

Digitaltechnik & Rechnersysteme

MIPS Prozessor

Martin Kumm



WiSe 2022/2023

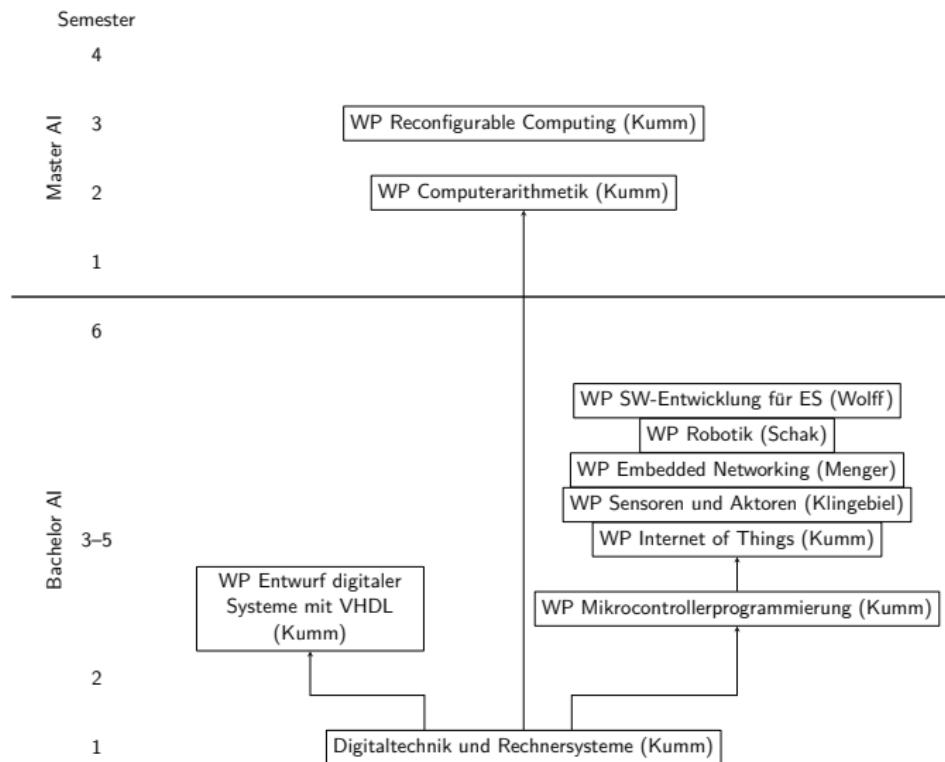
Was bisher geschah...

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

Ein paar Klausurhinweise...

- Termin: 27.02.2024, 15:00 Uhr, Raum 52.009, 46.107
- Hilfsmittel:
 - Handgeschriebene Formelsammlung (max.ein DINA4 Blatt, beidseitig)
 - Referenzblatt »MIPS32 Befehlssatz (Auszug) und Register« (wird mit Klausur ausgeteilt)
 - nicht-programmierbarer Taschenrechner
- Inhalten, die **nicht** geprüft werden:
 - MOSFET-Schaltungen
 - Hazards
 - MIPS Assembler Programmierung
- Am 13.02. 09:00–17:00 Uhr findet ein Intensiv-/Crashkurs für Digitaltechnik und Rechnersysteme statt
 - Primär für Wiederholer
 - Fragen zu Übungsaufgaben oder alten Klausuren

AI Schwerpunkt Embedded Systems



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 (<i>values</i>) |
| 4–7 | \$4 – \$7 | \$a0–\$a3 | Speicherung von Argumenten (<i>arguments</i>) |
| 8–15 | \$8 – \$15 | \$t0–\$t7 | Temporäre Variablen (<i>temporaries</i>), nicht gesichert |
| 16–23 | \$16 – \$23 | \$s0–\$s7 | Gesicherte Variablen (<i>saved</i>) |
| 24–25 | \$24 – \$25 | \$t8–\$t9 | Mehr temporäre Variablen |
| 26–27 | \$26 – \$27 | \$k0–\$k1 | Reserviert für Betriebssystem |
| 28 | \$28 | \$gp | <i>global pointer</i> |
| 29 | \$29 | \$sp | <i>stack pointer</i> |
| 30 | \$30 | \$fp | <i>frame pointer</i> |
| 31 | \$31 | \$ra | <i>return address</i> |

