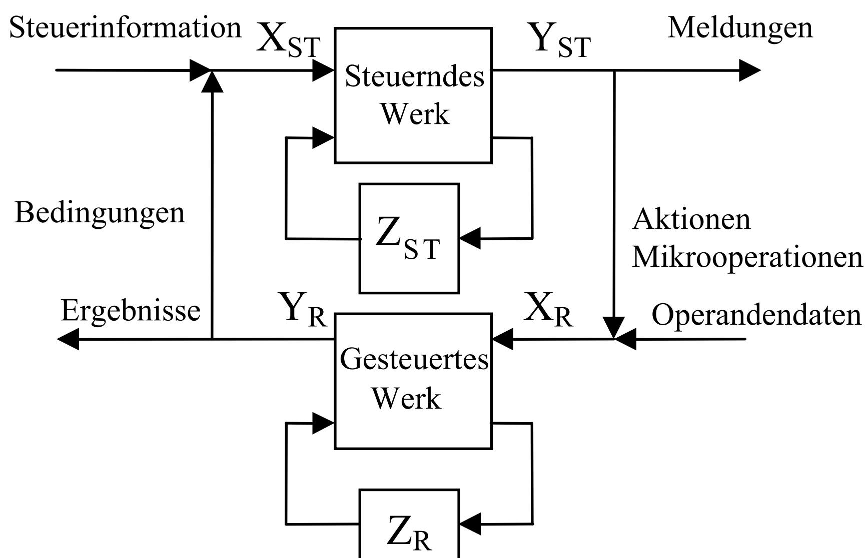
8 Steuerwerke

8.1 Steuerwerk

Wir kennen jetzt den Aufbau von Rechenwerken, z.B. einem Addierer oder einer logischen Einheit. In einem Computer müssen diese Einheiten aber zur richtigen Zeit aktiviert werden, Daten müssen über Multiplexer an die richtigen Einheiten geführt werden, mit anderen Worten: Die Abläufe müssen gesteuert werden. Diese Funktionen übernehmen ebenfalls Schaltwerke, sogenannte **Steuerwerke**. Wir wollen in diesem Abschnitt an einem einfachen Beispiel die Funktion eines Steuerwerks studieren.

Die folgende Folie zeigt das grundsätzliche Prinzip eines Steuerwerks: Es gibt ein steuerndes (Schalt-)werk und ein gesteuertes Werk. Das gesteuerte Werk ist häufig ein Rechenwerk. Das Steuerwerk hat - wie jedes Schaltwerk - Eingaben, Ausgaben und Zustände. Die Eingaben sind einerseits die Befehle der übergeordneten Instanz (z.B. des Benutzers oder eines übergeordneten Steuerwerks), andererseits die **Bedingungen.** Dies sind Ausgaben des Rechenwerks, mit dem es dem Steuerwerk Informationen über den Ablauf der gesteuerten Aufgabe gibt. Die Ausgaben des steuernden Werks sind einerseits die Meldungen zur übergeordneten Instanz und andererseits die **Mikrobefehle (Aktionen)**, die als Eingaben in das gesteuerte Werk gehen, und dadurch die Abläufe dort kontrollieren.

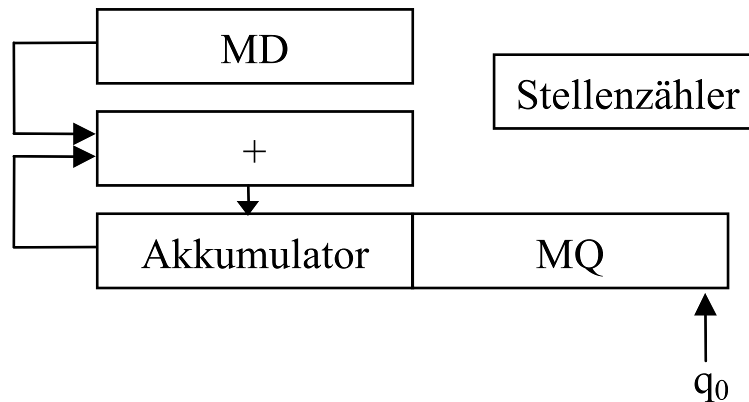
Steuerwerksprizip:



Multiplizierwerk:

Das folgende Beispiel soll das Steuerwerksprinzip illustrieren:

Ein Multiplizierwerk ist zu steuern, das zwei Zahlen (z.B. mit jeweils vier Bit) multiplizieren kann. Das Multiplizierwerk besteht aus einem Multiplikanden-Register MD, einem Akkumulatorregister AKK, das als paralleles Register und als Schieberegister genutzt werden kann, einem Multiplikatorregister MQ mit derselben Eigenschaft, einem Zähler, genannt Stellenzähler (SZ) und einem Addierer. Mit q_0 wird die letzte Stelle von MQ bezeichnet.



Für eine Multiplikation werden Multiplikand und Multiplikator in die entsprechenden Register geladen, der Akkumulator wird mit 0 vorbesetzt. Sodann sendet die Steuerung einen Mikrobefehl „-n“, der den Stellenzähler auf -n setzt. n soll dabei die Anzahl der Stellen der zu multiplizierenden Operanden sein. Wenn q_0 1 ist, wird jetzt die Mikrooperation „+“ generiert, die das Addieren des gegenwärtigen Akkumulators mit dem Register MD bewirkt, das Ergebnis wird im Akkumulator gespeichert. Danach werden die Mikrooperationen „S“ und „SZ“ generiert. S bewirkt ein Verschieben um ein Bit nach rechts im Schieberegister bestehend aus Akkumulator und MQ. SZ bewirkt ein Inkrementieren des Stellenzählers. Wenn der Stellenzähler den Wert 0 erreicht hat, terminiert der Prozess, das Ergebnis steht im Schieberegister. Wenn der Stellenzähler nicht 0 ist, wird wiederum q_0 interpretiert. Wenn q_0 0 ist, wird die Mikrooperation „0“ generiert, die kein Register verändert.

Die folgende Tabelle zeigt ein Beispiel für die Multiplikation der Zahlen 0101 und 1011 (5 * 11).

8.1.1 Abfolge im Multiplizierwerk:

MD = 0101

Akku	**MQ**	**SZ**	**q0**	**SZ=0**	**Start**	**Microoperation**
0 0 0 0	1 0 1 1	000	1	1	1	-n
0 0 0 0	1 0 1 1	100	1	0	0	+
0 1 0 1	1 0 1 1	100	1	0	0	S,SZ
0 0 1 0	1 1 0 1	101	1	0	0	+
0 1 1 1	1 1 0 1	101	1	0	0	S,SZ
0 0 1 1	1 1 1 0	110	0	0	0	0
0 0 1 1	1 1 1 0	110	0	0	0	S,SZ
0 0 0 1	1 1 1 1	111	1	0	0	+
0 1 1 0	1 1 1 1	111	1	0	0	S,SZ
0 0 1 1	0 1 1 1	000	1	1	0	

Wir wissen bereits, wie wir ein solches Rechenwerk bauen müssten, denn es besteht nur aus uns bekannten Komponenten (Register, Schieberegister, Zähler, Addierer). An dieser Stelle interessiert uns nun aber, wie wir die Mikrooperationen zum richtigen Zeitpunkt generieren können, also wie wir dieses Rechenwerk „steuern“ können. Dazu machen wir uns klar:

