Digitaltechnik & Rechnersysteme

MIPS Prozessor

Martin Kumm

Hochschule Fulda
University of Applied Sciences

AI | Angewandte Informatik

WiSe 2022/2023

Hinweis!

Nächste Woche: (kurzer) Ausblick + Wrapp-Up

Termin um offene Fragen aus Vorlesung, Aufgaben, alten Klausuren, etc. zu besprechen.

Kommen Sie hier mit konkreten Fragestellungen!

Zum Thema Assembler...

Aufgrund zahlreicher Rückfragen zum Thema Assembler:

Ihr "PC" (einschließlich Windows/Linux Laptops, Mac's (außer M1,M2)) unterstützt den **x86** Befehlssatz welcher um den **amd64** Befehlssatz erweitert wurde:

Intel® 64 and IA-32 Architectures Software Developer's Manual
(Achtung, 5060 Seiten!)

bzw. `https://www.felixcloutier.com/x86/`

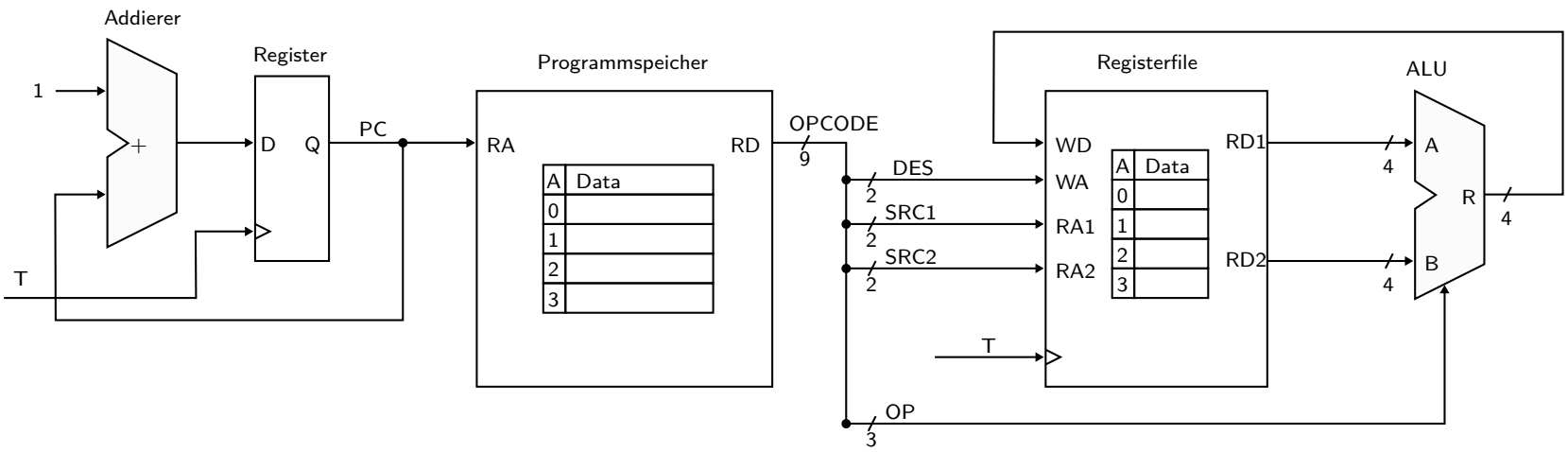
Für die moderne ARM8-basierte CPUs (wie Apple M1), siehe
(Achtung, 11.952 Seiten!)

Arm Architecture Reference Manual for A-profile architecture

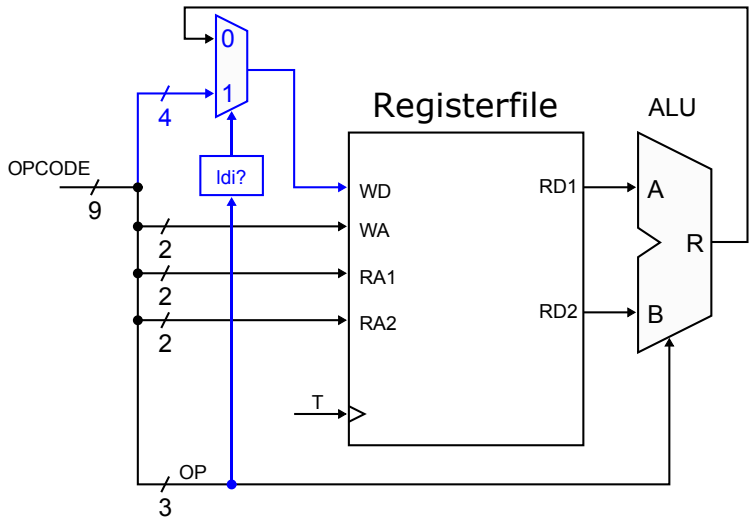
Was bisher geschah...

