



Herzlich willkommen
zum zweiten ERA
Tutorium des
Semesters!



QR-Codes



 SCAN ME

Montag 10.00



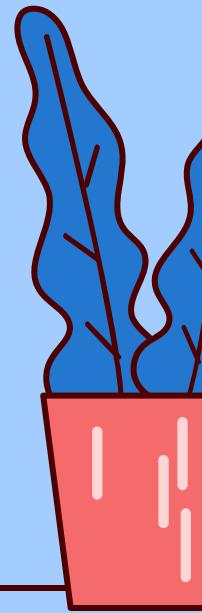
 SCAN ME

Montag 14.00



 SCAN ME

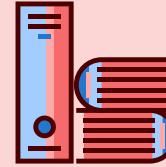
Mein Web





Zusammenfassung

Dies ist kein Ersatz für die VL!
Keine Garantie auf Korrektheit.
Nur eine Erinnerung an die
wichtigsten Themen der Woche.



Das Blatt gem. lösen

Es ist nicht nötig, das Blatt
vorher zu lösen. Du
bekommst für jede Übung
etwas Zeit, bevor wir sie
gemeinsam lösen.

Spaß mit Assembly

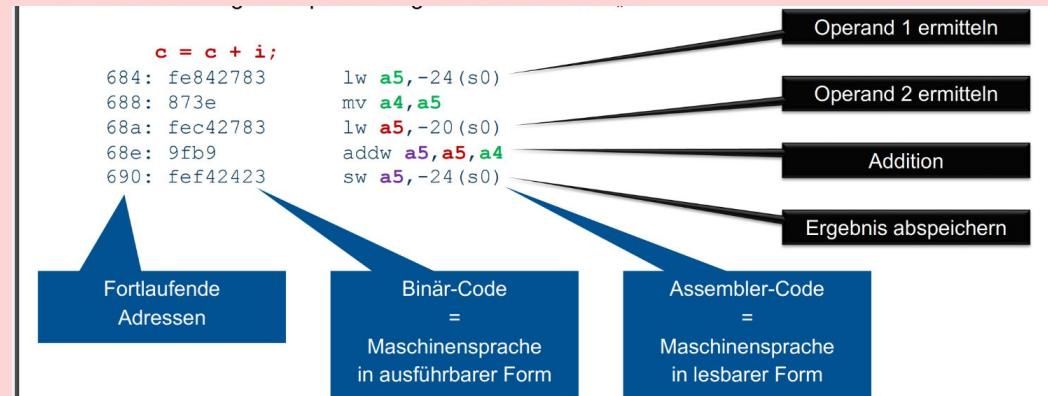
Assemblersprache

Maschinen-spezifische Programmiersprache

- Niedrigste Abstraktionsbene
- Primitive Befehle
 - Elementare Berechnungsschritte
 - Keine komplexen Datenstrukturen
- 1:1 Übersetzung in Binärkode
- Manuelle Verwaltung von Ressourcen
 - Speicherelemente
 - Datenübertragungen

Oft nahe an der Implementierung

- Einzelne Befehle direkt implementiert
- Kodierung passend zur internen Architektur
- Festgelegte "Breite"
 - Daten und/oder Instruktionen
- Limitierungen möglich
- Oft spezifische Erweiterungen



RISC-V

RISC-V

- Frei und offen
 - Jeder kann RISC-V ISA nutzen
 - Keine Lizenzkosten
 - Mehr Wettbewerb
- Einfach und modular
 - 5. RISC Architektur von UC Berkeley
 - Viel einfacher als ARM und x86
 - Erweiterbar: Custom-Instruktionen
- Für Cloud und Edge
 - Von Mikrocontrollern bis zu Supercomputern



- Security
 - Entwurf von eigenen Cores
 - Offene Cores
 - ➔ keine Geheimnisse
- Community Unterstützung und Stabilität
 - RISC-V Foundation „besitzt“ RISC-V Spezifikation
 - <https://riscv.org>
 - <https://riscv.org/specifications/>

1

Komplexität der ISA: RISC vs. CISC

CISC: Complex Instruction Set Computer

- Komfortabel und relativ mächtig
- Vorteil: einfache Programmierbarkeit,
- Realisiert durch Mikroprogramme
- Nachteile: komplexe (langsame) Implementierung insbesondere Dekodierung
 - Evtl. viele Funktionen ungenutzt

RISC: Reduced Instruction Set Computer

- Minimalistisch, auf das absolut Notwendige beschränkt
- Vorteil: einfache, effiziente, schnelle Implementierung
- Realisiert durch feste Verdrahtung
- Nachteil: schwierigere Programmierbarkeit
 - Aber: das erledigt der Compiler (?)

