Prozessor-Architektur, DLX-Befehlssatz und Assembler

Entwurfsprinzipien beim Prozessor-Design

Klein ist schnell: Kleinere Hardwareeinheiten können schneller sein, weil Leitungen kürzer sind, Transistoren kleiner usw.

Make the common case fast: Wenn eine Entwurfsentscheidung zu treffen ist, ist zu untersuchen, wie häufig eine Verbesserung ggf. benutzt wird.

Die CPU-Performance-Gleichung

Jeder Computer hat einen Takt mit einer festen Taktfrequenz, z.B. 4 GHz. Der Takt teilt die Zeit in lauter kleine Abschnitte, so genannte Taktzyklen. Die Länge der Taktzyklen ist genau der reziproke Wert der Taktfrequenz:

Beispiel: f= 4GHz => Zykluszeit = $1/(4 * 10^9 * 1/s) = 1/4 * 10^{-9}$ s = $0{,}25\ 10^{-9}$ s =0,25 ns

Die CPU-Zeit eines Programms kann daher auf drei Arten angegeben werden:

a) CPU-Zeit = Anzahl der Taktzyklen für das Programm * Zykluszeit

oder

b) CPU-Zeit = Anzahl der Taktzyklen für das Programm / Taktfrequenz

c) Andererseits können wir auch die Anzahl der Instruktionen (IC=instruction count) zählen, die in dem Programm ausgeführt werden.

Wenn wir die Gesamtanzahl der Taktzyklen des Programms kennen, dann können wir den Durchschnittswert für die Anzahl der Taktzyklen pro Befehl CPI, clocks per instruction ermitteln

CPI = Anzahl der Taktzyklen des Programms / IC

Der CPI Wert ist für die quantitative Analyse ein ausgesprochen wichtiger Parameter, der Einsicht in die Effizienz unterschiedlicher Instruktionssätze gibt. Wir werden ihn häufig benutzen.

Durch Einsetzen dieser letzten Gleichung in die CPU-Zeit Gleichung erhalten wir

CPU-Zeit = IC * CPI * Zykluszeit = IC * CPI / Taktfrequenz

Ausführlicher hingeschreiben bedeutet diese erste Formel:

$$CPU\ Zeit = Anzahl\ Instruktionen\ für\ Programm \times \frac{Taktzyklen}{Instruktion} \times \frac{Sekunden}{Taktzyklus} = Sekunden\ für\ Programm$$

Die CPU-Zeit hängt von diesen drei Parametern gleichermaßen ab. Eine Verbesserung in einem von ihnen um 10% verbessert die Performance um 10%.

Die RISC Idee

RISC (reduced instruction set computer): Philosophie des Prozessor-Designs, welches die Hardware vereinfacht (und damit schneller macht). Durch Vereinfachung des Befehlformats, der Speicheradressierung, des Befehlssatzes und des Registerzugriffs wird die Leistung eines Rechners signifikant gesteigert.

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.

Die Register-Architektur: Akkumulator

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
```

Nachteil ist, dass immer auf den Hauptspeicher zugegriffen werden muss.

Die Register-Architektur: General Purpose Register

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.

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
```

