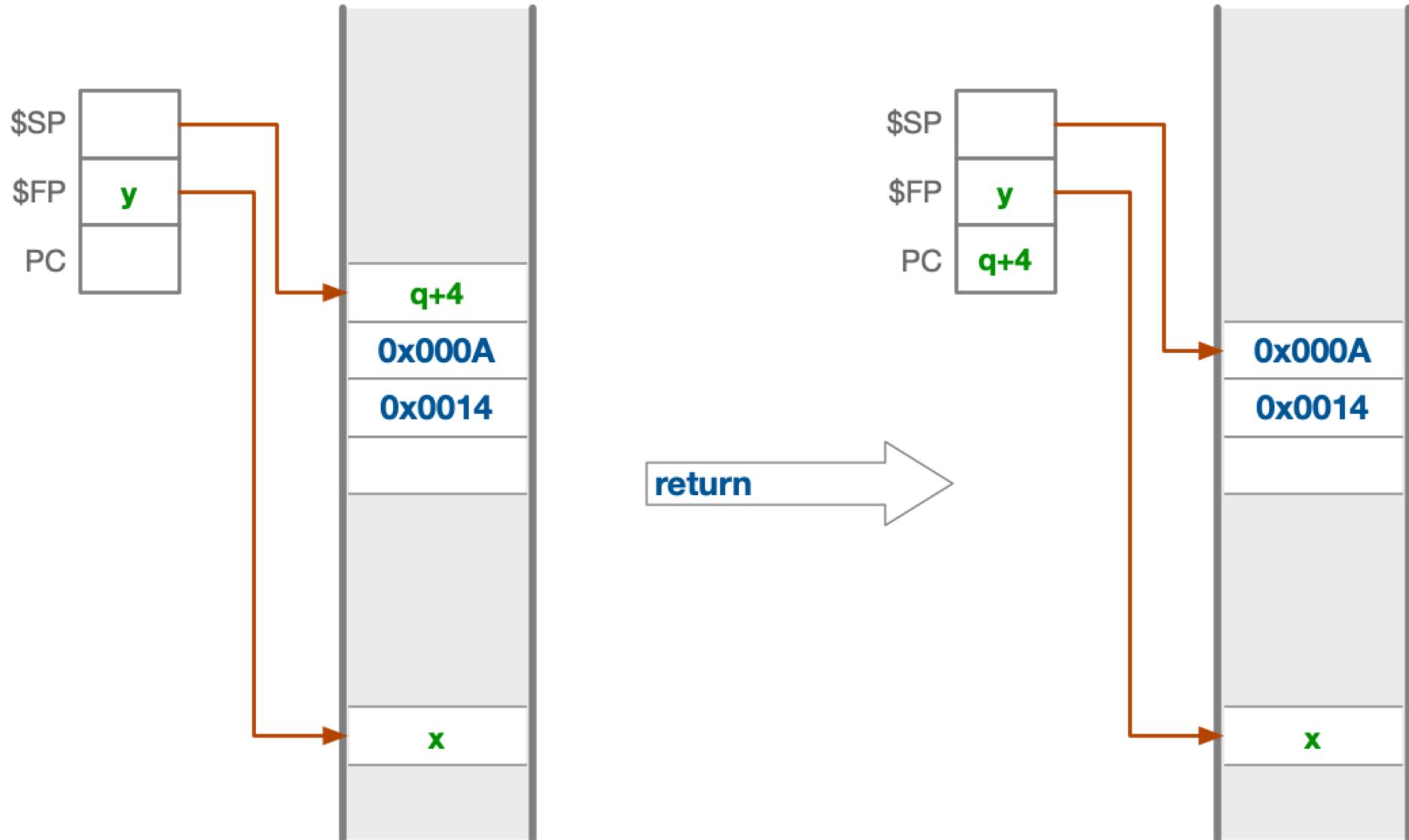
Allociere Speicher für lokale Variablen



Funktionsende: FP wiederherstellen und pop FP



Returning: PC wiederherstellen und pop PC



Stack overflow

Die Rücksprungadresse liegt auf dem Stack.
=> Diese kann auch überschrieben werden!

```
void f () {           int main() {
    int a[10];         int x = 10;
    a[15] += 7;        f();
}                         x = 20;
                           printf("x=%d!\n", x);
}
```

Output: x=10!

