GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Informatik II

SS 2005

Kapitel 4: Assembler Programmierung

Dr. Michael Ebner
Dipl.-Inf. René Soltwisch

Lehrstuhl für Telematik
Institut für Informatik

Assembler Programmierung – Motivation

- Was ist ein Programm?
  - Eine Reihe von Befehlen, die der Ausführung einer Aufgabe dient
  - Dazu wird das Programm sequentiell abgearbeitet und Befehl für Befehl abgearbeitet. Der Prozessor arbeitet dabei zustandsbasiert
- Problem:
  - Welche Befehle kennt der Prozessor?
  - Sind das dieselben Befehle, die ein Mensch verwendet?
- Sich Schritt für Schritt dem Prozessor nähern…
  - Anweisung: Addiere die Zahlen von eins bis hundert und speichere das Ergebnis!
  - Integer: Ergebnis := Sum(1+2+3+4+…+100);
  - ….aber das sind immer noch nicht die Befehle, die ein Prozessor versteht.

Assembler Allgemein – Befehle (Maschinensprache)

- Die Befehle, die der Computer versteht und ausführen kann, sind Bitfolgen
- Maschinensprache, in der Regel für jede CPU unterschiedlich.
  - Ausnahme: (aufwärts-) kompatible Rechnerfamilien
  - Beispiel: 286, 386, 486, Pentium
- Der Befehl muss zwei Angaben enthalten (v. Neumann-Rechner):
  - durchzuführende Operation (Was!) Operations-Code
  - verwendeter Operand (Womit!) Adresse
- Häufig gehen diese Angaben auch direkt in den Op-Code ein. Nach der Anzahl der Operanden im Adressteil unterscheidet man:
  - 8-Bit-Mikrocomputer arbeiten fast immer mit Einadress-Befehlen (1. Quelle und Zieladresse implizit gegeben, z.B. Akkumulator), 16-Bit-CPUs dagegen oft mit Zweiadress-Befehlen.
  - Ein Befehl kann aus einem oder mehreren Speicherworten bestehen
  - Bei Computern mit großer Wortlänge können auch mehrere Befehle in einem Speicherwort stehen
  - Bei 8-Bit-Mikros besteht der OP-Code aus einem Byte, der Adress-Teil aus einem oder zwei Bytes (= eine Adressangabe)

Assembler vs. Maschinensprache

- Gesamtheit aller vom Prozessor ausführbaren Befehle (Maschinenbefehle) heißt Maschinensprache
- Prozessoren können nur numerische Instruktionen ausführen.
  - 0000 0010 0010 0100 0000 0010 0000
  - Addiere den Inhalt der Register 17 und 18 und speichere das Ergebnis im Register 8 ab
  - für den Menschen nicht gut verständlich
- Ein Assembler setzt die Befehle um und verwaltet symbolische Adressen
- Der Wortschatz eines Assembler heißt Assemblersprache oder auch einfach Assembler

Assembler

- Aber auch die Maschinenbefehle (bzw. Assemblerbefehle) sind für den Menschen schwer zu verstehen, selbst wenn er nicht direkt in der binären Maschinensprache programmieren muss.
- Es gibt keine Variablen
- Einige Befehle stehen nicht direkt zur Verfügung.
- Jeder Prozessor ist anders, also auch jeder Assembler
- Das bedeutet, dass ein Programmierer ein Assemblerprogramm (z.B. Summe von 1 bis 100) für jeden Prozessor neu schreiben muss.

Vom Assembler zur Hochsprache

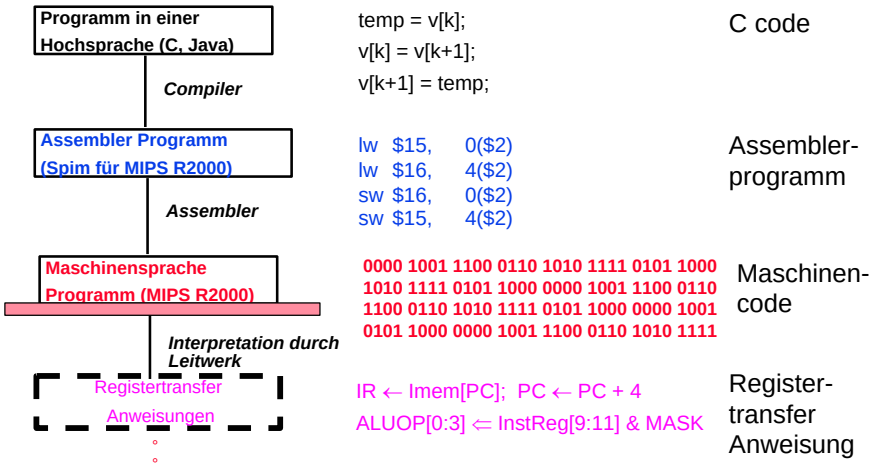
- Lösung des Problems: eine Hochsprache
- Es werden dem Programmierer mehr Befehle angeboten
- Speicherverwaltung wird übernommen
- Die Befehle sind Prozessor unabhängig

Wer vermittelt nun zwischen Hochsprache und Maschinensprache?