Typen von GPR-Architekturen

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
```

Typen von GPR-Architekturen

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 aber mit aufwendigem CPU-Speicher-Verkehr bezahlt.

Typen von GPR-Architekturen

Typ	Vorteile	Nachteile
Register-Register	Einfaches Befehlsformat fester Länge. Einfaches Modell zur Code Generierung. Instruktionen brauchen alle etwa gleichviel Takte.	Höherer IC als die beiden anderen.
Register-Speicher	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	Sehr kompakt. Braucht keine Register für Zwischenergebnisse.	Hoher CPI. Viele Speicherzugriffe. Speicher-CPU-Flaschenhals.

Speicher-Adressierung

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 (Hersteller: 80x86, VAX, Alpha).

Big Endian bedeutet: Das Byte mit Adresse xxxx....x00 ist das Most Significant Byte (Hersteller: MIPS, Motorola, Sparc).

In Big Endian ist die Adresse eines Wortes die des MSByte, in Little Endian die des LSByte.

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 an 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

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.

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 ziegt 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

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.

Sprungbefehle und Kontrollfluss-Steuerung

Wir haben bereits die ALU und ihre Befehle kennen gelernt. Weiterhin ist wichtig, wie der Kontrollfluss des Systems vom Programmierer gesteuert werden kann. Dazu gibt es die Sprungbefehle. Es gibt vier prinzipiell verschiedene Sprungbefehle:

Conditional branch:	Bedingte Verzweigung
Jumps:	Unbedingter Sprung
Procedure Calls:	Aufruf einer Prozedur, eines Unterprogramms
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. 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.

Die DLX-560 Architektur

DLX, ausgesprochen deluxe

- Datenformate und Befehlssatz
- Bauteile und Datenpfade
- Assemblerprogrammierung

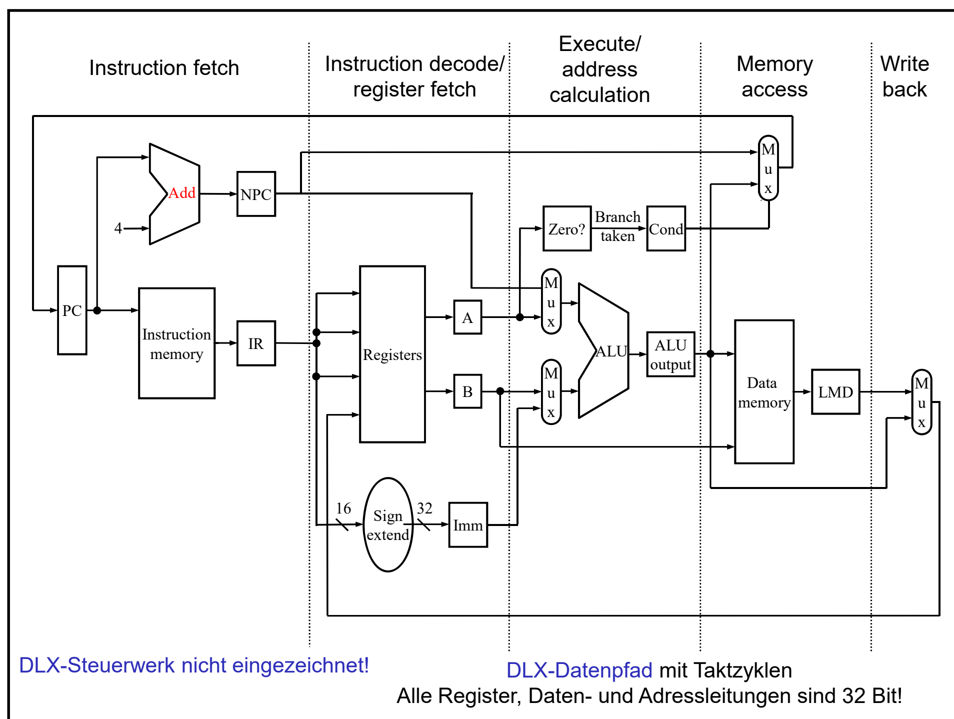
Das Verständnis dieser Architektur ist wichtig, sowohl für die Entwicklung von Hardware als auch für die Entwicklung von schneller und zuverlässiger Software.

Aus den vorangegangenen Kapiteln haben wir eine Reihe von Lehren gezogen, die wir jetzt in einer Beispielarchitektur umsetzen wollen:

Was sind diese Vorgaben?

Vorgaben DLX

- GPR-Architektur, load-store (=Register-Register)
- Adressierung: Displacement, Immediate, Indirect
- schnelle einfache Befehle (load, store, add, ...)
- 8-Bit, 16-Bit, 32-Bit Integer
- 32-Bit, 64-Bit Floating-point
- feste Länge des Befehlsformats, wenige Formate
- Mindestens 16 GPRs



DLX-Steuerwerk nicht eingezeichnet!

DLX-Datenpfad mit Taktzyklen
Alle Register, Daten- und Adressleitungen sind 32 Bit!

Aufbau DLX

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
- Es gibt 32 FP-Register. Jedes ist 32 Bit lang, F0,..,F31. Diese können wahlweise als einzelne Single-Register verwendet werden oder paarweise als Double-Register F0, F2,...,F30.
- Zwischen den Registern unterschiedlicher Art gibt es spezielle Move-Befehle

Aufbau DLX

Datentypen:

- Integer-Typen als unsigned Integer oder im 2-er Komplement:
  - 8-Bit Bytes (B)
  - 16-Bit Halbworte (H)
  - 32-Bit Worte (W)
- Floatingpoint Typen im IEEE Standard 754
  - 32-Bit Single Precision (F)
  - 64-Bit Double Precision (D)
- Laden von Bytes und Halbworten wahlweise mit führenden Nullen (unsigned) oder mit Replikation der Vorzeichenstelle (2-er Komplement)

Aufbau DLX

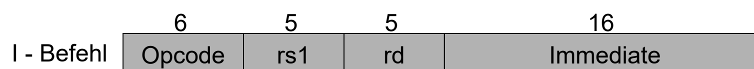
Adressierungsarten:

- Displacement und Immediate. Durch geschickte Benutzung von R0 und 0 können damit vier Adressierungsarten realisiert werden:
  - Displacement: LW R1, 1000(R2);
  - Immediate: LW R1, #1000;
  - Indirect: LW R1, 0(R2);
  - Direct: LW R1, 1000(R0);

Displacement ist dabei die mit Abstand wichtigste Adressierungsart