- Wir überspringen den Befehl **x = 20;** !
- Wo ist die Rücksprungadresse gespeichert (a[15])?
- Was könnte die neue Rücksprungadresse sein (erhöht um 7)?

Organisation des Stack

- ❑ Organisation des Stack: a[0], . . . , a[9], alter FP, alter PC
- ❑ Daher die Rücksprungadresse an der Stelle a[11] erwartet.
- ❑ Nicht immer!! Compiler Optimierungen können leere Stellen zwischen Array den folgenden Daten erzeugen.
- ❑ Ein blick auf den dekompliierten Code.

```
0x8048344 <f>:    push %ebp  
0x8048345 <f+1>:  mov $esp,%ebp  
0x8048347 <f+3>:  sub $0x38,%esp    # 56 Byte für 10 int
```

- Speicher allokiert nach alten FP ist $0x38 = 56 = 4 * 14$ Bytes.
- Daher ist die Rücksprungadresse an der Adresse a[15]

a[15]+=7

```
...
0x8048369 <main+23>:  call  0x8048344 <f>
0x804836e <main+28>:  movl  0x14,0 xfffffffc (%ebp) # x = 20
0x8048375 <main+35>:  sub   $0x8,%esp  # Neue Rücksprungadresse
...
```

- ❑ Befehl x = 20; benötigt 35 - 28 = 7 Bytes.
- ❑ Daher addieren wir in der Funktion f a[15] +=7 um diesen Befehl zu überspringen
- ❑ Neben dem verändern von Daten kann so auch der Ablauf verändert werden!
 - Ohne direkt die Befehle im Speicher zu überschreiben

Sicherstellen geeigneter Inputs

Schwächen können auch von User ausgenutzt (exploited) werden in dem sie geeignete Inputs liefern.

```
int main(int argc, char *argv[]) {  
    char s[1024];  
    strcpy(s, argv[1]);  
    ...  
}
```

- ❑ Ein geeigneter Input kann die Rücksprungadresse überschreiben.
- ❑ Im Minimum kann das Programm abrupt terminieren.
- ❑ Ein guter Angreifer kann gewünschten Code ausführen (shellcode)
 - Dieser Code ist ein Teil des Input-Strings

MIPS Immediate Befehlsformat

- Kleine Konstanten werden beim programmieren oft benutzt
- Mögliche Ansätze?
 - Lege “typische Konstanten” in den Speicher und Lade diese
 - Nutze fest verdrahtete Register (wie \$zero) für Konstanten wie 1
 - Nutze spezielle Befehle die Konstanten enthalten!

addi \$sp, \$sp, 4 # \$sp = \$sp + 4

slti \$t0, \$s2, 15 # \$t0 = 1 if \$s2 < 15

- Befehlsformat (I Format):



- Die Konstante ist direkt in der Befehle selber definiert!
 - Das Immediate Format limitiert Wertebereich auf $+2^{15}-1$ to -2^{15}

Wie sieht es mit grösseren Konstanten aus?

- Wir möchten auch in der Lage sein eine 32 Bit Konstante in ein Register laden zu können.
Dazu müssen wir zwei Befehle nutzen
- Erst ein neuer "load upper immediate" Befehl