MIPS Befehlssatzarchitektur

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

| | | | | | | | |
|---|----------|-------|-------|---------|-----------|-------|--|
| R | opcode | rs | rt | rd | shamt | funct | |
| | 31 26 25 | 21 20 | 16 15 | 11 10 | 6 5 | 0 | |
| I | opcode | rs | rt | | immediate | | |
| | 31 26 25 | 21 20 | 16 15 | | | | |
| J | opcode | | | address | | | |
| | 31 26 25 | | | | | | |

- 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)

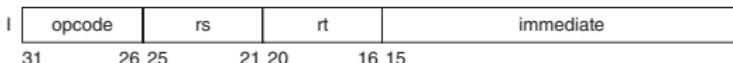
berechnet: $\$1 \leftarrow \$2 + \$3$

(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: $\$1 \leftarrow \$2 + 42$

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 = $4 \times$ 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 | <code>add \$1,\$2,\$3</code> | $\$1 \leftarrow \$2 + \$3$ |
| Subtract | R | <code>sub \$1,\$2,\$3</code> | $\$1 \leftarrow \$2 - \$3$ |
| Add immediate | I | <code>addi \$1,\$2,20</code> | $\$1 \leftarrow \$2 + 20$ |
| AND | R | <code>and \$1,\$2,\$3</code> | $\$1 \leftarrow \$2 \wedge \$3$ |
| OR | R | <code>or \$1,\$2,\$3</code> | $\$1 \leftarrow \$2 \vee \$3$ |
| NOR | R | <code>nor \$1,\$2,\$3</code> | $\$1 \leftarrow \neg (\$2 \vee \$3)$ |
| AND immediate | I | <code>andi \$1,\$2,20</code> | $\$1 \leftarrow \$2 \wedge 20$ |
| OR immediate | I | <code>ori \$1,\$2,20</code> | $\$1 \leftarrow \$2 \vee 20$ |
| Shift left logical | I | <code>sll \$1,\$2,10</code> | $\$1 \leftarrow \$2 << 10$ |
| Shift right logical | I | <code>srl \$1,\$2,10</code> | $\$1 \leftarrow \$2 >> 10$ |

F: (Befehls-)Format

MIPS32 Befehlssatz (Auszug)

Speicher-, Sprung- und Verzweigungs-Befehle

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

MIPS32 Instruction Set Manual



Befehle beschrieben im „MIPS32 Instruction Set Manual“

ADD Add Word

| 31 | 26 25 | 21 20 | 16 15 | 11 10 | 6 5 | 0 |
|-------------------|---------|---------|---------|--------|------------|---------------|
| SPECIAL 000000 | rs 6 | rt 5 | rd 5 | 0 5 | 00000 6 | ADD 100000 |

Format: ADD rd, rs, rt

MIPS32

Purpose: Add Word

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

Description: GPR[rd] \leftarrow 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  $\leftarrow$  (GPR[rs]31 | GPR[rs]31..0) + (GPR[rt]31 | GPR[rt]31..0)
if temp31  $\neq$  temp31 then
    SignalException(IntegerOverflow)
else
    GPR[rd]  $\leftarrow$  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

| R | opcode | rs | rt | rd | shamt | funct | |
|---|--------|---------|-------|-----------|-------|-------|---|
| | 31 | 26 25 | 21 20 | 16 15 | 11 10 | 6 5 | 0 |
| I | opcode | rs | rt | immediate | | | |
| | 31 | 26 25 | 21 20 | 16 15 | | | |
| J | opcode | address | | | | | |
| | 31 | 26 25 | | | | | |

(opcode = 14₁₀)

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.:

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

oder

```
ori $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

$\$1 \leftarrow 0x1234 \ll 16$

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 = 0xaff;
```

MIPS32 Assembler:

```
addi $1,$0,0x1234 # load base address into $1  
addi $2,$0,0xaff # 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
```

Wohin geht die Reise?