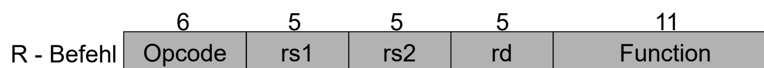
Befehlsformat

I - Befehl



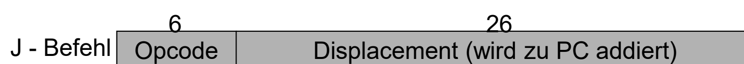
Typische Immediate-Befehle (rd ← rs1 op immediate)
Loads und Stores von Bytes, Worten, Halbworten (rs1 unbenutzt)
Bedingte Verzweigungen (rs1 : register, rd unbenutzt)
Jump register, Jump and link register
(rd = 0, rs1 = destination, immediate = 0)

R - Befehl



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

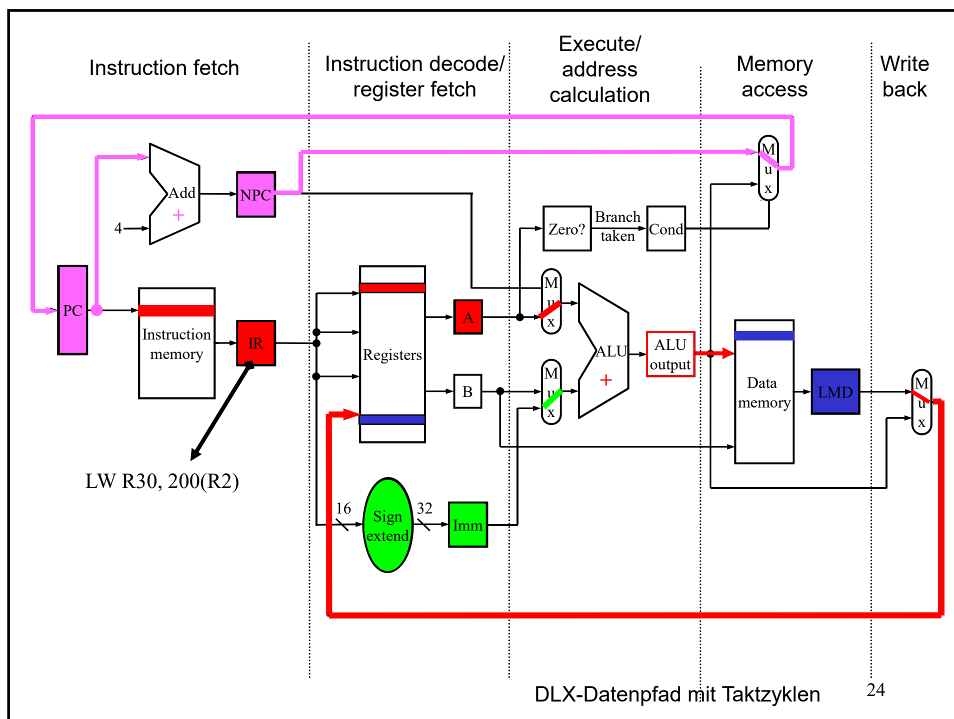
J - Befehl



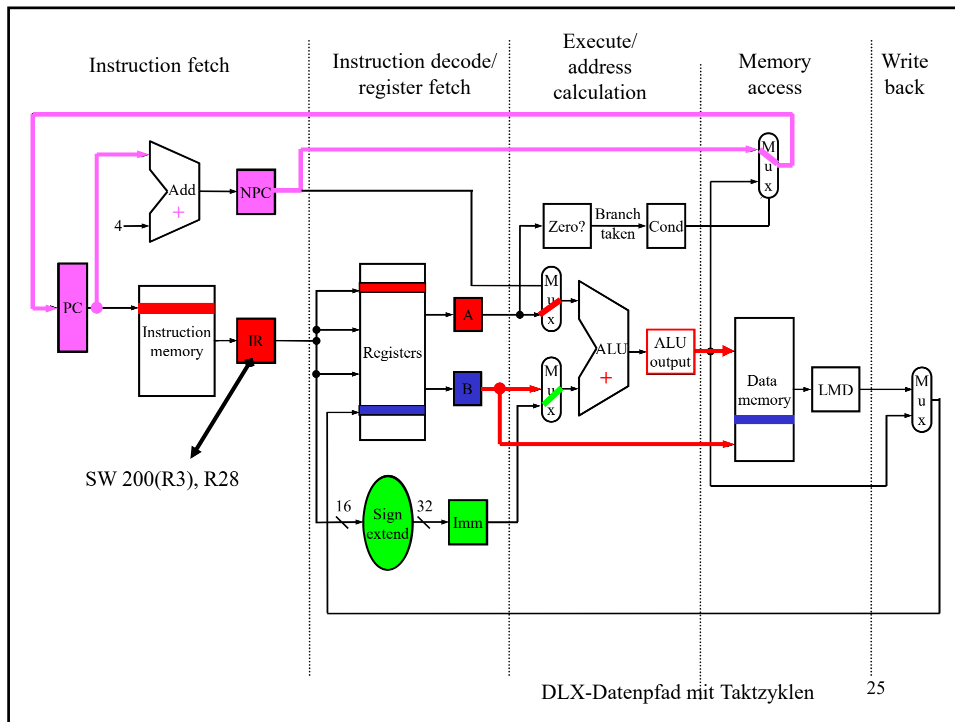
Jump und Jump and link
Trap und Return from exception

Befehle mit Speicherzugriff

Befehl		Name	Bedeutung
LW	R1,30(R2)	Load word	Regs [R1] $\leftarrow_{32}$Mem [30+Regs [R2]]
LW	R1,1000(R0)	Load word	Regs [R1] $\leftarrow_{32}$ Mem [1000+0]
LB	R1,40(R3)	Load byte	Regs [R1] $\leftarrow_{32}$ (Mem [40+Regs [R3]]$_0$)^{24} # # Mem[40+Regs[R3]]
LBU	R1,40(R3)	Load byte unsigned	Regs [R1] $\leftarrow_{32}$ 0^{24} # # Mem[40+Regs [R3]]
LH	R1,40(R3)	Load half word	Regs [R1] $\leftarrow_{32}$ (Mem[40+Regs [R3]]$_0$) ^{16} # # Mem [40+Regs [R3]] # # Mem[41+Regs [R3]]
LF	F0,50(R3)	Load float	Regs [F0] $\leftarrow_{32}$ Mem [50+Regs [R3]]
LD	F0,50(R2)	Load double	Regs [F0] # #Regs [F1] $\leftarrow_{64}$ Mem[50+Regs [R2]]
SW	500(R4),R3	Store word	Mem [500+Regs [R4]] $\leftarrow_{32}$ Regs [R3]
SF	40(R3),F0	Store float	Mem [40+Regs [R3]] $\leftarrow_{32}$ Regs [F0]
SD	40(R3),F0	Store double	Mem[40+Regs [R3]] $\leftarrow_{32}$ Regs [F0]; Mem[44+Regs [R3]] $\leftarrow_{32}$ Regs [F1]
SH	502(R2),R3	Store half	Mem[502+Regs [R2] $\leftarrow_{16}$ Regs [R3]$_{16...31}$
SB	41(R3),R2	Store byte	Mem[41+Regs [R3]] $\leftarrow_{8}$ Regs [R2]$_{24...31}$



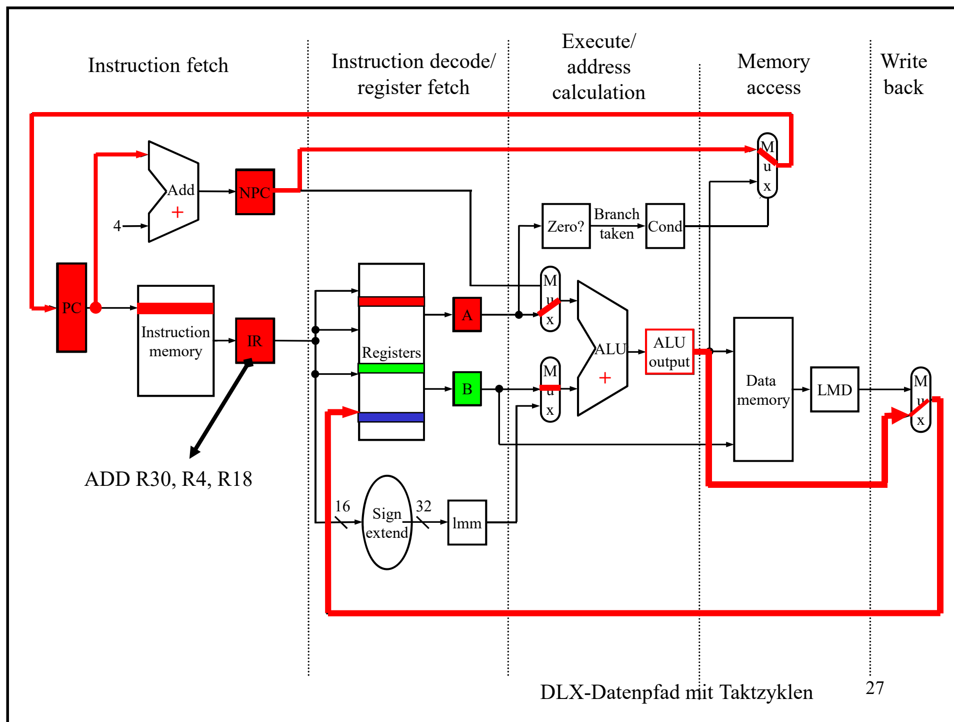
DLX-Datenpfad mit Taktzyklen



DLX-Datenpfad mit Taktzyklen

ALU-Befehle

Befehl		Name	Bedeutung
SUB	R1, R2, R3	Subtract	Regs [R1] ← Regs[R2] – Regs[R3]
ADDI	Rl, R2, #3	Add immediate	Regs [Rl] ← Regs [R2]+3



DLX-Datenpfad mit Taktzyklen

Weitere Arithmetik- / Logik-Befehle

Befehl		Name	Bedeutung
SLLI	R1,R2,#5	Shift left logical immediate	Regs [R1] ← Regs [R2]<<5
SLT	R1, R2, R3	Set less than	if (Regs[R2]<Regs[R3]) Regs [R1]
			Else Regs [R1] ←0

Sprungbefehle

Befehl		Name	Bedeutung
J	name	Jump	PC ← name; ((PC+4) - 2^{25}) ≤ name < ((PC+4)+2^{25})
JAL	name	Jump and link	Regs[R31]←PC+4; PC ← name; ((PC+4) – 2^{25}) ≤ name < ((PC+4) +2^{25})
JALR	R2	Jump and link register	Regs [R31] ← PC+4; PC ← Regs [R2]
JR	R3	Jump register	PC ← Regs [R3]
BEQZ	R4,name	Branch equal zero	if (Regs [R4] =0) PC←name; ((PC+4) – 2^{15}) ≤ name < ((PC+4) +2^{15})
BNEZ	R4,name	Branch not equal zero	if (Regs [R4] ≠ 0) PC←name; ((PC+4) – 2^{15}) < name < ((PC+4) +2^{15})

PC-relative Sprungbefehle

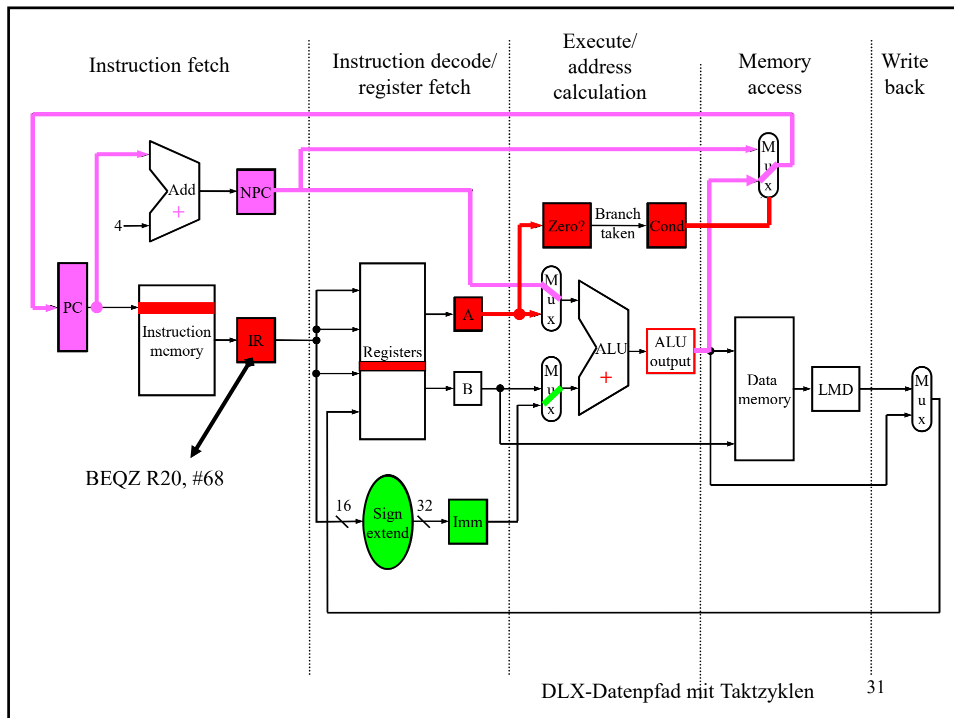
Die Syntax der PC-relativen Sprungbefehle ist etwas irreführend, da der als Operand eingegebene Parameter als Displacement zum PC zu verstehen ist.

Tatsächlich müßte die erste Zeile heißen:

J offset bedeutet: PC <--- PC+4 + offset mit $-2^{25} <= offset < +2^{25}$

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 Sprungmarken (Label) ins Assemblerprogramm schreibt.

Ein Precompiler wandelt die Namen in die Offsets um, so daß anschließend die tatsächlichen Offsets verwendet werden können. Für Entwickler und Leser ist ein solches mit Marken geschriebenes Programm leichter verständlich.



DLX-Datenpfad mit Taktzyklen

Gleitkommabefehle

Befehl		Name	Bedeutung
ADDS	F2, F0, F1	Add single precision floating point numbers	Regs[F2] ← Regs[F0] + Regs[F1]
MULTD	F4, F0, F2	Multiply double precision floating point numbers	Regs[F4]##Regs[F5] ← Regs[F0]##Regs[F1] * Regs[F2]##Regs[F3]

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, siore word (to/from integer registers)
LF, LD, SF, SD	Load SP float, load DP float, store SP float, store DP float