Heute: Oft keine saubere Trennung mehr möglich

- CISC Architekturen haben einen RISC Kern
- Erweiterungen in RISC Architekturen oft CISC-like

Arithmetische Operationen

Befehlssatz = Assemblersprache

Kern einer jeden „Instruction Set Architecture“ (ISA)

Drei Hauptklassen von Befehlen:

- Arithmetische und logische Operationen
- Datentransfer
- Steuerung des Programmablaufs = Sprünge und Unterprogrammaufrufe

- ADD, SUB, NEG
 - identische Befehle für Zahlen mit und ohne Vorzeichen (Zweierkomplement!)
- In manchen ISAs: Spezialbefehle
 - CMP (Vergleich), INC (Hochzählen), DEC (Herunterzählen), ...
- MUL, DIV
 - i.a. je eine Variante für Zahlen mit und ohne Vorzeichen
 - Ergebnis oft in zwei Registern
 - Multiplikation: doppelt großes Ergebnis
 - Division: Ergebnis und Divisionsrest
 - Längere Ausführungszeiten

hier
gespeicherte
↓
bleiben unverändert!

Addition: **add a, b, c** $a = b + c$
□ add:
□ b, c:
□ a:
Mnemonic legt auszuführende Operation fest
Quelloperanden (source operands) auf der die Operation ausgeführt wird
Zieloperand (destination)

Subtraktion: **sub a, b, c** $a = b - c$
□ sub:
□ b, c:
□ a:
Mnemonic legt auszuführende Operation fest
Quelloperanden (source operands) auf der die Operation ausgeführt wird
Zieloperand (destination)

Komplexere Operationen

$a = b + c - d;$

add t, b, c # t = b + c
sub a, t, d # a = t - d

Nur 12 Bits!!!

Addition: **addi a, b, <imm>** $a = b + <imm>$
□ addi:
□ b:
□ imm:
□ a:
Mnemonic legt auszuführende Operation fest
Quelloperanden (source operands) auf der die Operation ausgeführt wird
Immediate/Konstante
Zieloperand (destination)

Eigenschaften

- Gibt es nur für bestimmte Befehle
 - Z.B. kein „**subi**“ Befehl
 - Nutzung negativer Konstanten („sign extension“ wird angewendet)
- Konstanten auf 12bit beschränkt
- Begrenzte Länge einer Instruktion

Nützliche Operationen

Erzeugung von 32-BIT Konstanten

Größere Konstanten müssen in zwei Schritten erzeugt werden

- load upper immediate (`lui`) und `addi`
- `lui`: lädt eine Konstante in die oberen 20 Bits des Zielregisters und 0 in die unteren 12 Bits
- `addi`: addiert die unteren 12 Bits dazu

C Code

```
int a = 0xFEDC8765;
```

RISC-V Assembler

```
# s0 = a  
lui s0, 0xFEDC8    # s0 = 0xFEDC8000  
addi s0, s0, 0x765
```

Wichtig:

Bei `addi` 12-Bit-Immediate Vorzeichenerweiterung (sign extension) verwenden

- Zusätzliche Bits mit Vorzeichenbit füllen
- Ersetzt die unteren 12 Bits und die müssen zu den oberen 20 Bits passen

Warum haben wir kein slai?



`addi a0, a0, 0xF00` → $a0 - a0 + 0xFFFFFFF00$

`>>` = logischer Shift (Null Bits nachgeführt)

`>>>` = arithmetischen Shift (erhält Vorzeichen, 0 oder 1 nachgeführt)

Beispiel: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0111_{2K} = -25$

`sra t0, t0, 4` ⇒

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{2K} = -2$

<code>sll rd, rs1, rs2</code>	shift left logical	$rd := rs1 \ll rs2_{4:0}$
<code>slli rd, rs1, uimm</code>	shift left logical immediate	$rd := rs1 \ll uimm$
<code>srl rd, rs1, rs2</code>	shift right logical	$rd := rs1 \gg rs2_{4:0}$
<code>srli rd, rs1, uimm</code>	shift right logical immediate	$rd := rs1 \gg uimm$
<code>sra rd, rs1, rs2</code>	shift right arithmetic	$rd := rs1 \ggg rs2_{4:0}$
<code>srai rd, rs1, uimm</code>	shift right arithmetic immediate	$rd := rs1 \ggg uimm$

Call the convention a calling convention .

always
code
accordingly!
(oder
Punktabzug)