- Minimal-Computer
  - Bestehend aus Registerfile, ALU, Programmspeicher, Programmzähler
- Erweiterungen am Minimal-Computer
  - Laden von Registerinhalten (`ldi`)
  - Sprungbefehle (`jump`, `bne`)

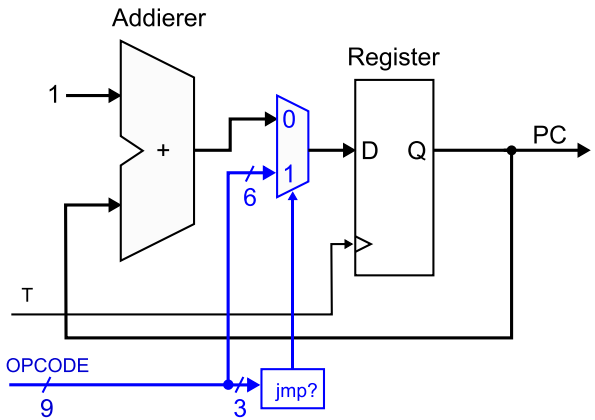
Minimal-Computer



LDI-Erweiterung:



JMP-Erweiterung:



Speicheranbindung

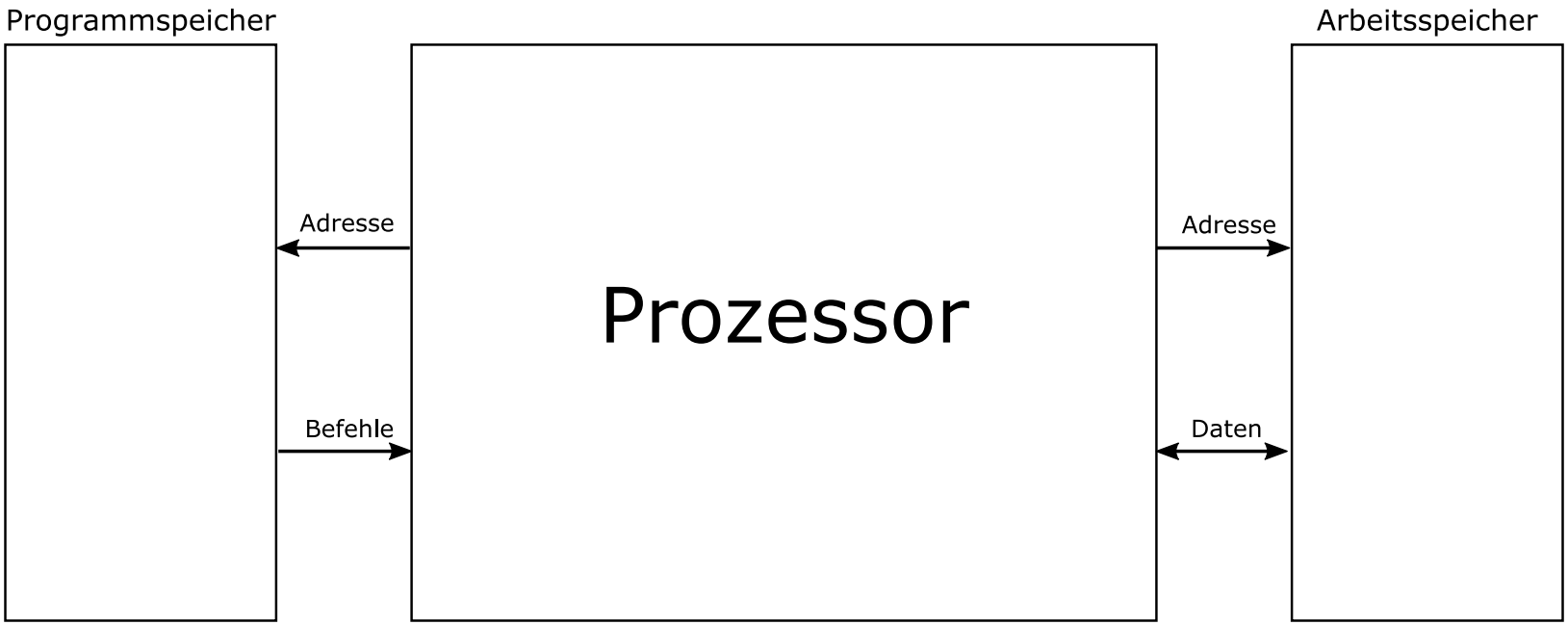
Registerfile ist von der Größe sehr limitiert (z.B. 32 Register beim MIPS Prozessor)