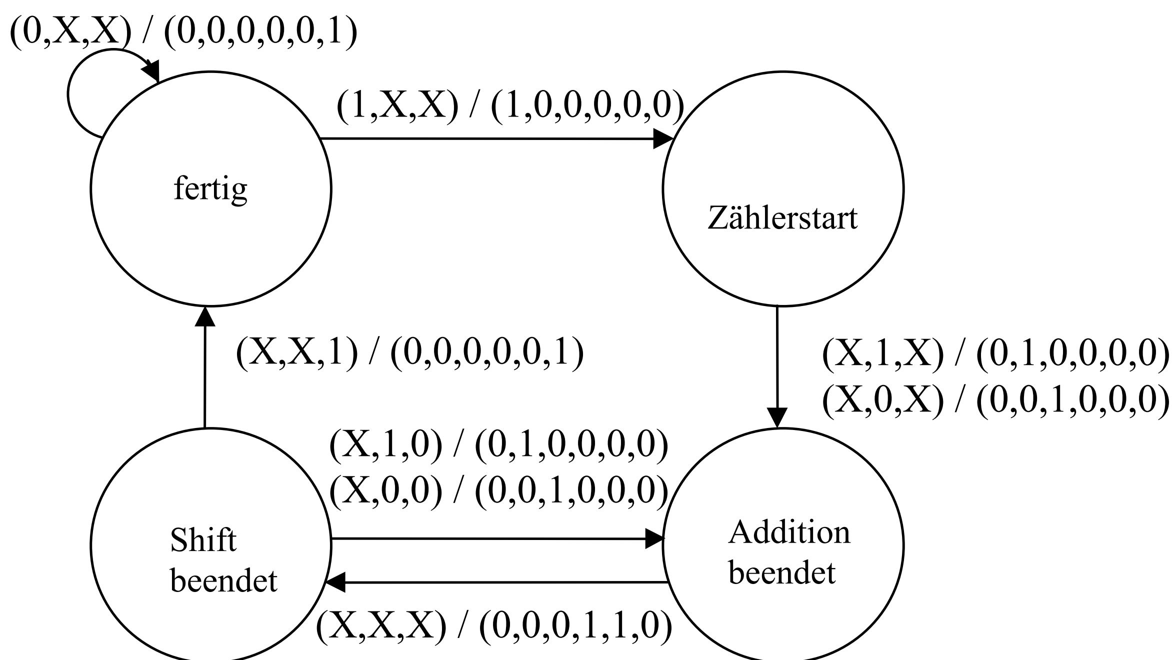
- Wenn ein „Start“-Signal kommt, muss der Stellenzähler mit „-n“ initialisiert werden.

- Wenn q_0 interpretiert wird, muss MD genau dann auf den Akkumulator addiert werden, wenn $q_0 = 1$ ist.
- Nach jedem solchen Additionsschritt muss der Akkumulator und das MQ um ein Bit geschoben und der Stellenzähler um eins erhöht (inkrementiert) werden.
- Wenn irgendwann der Stellenzähler den Wert 0 erreicht, ist die Multiplikation beendet, das Ergebnis steht im Schieberegister aus Akkumulator und MQ.

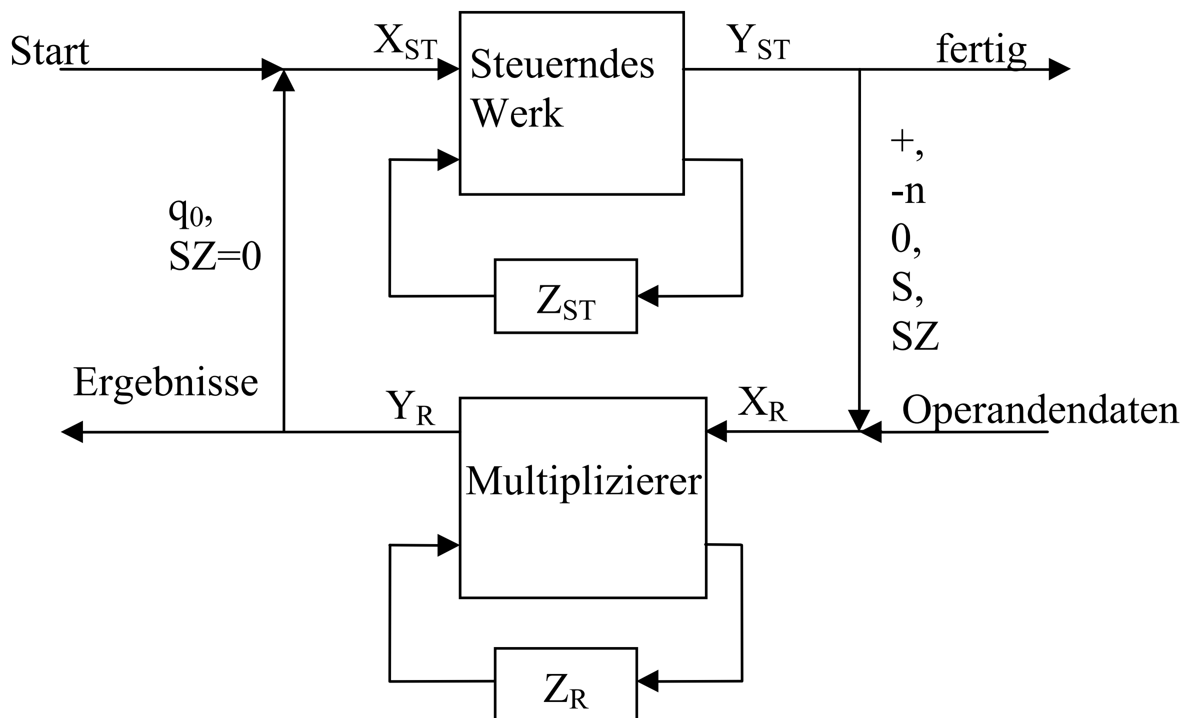
Die folgende Folie zeigt, welche dieser Signale Ein- und Ausgaben welcher Werke sind.

Aus diesen Informationen können wir danach einen Automatengraphen entwickeln.

Steuerwerk:



Eingaben: (Start, q_0, SZ=0) Ausgaben (-n, +, 0, S, SZ, fertig)



Wir codieren die Zustände mit

00: fertig

01: Zählerstart

10: Addition beendet

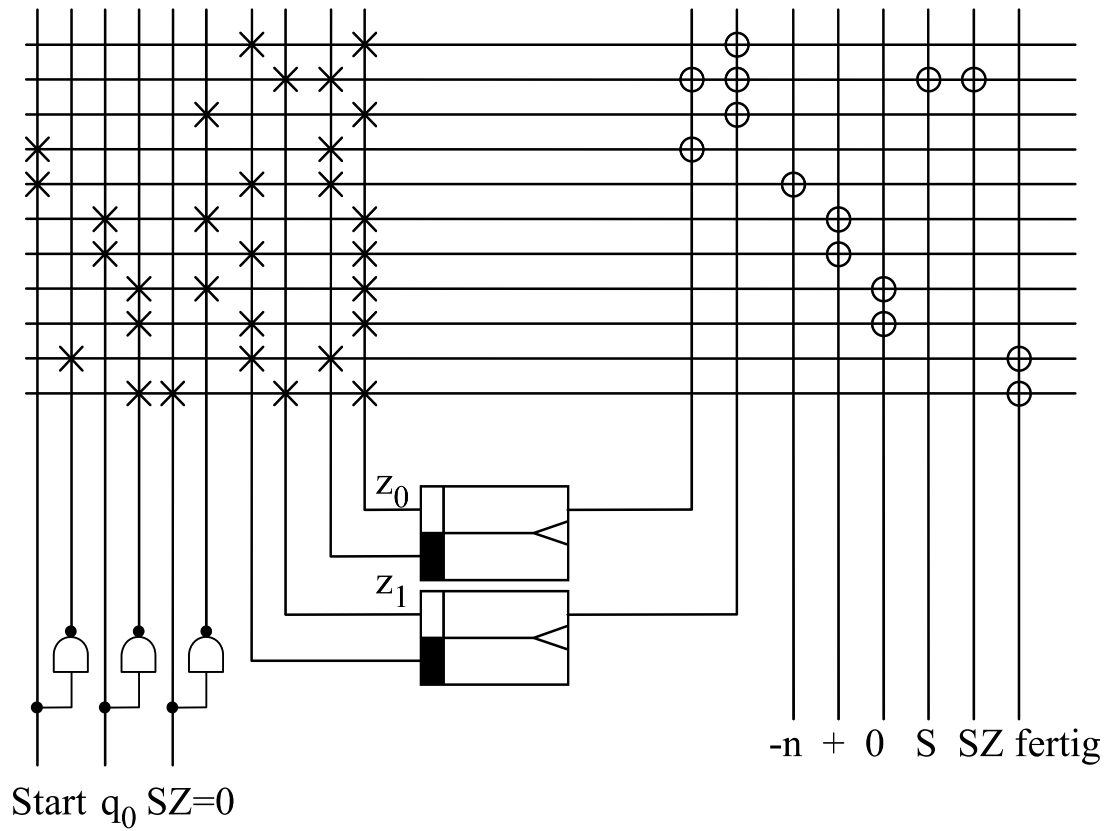
11: Shift beendet

Dann entspricht dieser Automat der folgenden Wertetabelle:

Start	**q0**	**SZ=0**	**z1**	**z0**	**z'1**	**z'0**	**-n**	**+**	**0**	**S**	**SZ**	**fertig**
0	X	X	0	0	0	0	0	0	0	0	0	1
1	X	X	0	0	0	1	1	0	0	0	0	0
X	1	X	0	1	1	0	0	1	0	0	0	0
X	0	X	0	1	1	0	0	0	1	0	0	0
X	X	X	1	0	1	1	0	0	0	1	1	0
X	1	0	1	1	1	0	0	1	0	0	0	0
X	0	0	1	1	1	0	0	0	1	0	0	0
X	0	1	1	1	0	0	0	0	0	0	0	1

Das ergibt nach Minimierung die Schaltwerksrealisierung auf der nächsten Folie:

Realisierung als FPLA:



9 Befehlssatzarchitektur und Assembler-Programmierung

Die Befehlssatzarchitektur ist der Teil der Maschine, die sichtbar ist für den Programmierer und Compiler-Schreiber.

9.1 Die RISC Idee

Mitte der 80-er Jahre wurde die Prozessorlandschaft revolutioniert durch ein neues Grundkonzept: RISC (reduced instruction set computer). Nachdem vorher die Philosophie der Architekten darin bestand, möglichst komfortable Funktionen bereitzustellen, damit der Anwender alle erdenklichen Operationen als Maschineninstruktionen der Hardware vorfand, fand man jetzt zurück zur Einfachheit:

Es zeigte sich, dass durch Vereinfachung des Befehlformats, der Speicheradressierung, des Befehlssatzes und des Registerzugriffs die Leistung eines Rechners signifikant gesteigert werden konnte. Dies wurde erreicht, indem man quantitative Entwurfsprinzipien entwickelte, mit denen man die Qualität von Rechnerarchitekturentscheidungen bewerten kann. Dieser Ansatz wurde von wenigen Grundprinzipien geleitet:

1. Klein ist schnell: Kleinere Hardwareeinheiten können schneller sein, weil Leitungen kürzer sind, Transistoren kleiner usw.
2. Make the common case fast: Wenn eine Entwurfsentscheidung zu treffen ist, ist zu untersuchen, wie häufig eine Verbesserung ggf. benutzt wird.
3. Amdahls Gesetz: Die Gesamtbeschleunigung einer Architekturveränderung ist begrenzt durch den Anteil A_u an der ursprünglichen Berechnung, der nicht beschleunigt werden kann:

$$\text{Speedup} = \frac{1}{A_u + \frac{1 - A_u}{\text{Teilbeschleunigung}}}$$

4. Die CPU-Performance-Gleichung: Die Ausführungszeit eines Programms kann berechnet werden als

CPU-Zeit = IC * CPI * Zykluszeit;

Dabei ist IC (instruction count) die Anzahl der Instruktionen des Programms, CPI (clocks per instruction) die mittlere Anzahl von Takten für die Ausführung einer Instruktion und Zykluszeit die Zeit für einen Taktzyklus also der reziproke Wert der Taktfrequenz.

Die RISC Idee zeichnet sich durch vier Charakteristika aus:

1. Einfacher Instruktionssatz
2. Feste Befehlswortlänge, wenige Befehlsformate
3. Wenige Adressierungsarten, einfache Adressierungsarten
4. Registermaschine mit Load-Store Architektur

Diese Punkte werden im folgenden näher erläutert.

9.2 Akkumulator-Architektur

Die Akkumulator-Architektur hat ein ausgezeichnetes Register: den Akkumulator. Dieser ist in jeder Operation als ein (impliziter) Operand beteiligt. Load und Store wirken nur auf den Akkumulator. Alle arithmetischen und logischen Operationen benutzen den Akkumulator sowohl als einen Operanden als auch als Zielregister. Somit kommen alle Operationen mit nur

einer Adresse aus. Dies ist ein Vorteil, wenn das Befehlsformat klein ist, so dass nicht genügend Bits für die Adressen mehrerer Operanden zur Verfügung stehen. Daher ist die Akkumulator-Architektur (insbesondere im 8-Bit Bereich, z.B. Mikrocontroller) auch heute noch gebräuchlich.

Eine typische Befehlsfolge zur Addition zweier Zahlen in der Akkumulator-Architektur sieht folgendermaßen aus:

Load A

Add B

Store C

9.3 General Purpose Register Architekturen (GPR-architectures)

General purpose Register sind innerhalb eines Taktzyklus zugreifbare Speicher im Prozessor, die als Quell und Zieladresse zugreifbar sind. Sie sind nicht für bestimmte Maschinenbefehle reserviert.

GPR-Architekturen sind heute die gebräuchlichsten. Fast jeder nach 1980 entwickelte Prozessor ist ein GPR-Prozessor. Warum?

1. **Register sind schneller als Hauptspeicher.**
2. **Register sind einfacher und effektiver für einen Compiler zu nutzen.**

   Beispiel:

   Operation G = (A*B)+(C*D)+(E*F)

die Multiplikationen können in beliebiger Reihenfolge ausgeführt werden. Dies hat Vorteile in Sinne von Pipelining. Auf einer Akkumulator Architektur würde durch Umladen des Akkumulators gebremst.

3. **Register können Variablen enthalten.**

Das verringert den Speicherverkehr und beschleunigt das Programm (da Register schneller sind als Speicher). Sogar die Codierung wird kompakter, da Register mit weniger Bits adressierbar sind als Speicherstellen. Dies erlaubt ein einfacheres Befehlsformat, was sich in der Compiler-Effizienz und im Speicherverkehr positiv auswirkt.

Für Compiler-Schreiber ist es optimal, wenn alle Register wirklich GPRs sind. Ältere Maschinen treffen manchmal Kompromisse, indem bestimmte Register für bestimmte Befehle vorzusehen sind. Dies reduziert jedoch faktisch die Anzahl der GPRs, die zur Speicherung von Variablen benutzt werden können.

Zwei Kriterien unterteilen die Klasse der GPR-Architekturen:

1. **Wie viele Operanden kann ein typischer ALU-Befehl haben?**
2. **Wie viele Operanden dürfen Speicheradressen sein?**

Bei einem 3-Operanden Format gibt es zwei Quell-Operanden und einen Zieloperanden.

Bei einem 2-Operanden Format muss ein Operand als Quelle und Ziel dienen, d.h. er wird durch die Operation zerstört.

9.3.1 Typen von GPR-Architekturen

1. **Register-Register-Maschinen (load-store-Architekturen)**

   Alle arithmetischen Operationen greifen nur auf Register zu. Die einzigen Befehle, die Speicherzugriffe machen sind **load** und **store**. Eine typische Befehlsfolge für das obige Beispiel wäre:

   Load R1, A

   Load R2, B

   Add R3, R1, R2

   Store C, R3

2. **Register-Speicher Maschinen**

   Bei Register-Speicher-Maschinen können die Befehle ihre Operanden aus dem Hauptspeicher oder aus den Registern wählen. Dies erlaubt gegenüber den load-store-Architekturen einen geringeren IC. Dafür muss aber der größere Aufwand eines Speicherzugriffs in jeder Operation in Kauf genommen werden, was in der Regel auf Kosten der CPI geht. Bekannteste Vertreter der Klasse ist die Intel 80x86-Familie und der Motorola 68000. Das obige Beispiel sieht auf einer Register-Speicher-Architektur so aus:

   Load R1, A

   Add R1, B

   Store C, R1

3. **Speicher-Speicher-Maschinen**

   Diese verfügen über die größte Allgemeinheit, was den Zugriff auf Operanden angeht. Operanden können aus Registern oder Speicherzellen geholt werden, die Zieladresse kann ebenfalls ein Register oder eine Speicherzelle sein. Unterschiedliche in der Praxis übliche Befehlsformante erlauben 2 oder 3 Operanden in einem Befehl. Im Falle einer 3-Operanden Maschine ist das Beispiel mit **Add C,A,B** abgehandelt. Dieser Vorteil wird mit aufwendigem CPU-Speicher-Verkehr bezahlt.