lui \$t0, 1010101010101010

16	0	8	1010101010101010
----	---	---	------------------

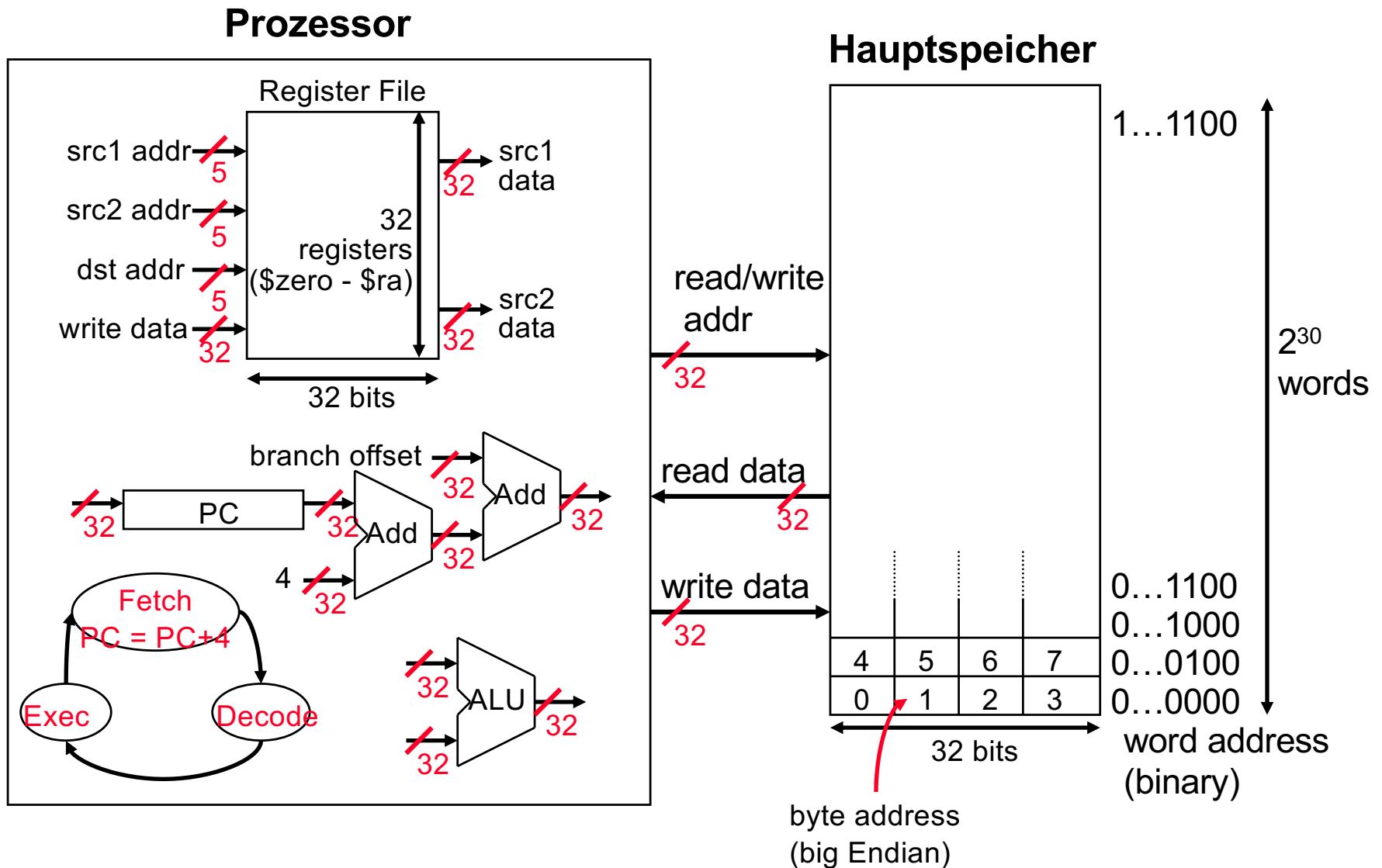
- Dann "or immediate" (logical OR)