- Ein Compiler übersetzt das Hochsprachenprogramm in Assemblerprogramm.
  - Diesen Vorgang nennt man kompilieren
- Ein Assembler macht aus einem Assemblerprogramm ein Maschinensprachenprogramm
- Anstelle des Compilers kann auch ein Interpreter verwendet werden. Dieser erzeugt auch Maschinencode, aber erst zur Laufzeit. Im Unterschied zum Interpreter übersetzt der Compiler das gesamte Programm.

Wo wir jetzt sind – ein kurzes Fazit

- Vorher Zahlendarstellung / Gatter → ALU
- von Neumann Rechner
- Jetzt Assembler Programmierung
- Später:
  - Compilerbau: Wie bekomme ich aus einer Hochsprache ein Assemblercode? Wie optimiere ich den Compiler? In welcher Sprache schreibe ich einen solchen Compiler?
  - Formale Sprachen: Formale Analyse der Sprachen und ihrer Mächtigkeit, usw.



Wie können wir nun Programmieren?

Wir brauchen:

- Einen Simulator:
  - MIPS R2000 – Stanford Projekt: Microprocessor without Internal Pipeline Stages
  - RISC Prozessor
  - SPIM (unser Simulator) unterliegt der GNU-Lizenz
    - http://www.cs.wisc.edu/~larus/spim.html
    - Linux: xspim; Windows© PCSpim
- Einen Texteditor
  - Zum Erstellen der Programme

Arbeitsweise des R 2000

1. Der Befehl wird geladen
2. Der Befehl wird decodiert
3. Daten werden in die Register geladen
   - Die zugehörigen Operanden werden geladen
4. In Abhängigkeit des aktuellen Befehls werden die Register verändert.
   - z.B. der Inhalt von zwei Registern wird addiert und das Ergebnisse in ein anderes Register geschrieben.
5. Daten werden aus den Registern in den Speicher geschrieben

Adressierung des Speichers

- Warum Hexadezimal?

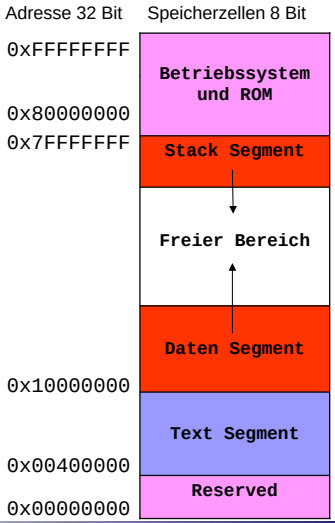
- Größe des Speichers
- Jede Zelle ist 8 Bit (1 Byte) Groß und hat eine 32 Bit Adresse
  - Kilobyte 1KB = 1024 Byte = 2^{10}Byte
  - Megabyte 1MB = 1024 KB = 2^{20} Byte
  - Gigabyte 1GB = 1024 MB = 2^{30} Byte

Der Speicher

- Von Neumann-Rechner haben gemeinsamen Speicher für Daten und Programm
- Der Programmierer muss verhindern, dass der Rechner versucht, die abgelegten Daten als Programm auszuführen.
- Der Prozessor soll das Programm nicht mit Daten überschreiben
- Selbstmodifizierender Code ist möglich (Beispiel: Computer Viren)
- Lösung: Segmente für Daten, Programm, OS und Stack
  - Mit den Direktiven .data und .text geben wir an, wohin die folgenden Befehle geschrieben werden
- Der Programmierer kann aber auch direkt in den Speicher schreiben.

Hauptspeicher Verwaltung

- Der Speicher besteht aus 2^{32} Zellen à einem Byte
- .text und .data sind Direktiven, die angeben, was Programm und was Daten sind.