9.3.2 Zusammenfassung der Typen von GPR-Architekturen:

Typ	Vorteile	Nachteile
Register-Register (0,3)	Einfaches Befehlsformat fester Länge. Einfaches Modell zur Code Generierung. Instruktionen brauchen alle etwa gleichviel Takte.	Höherer IC als die beiden anderen.
Register-Speicher (1,2)	Daten können zugegriffen werden, ohne sie erst zu laden. Instruktions-Format einfach.	Jeweils ein Operand in einer zweistelligen Operation wird zerstört. CPI variiert je nachdem von wo die Operanden geholt werden.
Speicher-Speicher (3,3)	Sehr kompakt. Braucht keine Register für Zwischenergebnisse.	Hoher CPI. Viele Speicherzugriffe. Speicher-CPU-Flaschenhals.

9.4 Speicheradressierung

Unabhängig davon, welche Art von Maschine man gewählt hat, muss festgelegt werden, wie ein Operand im Hauptspeicher adressiert werden kann.

Was können Operanden sein?

Bytes (8 Bit), Half Words 16 Bit), Words (32 Bit), Double Words (64 Bit).

Es gibt unterschiedliche Konventionen, wie die Bytes innerhalb eines Wortes anzuordnen sind:

Little Endian bedeutet: Das Byte mit Adresse xxxx....x00 ist das Least Significant Byte.

Big Endian bedeutet: Das Byte mit Adresse xxxx....x00 ist das Most Significant Byte.

In Big Endian ist die Adresse eines Wortes die des MSByte, in Little Endian die des LSByte.
Little Endian: 80x86, VAX, Alpha; Big Endian: MIPS, Motorola, Sparc;

In vielen Maschinen müssen Daten entsprechend ihrem Format ausgerichtet **(aligned)** sein. Das heißt, dass ein Datum, das s Bytes lang ist, an einer Speicheradresse steht, die kongruent 0 mod s ist. Beispiele für ausgerichtete und nicht ausgerichtete Daten sind in der folgenden Tabelle:

Ausrichtung auf Wortgrenzen:

Adressiertes Objekt	Ausgerichtet auf Byte	Nicht ausgerichtet auf Byte
Byte	0,1,2,3,4,5,6,7	Nie
Half word	0,2,4,6	1,3,5,7
Word	0,4	1,2,3,5,6,7
Double word	0	1,2,3,4,5,6,7

Warum Speicherausrichtung?

Speicher sind von der Hardware der DRAM- oder heute auch SDRAM-Bausteine her auf die Ausrichtung auf Wortgrenzen hin optimiert. Deshalb ist bei den Maschinen, die keine Ausrichtung fordern, der Zugriff auf nicht ausgerichtete Daten langsamer als auf ausgerichtete Daten. Zum Beispiel müssen bei einem 32-Bit Zugriff zwei 32-Bit Daten über den Datenbus

gelesen werden, und von diesen müssen per Maskierung die richtigen 32 Bit ermittelt werden, die dann den gesuchten Operanden ausmachen.

Um diesen Nachteil dem Programmierer zu ersparen, erzwingen moderne Prozessoren die Ausrichtung auf Wortgrenzen.

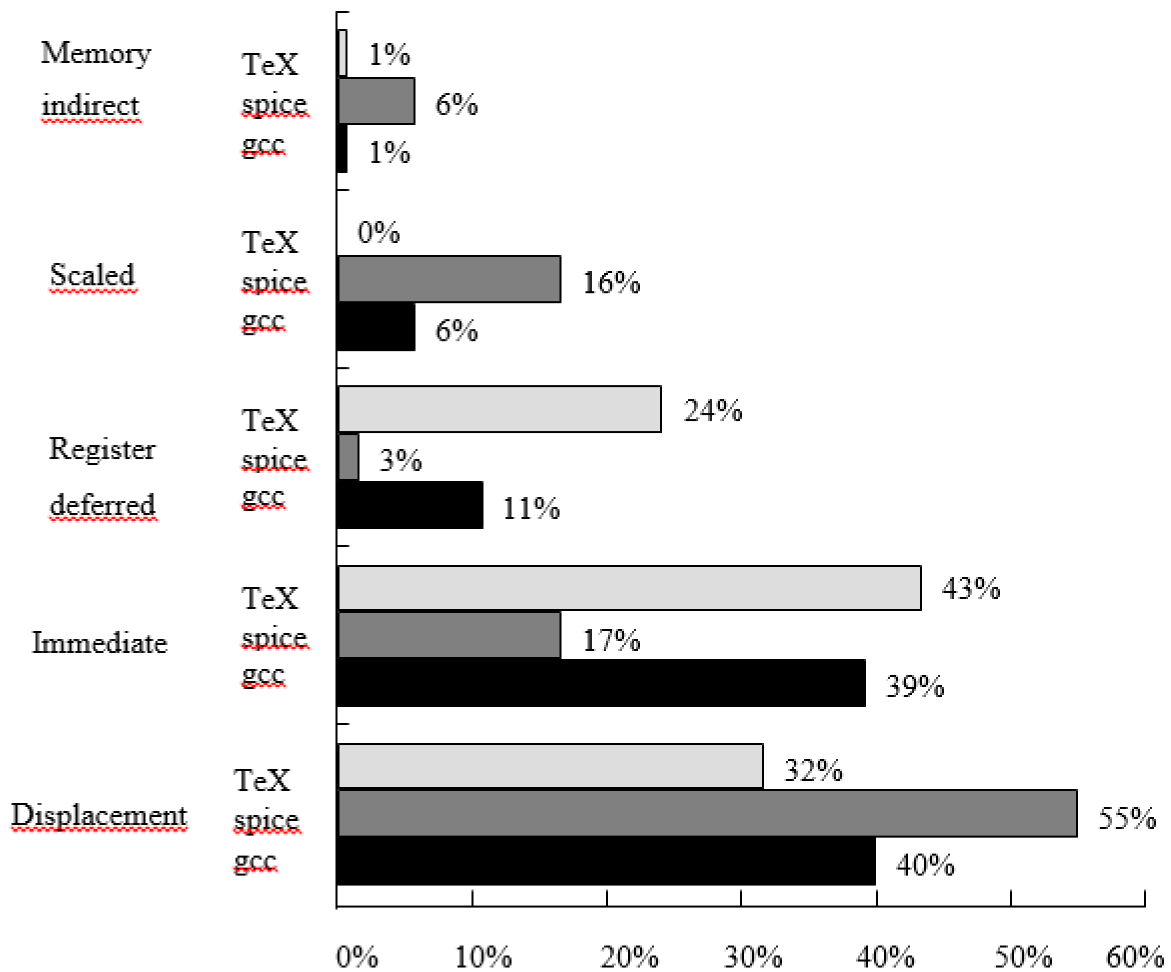
9.4.1 Adressierungsarten

Wir wissen jetzt, welche Bytes wir bekommen, wenn wir eine Adresse im Speicher zugreifen. Dieser Abschnitt soll die Möglichkeiten aufzeigen, wie Speicheradressen im Befehl anzugeben sind.

Definition: Die *effektive Adresse* ist die aktuelle Speicheradresse, die durch die jeweilige Adressierungsart angesprochen wird.

Die folgende Tabelle zeigt die in heutigen Rechnern verwendeten Adressierungsarten.