Problem ist hier wieder die Adressierung der Register im Programmcode
$\Rightarrow$ Größeres Registerfile $\rightarrow$ größere Befehlsworte

Hierzu wird Prozessor mit weiterem Speicher verbunden

Speicher kann separat sein (Arbeitsspeicher) oder gemeinsam mit dem Programmspeicher genutzt werden

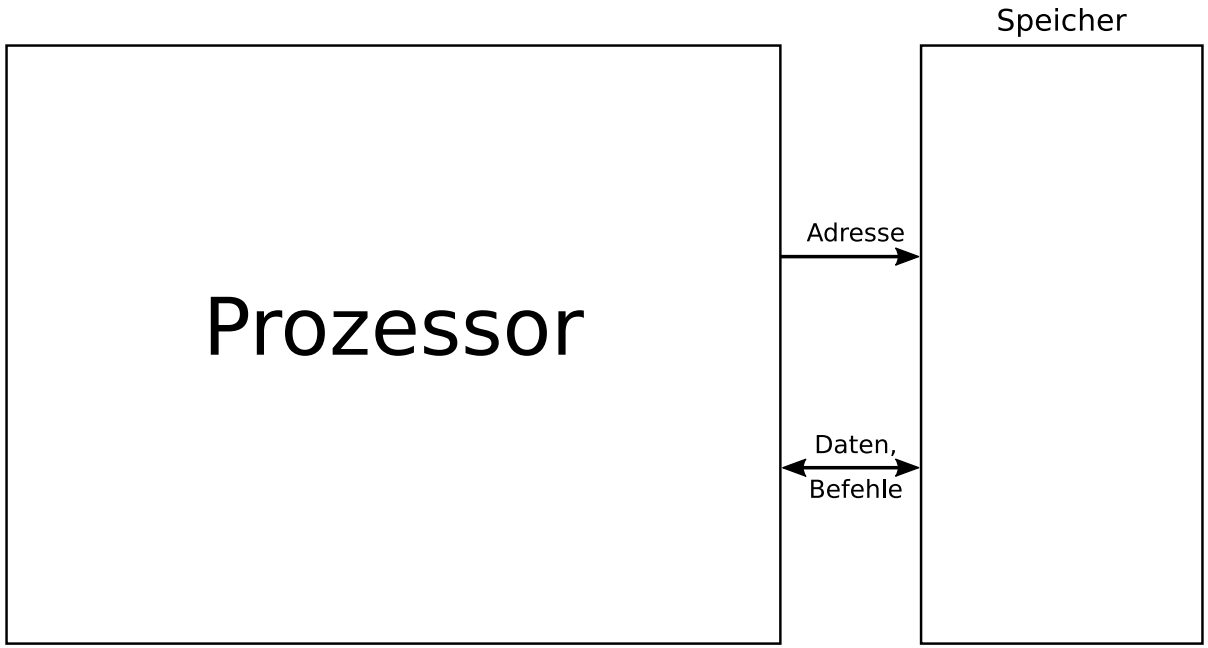
Harvard-Architektur



Trennung zwischen Arbeits- und Programmspeicher

Dadurch effizient da in einem Takt sowohl Programmcode als auch Daten gelesen/geschrieben werden können

Von-Neumann-Architektur



Arbeits- und Programmspeicher teilen sich den gleichen physikalischen Speicher

Sehr universell, Prozessor kann selbst eigenen Code manipulieren (allerdings sicherheitskritisch!)

Harvard vs. Von-Neumann-Architektur

Von-Neumann-Architektur:

- Vom Mathematiker John von Neumann entwickelt
- Ermöglichte erstmals die Programme ohne Hardwareänderung zu verändern (programmieren)!
- Heute immer noch in vielen modernen Computern zu finden, z.B. x86 (moderne PCs)

Harvard-Architektur:

- Von IBM und der Harvard-Universität entwickelt
- In jedem Takt können sowohl Daten als auch Programmcode geladen werden
- Daten- und Befehlswortbreite kann unterschiedlich sein
- Speichertyp (SRAM, Flash-Speicher, etc.) kann unterschiedlich sein
- Häufig in Mikrocontrollern oder eingebetteten Systemen zu finden (z.B. manche ARM CPUs wie z.B. ARM9)

Befehlserweiterung

Um den Speicher anzusprechen muss erneut der Befehlsvorrat erweitert werden.