MOVI2S, MOVS2I	Move from/to GPR to/from a special register
MOVF, MOVD	Copy one FP register or a DP pair to another register or pair
MOVFP2I,MOVI2FP	Move 32 bits from/to FP registers 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,DIV,DIVU	Multiply and divide, signed and unsigned; operands must be FP registers; all operations take and yield 32-bit values
AND,ANDI	And, and immediate
OR,ORI,XOR,XORI	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 I) 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
BFPT,BFPF	Test comparison bit in the FP status register and branch; 16-bit offset from PC+4
J, JR	Jumps: 26-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
Floating point	**FP operations on DP and SP formats**
ADDD,ADDF	Add DP. SP numbers
SUBD,SUBF	
MULTD,MULTF	Multiply DP, SP floating point
DIVD, DIVF	Divide DP, SP floating point
CVTF2D, CVTF2I, CVTD2F, CVTD2I, CVTI2F, CVTI2D	Convert instructions: CVTx2y converts from type x to type y, where x and y are I (integer), D (double precision), or F (single precision). Both operands are FPRs.
_D, _F	DP and SP compares: "_" = LT, GT, LE, GE, EQ, NE; sets bit in FP status register

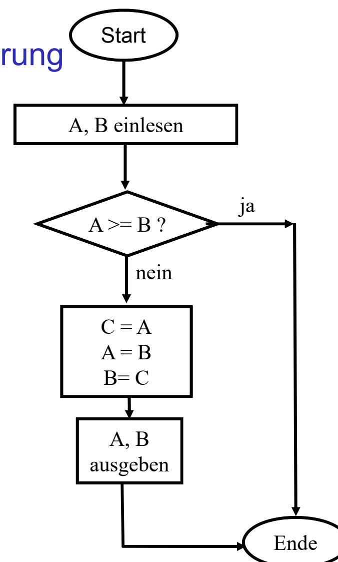
DLX-Assembler Befehlssatz (nur diese Befehle sind im Hüsertool implementiert und sollen verwendet werden)

Die Befehle werden in der Form *Instr. / Ziel / Quelle(n)* verwendet. Bsp: ADDI R3 R2 #15 ≈ R3:=R2+15

	Instr.	Description	Format	Operation (C-style coding)	
ADD	add	R	Rd = Rs1 + Rs2		
ADDI	add immediate	I	Rd = Rs1 + extend(immediate)		
AND	and	R	Rd = Rs1 & Rs2		
ANDI	and immediate	I	Rd = Rs1 & extend(immediate)		
BEQZ	branch if equal to zero	I	PC += (Rs1 == 0 ? 4 + extend(immediate))		
BNEZ	branch if not equal to zero	I	PC += (Rs1 != 0 ? 4 + extend(immediate))		
J	jump	J	PC += 4 + extend(immediate)		
JAL	jump and link	J	R31 = PC + 4 ; PC += 4 + extend(immediate)		
JALR	jump and link register	I	R31 = PC + 4 ; PC = Rs1		
JR	jump register	I	PC = Rs1		
LW	load word	I	Rd = MEM[Rs1 + extend(immediate)]		
OR	or	R	Rd = Rs1 | Rs2		
ORI	or immediate	I	Rd = Rs1 | extend(immediate)		
SEQ	set if equal	R	Rd = (Rs1 == Rs2 ? 1 : 0)	if (True) then 1 else 0	
SEQI	set if equal to immediate	I	Rd = (Rs1 == extend(immediate) ? 1 : 0)	if (True) then 1 else 0	
SLE	set if less than or equal	R	Rd = (Rs1 <= Rs2 ? 1 : 0)	if (True) then 1 else 0	
SLEI	set if less than or equal to immediate	I	Rd = (Rs1 <= extend(immediate) ? 1 : 0)	if (True) then 1 else 0	
SLL	shift left logical	R	Rd = Rs1 << (Rs2 % 32)		
SLLI	shift left logical immediate	I	Rd = Rs1 << (immediate % 32)		
SLT	set if less than	R	Rd = (Rs1 < Rs2 ? 1 : 0)	if (True) then 1 else 0	
SLTI	set if less than immediate	I	Rd = (Rs1 < extend(immediate) ? 1 : 0)	if (True) then 1 else 0	
SNE	set if not equal	R	Rd = (Rs1 != Rs2 ? 1 : 0)	if (True) then 1 else 0	
SNEI	set if not equal to immediate	I	Rd = (Rs1 != extend(immediate) ? 1 : 0)	if (True) then 1 else 0	
SRA	shift right arithmetic	R	as SRL & see below		
SRAI	shift right arithmetic immediate	I	as SRLI & see below		
SRL	shift right logical	R	Rd = Rs1 >> (Rs2 % 32)		
SRLI	shift right logical immediate	I	Rd = Rs1 >> (immediate % 32)		
SUB	subtract	R	Rd = Rs1 - Rs2		
SUBI	subtract immediate	I	Rd = Rs1 - extend(immediate)		
SW	store word	I	MEM[Rs1 + extend(immediate)] = Rs2		
XOR	exclusive or	R	Rd = Rs1 ^ Rs2		
XORI	exclusive or immediate	I	Rd = Rs1 ^ extend(immediate)		

Beachten Sie: Die Befehle SRA und SRAI füllen die vorderen Bits des Registers mit dem aktuellen Vorzeichenbit auf.
Ergänzung: Die Befehle HALT oder TRAP #0 beenden das Programm.

Beispiele für Assembler-Programmierung



Sortiere zwei Zahlen A und B absteigend:
A steht an Adresse 1000, B an Adresse 1004
Die größere Zahl soll am Ende in 1000 stehen,
die kleinere in 1004

Input: Natürliche Zahlen A und B
Output: A und B in der Reihenfolge der Größe
Methode: Tausche Inhalte A <-> B, falls A < B