Datentypen

- Ganze Zahlen
  - .word – 32-Bit-Zahlen
  - .half – 16-Bit-Zahlen
  - .byte – 8-Bit-Zahlen
- Zeichenketten
  - .ascii – 8Bit pro Zeichen (256 verschiedene Zeichen)
    - .ascii “Das ist ein Beispieltext. Und ohne “
    - .asciiz “Zeilenumbruch geht es weiter.“
- Floating Point wird vom Coprozessor unterstützt.

Speicher – Litte Big Endian

- Litte-Endian vs. Big-Endian
- Big Endian „denkt“, dass das erste Byte, das er liest, das größte ist.
- Little Endian „denkt“, dass das erste Byte, das er liest, das kleinste ist
  - Big Endian: Mainfraim, IBM
  - Litte Endian: PC, Intel, AMD
  - Bi Endian: PowerPC
- Daten sollten im Speicher ausgerichtet (Aligned) sein
  - Beispiel: 00000000 00000000 00000100 00000001 (1025)

Adresse	Big-Endian Darstellung von 1025	Little-Endian Darstellung von 1025
00	00000000	00000001
01	00000000	00000100
02	00000100	00000000
03	00000001	00000000

R 2000 Register

- 32 General Purpose Register mit 32 Bit Wortbreite
  - $0 bis $31 oder per „Name“
  - $zero enthält immer(!) den Wert 0
  - $at temporäres Assemblerregister
  - $v0, $v1 Funktionsergebnisse
  - $a0 bis $a3 Argumente für Prozeduraufrufe
  - $t1 bis $t9 temporäre Register
  - $s0 bis $s7 langlebige Register
  - $k0, $k1 Kernel Register
  - $gp Zeiger auf Datensegment
  - $sp Stack Pointer
  - $fp Frame Pointer
  - $ra Return Address
- Zusätzlich lo und hi (Spezialregister)

0	$zero	16	$s0
1	$at	17	$s1
2	$v0	18	$s2
3	$v1	19	$s3
4	$a0	20	$s4
5	$a1	21	$s5
6	$a2	22	$s6
7	$a3	23	$s7
8	$t0	24	$t8
9	$t1	25	$t9
10	$t2	26	$k0
11	$t3	27	$k1
12	$t4	28	$gp
13	$t5	29	$sp
14	$t6	30	$fp
15	$t7	31	$ra

Die Assembler Befehle

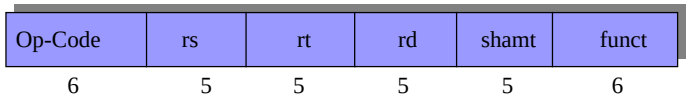
- Befehlsformat
  - <Marke>: <Befehl1> <Arg_1> <Arg_2> <Arg_3> # <Kommentar>
  - <Marke>: <Befehl2>, <Arg_1>, <Arg_2>, <Arg_3> # <Kommentar>
- Befehle
  - Lade- / Speicherbefehle
  - Arithmetische Operationen (+ , * , - , /)
  - Logische Operationen (AND , OR , XOR , NAND , …)
  - Schiebe- / Rotationsbefehle
  - Sprungbefehle
- Pseudobefehle werden vom Assembler in mehrere Befehle umgesetzt → in die Befehle, die der Prozessor verarbeiten kann.
- Direktiven sind Assembleranweisung, die NICHT in Befehle umgesetzt werden; z.B. um Platz zu lassen für Variablen (beginnen mit einem Punkt z.B. „.data“)

Assembler Programme

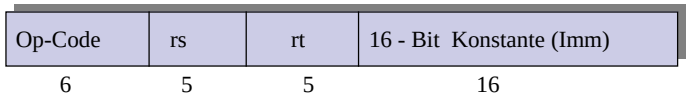
- Sequentielle Abarbeitung von Befehlen
  - Sprünge sind möglich
- Nur ein Befehl pro Zeile
- „#“ leitet einen Kommentar ein
- „<name>:“ ist ein Label
  - Das Label „main:“ muss immer vorhanden sein
  - Label können angesprungen werden z.B. „while:“ oder „for:“
  - Den Namen des Labels entscheidet der Programmierer (bis auf „main:“)

Die drei Instruktionsformate

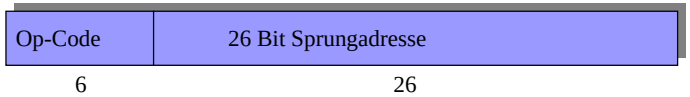
Register Format (R)



Immediate Format (I)



Sprung Format (J)