Adressierungsart	**Beispiel**	**Bedeutung**	**Anwendung**
Register	Add R4 , R3	Regs[R4]:=Regs[R4] +Regs[R3]	Wert ist im Register.
Immediate	Add R4 , #3	Regs [R4] :=Regs [R4]+3	Operand ist eine Konstante
Displacement	Add R4 , 100(R1)	Regs [R4] :=Regs [R4]+Mem [100+Regs[R1]]	Lokale Variable
Register deferred or indirect	Add R4 , (R1)	Regs [R4] :=Regs [R4]+Mem [Regs[R1]]	Register dient als Pointer.
Direct or absolute	Add R1 , (1001)	Regs [R1] :=Regs [R1]+Mem [1001]	Manchmal nützlich für Zugriff auf statische Daten
Indexed	Add R3 , (R1 + R2)	Regs [R3] :=Regs [R3]+Mem [Regs[R1]+Regs[R2]]	Nützlich für array-Addressierung: R1=base of array; R2=index amount
Memory indirect	Add R1 , @(R3)	Regs[R1]:=Regs[R1]+ Mem[Mem[Regs[R3]]]	Wenn R3 die Adresse eines Pointers *p* enthält, dann bekommen wir **p*.
Autoincrement	Add R1 , (R2)+	Regs [R1] :=Regs [R1]+ Mem[Regs[R2]]Regs [R2]	Nützlich für arrays mit Schleifen. R2 zeigt auf den array-Anfang; jeder Zugriff erhöht R2 um die Größe d eines array Elements
Autodecrement	Add R1 , -(R2)	Regs [R2] :=Regs [R2]-*d* Regs [R1] :=Regs [R1]+ Mem[Regs [R2]]	Genauso wie Autoincrement
Scaled	Add R1 , 100 (R2)[R3]	Regs [R1] :=Regs [R1]+ Mem[100 + Regs [R2] + Regs	Indizierung von Feldern mit Datentypen der Länge d

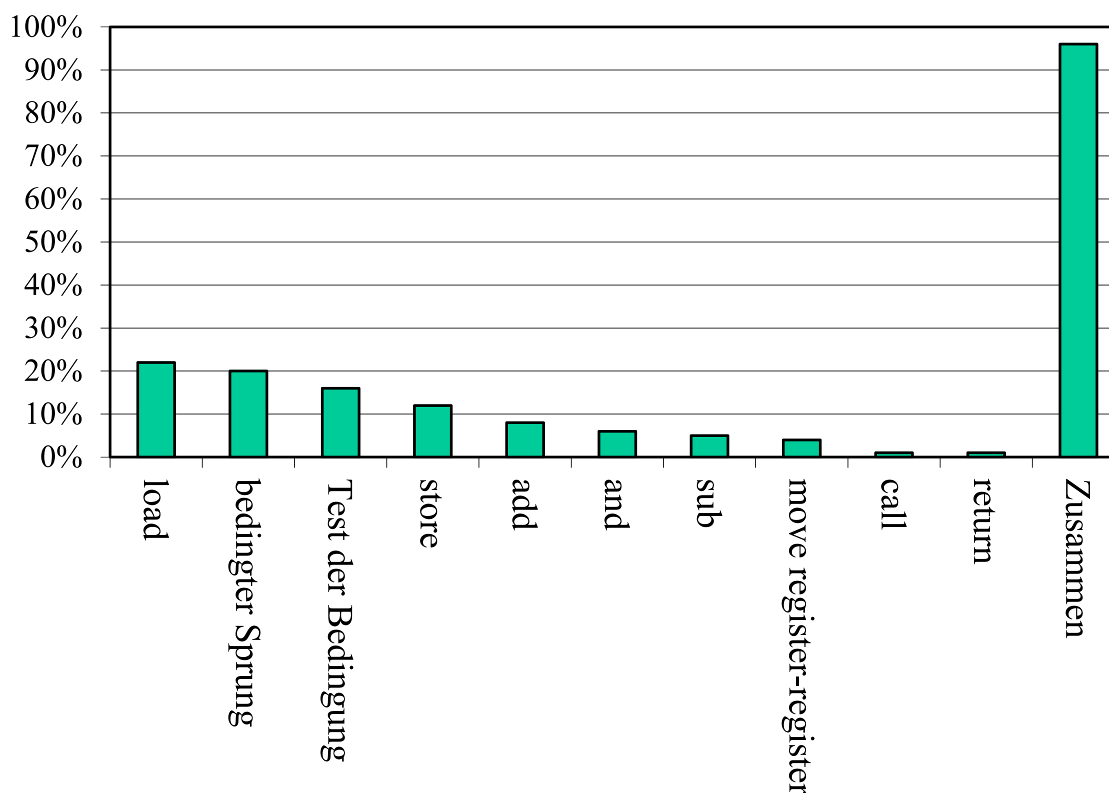


Häufigkeit der Adressierungsarten im Spec 92 Benchmark

Operationen in einem Befehlssatz

Die folgende Tabelle zeigt die Verteilung der 10 häufigsten Instruktionen im SPECint92 gemessen an IBM-kompatiblen Maschinen.

Befehlshäufigkeit:



Faustregel:

Die am häufigsten genutzten Befehle sind die einfachsten Befehle eines Befehlssatzes! Wichtigste Message ist: **Die einfachsten Operationen sind die wichtigsten. Sie müssen schnell sein.**

Load, store, add, subtract, move register register, and, shift, compare equal, compare not equal, branch, jump, call und return.

Für die **Sprungentfernung** von der aktuellen Stelle sollten mindestens **8 Bit** (12 Bit erreicht bereits fast 100% der Fälle) zur Verfügung stehen.

Wir brauchen **PC-relative und Register-indirekte Adressierung für Sprünge**.

9.5 Sprungbefehle

Es gibt vier prinzipiell verschiedene Sprungbefehle:

Conditional branch	**Bedingte Verzweigung**
Jumps	**Unbedingter Sprung**
Procedure Calls	**Aufruf**
Procedure Returns	**Rückkehr aus Prozeduren**

Die Zieladresse für Sprünge muss im Befehl codiert sein. (Ausnahme Returns, weil da das Ziel zur Compilezeit noch nicht bekannt ist). Die einfachste Art ist ein Displacement (offset), das zum **PC (Program Counter)** dazu addiert wird. Solche Sprünge werden **PC-relativ** genannt.

PC relative Branches und Jumps haben den Vorteil, dass das Sprungziel meist in der Nähe des gegenwärtigen PC Inhalts ist, und deshalb das Displacement nur wenige Bits braucht.

Außerdem hat die Technik, Sprungziele PC-relativ anzugeben, den Vorteil, dass das Programm ablauffähig ist, egal wohin es geladen wird. Man nennt diese Eigenschaft **position independence**.

Für die PC-relative Adressierung ist es sinnvoll, die Länge des erforderlichen Displacements zu kennen.

9.5.1 Befehlsformate und Codierung

Für die Länge der Befehlsworte sind die Anzahl der Register und die Adressierungsarten entscheidend. Entscheidender als die Länge des opcodes, da sie öfter in einem Befehl auftauchen können.

Der Architekt muss zwischen folgenden Zielen balancieren:

1. So viele Register und so viele Adressierungsarten wie möglich zu haben.
2. Das Befehlsformat so kompakt wie möglich zu machen.
3. Die Längen der Befehlsformate einfach zugreifbar zu machen.

Letzteres bedeutet insbesondere: Vielfaches eines Bytes. Moderne Architekturen entscheiden sich in der Regel für ein festes Befehlsformat, wobei Einfachheit für die Implementierung gewonnen wird, aber das Optimum an durchschnittlicher Codelänge nicht erreicht wird.

Ein festes Befehlsformat bedeutet nur wenige Adressierungsmodi.

Die Länge der 80x86-Befehle können von 1 bis 5 Byte variieren.

Die der MIPS, Alpha, i860, PowerPC sind fest 4 Byte.

9.5.2 Zusammenfassung Codierung

Da wir an der Optimierung in Hinsicht auf Performance interessiert sind, (und nicht vorrangig an Optimierung in Hinsicht auf Code-Dichte, entscheiden wir uns für ein festes Instruktionsformat mit 32 Bit.

9.6 DLX-Architektur

Der DLX ist ein Prozessor mit eingeschränktem Befehlssatz, der für Übungszwecke konzipiert wurde. Folgende Vorgaben sollen erfüllt werden:

- GPR-Architektur, load-store
- Adressierung: Displacement, Immediate, Indirect
- schnelle einfache Befehle (load, store, add, ...)
- 8-Bit, 16-Bit, 32-Bit Integer
- feste Länge des Befehlsformats, wenige Formate
- Mindestens 16 GPRs

9.6.1 Register

Der Prozessor hat **32 GPRs.**

Jedes Register ist 32-Bit lang.