Beispielbefehl: `lw DES,ADDR` (*load word*)

Lade Datenwort von (Arbeits-)speicheradresse `ADDR` nach Zielregister `DES`

Beispiel:
`lw r1,5` Lade Datenwort von Adresse 5 nach Zielregister `r1`

Analog dazu existieren Speicherbefehle wie z.B.: `sw SRC,ADDR` (*store word*)

Computer

Um aus einem Prozessor und Speicher einen echten Computer zu machen wird weitere Peripherie benötigt.

Um Ein- und Ausgaben zu erlauben sind Peripheriegeräte wie Tastatur, Maus/Trackpad, Bildschirm, Mikrofon/Kamera, Netzwerk etc. notwendig.

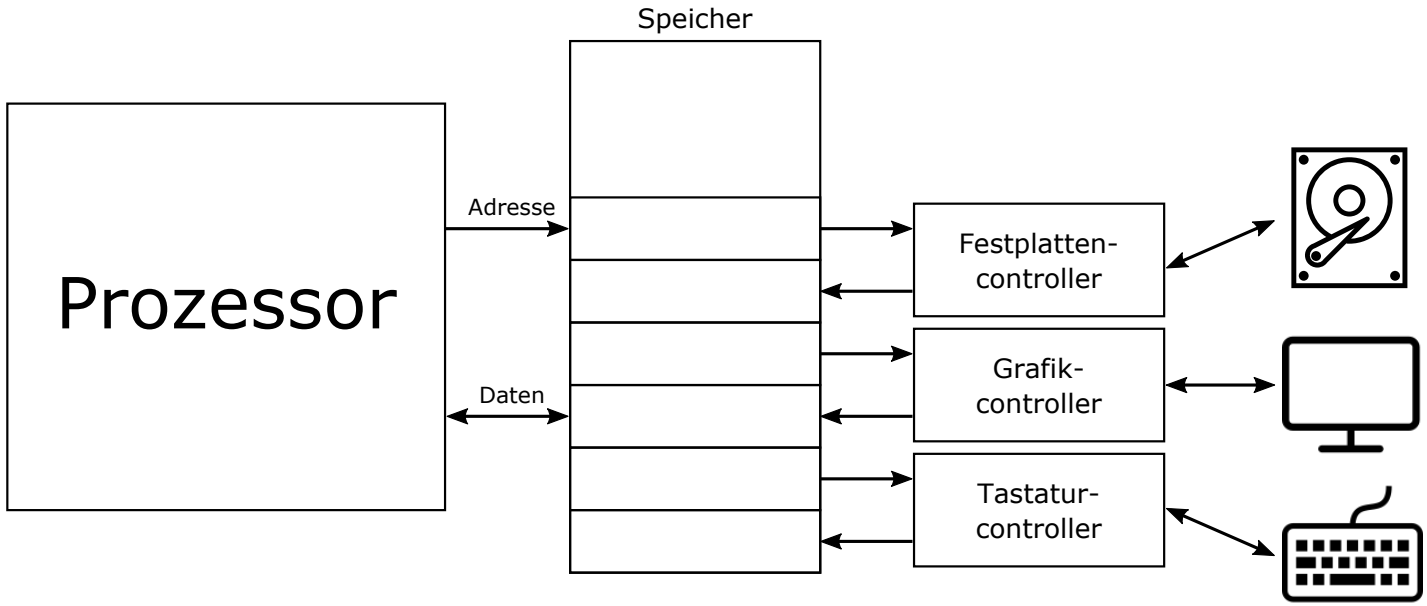
Bisher musste unser Programm in einem festen programmierten Speicher (*read only memory*, ROM) stehen.

Um dynamisch Programme zu laden sind Massenspeicher nötig, wie z.B. Festplattenlaufwerke (*hard disk drive*, HDD), oder *Solid State Discs* (SSD)

Peripherie

Die Peripherie wird über spezifische Controller angesprochen

Oft sind hier andere Schnittstellen beteiligt, z.B. *Universal Serial Bus* (USB), *Peripheral Component Interconnect* (PCI)



Die Ansteuerung der Peripherie-Controller erfolgt über eigene Register welche i.d.R auf den Arbeitsspeicher abgebildet werden.