Assembler Programm – ein Beispiel

```
# Beispielprogramm V. 001
# berechnet den Umfang eines Dreiecks mit den Kanten x, y, z
# kein sehr sinnvolles Programm
.data
x:              .word   12
y:      .word   14
z:      .word   5
u:      .word   0
.text
main:   lw      $t0, x
        lw      $t1, y
        lw      $t2, z
        add     $t0, $t0, $t1  # $t0 := x + y
        add     $t0, $t0, $t2  # $t0 := x + y + z
        sw      $t0, u         # u := x + y + z
        li      $v0, 10        # EXIT
        syscall
```

Ladebefehle und Adressierungsarten I

- lw Rd, Adr
  - Der Befehl lw (load word) lädt ein Wort in ein Register
  - Rd und Adr sind die Argumente. Adr ist die Adresse im Hauptspeicher, von wo der Befehl in das Register Rd geladen wird.
- Adressierungsmodi I – „...wie wird die Adresse angegeben?“
  - Register Indirekt (RS): Der Wert steht an der Adresse, die im Register RS angegeben ist.
  - Direkt: Die Adresse wird direkt angegeben über ein Label oder Label + Konstante.
  - Indexierte – Label (RS): Der Wert wird durch ein Label angegeben + den Wert aus Register RS (+ Konstante)

Ladebefehle und Pseudoinstruktionen

- Weitere Befehle
  - lb Rd, Adr : load byte 8 Bit
  - lbu Rd, Adr : load unsigned byte 8 Bit
  - lh Rd, Adr : load halfword 16 Bit
  - lhu Rd, Adr : load unsigned halfword 16 Bit
  - ld Rd, Adr : load double-word ← ist eine Pseudoinstruktion
- Pseudoinstruktionen werden vom Assembler aufgelöst

  ld Rd, Adr := lw Rd, Adr
  lw Rd+1,Adr+4
- Daten müssen aligned im Speicher sein
  - Adressen von Wörtern müssen durch 4 teilbar sein.
  - Adressen von Halbwörtern müssen gerade sein.
  - Oder es werden die Befehle ulw, ulh und ulhu verwendet.

… und nun das Ganze noch mal zum Speichern

sw Rs, Adr	store word
sb Rs, Adr	store byte
sh Rs, Adr	store halfword
sd Rs, Adr	store double-word (Pseudoinstruktion)
ush Rs, Adr	unaligned store halfword
usw Rs, Adr	unaligned store word

Register Transfer Befehle und Adressierungsarten II

Register direkte Adressierung:

Es wird direkt auf die Register zugegriffen unter Nennung der Registernamen Rd (destination) und Rs (source).

move Rd,Rs move Rs → Rd

Unmittelbare Adressierung:

Dort wird eine Konstante unmittelbar in ein Register übertragen. Die Konstante ist Teil des Programmcodes, nicht der Daten.

li Rd, Imm	load immediate (Imm ist dann eine Konstante)
lui Rs, Imm	load upper immediate Rs ← Imm * 2^{16}

Arithmetische Befehle (Addition & Subtraktion)

- Können nur direkt auf den Registern ausgeführt werden
- Addition und Subtraktion (mit Overflow)
  - add Rd, Rs1, Rs2 Rd := Rs1 + Rs2
  - addi Rd, Rs1, Imm Rd := Rs1 + Imm
  - sub Rd, Rs1, Rs2 Rd := Rs1 – Rs2
- add kann auch mit einer Konstanten (unmittelbaren Operanden) verwendet werden. Der Assember benutzt dann automatisch addi.
- sub und alle weiteren arithmetischen Befehle können mit einer Konstanten verwendet werden. ☺

Überläufe und Ausnahmen

- Was passiert, wenn die Summe zweier positiver Zahlen negativ wird?
- →Es tritt eine Ausnahme (Exception) auf. Ausnahmen sind Fehler, die während der Laufzeit des Programms auftreten.
  - Der Exception Handler beendet in diesem Fall das Programm!
- Ein klassisches Beispiel ist auch die Division durch Null.
- Beispiel: Die Halfbytes (4 Bit) a und b werden addiert.

A= 0111	7
B= 0001 +	1
----------	---
S= 1000	-8

Addition ohne Überlauf

addu Rd, Rs1, Rs2
addiu Rd, Rs1, Imm
aubu Rd, Rs1, Rs2

U sollte nicht durch unsigned übersetzt werden, da auch bei Addition von vorzeichenlosen Zahlen Überläufe auftreten können.