Wie werden Computer in 50 Jahren aussehen?
(Wenn Sie in Rente gehen (?))

»Prognosen sind schwierig, besonders wenn sie die Zukunft betreffen« (Urheber unklar)

»Der beste Weg, die Zukunft vorauszusagen, ist, sie selbst zu gestalten.« (Abraham Lincoln)

Wohin geht die Reise?

Vor über 60 Jahren, sahen Computer noch so aus:



Z23 Computer von 1959 (Quelle: [Z23 Beschreibung](#))

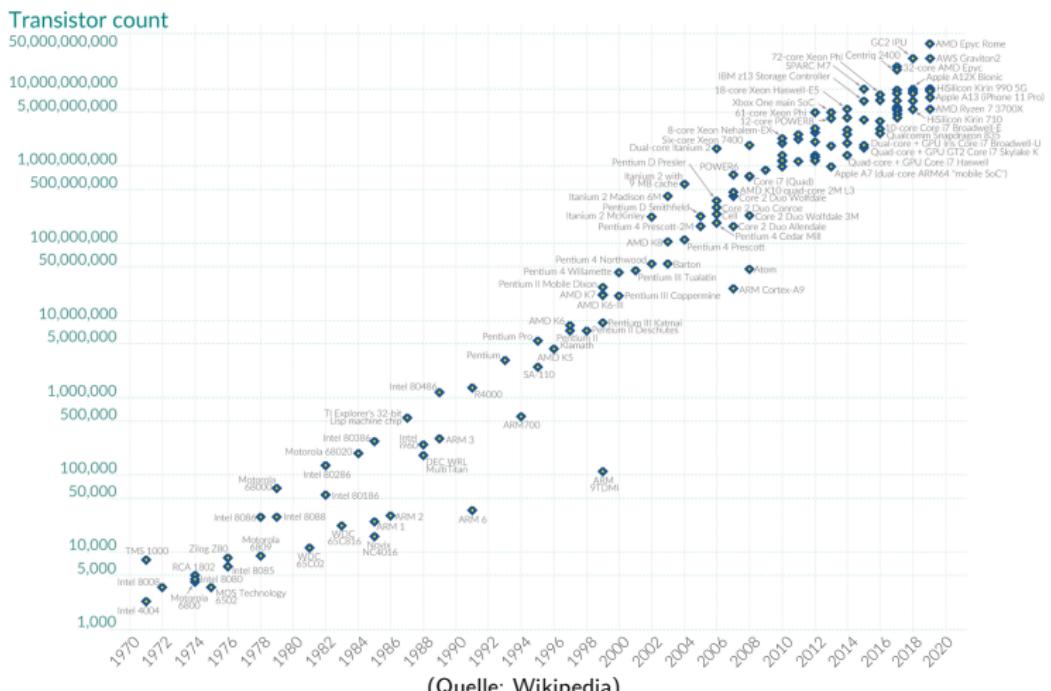
Er verwendete 2700 Transistoren und 7700 Dioden, lief mit 150 kHz (das sind 0,00015 GHz!) verbrauchte 4 kW und wog 1 t.

Kosten: 180.000 DM, heute: \approx 430.460€, ca. 41€ pro Halbleiter

Funktionsfähiges Modell im Zuse-Museum: zuse-museum-huenfeld.de

Mooresches Gesetz

Mooresches Gesetz (*moore's law*): Die Komplexität integrierter Schaltkreise verdoppelt sich alle 18 Monate



Viele Transistoren!