```
if (A < B) Then
{
        C = A
        A = B
        B = C
        Output A
        Output B
}
Endif
```


Sortiere zwei Zahlen A und B absteigend:

Wir wollen dafür ein Assemblerprogramm schreiben. Wir setzen voraus, dass die Operanden A und B an den Adressen 1000 und 1004 im Speicher stehen und das Ergebnis sortiert an denselben Adressen entstehen soll:

```
/Register:
/R1, R2:  A, B
/R3, R4:  Hilfsregister
Start:    LW      R1, 1000(R0)    /Lade A nach R1
          LW      R2, 1004(R0)    /Lade B nach R2
          SLT     R3, R2, R1      /Setze R3 (ungleich 0), falls B < A
          BNEZ    R3, Ende        /Zahlen bereits in der richtigen Reihenfolge.
                                  /Sonst vertausche A und B durch Ringtausch:
          ADD     R4, R1, R0      /R4 := R1 => C = A
          ADD     R1, R2, R0      /R1 := R2 =>  A = B
          ADD     R2, R4, R0      /R2 := R4 => B = C
          SW      1000(R0), R1    /Speichern des Maximums
          SW      1004(R0), R2    /Speichern des Minimums
Ende:     HALT                    /Ende des Programms
```

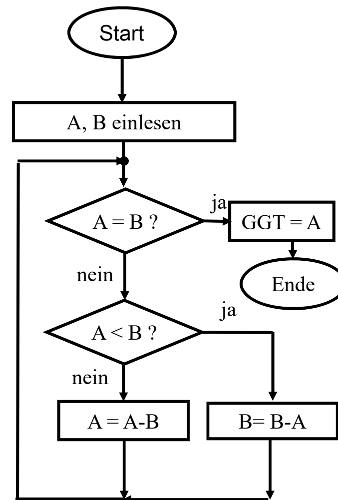
Ganz wichtig:
Programmbeschreibung, Registerbelegung, Kommentare (nicht generisch)
Test mit https://huesersohn.github.io/dlx/ => Vermeidung syntaktischer Fehler

Beispiel: Berechnung des GGT zweier Zahlen A und B.

Input: Natürliche Zahlen A und B
Output: GGT(A,B)
Methode: Euklidischer Algorithmus

```
while (A != B)    // GGT noch nicht gefunden
{
    If  (A<B)
        B = B-A  // B größer, ziehe A ab
    else
        A = A-B  // A größer, ziehe B ab

}
GGT = A   // A == B, gebe GGT aus
```



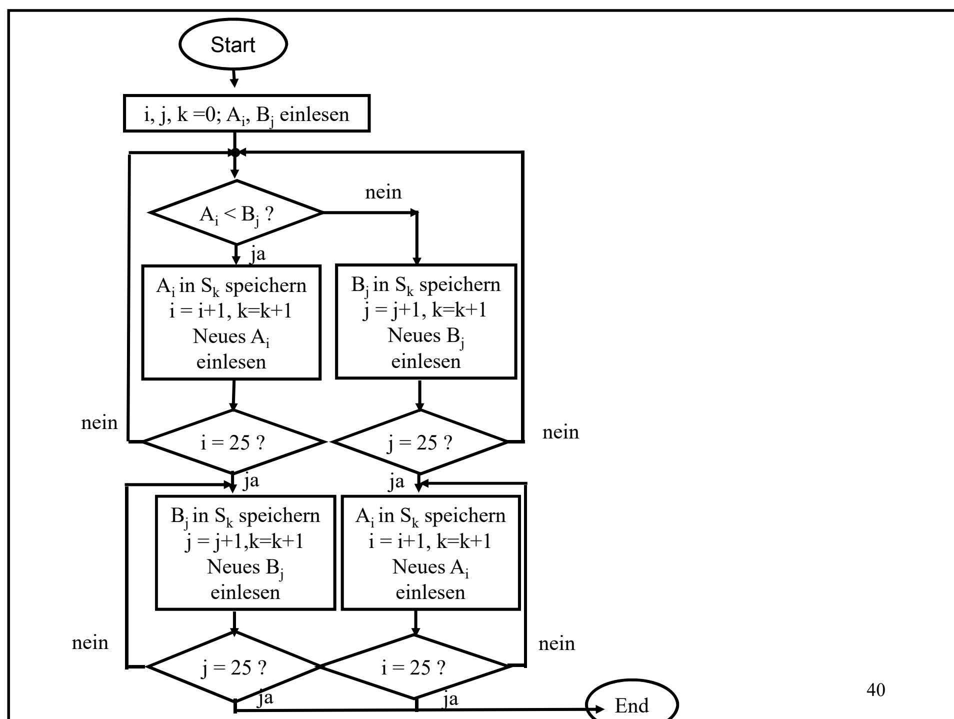