Sie werden mit R0,..,R31 bezeichnet.

R0 hat den Wert 0 und ist nicht beschreibbar (Schreiben auf R0 bewirkt nichts)

R31 übernimmt die Rücksprungadresse bei Jump and Link-Sprüngen

9.6.2 Datentypen

8-Bit Bytes.

16-Bit Halbworte.

32-Bit Worte.

Für Integers. All diese entweder als unsigned Integer oder im 2-er Komplement.

Laden von Bytes und Halbworten kann wahlweise mit führenden Nullen (unsigned) oder mit Replikation der Vorzeichenstelle (2-er Komplement) geschehen.

9.6.3 Adressierungsarten

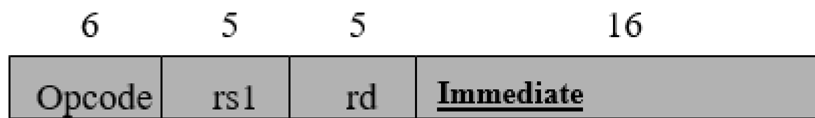
Displacement und Immediate

Durch geschickte Benutzung von R0 und 0 können damit vier Adressierungsarten realisiert werden:

Displacement:	**Load R1, 1000(R2);**
Immeditate:	**Load R1, #1000;**
Indirect:	**Load R1, 0(R2);**
Direct:	**Load R1, 1000(R0);**

9.6.4 Befehlsformate:

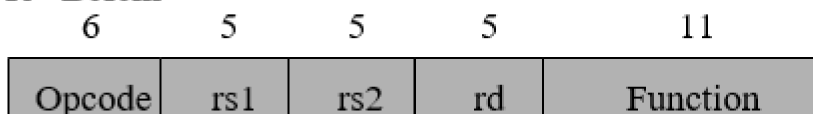
I - Befehl



Loads und Stores von Bytes, Worten, Halbworten
Alle Immediate-Befehle (rd ← rs1 op immediate)

Bedingte Verzweigungen (rs1 : register, rd unbenutzt)
Jump register, Jump and link register
(rd = 0, rs1 = destination, immediate = 0)

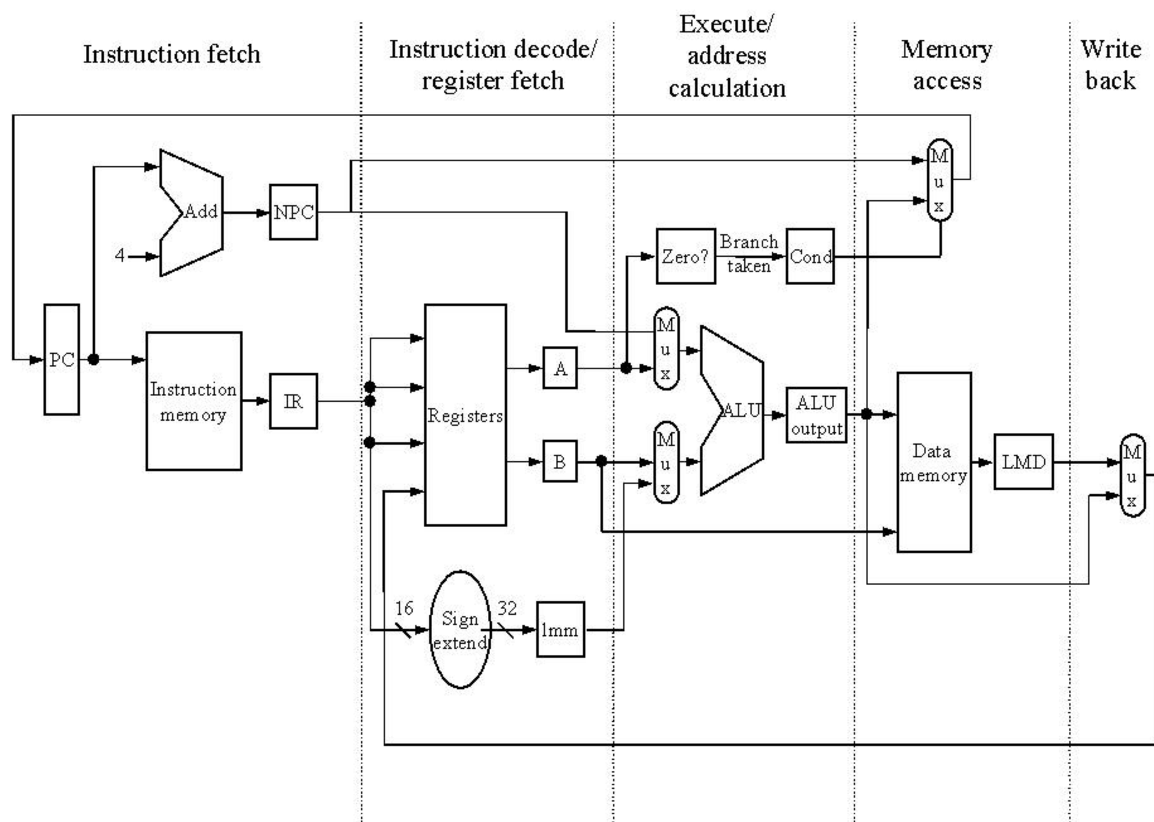
R - Befehl



Register-Register ALU Operationen: rd ← rs1 func rs2
Function sagt, was gemacht werden soll: Add, Sub, ...
Read/write auf Spezialregistern und moves

9.6.5 Datenpfad der DLX

Die folgende Abbildung zeigt den Datenpfad der DLX. Es ist bereits die gesamte Hardware des Prozessors mit Ausnahme des Steuerwerks, das die Informationen zur Steuerung der Multiplexer und der Alu liefert.



Die Syntax der PC-relativen Sprungbefehle ist etwas irreführend, da der als Operand eingegebene Parameter als Displacement zum PC zu verstehen ist.

Tatsächlich müsste die erste Zeile heißen:

J offset bedeutet PC <--- PC+4 + offset mit -2^{15} <= offset < $+2^{15}$

Das würde aber heißen, dass man in Assemblerprogrammen die Displacements bei relativen Adressen explizit angeben muss. Dies macht die Wartung eines solchen Programms unglaublich schwierig. Daher erlaubt man, dass man Namen für die effektiven Adressen einführt, die man wie Marken ins Assemblerprogramm schreibt. Ein Compiler verwendet natürlich die tatsächlichen Offsets, aber für den Leser ist ein solches mit Marken geschrieben Programm leichter verständlich.

Instruction type/opcode	**Instruction meaning**
Data transfers	**Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-bit displacement + contents of a GPR**
LB,LBU,SB	Load byte, load byte unsigned, store byte
LH, LHU, SH	Load half word, load half word unsigned, store half word
LW, SW	Load word, store word (to/from integer registers)
Arithmetic/logical	**Operations on integer or logical data in GPRs; signed arithmetic trap on overflow**
ADD, ADDI, ADDU, ADDUI	Add, add immediate (all immediates are 16 bits); signed and unsigned
SUB, SUBI, SUBU, SUBUI	Subtract, subtract immediate; signed and unsigned
MULT,MULTU	Multiply, signed and unsigned; all operations take and yield 32-bit values
AND,ANDI	And, and immediate
OR,ORI,XOR,XORI LHI	Or, or immediate, exclusive or, exclusive or immediate
LHI	Load high immediate – loads upper half of register with immediate
SLL, SRL, SRA, SLLI, SRLI, SRAI	Shifts: both immediate (S_l) and variable form (S_); shifts are shift left logical, right logical, right arithmetic
S_, S_ I	Set conditional: ”_ ” may be LT, GT, LE, GE, EQ, NE
Control	**Conditional branches and jumps; PC-relative or through register**
BEQZ,BNEZ	Branch GPR equal/not equal to zero; 16-bit offset from PC+4
J, JR	Jumps: 16-bit offset from PC+4 (J) or target in register (JR)
JAL, JALR	Jump and link: save PC+4 in R31, target is PC-relative (JAL) or a register (JALR)
TRAP	Transfer to operating system at a vectored address
RFE	Return to user code from an exception; restore user mode

Beispiel:

Man kann den größten gemeinsamen Teiler (GGT) zweier Zahlen mit dem so genannten Euklidischen Algorithmus berechnen. Die Idee ist, dass eine Zahl, die A und B teilt, auch A-B teilen muss. So wird beim Euklidischen Algorithmus solange die kleinere Zahl von der größeren abgezogen, bis beide gleich sind. In diesem Moment haben sie den Wert des GGT der beiden Zahlen. Wir betrachten zunächst ein Programm im Pseudocode einer höheren Programmiersprache.