Peripherie-Adressierung

D.h. spezielle Adressen im Speicher haben eine besondere Bedeutung und wirken sich direkt auf die Peripherie aus

Beispiele:

- Eine Speicherstelle repräsentiert die letzte Tastatureingabe
- Eine Speicherstelle repräsentiert den Pixel-Farbwert von Bildschirmkoordinate (x, y)
- Eine Speicherstelle repräsentiert die Speicheradresse der Festplatte, von der gelesen werden soll
- Eine Speicherstelle repräsentiert das Datenwort, dass von der Festplatte zuletzt gelesen wurde
- ...

MIPS Prozessor

Im Folgenden behandeln wir den MIPS32 32-Bit Prozessor von John L. Hennessy (Stanford)

MIPS: Microprocessor without interlocked pipeline stages („Mikroprozessor ohne verschränkte Pipeline-Stufen“)

Pipeline-Stufen = Register um kritischen Pfad zu reduzieren

MIPS ist ein Reduced Instruction Set Computer (RISC)

Im Gegensatz zum Complex Instruction Set Computer (CISC) kommt dieser mit deutlich einfacheren (und schnelleren) Befehlen aus.

MIPS Register

MIPS hat 32 Register, welche mit `$0` bis `$31` bezeichnet werden

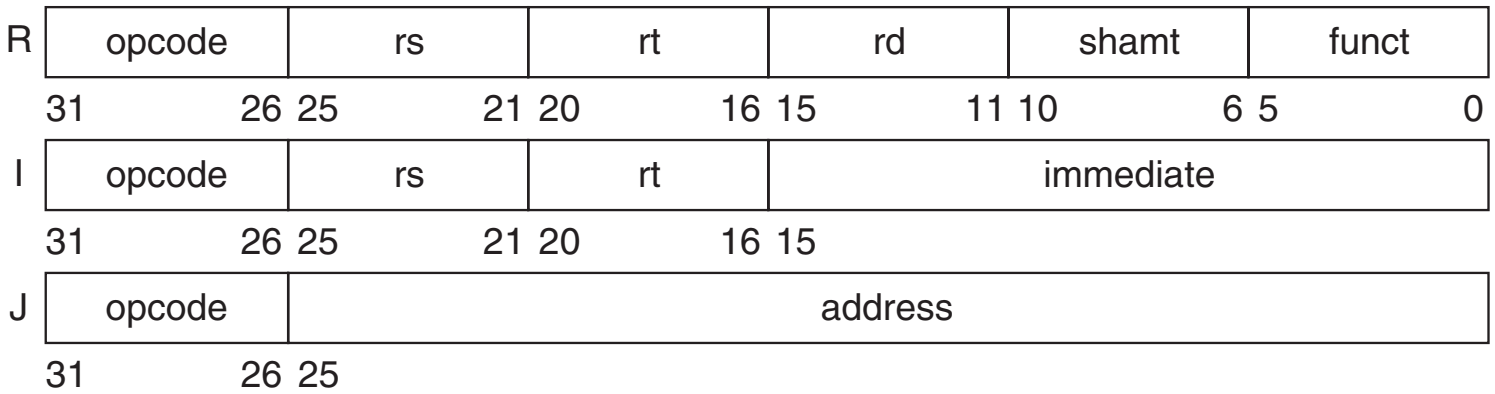
Register `$0` hat immer den Wert Null, `$1` bis `$31` sind generisch verwendbar

Es gibt alternative Namen/Bedeutung, welche nur im Zusammenhang mit Compilern eine Rolle spielen:

Nummer	Name	Alt. Name	Bedeutung
0	`$0`	`$zero`	Ist immer konstant 0
1	`$1`	`$at`	Reserviert für Assembler
2–3	`$2` – `$3`	`$v0`–`$v1`	Speicherung von Ergebnissen (*values*)
4–7	`$4` – `$7`	`$a0`–`$a3`	Speicherung von Argumenten (*arguments*)
8–15	`$8` – `$15`	`$t0`–`$t7`	Temporäre Variablen (*temporaries*), nicht gesichert
16–23	`$16` – `$23`	`$s0`–`$s7`	Gesicherte Variablen (*saved*)
24–25	`$24` – `$25`	`$t8`–`$t9`	Mehr temporäre Variablen
26–27	`$26` – `$27`	`$k0`–`$k1`	Reserviert für Betriebssystem
28	`$28`	`$gp`	*global pointer*
29	`$29`	`$sp`	*stack pointer*
30	`$30`	`$fp`	*frame pointer*
31	`$31`	`$ra`	*return address*