Multiplikation und Division

- Problem eines Überlaufs ist sehr wahrscheinlich und es werden sogar doppelt so viel Bits zum Speichern benötigt.
- Beispiel: $(1 * 2^{32}) * (1* 2^{32}) = 1 * 2^{32+32} = 1* 2^{64}$
- Der MIPS verfügt über zwei Spezialregister hi und lo
- Es gibt aber nur 4 Maschinenbefehle für Multiplikation und Division

div Rd, Rs hi:=Rd MOD Rs; lo:= Rd DIV Rs
divu Rd, Rs hi:=Rd MOD Rs; lo:= Rd DIV Rs
mult Rd, Rs hi:=Rd * Rs DIV 2^{32}; lo:=Rd * Rs MOD 2^{32}
multu Rd, Rs hi:=Rd * Rs DIV 2^{32}; lo:=Rd * Rs MOD 2^{32}

Die Ergebnisse von Maschinenbefehlen werden immer in hi und lo gespeichert.

Multiplikation und Division

- Diese Pseudobefehle arbeiten direkt auf den Registern:

div Rd, Rs1, Rs2	Rd:=Rs1 MOD Rs2 mit Überlauf
divu Rd, Rs1, Rs2	Rd:=Rs1 MOD Rs2 ohne Überlauf
mul Rd, Rs1, Rs2	Rd:= Rs1 * Rs2 ohne Überlauf
mulo Rd, Rs1 ,Rs2	Rd:= Rs1 * Rs2 mit Überlauf

Das Ergebnis wird in Rd gespeichert.

Ob div ein Maschinenbefehl oder Pseudobefehl ist, erkennt der Assembler an der Anzahl der Parameter.

- ➔ Logische Operationen
- ➔ Sprünge

→ Jetzt erst mal weiter mit dem SPIM Ein- und Ausgaben…

SPIM – Ein- und Ausgaben

Der Befehl syscall

Der Befehl hat keine Argumente.

Ein Parameter wird über das Register $v0 übergeben.

Das Verhalten des Simulators ist abhängig vom Register $v0

$v0=1	print_int	Wert in $a0 wird dezimal ausgegeben auf dem Bildschirm
$v0=4	print_string	Die mit Chr0 endende Zeichenkette wird ausgegeben. $a0 muss die Adresse enthalten, an der die Zeichenkette im Speicher beginnt.

Syscall Parameter

- Vor Aufruf von syscall muss der Parameter in $v0 gegeben werden

1	print_int	Wert in $a0 wird dezimal ausgegeben
2	print_float	32 Bit Gleitkommazahl aus Register $f12/13 wird ausgegeben
3	print_double	64 Bit …
4	print_string	Die an ($a0) beginnende und auf Chr 0 endende Zeichenkette wird ausgegeben.
5	read_int	Schreibt eine Dezimalzahl in $v0
6	read_float	Schreibt eine Gleitkommazahl in $f0 (32 Bit)
7	read_double	Schreibt eine Gleitkommazahl in $f0/1 (64 Bit)
8	read_string	Schreibt eine Zeichenkette bis Chr 0 in den Speicher. Startadresse: $a0 Max. Länge $a1
9	sbrk	Reserviert Speicher
10	exit	Beendet das Programm

Logische Befehle

and Rd, Rs1, Rs2	Rd:=Rs1 AND Rs2
andi Rd, Rs, Imm	Rd:=Rs AND Imm
nor Rd, Rs1, Rs2	Rd:=Rs1 NOR Rs2
or Rd, Rs1, Rs2	Rd:=Rs1 OR Rs2
ori Rd, Rs, Imm	Rd:=Rs AND Imm
xor Rd, Rs1, Rs2	Rd:=Rs1 XOR Rs2
xori Rd, Rs, Imm	Rd:=Rs AND Imm
not Rd, Rs	Rd:= NOT(Rs)
	:= xori Rd, Rs, -1

Rotations- und Schiebebefehle

- Rotation: Alle Bits in einem Register werden nach links oder rechts verschoben; das Letzte kommt dann an den Anfang.
- Schiebebefehle: Wie Rotation allerdings fällt das letzte Bit weg. Das erste Bit wird durch 0 ersetzt (logische Schiebebefehle) oder das höchstwertige Bit wird übernommen, damit das Vorzeichen erhalten bleibt (arithmetische Schiebebefehle).

rol Rd, Rs1, Rs2 rotiert Rs1 um Rs2 Stellen nach links (ror für rechts)

sll und sllv (shift left logical); srl und srlv (shift right logical); sra und srav (shift right arithmetic)

V steht für Variable, was bedeutet, dass ein Register die Anzahl der Schritte bestimmt; ansonsten ist der Wert eine Konstante.