```
/Programmbeschreibung GGT:
/Das Programm liest 2 positive Integerwerte A und B aus den Adressen 1000 bzw. 1004,
/berechnet GGT(A,B) und speichert ihn an Adresse 1008.
/Beispiel: GGT(28,12)=4

/Registerbelegung:
/R1,R2:          A, B
/R3:    Hilfsregister
Start:  LW      R1, 1000(R0) /Lade A nach R1
        LW      R2, 1004(R0) /Lade B nach R2
Loop:   SEQ     R3, R1, R2   /while: R3 wird gesetzt, wenn (A==B) => A ist GGT
        BNEZ    R3, Ende     / => Geh zum Ende, while ist false
        SLT     R3, R1, R2   /if: setze R3, wenn (A<B)
        BNEZ    R3, AklB     /Falls A<B gehe zu AklB (then-Zweig)
        SUB     R1, R1, R2   /else-Zweig: A > B  => A:=A-B,
        J       Loop         /Geh zum Anfang der Schleife (while)
AklB:   SUB     R2, R2, R1   / then-Zweig: A < B  => B:=B-A
        J       Loop         /Geh zum Anfang der Schleife (while)
Ende:   SW      1008(R0), R1 /Speichere GGT, der in A steht.
        Halt                 /Programmende
```

Weitere Beispiele für Assembler-Programmierung

Mergen (zusammenbringen) zweier sortierter Listen der Länge 25. Die eine ist ab Adresse 1000 im Speicher, die andere ab Adresse 1100 abgelegt. Die sortierte Gesamtliste S soll ab Adresse 1200 in den Speicher geschrieben werden. Sortiert heißt: Das kleinste Element steht in der Zelle mit der kleinsten Adresse und von da an aufsteigend. Es sind gleichgroße Elemente hintereinander erlaubt.

Input: Zwei sortierte Listen A und B
Output: Eine sortierte Gesamtliste S
Methode:
Das jeweils kleinste Element der einen Liste wird mit dem jeweils kleinsten Element der zweiten Liste verglichen. Das kleinere von beiden wird in die Gesamtliste gespeichert. Ein neues nunmehr kleinstes Element wird aus der Liste geladen, aus der das eben gespeicherte Element kam.
Sobald eine der Listen verbraucht ist, speichert man die restlichen Elemente der anderen Liste in der Reihenfolge ab, in der sie bereits sortiert sind.



```
/Programmbeschreibung:
/Merge 2 aufsteigend sortierte Listen A, B der Länge 25 in eine Gesamtliste S.
/A beginnt an Adresse 1000, B an 1100, S soll ab 1200 gespeichert werden.

/Registerbelegung:
/R1,R2,R3:  Zeiger auf Listen A, B bzw. S
/R4,R5:     Ai, Bj
/R6:        Hilfsregister für Vergleiche

START:  ADD    R1, R0, R0       /Initialisiere Zeiger auf Listen R1=A,R2=B,R3=S mit 0
        ADD    R2, R0, R0       / R1=i, R2=j, R3=k: Zähler der Ergebnisliste
        ADD    R3, R0, R0
        LW     R4, 1000(R1)     /Lade Ai
        LW     R5, 1100(R2)     /Lade Bj
LOOP:   SLT    R6, R4, R5       /Ist (Ai<Bj)?
        BNEZ   R6, AkB          /Spring ggf. zu AkB (A kleiner B)
        SW     1200(R3), R5     /Sk:= Bj
        ADDI   R3, R3, #4       /Erhöhe Zeiger k von S
        ADDI   R2, R2, #4       /Erhöhe Zeiger j von B
        LW     R5, 1100(R2)     /Lade nächstes Bj
        SLTI   R6, R2, #100     /Sind noch weitere Elemente in Liste B?
        BNEZ   R6, LOOP         /Springe ggf. zu Loop und vergleiche weiter.
```

/Fortsetzung auf der nächsten Folie


```
ARAUS:  SW     1200(R3), R4     /Ab hier wird Rest von A herausgeschrieben
        ADDI   R3, R3, #4       /Erhöhe Zeiger k von S
        ADDI   R1, R1, #4       /Erhöhe Zeiger i von A
        LW     R4, 1000(R1)     /Lade nächstes Ai
        SLTI   R6, R1, #100     /Sind noch weitere Elemente in Liste A?
        BNEZ   R6, ARAUS        /Springe ggf. zu ARAUS.
        J      ENDE             /Sonst springe zum Ende
AkB:    SW     1200(R3), R4     /Sk:= Ai
        ADDI   R3, R3, #4       /Erhöhe Zeiger k von S
        ADDI   R1, R1, #4       /Erhöhe Zeiger i von A
        LW     R4, 1000(R1)     /Lade nächstes Ai
        SLTI   R6, R1, #100     /Sind noch weitere Elemente in Liste A?
        BNEZ   R6, LOOP         /Springe ggf. zu Loop und vergleiche weiter.
BRAUS:  SW     1200(R3), R5     /Ab hier wird Rest von B herausgeschrieben
        ADDI   R3, R3, #4       /Erhöhe Zeiger k von S
        ADDI   R2, R2, #4       /Erhöhe Zeiger j von B
        LW     R5, 1100(R2)     /Lade nächstes Bj
        SLTI   R6, R2, #100     /Sind noch weitere Elemente in Liste B?
        BNEZ   R6, BRAUS        /Springe ggf. zu BRAUS.
Ende:   HALT
```

Leitfaden für die Bearbeitung von Programmieraufgaben

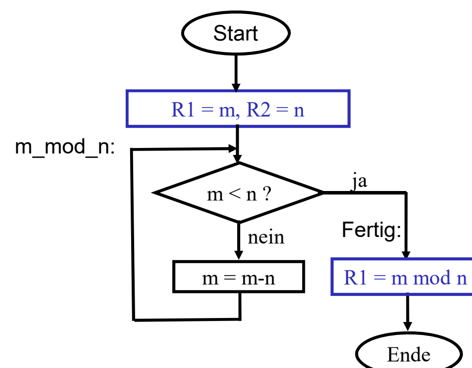
- Nehmen Sie sich die Zeit, die Aufgabe gründlich zu lesen.
- Kleine Beispiele überlegen und aufschreiben, die unterschiedliche Fälle abdecken.
- Welche Zahlenformate sind geeignet?
- Welche Grenzfälle gibt es?
- Welche Probleme können auftreten (z.B. Über-, Unterlauf, Rundungsfehler, Division durch 0)?
- Entwerfen Sie einen Algorithmus und optimieren Sie diesen (also nicht gleich Code schreiben).
- Kleine Code-Stücke schreiben und testen.
- Programm gut kommentieren und beschreiben (auch dabei entdeckt man oft noch Fehler).
- Programm anhand der kleinen Beispiele schrittweise durchlaufen lassen.
- Zum Schluss auch größere Beispiele testen.

In der Klausur dürfen Sie das Hüser-Tool nicht verwenden. Sie sollten deshalb genügend üben, damit Sie auch ohne die Hilfen des Hüser-Tools (möglichst) fehlerfreien Code schreiben können.

Unterprogramme

Betrachten wir zunächst folgendes Programm m_mod_n :

Berechne m modulo n für positive Integerwerte m und n.
Beim Programmstart sollen m in R1 und n in R2 stehen.
Am Programmende soll das Ergebnis in R1 stehen.
Das Ergebnis ist 0 <= R1 < n=R2
Die eigentliche Operation beginnt bei m_mod_n: und endet bei Fertig:



Unterprogramme

Aufgabe: Berechne e = (b mod a) + (d mod c) durch Wiederverwendung m_mod_n

Einfaches Beispiel als C-Programm:

```
#include <stdio.h>
int Up_m_mod_n(int m, int n); // UP für modulo-Berechnung

int main()   // Hauptprogramm
{
   int a=5, b=19, c=3, d=7, e=0,r;              // initialisiere
   r= Up_m_mod_n(b,a);                          // Übergebe an UP, berechne Resultat
   printf("Berechne %i mod %i = %i\n",b, a, r);
   e=r;                                         //Speicherung erstes Resultat
   r= Up_m_mod_n(d,c);                          // Übergebe an UP, berechne Resultat
   printf("Berechne %i mod %i = %i\n",d, c, r);

   e=e+r;                                       // summiere Ergebnis
   printf("Summe = %i\n",e);
   return 0;
}
```


Unterprogramm in C

Betrachten wir zunächst folgendes Programm m_mod_n :

```
#include <stdio.h>
// UP für modulo


int Up_m_mod_n(int m, int n)
{                                    //Definition der Funktion, C-style
   int r = m;                        // Zwischenvariable, lokal im UP

    while (r >= n)                   // wenn r echt kleiner n
    r=r-n;                           // ziehe n solange ab, bis r echt kleiner n

   return r;                         // Rückgabe
}
```

Unterprogramm in C

Jetzt UP in Assembler-Style:

```
#include <stdio.h>
// UP für modulo


int Up_m_mod_n(int m, int n)            //Definition der Funktion
{
    int r = m;                    // Zwischenvariable, lokal im UP

m_mod_n:
    if (r < n)                    // wenn r echt kleiner n
        goto Fertig;              // dann m mod n gefunden, beende UP
    else
        r=r-n;                    // ziehe n solange ab, bis r echt kleiner n

    goto loop;                    // schlechte C-Prog. aber OK für Assembler

Fertig: return r;                 //UP beenden, Ergebnis zurück geben
}
```


Unterprogramme

Programmcode:
/Berechne m modulo n für positive Integerwerte m und n.
/Beim Programmstart sollen m in R1 und n in R2 stehen.
/R3 wird als Hilfsregister für Vergleiche verwendet.
/Am Programmende soll das Ergebnis in R1 stehen.

```
m_mod_n:                /Schleifenbeginn
SLT     R3,R1,R2        /R3 wird genau dann gesetzt, wenn R1<n .
BNEZ    R3,Fertig       /in R1steht das Ergebnis m modulo n => Fertig
SUB     R1,R1,R2        /R1 soll solange um n reduziert werden, bis R1<0
J       m_mod_n         /Springe zum Schleifenanfang
Fertig:                 /Programmende
HALT
```

Welche Probleme treten auf, wenn m_mod_n als Unterprogramm verwendet werden soll?
- Rücksprungadresse?
- Registerbelegung?
- Parameterübergabe?

=> Stacks (Kellerspeicher) verwenden!

Der Stack

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

Mit dem push-Befehl wird ein neuer Wert auf den Stack gelegt (von oben in den Keller gelegt), zusätzlich zu den bereits im Stack befindlichen Werten.

Mit dem pop-Befehl wird der oberste Wert vom Stack gelesen, wobei er im Moment des Lesens aus dem Stack gelöscht wird (von oben aus dem Keller geholt).

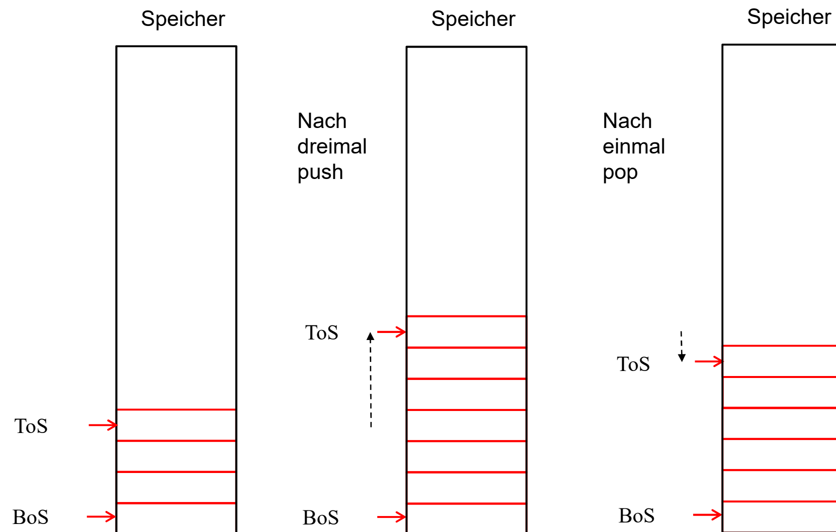
Man kann sich das auch so wie eine runde Tablettendose vorstellen, deren Querschnitt die Form der Tabletten hat: Wenn man eine neue Tablette hinein tut, kann man zuerst nur diese wieder heraus holen. Erst wenn man die oberste Tablette heraus genommen (gelesen) hat kann man auf die nächste, darunter liegende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

Der Stack

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 (BoS), 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 werden drei weitere Elemente in den Stack geschrieben, jeweils 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 eine byte-adressierbaren Speicher jeweils um 1, 2 oder 4 verändern.

Der Stack



Der Stack und Unterprogramme

Stacks werden in der Informatik an vielen Stellen verwendet. Ein Beispiel dafür finden wir in der Assembler-Programmierung, wenn wir geschachtelte Unterprogramme implementieren sollen, die von unterschiedlichen Stellen aufgerufen werden sollen.

Wir müssen uns dann 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.

Lokale Variablen und Register in Unterprogrammen

Auch für andere Register kann die Rettung der Registerinhalte wichtig sein.
Im bereits vorgestellten Programm m_mod_n verwenden wir R1 bis R3.
Wenn wir nun aus diesem Programmstück ein Unterprogramm entwickeln wollen, dass von beliebigen Stellen aufgerufen werden kann, müssen wir uns Gedanken darüber machen, ob R1 bis R3 vom aufrufenden Programm anders verwendet wurden und die Registerinhalte von R1 bis R3 beim Aufruf des Unterprogramms zunächst gerettet und bei der Beendigung wieder hergestellt werden müssen.

Der Speicherbereich, in dem die Register gerettet werden, ist sinnvollerweise als Stack organisiert:

- Unmittelbar nach dem Aufruf des Unterprogramms werden die Registerinhalte mit wiederholten push-Operationen auf den Stack geschrieben.
- Unmittelbar vor dem Rücksprung aus dem Unterprogramm werden die Registerinhalte durch pop-Operationen vom Stack geholt und in den Registern wieder hergestellt.

Der DLX hat leider keinen speziellen push/pop-Befehl, sondern ToS wird über ein selbst gewähltes Register (z.B. R30) und dessen Inkrementierung / Dekrementierung realisiert. R30 darf dann nur für ToS genutzt werden!

Parameterübergabe in Unterprogrammen mit dem Stack

Des weiteren kann mit Hilfe des Stacks die Parameterübergabe erfolgen:
Bei unserem Beispiel m_mod_n kann das aufrufende Programm die Parameter m und n auf den Stack pushen und zusätzlich einen Speicherplatz für das Ergebnis reservieren.
Das Unterprogramm kann dann die Parameter m und n vom Stack holen und am Ende des Unterprogramms das Ergebnis an die reservierte Stelle des Stacks schreiben.

Beispiel:
/Berechne R5:=(R2 mod R1) + (R4 mod R3) mit Hilfe
/ des angepassten Unterprogramms Up_m_mod_n.
/Voraussetzungen:
/ In R1 bis R4 stehen positive Integerwerte. Ergebnis < 2^31.
/ R5: Ergebnis
/ R6: Hilfsregister
/ R7 bis R29 werden nicht überschrieben
/ R30: ToS (=Top of Stack), zu Beginn ToS=1000

Hauptprogramm zum Unterprogramm Up_m_mod_n