Aufrufkonvention

Soviel wie möglich via Register

- Eingabedaten
- Rückgabewert

Gezielte Aufgaben

- a0-a7: Eingaben
- a0-a1: Ergebnisse
- ra: Rücksprungadresse

Caller-saved

- Aufrufer/Caller muss Werte selbst sichern, dürfen von Callee verändert werden

Callee-saved

- Aufgerufene Funktion darf Werte nicht verändern oder muss sie wieder herstellen

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

From: <https://riscv.org/technical/specifications/>

Load & Store

Datenspeicherung

Operanden nur aus Registern und Konstanten

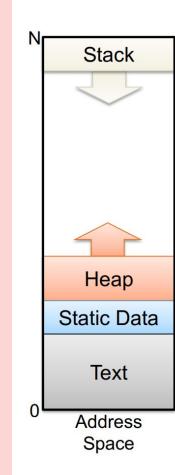
- Register sind schnell, aber auch sehr begrenzt
- Nur als Zwischenspeicher gedacht

Zusätzlich: Hauptspeicher

- Große Speicherkapazität
- Ablage aller zu bearbeitenden Daten
- Nicht zum Verwechseln mit „Hintergrundspeicher“
 - Hauptspeicher meist flüchtig
 - Daten müssen in den Hauptspeicher geladen werden
- Meist als „Random Access Memory“ oder RAM
 - Man kann auf jedes gespeicherte Element zugreifen
 - Kein „vorspulen“ / „zurückspulen“

Konsequenz

- Daten müssen übertragen werden
- „Load“ und „Store“ Befehle → RISC-V ist eine „Load and Store Architecture“
- Andere ISAs erlauben Operanden direkt aus dem Hauptspeicher



wenn die Peile
sich treffen ...
KABOOM

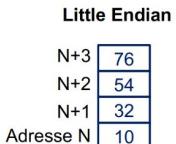
Bytes. Bytes everywhere.

Byte-Reihenfolge

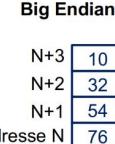
Beispiel: Ablage des Worts

24	16	8	0
76	54	32	10

 =0x76543210



Beispiel: Intel IA-32, typisch für PC



Beispiel: SPARC, Motorola, MIPS64

- Heute: Little Endian dominiert (x86 / x86_64)
- Die meisten anderen Architekturen bieten Wechselmöglichkeit (auch RISC-V)

Speicher/Zellenbreite

Speicherzellen sind üblicherweise mehrere Byte breit

Entspricht der Verarbeitungsgröße

- 16bit vs. 32bit vs. 64bit vs. 128bit Rechner
- Anzahl der Datenleitungen
- Warum? Effizienterer Transfer, Einfachere Logik

In der Praxis sind Speicher byte-adressierbar

- Adresse eines Datums bezieht sich immer auf Bytengrenzen
- Unabhängig vom Datenformat
- Mehrere Adressen zeigen in eine Zelle

Sprünge (aka cheat code for if/for)

Unbedingter Sprung

```
j      target      # Sprunge an Adresse mit label "target"
srai   s1, s1, 2    # nicht ausgeführt
addi   s1, s1, 1    # nicht ausgeführt
sub    s1, s1, s0   # nicht ausgeführt

target:
add   s1, s1, s0   # s1 = 1 + 4 = 5
```

Bedingter Sprung

Option 1

```
addi s0, zero, 4
addi s1, zero, 1
slli s1, s1, 2
beq s0, s1, target
addi s1, s1, 1
sub s1, s1, s0

# Add Immediate / s0 = 0 + 4 = 4
# Add Immediate / s1 = 0 + 1 = 1
# Shift Left / s1 = 1 << 2 = 4
# s0 = s1: Sprung wird ausgeführt (eq)
# nicht ausgeführt
# nicht ausgeführt
```

Option 2

```
addi s0, zero, 4
addi s1, zero, 1
slli s1, s1, 2
bne s0, s1, target
addi s1, s1, 1
sub s1, s1, s0

# Add Immediate / s0 = 0 + 4 = 4
# Add Immediate / s1 = 0 + 1 = 1
# Shift Left / s1 = 1 << 2 = 4
# s0 = s1: Sprung nicht wird ausgeführt (ne)
# Addition: s1 = 4 + 1 = 5
# Subtraktion: s1 = 5 - 4 = 1
```

Sprungziel

```
target:          # label
add s1, s1, s0   # s1 = 4 + 4 = 8 (beq)      s1 = 1 + 4 = 5 (bne)
```