MIPS Befehlssatzarchitektur

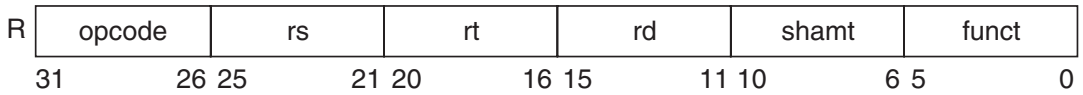
In der MIPS Befehlssatzarchitektur werden 3 Befehlsformate unterschieden, R-, I- und J-Befehle:



- R: Register, erlaubt 3 Register als Quelle/Ziel
- I: Immediate, erlaubt 16 Bit Konstante im Befehlswort
- J: Jump, maximaler Platz für Sprungadresse

MIPS Befehlssatzarchitektur

R-Befehlsformat:



- `opcode`: Befehlscode, bei R-Befehl immer 0
- `rs`: 1. *source register*, 1. Quellregister
- `rt`: 2. *source register*, 2. Quellregister
- `rd`: *destination register*, Zielregister
- `shamt`: *shift amount*, wird nur für Bitverschiebung benötigt
- `funct`: *function code*, spezifiziert Variante der Operation

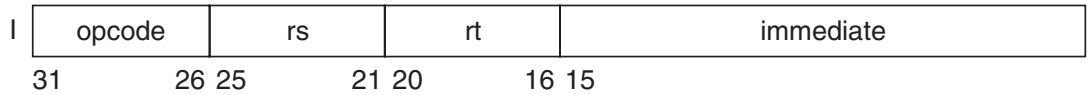
Beispiel: `add $1,$2,$3` (`opcode`=0, `shamt`=0, `funct`=32)

berechnet: `$1` ← `$2+$3`

(Achtung! Operanden-Reihenfolge anders als in Befehls-Kodierung)

MIPS Befehlssatzarchitektur

I-Befehlsformat:



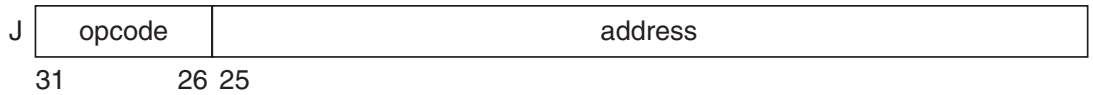
- `opcode`: Befehlscode
- `rs`: *source register*, Quellregister
- `rt`: *target register*, Zielregister
- `immediate`: *immediate constant or address* für Konstanten oder konstante Adressen

Beispiel: `addi $1,$2,42` (`opcode`=8, `immediate`=42)

berechnet: `$1` ← `$2+42`

MIPS Befehlssatzarchitektur

J-Befehlsformat:



- `opcode`: Befehlscode
- `address`: 26 Bit Adresse, adressiert ganze 32 Bit Wörter.

Da die Speicheradresse in der Regel Bytes adressiert ist
Byte-Adresse = 4×Wort-Adresse (Bit-Shift!)

26 Bit Wort-Adresse erlaubt somit Adressierung von
$4 \times 2^{26} = 2^{28} = 2^8 \times 2^{10} \times 2^{10}$ Bytes=256 MByte

Beispiel: `j 1000` (`opcode`=2, `address`=1000)

MIPS32 Befehlssatz (Auszug)

Arithmetische- und Logische Befehle:

Befehl	F.	Beispiel	Bedeutung
Add	R	`add $1,$2,$3`	$1 ← $2+$3
Subtract	R	`sub $1,$2,$3`	$1 ← $2−$3
Add immediate	I	`addi $1,$2,20`	$1 ← $2+20
AND	R	`and $1,$2,$3`	$1 ← $2∧$3
OR	R	`or $1,$2,$3`	$1 ← $2∨$3
NOR	R	`nor $1,$2,$3`	$1 ← $\overline{\$2 \vee \$3}$
AND immediate	I	`andi $1,$2,20`	$1 ← $2∧20
OR immediate	I	`ori $1,$2,20`	$1 ← $2∨20
Shift left logical	I	`sll $1,$2,10`	$1 ← $2<<10
Shift right logical	I	`srl $1,$2,10`	$1 ← $2>>10

F: (Befehls-)Format

MIPS32 Befehlssatz (Auszug)