Vergleiche

- Es werden zwei Werte verglichen und wenn die Bedingung (z.B. gleich, > , < ,….) erfüllt ist, wird in das Zielregister eine 1 geschrieben und sonst eine 0.
- Beispiel: seq Rd, Rs1, Rs2 Falls (Rs1)=(Rs2)→Rd:=1, sonst Rd:=0
- sne – ungleich; sge – größer oder gleich; sgeu – größer oder gleich (unsigned) usw.
- Bei den Vergleichen gibt es nur 4 Maschinenbefehle:
  - slt Rd, Rs1, Rs2 set less than
  - sltu Rd, Rs1, Rs2 set less than unsigned
  - slti Rd, Rs1, Imm set less than immediate
  - sltui Rd, Rs1, Imm set less than immediate unsigned

Sprünge

- Sprünge machen Programme erst mächtig. Der Spim unterstützt bedingte und unbedingte Sprünge (jumping) an eine Marke (Label).
- Im Gegensatz zu Hochsprachen unterstützt der Assembler bedingte Sprünge nur aufgrund von Größenvergleichen zweier Register oder eines Registers mit einer Konstanten.
  - b label – unbedingter Sprung zum Label
  - beq Rs1, Rs2 label – bedingter Sprung falls Rs1 = Rs2
  - Es gibt insgesamt 21 Befehle für bedingte Sprünge, die auch =,<>,<=,>=,<,>,=0, usw. abfragen.
- Mit den Sprüngen können wir die Schleifenkonstrukte und Fallunterscheidungen der Hochsprachen nachbilden.
- Eigentlich erwartet der Assembler einen Offset zu PC (Branching), wir verwenden allerdings Marken, damit sind die Befehle j und b identisch im Spim.

Kontrollstrukturen

- Sprünge (bedingte und unbedingte)
- If-then-else, Case
- Loop (n Durchläufe)
- While (Abbruchbedingung)

Beispiel: if–then–else in Assembler

```
if ($t8 < 0) then{
        $s0 = 0 - $t8;
        $t1 = $t1 + 1;
}
else{
        $s0 = $t8;
        $t2 = $t2 + 1;
}
```

```
main:   bgez    $t8, else       # if ($t8 is > or = zero) branch to else
        sub     $s0, $zero, $t8  # $s0 gets the negative of $t8
        addi    $t1, $t1, 1     # increment $t1 by 1
        b       next            # branch around the else code
else:
        ori     $s0, $t8, 0     # $s0 gets a copy of $t8
        addi    $t2, $t2, 1     # increment $t2 by 1
next:
```


Beispiel: While in Assembler

```
$v0 = 1;
while ($a1 < $a2) do{
        $t1 = mem[$a1];
        $t2 = mem[$a2];
        if ($t1 != $t2) go to break;
        $a1 = $a1 +1;
        $a2 = $a2 –1;}
        return
break:  $v0 = 0
        return
```

```
        li      $v0, 1          # Load $v0 with the value 1
loop:
        bgeu    $a1, $a2, done  # If( $a1 >= $a2) Branch to done
        lb      $t1, 0($a1)     # Load a Byte: $t1 = mem[$a1 + 0]
        lb      $t2, 0($a2)     # Load a Byte: $t2 = mem[$a2 + 0]
        bne     $t1, $t2, break # If ($t1 != $t2) Branch to break
        addi    $a1, $a1, 1     # $a1 = $a1 + 1
        addi    $a2, $a2, -1    # $a2 = $a2 - 1
        b       loop            # Branch to loop
break:
        li      $v0, 0          # Load $v0 with the value 0
done:
```


Beispiel: Schleifen in Assembler

```
$a0 = 0;
For ( $t0 =10; $t0 > 0; $t0 = $t0 -1) do {
        $a0 = $a0 + $t0;
}
```

```
        li      $a0, 0          #  $a0 = 0
        li      $t0, 10         # Initialize loop counter to 10
loop:
        add     $a0, $a0, $t0
        addi    $t0, $t0, -1    # Decrement loop counter
        bgtz    $t0, loop       # If ($t0 >0) Branch to loop
```


Komplexe Datenstrukturen – Felder

- Wir können im MIPS Felder deklarieren, indem wir die Felder explizit belegen.