Learn to use stacks so you don't stack shelves*

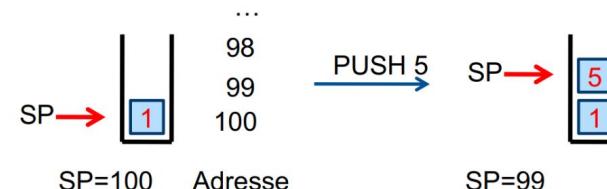


Implementierung von Kellern/Stacks

Keller für Programmausführung werden oft im Speicher so abgelegt, dass sie „nach unten“ wachsen: Richtung kleinere Adressen

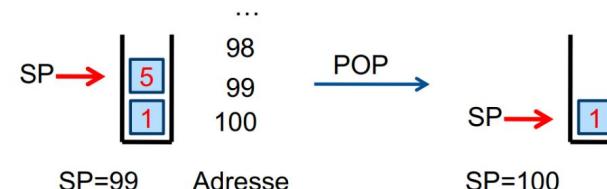
PUSH Funktionalität

- SP erniedrigen und in Adresse SP schreiben
- Prädekrement
- Viele ISAs haben dafür spezielle Befehle
- RISC-V
 - Anpassung von SP mit arithmetischen Befehlen
 - Schreiben mit „sd“ Anweisungen



POP Funktionalität

- Von Adresse SP lesen und SP erhöhen
- Postinkrement
- Viele ISAs haben dafür spezielle Befehle
- RISC-V
 - Lesen mit „ld“ Anweisungen
 - Anpassung von SP mit arithmetischen Befehlen



* for legal reasons this is a joke, respect to everyone who stacks shelves

- a) Machen Sie sich nochmals mit den Registern und deren üblicher Verwendung vertraut.
Die Register sind in Tabelle 1 abgebildet.

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0–2	x5–7	Temporary registers
s0/fp	x8	Saved register/Frame pointer
s1	x9	Saved register
a0–1	x10–11	Function arguments/Return values
a2–7	x12–17	Function arguments
s2–11	x18–27	Saved registers
t3–6	x28–31	Temporary registers

Abbildung 1: RISC-V 32-Bit Register

$lui \quad t1, \quad 0xF0000$

$t1 = 0x F0000\textcolor{red}{000}$

- b) Alle für uns relevanten Befehle sind in Tabelle 2 gelistet. Nutzen Sie möglichst wenige Befehle, um folgende Aufgaben zu bewältigen.

- Laden Sie in das Register a0 den Wert 0 (sog. Nullen). *es gibt mehr
als eine Möglichkeit
dafür*
- Laden Sie die Konstante 0xF0000000 in das Register t1.
- Laden Sie die Konstante 0xABAD1DEA in das Register a3. Verwenden Sie keine Pseudobefehle.

↳ Trick Question

$0xABAD1IDEA \rightarrow a3$

{ }
lui

addi

1010.1011.1010.1101.0001.1101.1110.1010

1010.1011.1010.1101.0010.0000.0000.0000
+ 1111.1111.1111.1111.1111.1101.1110.1010

11010.1011.1010.1101.0001.1101.1110.1010

lui t3, 0xABAD2
addi t2, -534

A B A D D E A
 |
 ?

- Laden Sie in das Register a0 den Wert 0 (sog. Nullen).

```
addi a0, zero, 0
```

- Laden Sie die Konstante 0xF0000000 in das Register t1.

```
lui t1, 0xF0000
```

- Laden Sie die Konstante 0xABAD1DEA in das Register a3. Verwenden Sie keine Pseudobefehle. $0xABAD1DEA = 1010.1011.1010.1101.0001.1101.1110.1010_2$

$$0xDEA = 1101.1110.1010_2 = -0010.0001.0110_2 = -(2 + 4 + 16 + 512) = -534$$

Problem: addi addiert nicht 0xDEA sondern 0xFFFF.FDEA (sign extension)

$$\begin{aligned}0xABAD1DEA &\stackrel{!}{=} (?) + 0xFFFF.FDEA \\ \Rightarrow (?) &= 0xABAD1DEA - 0xFFFF.FDEA \\ \Rightarrow (?) &= 0xABAD1DEA - (-0x00000216) \\ \Rightarrow (?) &= 0xABAD1DEA + 0x00000216 \\ \Rightarrow (?) &= 0xABAD2000\end{aligned}$$

Lösung:

```
lui a3, 0xABAD2  
addi a3, a3, -534
```

Probier's in qtrvsim!

if you start with an empty template,
write " .globl _start " before copying your code
.text
.globl _start:
_start:

- a) Welchen Wert hat das Register a0 nach der Ausführung des folgenden Programms:

main:

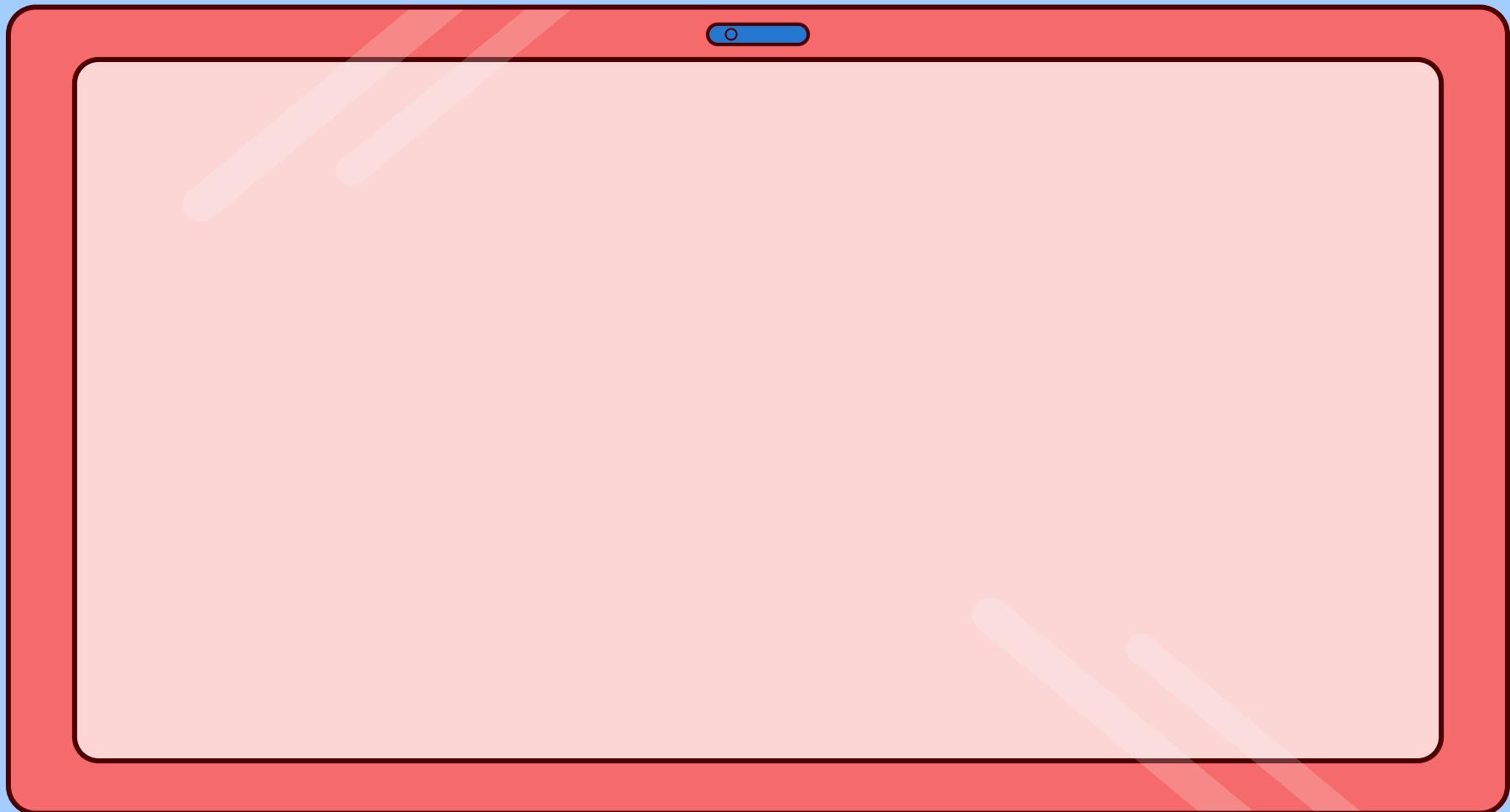
```
addi a0, zero, -534
lui a0, 0
srl a0, a0, 10
add a0, a0, a0
```

ebreak

warum dieser Wert?

b) Übersetzen Sie die folgenden Terme in RISC-V Assembler. Verwenden Sie dabei nur Befehle aus Tabelle 2.