AMD Ryzen 9 3900X mit 273 mm²,
9,89 Milliarden Transistoren

36,2 Millionen Transistoren pro mm²

Preis: 439,- Euro (19.11.2020),
also 0,0000000443 €/T $\hat{=}$ 4µ¢/Transistor

Reis: 500 g, ~23 520 Körner (\rightarrow Zählwaage)

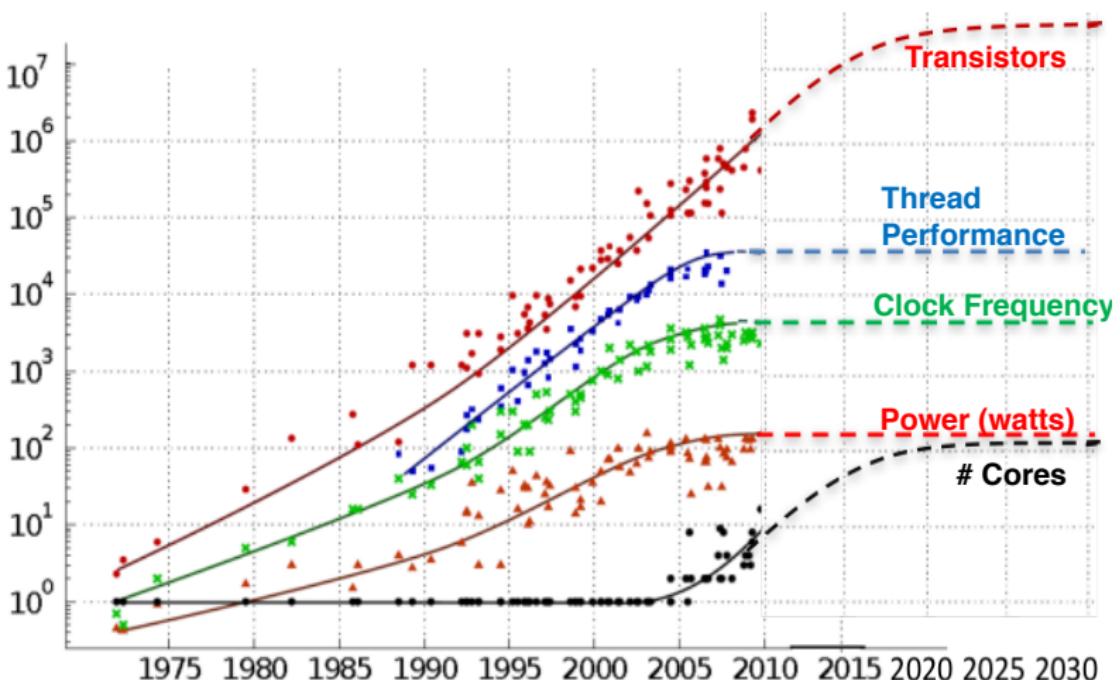
Preis: 1,49 Euro (05.02.2024),
also 0,00006335 €/Korn $\hat{=}$ 6,3mc

Man benötigt also 420 493 Packungen für 9,89 Milliarden Körner!

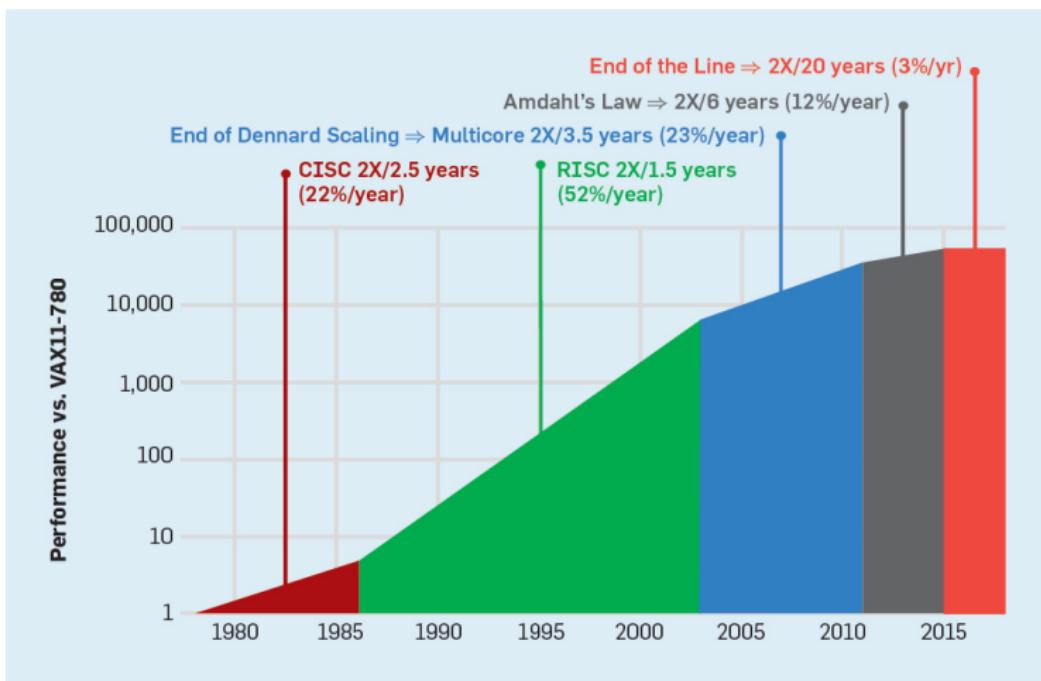
Für ein Korn Reis bekommt man 1420 Transistoren!