Berechnung des GGT zweier Zahlen A und B.

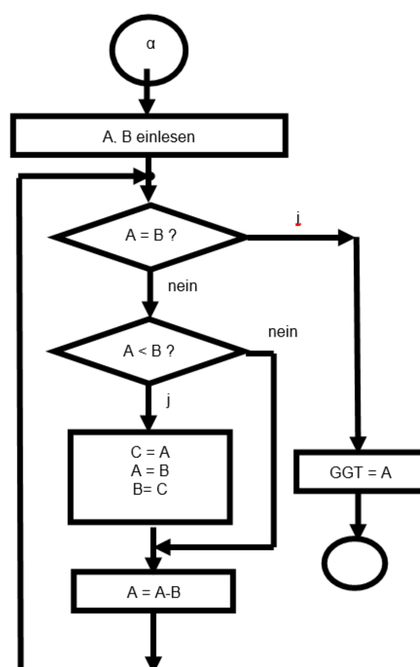
Input: Natürliche Zahlen A und B

Output: GGT(A,B)

Methode:

```
While A <> B Do
        {If  A<B Then
                {  C = A
                A = B
                B = C  }
        Endif
                A = A – B
        }
GGT = A
```

Man kann auch ein Flussdiagramm wählen, um sich die Funktion des Programms klarzumachen.



Wir wollen dafür ein Assemblerprogramm schreiben. Wir setzen voraus, dass die Operanden an den Adressen 1000 und 1004 im Speicher stehen und das Ergebnis an der Adresse 1008 entstehen soll:

Register:
R1 : A
R2 : B
R3 : Hilfsregister

```
Start     LW    R1, 1000(R0)     / Lade ersten Operanden
          LW    R2, 1004(R0)     / Lade zweiten Operanden
Loop      SEQ   R3, R1, R2       / Stelle fest, ob beide gleich sind
          BNEZ  R3, Ende         / Wenn ja, gehe zum Ende
          SLT   R3, R1, R2       / Prüfe, welches die kleinere Zahl ist
          BEQZ  R3, Weiter       / Wenn R1>R2 ist, mache nichts
          ADD   R3, R1, R0       / Andernfalls vertausche R1 und R2
          ADD   R1, R2, R0       / mit diesen drei
          ADD   R2, R3, R0       / Befehlen
Weiter    SUB   R1, R1, R2       / A := A-B
          J     Loop             / Führe die Schleife ein weiteres Mal aus
Ende      SW    1008(R0), R1     / Speichere das Ergebnis
          HALT                   / Beende das Programm
```

9.7 Stacks

Ein Stack (Kellerspeicher) ist ein Speicher, auf den nur auf bestimmte Weise zugegriffen werden kann.

Es kann immer nur der letzte gespeicherte Wert wieder gelesen werden, wobei er im Moment des Lesens aus dem Stack gelöscht wird.

Es kann immer nur ein neuer Wert zusätzlich zu den bereits im Stack befindlichen Werten geschrieben werden.

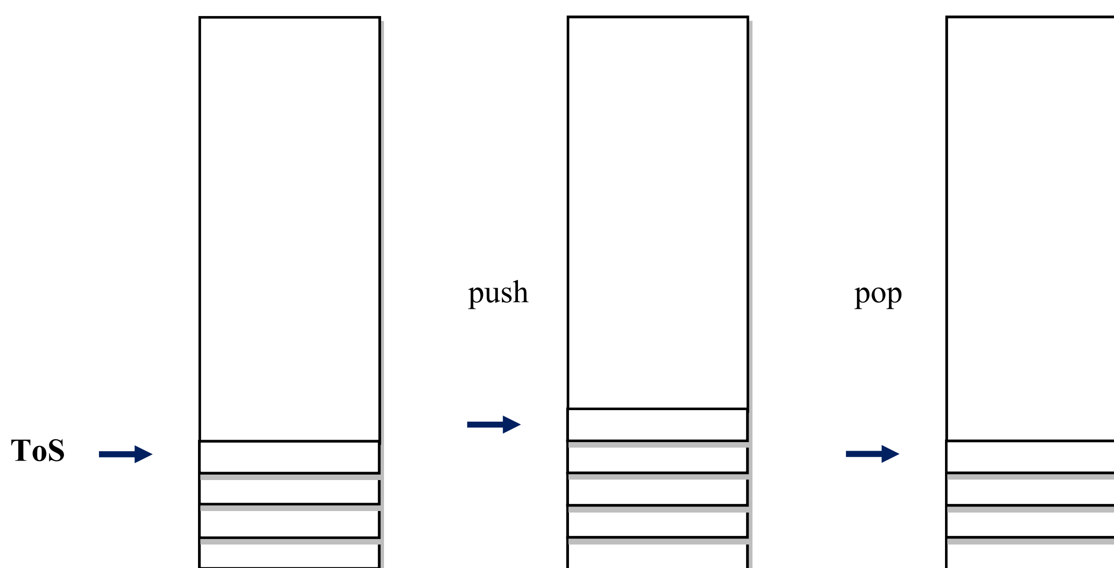
Man kann sich das so wie eine runde Tablettendose vorstellen, deren Querschnitt die Form der Tabletten hat: Wenn man eine neue Tablette hinein tut, kann man zuerst nur diese wieder heraus holen. Erst wenn man die erste Tablette heraus genommen (gelesen) hat kann, man auf die nächste, darunter liegende Tablette zugreifen.

Die Operationen auf dem Stack haben Namen: Das Schreiben in den Stack bezeichnet man als **push**-Operation, das Lesen vom Stack als **pop**-Operation.

Stacks werden in Computern realisiert als Teile von physikalisch vorhandenem RAM. Dabei benutzt man als Hilfsvariable einen Pointer (Zeiger) auf den **Top-of-Stack (ToS)**, also auf die Speicherzelle, in die zuletzt ein Element hineingeschrieben worden ist. Darüber hinaus merkt man sich ggf. in einer weiteren Hilfsvariablen den **Bottom-of-Stack**, also die Speicherstelle des ersten Elements, das im Stack steht, damit man eine Prüfmöglichkeit hat, wann der Stack durch wiederholtes Lesen leer geworden ist. Dieser Bottom-of-Stack verändert sich nie, während der Top-of-Stack bei jeder push-Operation hoch- und bei jeder pop-Operation heruntergezählt wird.

In der folgenden Grafik ist die Aktivität eines Stacks dargestellt:

Es sind vier Elemente im Stack gespeichert. Danach wird ein weiteres Element in den Stack geschrieben, indem der ToS erhöht wird und das neue Element an der Adresse des neuen ToS gespeichert wird. Danach wird ein Element gelesen, indem die Speicherzelle, auf die der ToS zeigt, gelesen wird und danach der ToS verringert wird.



Der Stack kann byte-, halbwort-, oder wortorientiert sein, wodurch sich ergibt, dass push und pop den ToS in einem Byte-adressierbaren Speicher jeweils um 1, 2 oder 4 verändern.

Stacks werden in der Informatik an vielen Stellen benötigt. Ein Beispiel dafür finden wir in der Assembler-Programmierung, wenn wir eine rekursive Funktion implementieren sollen, also wenn wir ein Unterprogramm verwenden, das sich während seiner Ausführung selbst aufruft.