- $a_0 := a_0 - a_1$
- $a_0 := a_0 + 2^{10}$
- $a_0 := a_0 + 2^{11}$
- $a_0 := a_0 \cdot 2 + 123$ (vorzeichenlose Multiplikation)
- $a_0 := a_0 \cdot 5$ (vorzeichenlose Multiplikation)



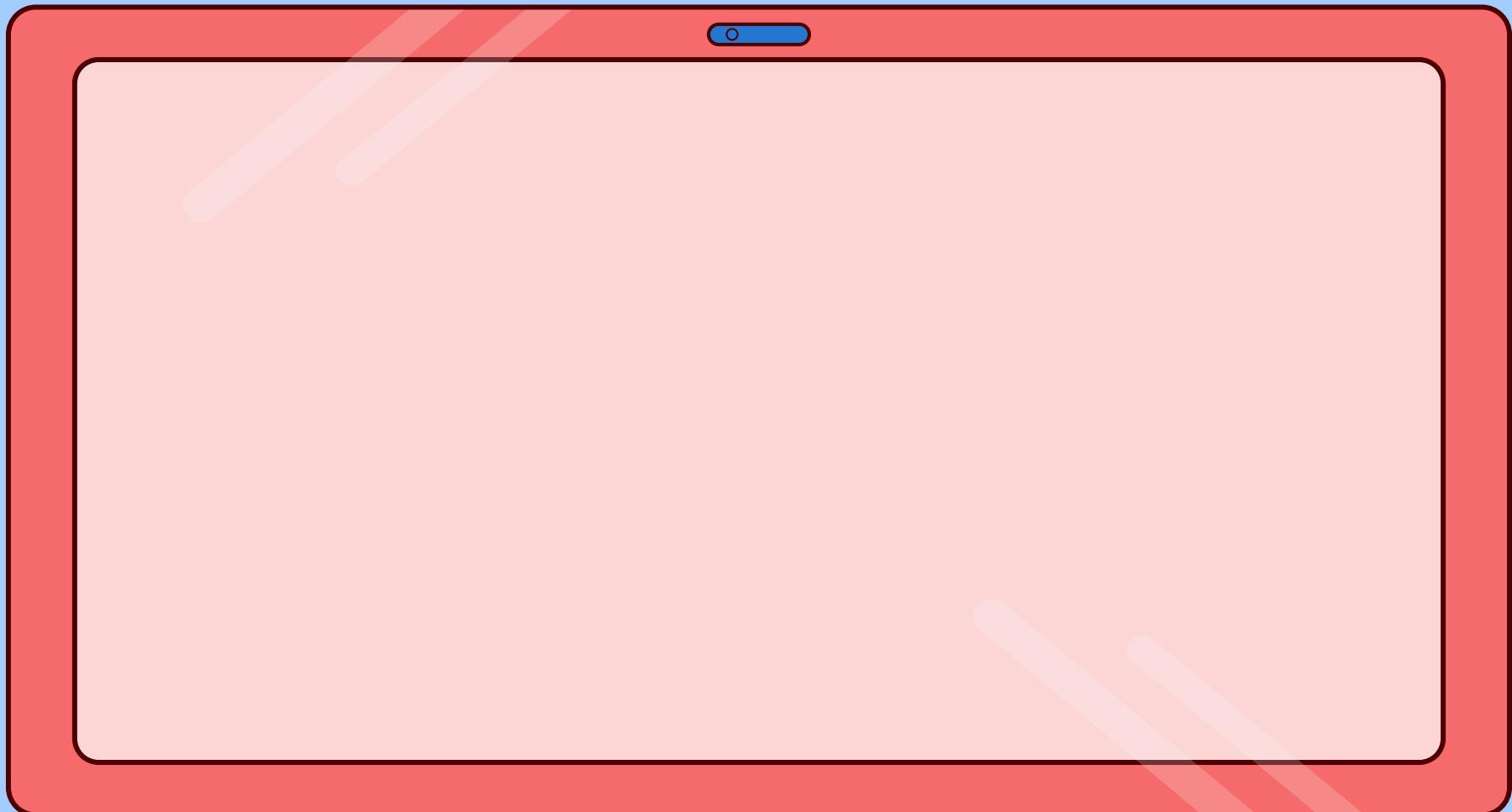
```
subtask1:  
sub a0, a0, a1  
  
subtask2:  
addi a0, a0, 1024  
  
subtask3:  
addi a1, zero, 1  
slli a1, a1, 11  
add a0, a0, a1
```



```
subtask4:  
slli a0, a0, 1  
addi a0, a0, 123  
  
subtask5:  
slli a1, a0, 2  
add a0, a0, a1
```

Setzen Sie die folgenden Aufgaben um indem Sie Bitmasken und -hacks verwenden.