Skalierung der Chip-Technologie



The End of the Free Lunch



Quelle: Hennessy/Patterson, A New Golden Age for Computer Architecture, Communications of the ACM, 2019

Mooresches Gesetz

Das Mooresche Gesetz war die letzten 50 Jahre eine selbsterfüllende Prophezeiung

Chip-Hersteller haben es sich zum Ziel gemacht das Mooresche Gesetz einzuhalten

In dieser Zeit hat sich die Transistorgröße von 50000 nm auf 3 nm reduziert, der Durchmesser eines Silizium-Atoms beträgt (je nach Definition) ca. 0.17 nm

Das Mooresche Gesetz hat daher eine natürliche Grenze und wird von vielen totgesagt

Die Power-Wall



Die Verkleinerung der Technologie erlaubt mehr Transistoren pro Fläche, mehr Geschwindigkeit und geringere Schaltspannung.

Durch die Erhöhung der Transistoren bei gleichzeitiger Reduktion der Spannung blieb die Leistungsdichte konstant.

Dieser Effekt wurde durch Dennard analysiert und nach ihm als *Dennard scaling* benannt.

Seit ca. 2006 gilt dieses „Gesetz“ nicht mehr, da in kleineren Technologien höhere Leckströme entstehen.

Seit dem stagnieren CPU-Taktfrequenzen, maßgeblich ist nun die *Thermal Design Power*

Dies wird oft als Power-Wall bezeichnet, da diese „Wand“ (momentan) nicht übersprungen werden kann.

Alternative Rechenkonzepte



Das Stagnieren der Taktfrequenz erfordert neue Konzepte um die Steigerung der Rechenkapazität aufrecht zu erhalten

Der momentane Trend: Mehr Rechenkerne

Aktuell (2023): Apple M3 Ultra: 60/76 CPU/GPU pro Chip

Dark Silicon: Viele spezialisierte effiziente Recheneinheiten die nur in bestimmten Anwendungen benötigt werden. Ansonsten sind sie abgeschaltet (*dark*)

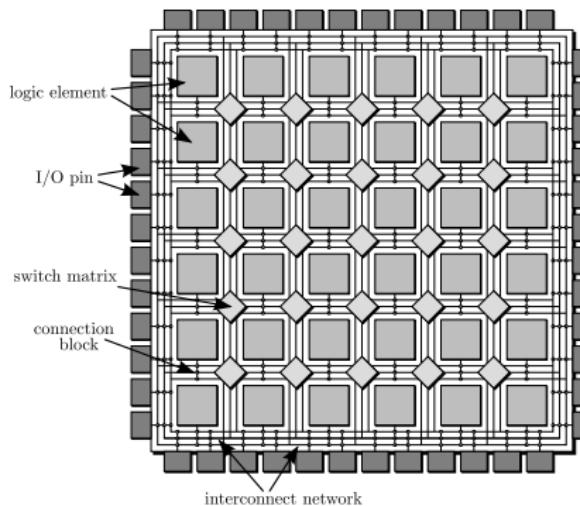
Generalisierte Grafikprozessoren: *General Purpose Computation on Graphics Processing Unit (GPGPU)*