```
/Programmbeschreibung und Registerbelegung s. vorangegangene Folie
Hp_Start:                /Start des Hauptprogramms
ADDI R30,R0,#1000        /ToS wird mit 1000 initialisiert
                         / erster UP-Aufruf, Übergabe vom HP auf Stack
SW   0(R30),R2           /R2, Speichern der Übergabeparameter auf dem Stack
SW   4(R30),R1           /R1
SW   8(R30),R0           /Reservierung für das Ergebnis und Initialisierung
ADDI R30,R30,#12         /Erhöhe ToS (3x push, daher 3x4=12 Byte)
JAL  Up_m_mod_n          /Berechne R2 mod R1 in Unterprogramm
SUBI R30,R30,#12         /Reduziere ToS (3x pop auf ToS)
LW   R5, 8(R30)  /Hole das Ergebnis (R2 mod R1) vom Stack und speichere es in R5
                         / Zweite UP-Aufruf, wieder hole Stackprozedur
SW   0(R30),R4           /R4, Speichern der Übergabeparameter auf dem Stack
SW   4(R30),R3           /R3, für den 2. Aufruf des Unterprogramms mit R4 mod R3
SW   8(R30),R0           /Reservierung für das Ergebnis und Initialisierung
ADDI R30,R30,#12         /Erhöhung ToS (3x push, daher 3x4=12 Byte)
JAL  Up_m_mod_n          /Berechne R4 mod R3
SUBI R30,R30,#12         /Reduziere ToS (3x pop auf ToS)

LW   R6, 8(R30)  /Hole das Ergebnis (R4 mod R3) vom Stack und speichere es in R6
ADD  R5,R5,R6    /Berechne das Gesamtergebnis R5=(R2 mod R1) + (R4 mod R3)
HALT             /Ende des Hauptprogramms
```


Angepasstes Unterprogramm Up_m_mod_n (1. Teil)

```
Up_m_mod_n:      /Start des Unterprogramms
/Berechnet m modulo n für positive Integerwerte m und n.
/m und n werden vom Stack geholt und in R1 bzw. R2 gespeichert.
/R3 wird als Hilfsregister für Vergleiche verwendet.
/Am Programmende wird das Ergebnis auf den Stack geschrieben.
/ Der Stack ist durch HP schon mit den Übergabeparametern belegt. ToS (R30) zeigt
auf die nächste freie Stackadresse
                         /Alle Operationen ab hier nur Parametersicherung
SW   0(R30),R1           /Retten der Registerinhalte R1 bis R3, die vom
SW   4(R30),R2           /Unterprogramm überschrieben werden.
SW   8(R30),R3
SW   12(R30),R31         /Rücksprungadresse sichern, falls innerhalb des Unter-
                         /programms weitere Unterprogramme aufgerufen werden
LW   R1,-12(R30)         /Holen der HP-Übergabeparameter vom Stack
LW   R2,-8(R30)
ADDI R30,R30,#16         /Erhöhen des Stackpointers (4 x push)
                         /Alle Operationen bis hier nur Parametersicherung
                         /Fortsetzung auf der nächsten Folie
```

Angepasstes Unterprogramm Up_m_mod_n (2. Teil)

```
m_mod_n:                  /ab hier beginnt eigentliche Operation des UPs
SLT   R3,R1,R2            /m<n?
BNEZ  R3, Fertig          /Falls m<n steht in R1 das Ergebnis m modulo n
SUB   R1,R1,R2            /m soll solange um n reduziert werden, bis m<n
J     m_mod_n
Fertig:                   / die eigentliche Operation des Ups endet hier
                          /Alle Operationen ab hier nur Parametersicherung
SUBI  R30,R30,#16         /Reduziere Stackpointer, 4 x pop
SW    -4(R30),R1          /Speichere das Ergebnis auf dem Stack
LW    R1,0(R30)           /Hole gerettete Registerinhalte vom Stack
LW    R2,4(R30)
LW    R3,8(R30)
LW    R31,12(R30)         /Hole Rücksprungadresse vom Stack
                          /Alle Operationen bis hier nur Parametersicherung
JR    R31                 /Rücksprung zum aufrufenden Programm
```