ori \$t0, \$t0, 1010101010101010

1010101010101010	0000000000000000
0000000000000000	1010101010101010

1010101010101010	1010101010101010
------------------	------------------

MIPS Organisation bis jetzt

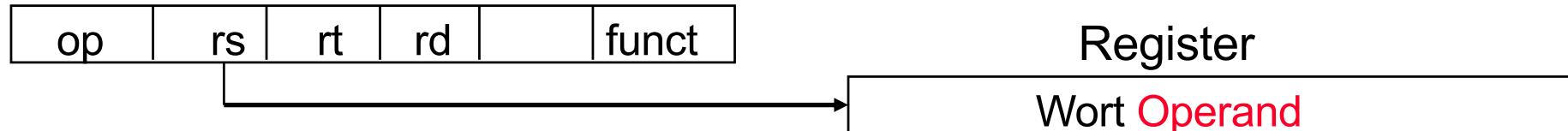


MIPS ISA bis jetzt

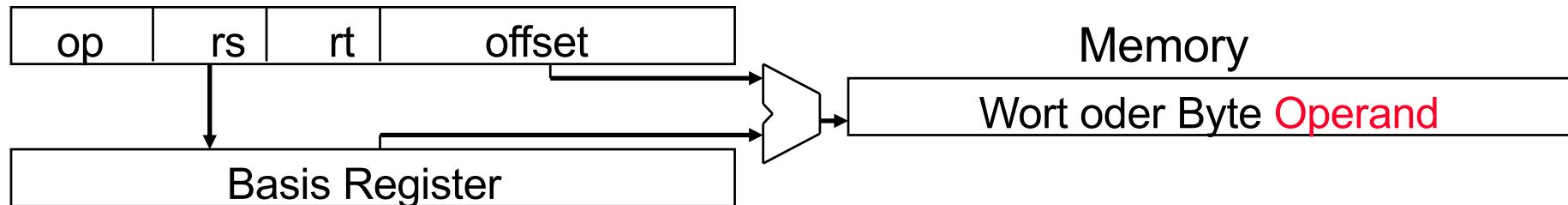
Category	Instr	Op Code/ Funct	Example	Meaning
Arithmetic (R & I format)	add	0 and 32	add \$s1, \$s2, \$s3	\$s1 = \$s2 + \$s3
	subtract	0 and 34	sub \$s1, \$s2, \$s3	\$s1 = \$s2 - \$s3
	add immediate	8	addi \$s1, \$s2, 6	\$s1 = \$s2 + 6
	or immediate	13	ori \$s1, \$s2, 6	\$s1 = \$s2 v 6
Data Transfer (I format)	load word	35	lw \$s1, 24(\$s2)	\$s1 = Memory(\$s2+24)
	store word	43	sw \$s1, 24(\$s2)	Memory(\$s2+24) = \$s1
	load byte	32	lb \$s1, 25(\$s2)	\$s1 = Memory(\$s2+25)
	store byte	40	sb \$s1, 25(\$s2)	Memory(\$s2+25) = \$s1
	load upper imm	15	lui \$s1, 6	\$s1 = 6 * 2 ¹⁶
Cond. Branch (I & R format)	br on equal	4	beq \$s1, \$s2, L	if (\$s1==\$s2) go to L
	br on not equal	5	bne \$s1, \$s2, L	if (\$s1 !=\$s2) go to L
	set on less than	0 and 42	slt \$s1, \$s2, \$s3	if (\$s2<\$s3) \$s1=1 else \$s1=0
	set on less than immediate	10	slti \$s1, \$s2, 6	if (\$s2<6) \$s1=1 else \$s1=0
Uncond. Jump (J & R format)	jump	2	j 2500	go to 10000
	jump register	0 and 8	jr \$t1	go to \$t1
	jump and link	3	jal 2500	go to 10000; \$ra=PC+4

Überblick über MIPS Adressierungsarten

- Registeradressierung – Operand ist in einem Register



- Basisadressierung – Operand ist im Hauptspeicher an der Adresse welche die Summe aus einem Registerinhalt und einer Konstante im Befehl ergibt



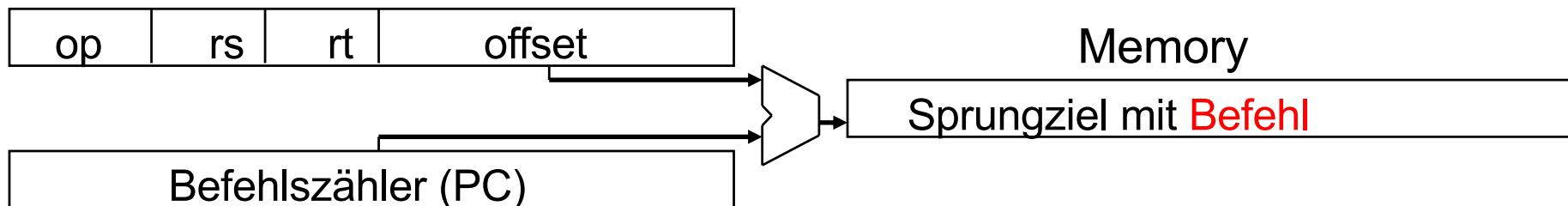
- Register relativ (indirekt) mit 0(\$a0)
- Pseudo-direkt mit addr(\$zero)