Single instruction, multiple data (SIMD): Viele Berechnungen erfolgen parallel auf Arrays/Matrizen von Daten

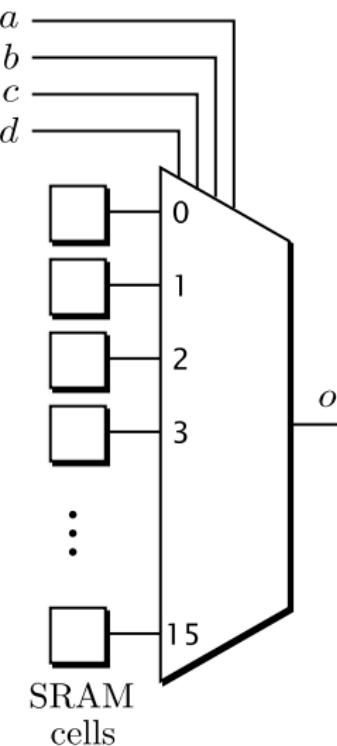
Rekonfigurierbares Rechnen

Rekonfigurierbares Rechnen (*reconfigurable computing*): Die Hardwarestruktur wird selbst programmierbar gemacht.

Konfiguration der Rechenarchitektur zur Laufzeit möglich durch
Field Programmable Gate Arrays (FPGAs)



Rekonfigurierbares Rechnen



- Ein Logikelement (*logic element*) kann beliebige Boolesche-Wahrheitstabellen realisieren
 - Realisierung durch MUX und SRAM-Zellen
 - SRAM-Zellen programmierbar
 - Durch programmierbares Routing-Netzwerk, sehr komplexe Schaltungen realisierbar (z.B. Multi-CPU Systeme)

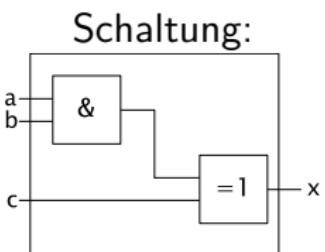
Entwurf mit VHDL



VHDL-Beschreibung:

```
entity beispiel is
  port(
    a: in  std_logic;
    b: in  std_logic;
    c: in  std_logic;
    x: out std_logic
  );
end entity;

architecture verhalten of beispiel is
begin
  process(a,b,c)
  begin
    x <= (a and b) xor c;
  end process;
end architecture;
```



Schaltung:

⇒ WP-Veranstaltung »Entwurf digitaler Systeme mit VHDL«

Quantencomputer



Alternative Quantencomputer?

Quantencomputer basieren auf den Gesetzen der Quantenmechanik

Die kleinste Informationseinheit bildet ein Quantenbit (Qubit)

Mehrere Qubits bilden ein Quantenregister und sind miteinander *verschränkt* (quantenphysikalisches Phänomen)

Der Zustand eines Quantenregisters mit n Qubits repräsentiert die Überlagerung aller möglichen 2^n Zustände (im Gegensatz zu einem bei gewöhnlichen Registern)

Dies verspricht mehr Information pro Qubit und somit eine potenziell größere Rechenleistung

Quantencomputer



Quantenüberlegenheit (engl. *Quantum Supremacy*) erstmals 2019 von Google gezeigt

Hierbei wurde von (sehr spezielles) Problem in 200 Sekunden gelöst für das ein Supercomputer 10.000 Jahre benötigen würde



Googles Quantencomputer
»Sycamore« (53-Qubit)



IBMs Quantencomputer
»Q« (53-Qubit)

Bilder erinnern an Zuses Z23 von vor 60 Jahren...

The End



Angewandte Informatik

Machen Sie es gut!



Und viel Erfolg bei der Klausur!