AI Angewandte Informatik

Speicher-, Sprung- und Verzweigungs-Befehle:

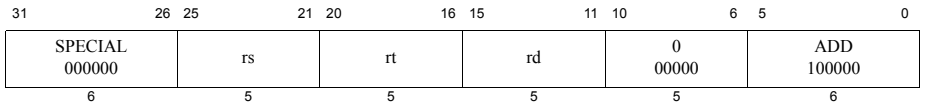
Befehl	F	Beispiel	Bedeutung
Load word	I	`lw $1,20($2)`	$1 ← Memory[$2 + 20]
Store word	I	`sw $1,20($2)`	Memory[$2 + 20] ← $1
Load half	I	`lh $1,20($2)`	($1 ← Memory[$2 + 20])$_{15:0}$
Store half	I	`sh $1,20($2)`	(Memory[$2 + 20] ← $1)$_{15:0}$
Load byte	I	`lb $1,20($2)`	($1 ← Memory[$2 + 20])$_{7:0}$
Store byte	I	`sb $1,20($2)`	(Memory[$2 + 20] ← $1)$_{7:0}$
jump	J	`j 2500`	PC ← PC + 10000
jump register	R	`jr $1`	PC ← $1
branch on equal	I	`beq $1,$2,25`	if($1 ≡ $2) PC←PC+100
branch on not equal	I	`bne $1,$2,25`	if($1 ≠ $2) PC←PC+100
set on less than	R	`slt $1,$2,$3`	if($2<$3) $1←1 else $1←0
set less than immediate	I	`slti $1,$2,20`	if($2<20) $1←1 else $1←0

MIPS32 Instruction Set Manual

Angewandte Informatik

Befehle beschrieben im „MIPS32 Instruction Set Manual“

ADD **Add Word**



Format: `ADD rd, rs, rt` **MIPS32**

Purpose: Add Word

To add 32-bit integers. If an overflow occurs, then trap.

Description: `GPR[rd] ← GPR[rs] + GPR[rt]`

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* to produce a 32-bit result.

- If the addition results in 32-bit 2’s complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs.
- If the addition does not overflow, the 32-bit result is placed into GPR *rd*.

Restrictions:

None

Operation:

```
temp ← (GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp
endif
```

Exceptions:

Integer Overflow

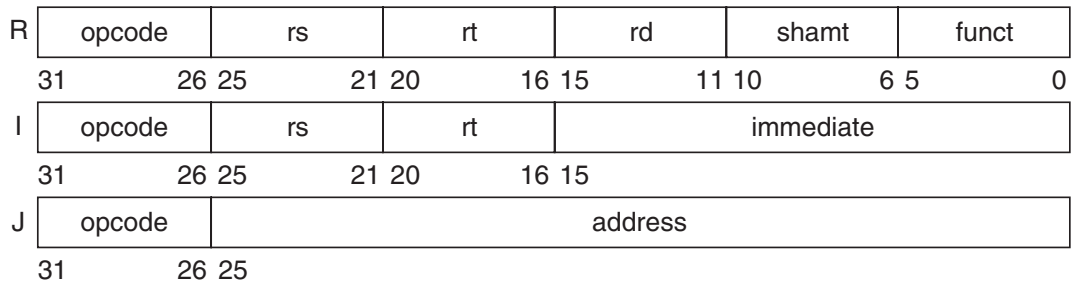
Programming Notes:

ADDU performs the same arithmetic operation but does not trap on overflow.

Vorlesungsaufgabe

Ermitteln Sie das Codewort der Anweisung

```
xori $8,$9,0xffff
```

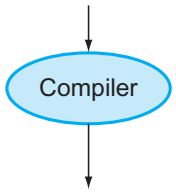


$(\text{opcode} = 14_{10})$

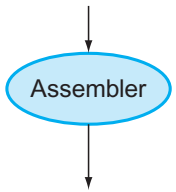
Von Hochsprache zu Maschinencode

AI Angewandte Informatik

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




```
swap:
      multi $2, $5,4
      add   $2, $4,$2
      lw    $15, 0($2)
      lw    $16, 4($2)
      sw    $16, 0($2)
      sw    $15, 4($2)
      jr    $31
```




```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
10001111000010010000000000000100
10101111000010010000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```

Arithmetische und Logische Befehle

Beispiel Addition (in C/Java-Syntax):

```
f = a + b;
```

Variablen `a`, `b`, `f` liegen in Registern `$1`, `$2`, `$3`.

MIPS32 Assembler:

```
add $3,$1,$2   #f = $3 = $1+$2 = a+b
```

Beispiel Bitweises-UND (in C/Java-Syntax):

```
f = a & b;
```

MIPS32 Assembler:

```
and $3,$1,$2   #f = $3 = $1&$2 = a&b
```

Arith. und Log. Befehle mit Konstanten

Beispiel Addition (in C/Java-Syntax):

```
f = a + 5;
```

Variablen `a` und `f` liegen in Registern `$1` und `$3`.

MIPS32 Assembler:

```
addi $3,$1,5   # f = $3 = $1+5 = a+5
```

Beispiel Bitweises-UND (in C/Java-Syntax):

```
f = a & 0x0f;
```

MIPS32 Assembler:

```
andi $3,$1,0x0f   # f = $3 = $1&0x0f = a&0x0f
```

Zuweisung einer Konstanten

Angenommen Variable `x` liegt in Register `$t0` und soll einer Konstante zugewiesen werden (in C/Java-Syntax):

```
x = 42;
```

Es existiert keine direkte Anweisung zur Zuweisung einer Konstanten (*immediate*) an ein Register.

Andere Operationen können hierzu verwendet werden, z.B.:

```
ori $1,$0,42        # $1 = 42|0 = 42
```

oder

```
addi $1,$0,42       # $1 = 42+0 = 42
```

Zuweisungen einer 32 Bit Konstanten

Immediate darf max. 16 Bit groß sein.

Was, wenn wir 32 Bit initialisieren wollen, z.B. `x = 0x12345678;`?

`lui` beschreibt die oberen 16 Bit, z.B.

```
lui $1,0x1234
```

ergibt

$1 ← 0x1234 << 16

Eine 32-Bit Zuweisung erfolgt somit über zwei Befehle:

```
lui  $1,0x1234     # $1 = 0x1234 << 16
addi $1,$1,0x5678  # $1 = $1 + 0x5678 = 0x12345678
```

Daten in Speicher schreiben

Angenommen `x` liegt im Speicher an Adresse `0x1234`.

Speichern ganzer Worte in den Speicher erfolgt über Befehl `sw`.

C-Code:

```
int *y;
y = 0x1234;  //direkte Zuweisung der Adresse, ↩
    geschieht normalerweise ueber malloc()
*y = 0xaffe;
```

MIPS32 Assembler:

```
addi $1,$0,0x1234 # load base address into $1
addi $2,$0,0xaffe # init the data
sw $2,0($1)       # store data to base address + 0
```

Daten aus Speicher laden

Angenommen `x` liegt im Speicher an Adresse `0x1234`.

Laden ganzer Worte aus dem Speicher erfolgt über Befehl `lw`.

C-Code:

```
int x;
int *y;
y = 0x1234; //direkte Zuweisung der Adresse, ↩
    geschieht normalerweise ueber malloc()
x = *y;
```

MIPS32 Assembler:

```
addi $1,$0,0x1234 # load base address into $1
lw $2,0($1)       # load data from base address + 0
```

if Statement (Beispiele)

C/Java:

```
if(i == 5) {
  //do something
}
else {
  //do something else
}
```

MIPS32 Assembler (`$1=i`):

```
addi $2,$0,5        # $2 = 5
beq $1,$2,do        # branch if $1==$2 (i==5)
# do something else
j end               # skip do case
do:
# do something
end:
```

if Statement (Beispiele)

C/Java:

```
if(i != 5) {
  //do something
}
else {
  //do something else
}
```

MIPS32 Assembler (`$1=i`):

```
addi $2,$0,5        # $2 = 5
bne $1,$2,do        # branch if $1!=$2 (i!=5)
# do something else
j end               # skip do case
do:
# do something
end:
```

if Statement (Beispiele)

C/Java:

```
if(i > 5) {
  //do something
}
else {
  //do something else
}
```

MIPS32 Assembler (`$1=i`):

```
addi $2,$0,5          # $2 = 5
slt $3,$2,$1          # set $3 when $2<$1 (5 < i)
beq $3,$0,else        # skip if part when test false
# do something
j end                 # skip else part
else:
# do something else
end:
```

for-Schleife

C/Java:

```
for(i=0; i != 10; i++)
{
  //loop body
}
```

MIPS32 Assembler (`$1=i`):

```
addi $1,$0,0      #$1=i=0
addi $2,$0,10     #$2=10 (loop limit)

loop:
#loop body
addi $1,$1,1      #i++
bne $1,$2,loop    #if i!=10 branch to loop
```