- Direkte Adressierung – Operand ist eine 16-bit Konstante in einem Befehl

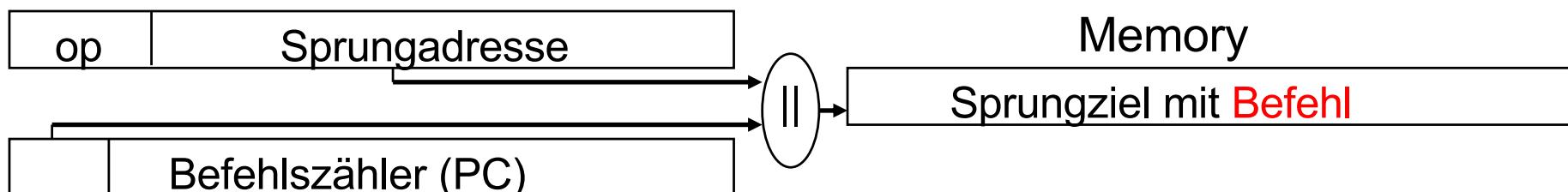


Review der MIPS Befehle zur Addressierung

- PC-relative Adressierung: Die **Adresse** ergibt sich aus der Summe des Befehlszählers (PC + 4) und einer 16-Bit Konstanten im Befehl



- Pseudodirekte Adressierung: Der neue **Befehlszähler** (PC) ergibt sich aus einer Konstanten (26 Bit) und den oberen 4 Bit des alten Befehlszähler (PC + 4).



MIPS (RISC) Entwurfsprinzipen

□ Regularität vereinfacht Entwurf

- Befehle mit fixer Länge – 32-Bits
- Kleine Anzahl an Befehlsformate
- Opcode entspricht immer den ersten 6 Bits

□ Ein guter Entwurf verlangt gute Kompromisse

- Drei Befehlsformate

□ Kleiner ist schneller

- Kleiner Befehlssatz
- Kleine Anzahl von Registern im Registerspeicher
- Kleine Anzahl von Adressierungsarten

□ Mach den häufigsten Fall schnell

- Arithmetische Operanden nur aus Registerspeicher
- Befehle mit „immediate“ Operanden erlauben (Konstanten)

Alternative Architekturen

- Design Alternativen:
 - Unterstütze mächtigere Befehle
 - Reduzieren der Anzahl auszuführender Schritte im Prozessor
 - Gefahren sind längere Taktzeiten und/oder höhere CPI^[1]
 - “*The path toward operation complexity is thus fraught with peril. To avoid these problems, designers have moved toward simpler instructions*”
- Ein (kurzer) Blick auf IA-32

[1]: CPI = Clock cycles per instruction

IA - 32

- ❑ 1978: Der Intel 8086 wird angekündigt (16 Bit Architektur)
- ❑ 1980: Der 8087 floating point Coprozessor wird hinzugefügt
- ❑ 1982: Der 80286 vergrößert Adressbereich auf 24 Bits, +Befehle
- ❑ 1985: Der 80386 vergrößert auf 32 bits, neue Adressierungsarten
- ❑ 1989-1995: Der 80486, Pentium, Pentium Pro mit neuen Befehlen
(Hauptsächlich zur Verbesserung der Leistung)
- ❑ 1997: 57 neue “MMX” Befehle hinzugefügt, Pentium II
- ❑ 1999: Der Pentium III fügt weitere 70 Befehle hinzu (SSE)
- ❑ 2001: Weitere 144 Befehle (SSE2)
- ❑ 2003: AMD erweitert die Architektur um die Adressierung auf 64 Bit zu erweitern, verbreitert alle Registers auf 64 Bits, ... (AMD64)
- ❑ 2004: Intel kapituliert und übernimmt AMD64 Erweiterungen (EM64T)

“This history illustrates the impact of the “golden handcuffs” of compatibility”

“adding new features as someone might add clothing to a packed bag”

“an architecture that is difficult to explain and impossible to love”

IA-32 Übersicht

- Komplexität:
 - Befehle von 1 bis 17 Bytes lang
 - Ein Operand kann nicht nur Source sondern muss auch Ziel sein
 - Ein Operand kann aus dem Hauptspeicher kommen
 - Komplexe Adressierungsarten
 - z.B., “base or scaled index with 8 or 32 bit displacement”
- In der Praxis:
 - Die meistgenutzten Befehle sind nicht zu schwierig zu nutzen
 - Compiler vermeiden ineffiziente Befehle