Wir müssen uns nämlich beim Aufruf eines Unterprogramms die Stelle im Programm merken, von der wir weggesprungen sind, um dorthin zurück gelangen zu können, wenn das Unterprogramm seine Berechnung ausgeführt hat. Dies geschieht zweckmäßigerweise mit einem JAL-Befehl (JAL steht für **J**ump **A**nd **L**ink). Dieser vollzieht einen Sprung an die angegebene Adresse und merkt sich den NPC, also die Speicherstelle des folgenden Befehls im Programmspeicher im Register R31. Am Ende des Unterprogramms kann man dann einfach mit einem JR R31 (JR steht für **J**ump **R**egister) an diese Stelle zurückspringen und somit kann das aufrufende Programm fortgesetzt werden.

Wenn nun aber R31 bereits belegt ist und ein neuer Aufruf eines Unterprogramms findet statt (z.B. weil ein Unterprogramm sich selbst aufruft), dann müssen die Inhalte der Register, die das aufrufende Programm benötigt, gerettet werden, um sie für den Rücksprung wieder herzustellen. Dies betrifft insbesondere das Register R31, das ja für den Rücksprung gebraucht wird.

Der Speicherbereich, in dem die Register gerettet werden, ist sinnvollerweise als Stack organisiert: Im Moment des Aufrufs werden die Registerinhalte mit wiederholten push-Operationen auf den Stack geschrieben und vor dem Rücksprung aus dem Unterprogramm werden die Registerinhalte durch pop-Operationen vom Stack geholt und in den Registern wieder hergestellt.

Das erste Beispiel soll eine rekursive Berechnung des GGT zweier Zahlen mit dem Euklidischen Algorithmus sein.

Wir verwenden ein Unterprogramm, das die kleinere Zahl von der größeren abzieht und sich danach selber wieder aufruft, falls die beiden Zahlen unterschiedlich sind. Wenn die beiden Zahlen gleich sind, beendet sich das Unterprogramm.

Die beiden Zahlen sollen dabei am Anfang im Speicher an Adresse 100 und 104 stehen. Der GGT soll an Adresse 108 in den Speicher geschrieben werden.

Der Bereich für den Stack beginnt im Speicher an Adresse 1000.

Jedes Mal, wenn das Unterprogramm Euklid aufgerufen wird, wird die Rücksprungadresse in Register 31 geschrieben. Da das Unterprogramm sich aber selber wieder aufruft, würde dieses Register in dem Moment überschrieben werden. Daher müssen wir es am Anfang des Unterprogramms auf den Stack retten, um es am Ende des Unterprogramms von dort wieder zurück zu holen.

Registerbelegung:
R1: erste Zahl
R2: zweite Zahl
R3: Hilfsvariable
R4: Stackpointer (ToS)
R31: Rücksprungadresse

```
Start   ADDI R4, R0, #1000   // Initialisieren ToS
        LW   R1, 100(R0)     // Lesen der ersten Zahl
        LW   R2, 104(R0)     // Lesen der zweiten Zahl
        JAL  Euklid          // Aufruf des Unterprogramms
Rück1   SW   108(R0), R1     // Wegschreiben des Ergebnisses
        HALT                 // Ende des Programms
Euklid  SW   0(R4), R31      // Retten von R31
        ADDI R4, R4, #4      // Hochzählen des ToS
        SUB  R3, R1, R2      // prüfen, ob fertig
        BEQZ R3, Ende        // wenn fertig ans Ende springen
        SLT  R3, R1, R2      // prüfen ob richtige Reihenfolge
        BEQZ R3, R1GRR2      // wenn ja, ist R1>R2
        ADD  R3, R0, R1      // Vertauschen von R1 und R2
        ADD  R1, R0, R2      // durch Ringtausch
        ADD  R2, R0, R3      //
R1GRR2  SUB  R1, R1, R2      // Verringern von R1 um R2
        JAL  Euklid          // rekursiver Aufruf des Unterprogramms
Ende    SUBI R4, R4, #4      // Herunterzählen des ToS
        LW   R31, 0(R4)      // Zurückholen von R31
        JR   R31             // Rücksprung
```

Ein weiteres Beispiel für ein Programm mit Stack sehen wir in dem folgenden Programm zur Berechnung der Fakultät einer natürlichen Zahl. Die Berechnung von n! erfolgt wieder rekursiv: Wenn n=0 ist, wird die das Ergebnis der Fakultät im Register R14 auf 1 gesetzt. Wenn n>0 ist, wird das Ergebnis als n*(n-1)! ermittelt, wobei (n-1)! mit demselben Unterprogramm berechnet wird.

In dem Unterprogramm für die Fakultät werden zunächst die Register, die für die Berechnung erforderlich sind, auf einen Stack gerettet, der im globalen Speicher an der Adresse 1000 beginnt. Zu diesem Zweck wird der ToS am Anfang des Programms im Register R2 auf 1000 gesetzt. Bei jedem Aufruf des Unterprogramms Fakultät werden drei Register auf den Stack geschrieben, nämlich R1 (enthält den aktuellen Wert von n), R3 (enthält den aktuellen Wert von k) und R31 (enthält die Rücksprungadresse, an die wir nach der Ausführung des Unterprogramms zurückspringen müssen). Wenn die drei Registerinhalte im Speicher stehen, muss der ToS um 12 Byte (3 Worte) erhöht werden.

Erreichen wir irgendwann im Unterprogramm Fakultät den Wert 0 für n, so können wir das Ergebnis der Fakultät auf 1 setzen.

Wenn dieser Punkt erreicht ist, müssen wir wieder nacheinander aus der Kette der aufgerufenen Unterprogramme herausspringen. Zu diesem Zweck werden jedes Mal die drei obersten Werte im Stack zurück in die entsprechenden Register geschrieben, wobei anstelle von R31 für den Rücksprung R20 verwendet wird. Dann wird die für die Fakultätsberechnung erforderliche Multiplikation der bisherigen Fakultät mit k ausgeführt, um danach ein weiteres Mal zurück zu springen.

Wenn in R20 die Rücksprungadresse auftaucht, die uns ins Hauptprogramm zurück führt, ist die Fakultät vollständig berechnet und das Ergebnis kann ausgegeben werden.

Die maximale Größe des Stacks kann häufig erst zur Laufzeit des Programms ermittelt werden, da man erst dann feststellt, wie oft solche rekursiven Aufrufe ausgeführt werden. Man spricht daher von einem Laufzeitstack, der im Speicher so angelegt werden muss, dass er auch im schlechtesten Fall genügend freie Speicherzellen hat, um das Programm korrekt auszuführen.

Zunächst wieder in einer höheren Programmiersprache:

```
function Fakultät(n:integer): integer;
var k:integer;
begin
if n=0 then Fakultät := 1
else
  begin
    k:=n;
    n:=n-1;
    Fakultät := k*Fakultät(n)
  end
end;
```

Und dann in der Maschinensprache:

Registerbelegung:
R1: n
R2: Stackpointer
R3: k
R4: Fakultät
R31: Rücksprungadresse

Hauptprogramm:

```
start      LW      R1, (100)R0        // n einlesen, z.B. n=4
           ADDI    R2, R0, #1000      // Stackpointer  initialisieren ToS=BoS
           ADDI    R3, R0, #0         // Offset für Adressierung initialisieren
           JAL     Fakultät           //  R1 Fakultät wird in F4 berechnet
rück1      SF      2000(R0), R4       //  Herausschreiben des Ergebnisses
           HALT
Fakultät   SW      0(R2), R1          //  Retten der Register R1, R3, R31
           SW      4(R2), R3          //  auf den Stack
           SW      8(R2), R31         //
           ADDI    R2, R2, #12        //  ToS um drei Worte hochzählen
           BNEZ    R1, weiter         //  Wenn R1=0 ist, dann R4 = 1
           ADDI    R4, R0, #1         //  setzen
           J       return             //  Rücksprung aus dem Unterprogramm
weiter     ADD     R3, R1, R0         //  k = n
           SUBI    R1, R1, #1         //  n = n-1
           JAL     Fakultät           //  rekursiver Aufruf des Unterprogramms
rück2      MULT    R4, R3, R4         //  Fakultät := Fakultät* k
return     LW      R20, -4(R2)        //  Rücksprungadresse steht oben im Stack
           LW      R3, -8(R2)         //  Zurückholen der
           LW      R1, -12(R2)        //  Register vom Stack
           SUBI    R2, R2, #12        //  ToS um drei Werte herunterzählen
           JR      R20                //  Rücksprung
```