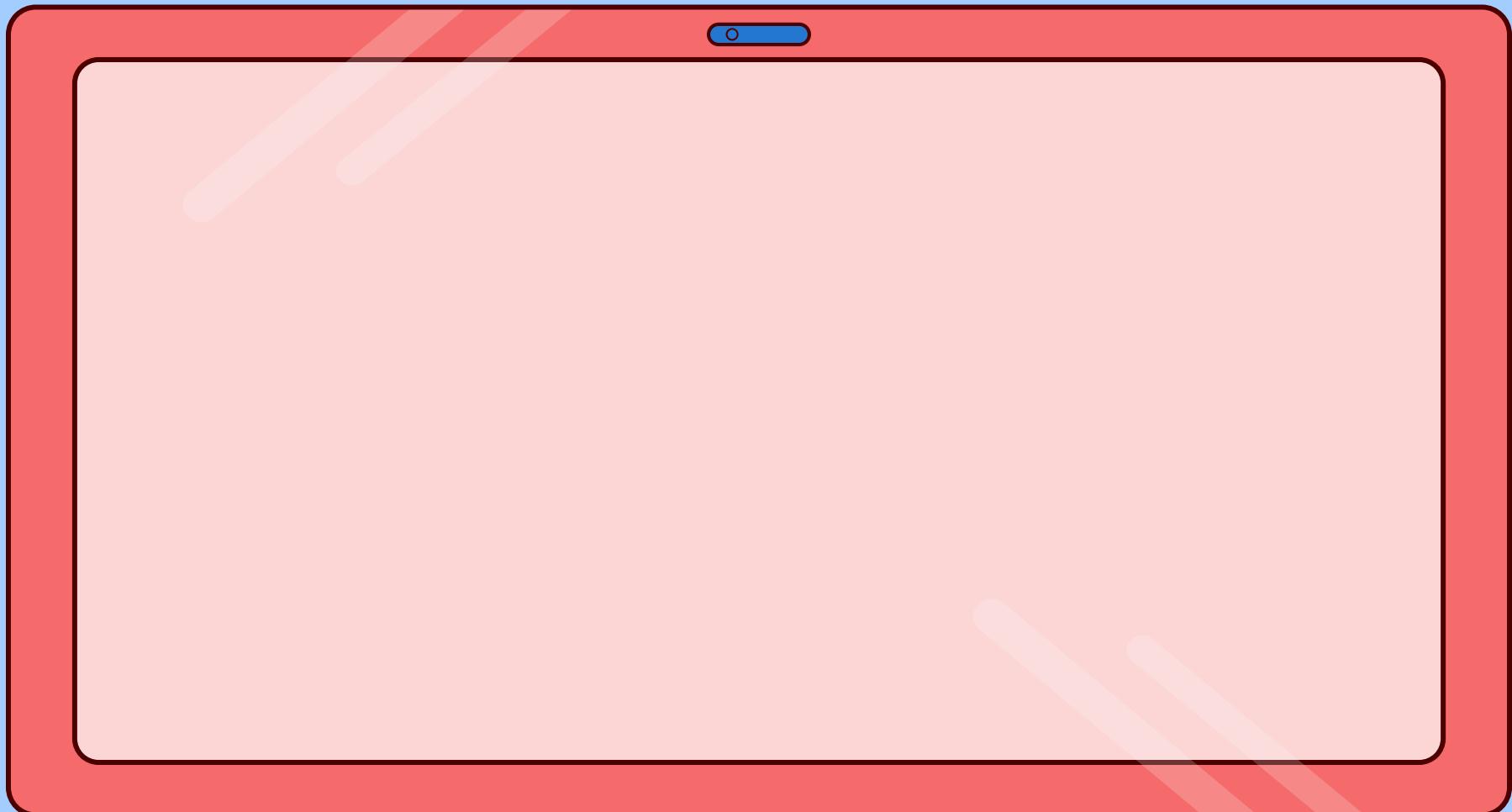
- a) $a_0 = \lfloor a_0/16 \rfloor$ (Ergebnis abgerundet, vorzeichenlose Division)
- b) $a_0 = a_0 \bmod 256$ (vorzeichenlos)
- c) Kopieren Sie die unteren 16 Bit von a_1 in die unteren 16 Bit von a_0 und die unteren 16 Bit von a_2 in die oberen 16 Bit von a_0 .
- d) Setzen Sie das a_1 -te Bit in a_0 auf Eins. Die restlichen Bits sollen unverändert bleiben.
- e) Bonus: Setzen Sie das a_1 -te Bit in a_0 auf Null. Die restlichen Bits sollen unverändert bleiben.



```
task_a:  
    srl a0, a0, 4  
  
task_b:  
    # Idee: Jedes Bit ab 8 ( $2^8$ ) ist Vielfaches von 256  
    # => z.B. verunden mit 000...01111111 (255)  
    andi a0, a0, 255  
  
task_c:  
    slli a1, a1, 16  
    srl a1, a1, 16 # Nullen von oberen 16 Bit von a1  
  
    # Verschieben von unteren 16 Bit von a2 nach oben  
    # (untere 16 Bit sind damit 0)  
    slli a2, a2, 16  
  
    or a0, a1, a2
```

```
task_d:  
addi a2, zero, 1  
sll a2, a2, a1  
or a0, a0, a2  
  
task_e:  
# einfache Lösung mit Code Recycling zuerst auf 1 setzen,  
# dann xor mit gleicher Maske (1 xor 1 = 0)  
addi a2, zero, 1  
sll a2, a2, a1  
or a0, a0, a2  
xor a0, a0, a2
```