*“what the 80x86 lacks in style is made up in quantity,
making it beautiful from the right perspective”*

IA-32 Register und Daten Adressierung

- Das 32-bit Register subset entstanden mit dem 80386



IA-32 Register Restriktionen

- Register sind nicht “general purpose” – beachte die Restriktionen unten

Mode	Description	Register restrictions	MIPS equivalent
Register Indirect	Address is in a register.	not ESP or EBP	lw \$s0,0(\$s1)
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	not ESP or EBP	lw \$s0,100(\$s1) #≤16-bit # displacement
Base plus scaled Index	The address is Base + ($2^{Scale} \times$ Index) where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	mul \$t0,\$s2,4 add \$t0,\$t0,\$s1 lw \$s0,0(\$t0)
Base plus scaled Index with 8- or 32-bit displacement	The address is Base + ($2^{Scale} \times$ Index) + displacement where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	mul \$t0,\$s2,4 add \$t0,\$t0,\$s1 lw \$s0,100(\$t0) #≤16-bit # displacement

FIGURE 2.42 IA-32 32-bit addressing modes with register restrictions and the equivalent MIPS code. The Base plus Scaled Index addressing mode, not found in MIPS or the PowerPC, is included to avoid the multiplies by four (scale factor of 2) to turn an index in a register into a byte address (see Figures 2.34 and 2.36). A scale factor of 1 is used for 16-bit data, and a scale factor of 3 for 64-bit data. Scale factor of 0 means the address is not scaled. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent mode would need two more instructions: a `lui` to load the upper 16 bits of the displacement and an `add` to sum the upper address with the base register `$s1`. (Intel gives two different names to what is called Based addressing mode—Based and Indexed—but they are essentially identical and we combine them here.)

IA-32 Typische Befehle

- Vier Hauptarten von Integer Befehlen:
 - Daten verschieben inklusive move, push, pop
 - Arithmetische und logische (Ziel Register oder Hauptspeicher)
 - Kontrollfluss (nutzen von Zustandscodes / Flags)
 - String Befehle, beinhaltet Strings verschieben und vergleichen

Instruction	Function
JE name	if equal(condition code) (EIP=name); EIP-128 ≤ name < EIP+128
JMP name	EIP=name
CALL name	SP=SP-4; M[SP]=EIP+5; EIP=name;
MOVW EBX,[EDI+45]	EBX=M[EDI+45]
PUSH ESI	SP=SP-4; M[SP]=ESI
POP EDI	EDI=M[SP]; SP=SP+4
ADD EAX,#6765	EAX= EAX+6765
TEST EDX,#42	Set condition code (flags) with EDX and 42
MOVSL	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

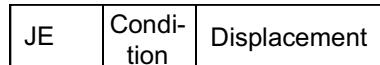
FIGURE 2.43 Some typical IA-32 instructions and their functions. A list of frequent operations appears in Figure 2.44. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)

IA-32 Befehlsformat

□ Typische Format: (Beachte die Längenunterschiede)

a. JE EIP + displacement

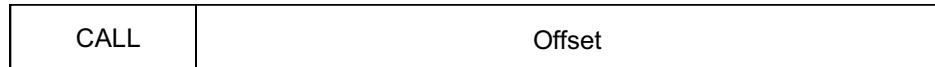
4 4 8



b. CALL

8

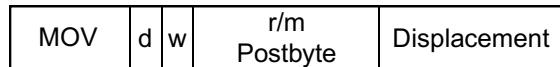
32



c. MOV EBX, [EDI + 45]

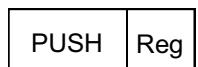
6 1 1 8

8



d. PUSH ESI

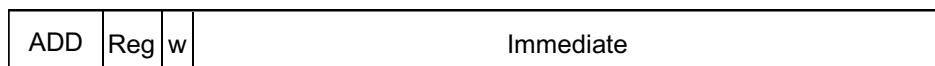
5 3



e. ADD EAX, #6765

4 3 1

32



f. TEST EDX, #42

7 1 8

32