Beispiel:

```
        .data
feld: .word 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

- Oder wir reservieren den Platz für 13 Wörter

Beispiel:

```
        .data
feld: .space 52
```

Hier wird nur Platz reserviert, aber **nicht initialisiert**!

- Ein Feld ist vergleichbar mit einem Array

```
sw $t0, feld($t0)  # feld[i] := i
```

Der Stack

- Ein LIFO Speicher
- Kellerzeiger (SP) zeigt auf 0x7FFF FFFC
- Einige CISC Prozessoren verfügen über Extrabefehle (push, pop) zum Einkellern von Daten. Beim Spim müssen wir die Kellerverwaltung selbst übernehmen.
- Unterbrechungen müssen beachtet werden
  - sw $t0, ($sp) # tep. Reg. $t0 auf Stack
  - addi $sp, -4 # Stack Pointer neu setzen
  - Besser:
  - addi $t0, -4
  - sw $t0, 4($sp)

Keller für mehr Daten

- Für den Fall, dass man mehrere Daten speichern will
  - addi $sp, -12 #gleich für 3 Worte der SP setzen
  - sw $t0, 12($sp) # t0 sichern
  - sw $t1, 8($sp) # t1 sichern
  - sw $t2, 4($sp) # t2 sichern
- Lesen analog mit dem Befehl lw
- Erst den Lesebefehl und dann den SP wieder neu setzen.
- Wir sind nun in der Lage, die ganzen temporären Register zu sichern.

Unterprogramme

- Aus Hochsprachen ist das Konzept von Unterprogrammen bekannt.
- In Hochsprachen heißen sie Prozeduren, Methoden, Subrotinen oder Funktionen.
- Der Grund von Unterprogrammen ist die Wiederverwendung von Programmteilen.
- Programme können leichter verständlich werden, wenn Unterprogramme sinnvoll eingesetzt werden.
- Auch Assembler unterstützen das Konzept von Unterprogrammen.

Beispiel: Das Unterprogramm ***swap*** als C Programm

swap vertauscht v[k] mit v[k+1]

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

Swap als MIPS Assembler Programm

```
swap:  addi   $29, $29, -12  # reserve space on stack
       sw     $2, 4($29)     # save $2 on stack
       sw     $15, 8($29)    # save $15 on stack
       sw     $16, 12($29)   # save $16 on stack
       sll    $2, $5, 2      # reg $2 = k * 4
       add    $2, $4, $2     # reg $2 = v + (k*4)
                             # $2 con. addr. of v[k]
       lw     $15, 0($2)     # $15 <-- v[k]
       lw     $16, 4($2)     # $16 <-- v[k+1]
       sw     $16, 0($2)     # $16 --> v[k]
       sw     $15, 4($2)     # $15 --> v[k+1]
       lw     $2, 4($29)     # restore $2 from stack
       lw     $15, 8($29)    # restore $15 from stack
       lw     $16, 12($29)   # restore $16 from stack
       addi   $29, $29, 12   # restore stackpointer
       jr     $31            # return
```


Unterprogramme

- Das Unterprogramm wird mit „`jal swap`“ aufgerufen.
- Die Parameter müssen in $4, $5 übergeben werden.
- Die Rückkehradresse wird automatisch in $31 gespeichert.
- `jr $31` ist der Rücksprung aus dem Unterprogramm und entspricht einem Return.

Konventionen für Prozeduren (Prolog)

- Für den Aufrufer (Caller)
  - Sichere $a0-$a3, $v0, $v1, da diese in der Prozedur verändert werden dürfen.
  - Speichere die zu übergebenden Argumente in $a0 bis $a3.
  - Weitere Argumente werden über den Stack übergeben (das fünfte ist das letzte Argument auf dem Stack).
  - Beachte: Call-by-Value vs. Call-by-Reference.
  - Prozedur wird mit jal gestartet.
- Für den Aufgerufenen (Callee)
  - Platz für Stackframe reservieren.
  - Sichere alle „callee-saved“ Register, die verändert werden.
    - $fp, $ra, $s0 - $s7
    - Achtung Fehlerquelle: der Befehl jal verändert $ra
    - Erstelle $fp durch $sp + Stackframe

Konventionen für Prozeduren (Epilog)

- Callee
  - Rückgabe des Funktionswerts in $v0, $v1
  - Register wieder herstellen
    - $fp als letztes Register wieder herstellen
  - Stackframe entfernen
    - $sp = $sp – Größe des Frames
  - Return (jr $ra)
- Caller
  - Auf dem Stack gesicherten Register wieder herstellen
  - Argumente vom Stack nehmen

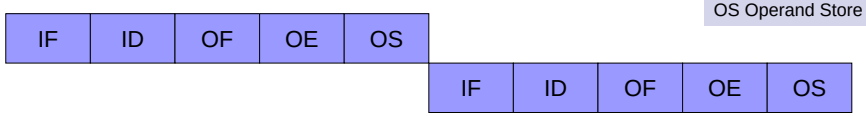
RISC- Prozessoren und Pipelining

- Ein-Zyklus-Maschinenbefehle
  - Möglichst alle Befehle laufen in einem Taktzyklus ab.
- Load/Store-Architektur
  - Nur über Load/Store-Befehle Zugriff auf den Hauptspeicher. Alle anderen Befehle: Reg.-Reg.
- Keine Mikroprogrammierung
  - Festverdrahtete Ablaufsteuerung
- Möglichst wenige Befehle und Adressierungsarten
  - Typischerweise ca. 50 Befehle, 2 Adressierungsarten: PC-relativ und indiziert.
  - Nur Befehle aufnehmen, wenn sie eine deutliche Geschwindigkeitssteigerung **im Mittel** bringen.
- Einheitliches Befehlsformat
  - 1 Wort = 1 Befehl (einfache Dekodierung)
- Aufwandsverlagerung in den Compiler
  - Optimierender Compiler erzeugt Code (später mehr dazu).

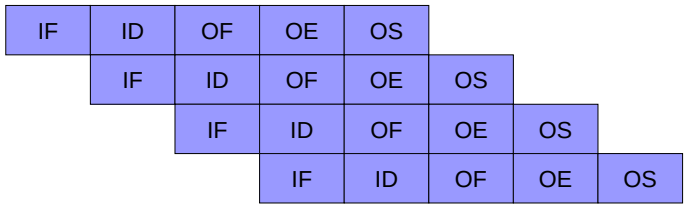
Pipelining

IF Instruction Fetch
ID Instruction Decode
OF Operand Fetch
OE Operation Execute
OS Operand Store

- Phasen-Pipelining im RISC
  - Ohne Pipelining



  - Mit Phasen-Pipelining



Pipelining

- Der Befehl wird nicht schneller ausgeführt, aber es können mehr Befehle pro Zeit ausgeführt werden.
  - → Parallelität und damit eine Geschwindigkeitssteigerung um die Anzahl der Pipelinestufen.
- Pipelinekonflikte
  - Datenfluss-Konflikt durch Datenabhängigkeiten (Data Interlock)
  - Beispiel:

```
add  $t0, $t1, $t2    # $t0 ← $t1 + $t2
addi $t0, 1           # $t0 ← $t0 + 1
```

Optimierung des Phasen-Pipelinings

- Data Forwarding
  - Operand kann vor dem Rückspeichern an den nächsten Befehl übergeben werden.
- Delayed Load
  - Beim Ladebefehl darf nicht sofort im nächsten Befehl auf den geladenen Operanden zugegriffen werden, sondern erst im übernächsten.
  - Es wird die „Blase“ vermieden, dafür müssen evt. nop Befehle eingeschoben werden.
  - Mit „Blase“ ist die Zeit gemeint, in der der Prozessor nicht weiß, was er tun soll und auf das Rückspeichern wartet.

Steuerfluss-Konflikte

- Delayed Branch (verzögerter Sprung)
  - Verzweigung wird erst ab dem übernächsten Befehl wirksam.
  - Das „Loch“ wird durch einen Befehl, der eigentlich vor dem Branch hätte ausgeführt werden sollen, gestopft.
- Für den Programmierer lästig, für optimierende Compiler kein Problem.

traditioneller Sprung	**verzögerter Sprung**	**optimierter verzögerter Sprung**
LOAD X, R1	LOAD X, R1	LOAD X, R1
ADD 1, R1	ADD 1, R1	BRANCH L
BRANCH L	BRANCH L	ADD 1, R1
ADD R1, R2	NOP	ADD R1, R2
SUB R3, R2	ADD R1, R2	SUB R3, R2
L: STORE R1, Y	SUB R3, R2	L: STORE R1, Y
	L: STORE R1, Y	

Fazit und Ausblick

- RISC hat sich durchgesetzt
  - Einsatz in PCs, Workstation, Parallelrechnern und zunehmend auch in Mikrocontrollern.
- Moderne CICS- Prozessoren haben mehr und mehr RISC Techniken übernommen
  - z.B. Intel Pentium-Familie
- Neue Technologien besonders RISC geeignet (z.B. Galliumarsenid)

Ausblick