- a) Was passiert, wenn man die in der Zentralübung erwähnte Formel zur Berechnung des Werts einer Binärzahl ($a = \sum_{i=0}^n a_i \cdot 2^i$) auf negative Indizes erweitert?
- b) Welchen Wertebereich kann man mit 8 binären Vorkommastellen (ohne Vorzeichen) und 8 binären Nachkommastellen (8.8) erreichen?
- c) Wieviele Bit bräuchte man mindestens, um Zahlen von 0 bis 100 darzustellen, sodass der Abstand zwischen zwei darstellbaren Werten maximal 0.005 beträgt?
- d) Wie sieht die Addition bzw. Subtraktion in Festkommarechnung auf einem Blatt Papier aus? Was muss man beachten?
- e) Wie sieht die Multiplikation in Festkommarechnung auf einem Blatt Papier aus? Was muss man beachten?
- f) Wie sieht die Zahl π im 8.8-Format aus? Was muss man beachten?
- g) Wie kann man beliebige Zahlen in Festkommazahlen umwandeln?



Die Formel funktioniert weiter. Für Nachkommastellen (rechts vom Komma) sind die Zweierpotenzen keine Ganzzahlen mehr, sondern rationale Zahlen: $2^{-1} = \frac{1}{2}$, $2^{-2} = \frac{1}{2^2} = \frac{1}{4}$ usw.

Vorkomma: $2^8 - 1 = 255$, Nachkomma: $2^{-8} = \frac{1}{256} = 0.00390625$. D.h. die Schrittweite unserer „Kommazahlen“ ist $\frac{1}{256}$. Somit ist die größte darstellbare Zahl: $(2^8 - 1) + (1 - 2^{-8}) = 255 + 0.99609375 = 255.99609375$ und damit der Wertebereich: $[0, 255.99609375]$

Gesucht ist eine Formel, mit der man von Schrittweite auf Nachkommabits kommen. Von der Aufgabe davor wissen wir, dass hier ein logarithmischer Zusammenhang besteht. Also:

$$\#\text{Nachkommastellen} = \lceil -\log_2(0.005) \rceil = 8$$

Kontrolle: $2^{-7} = 0.0078125$, also zu groß. $2^{-8} = 0.00390625$ passt. $2^{-9} = 0.001953125$ passt (offensichtlich) auch. 8 Bits ist damit die gesuchte minimale Antwort.

Man muss nichts Besonders beachten :)

Wenn man Vor- und Nachkommaanteil konkateniert und multipliziert, ist das Multiplikationsergebnis um die Nachkommastellen nach links verschoben. Dies muss nach der Multiplikation wieder korrigiert werden:

Beispiel mit 8+8 Vorkomma/Nachkommastellen:

$$a = 1.5 = (0000\ 0001 \cdot 1000\ 0000 = 384)$$

$$b = 3.25 = (0000\ 0011 \cdot 0100\ 0000 = 832)$$

$$384 \cdot 832 = 319488 = 000001001110000000000000$$

Verschiebung um $n = 8$ Stellen nach rechts (Division durch 256)

$$0000\ 0100 \cdot 1110\ 0000 = 4.875$$

Das heißt Festkommamultiplikation geht mit normaler Ganzzahlmultiplikation, wenn man danach eine Korrektur durch Rechtsverschiebung bzw. Teilen durch die passende Zweierpotenz durchführt.

Umwandlung von π ins 8.8-Format: $[3.141592 \cdot 256] = 804$ (einfach die Zahl mit $256 = 2^8$ multiplizieren und Nachstellen abschneiden)

Sollte anhand der letzten Aufgabe klar geworden sein ;)



Live love Dominic

804 → 00000011._{red}00100100