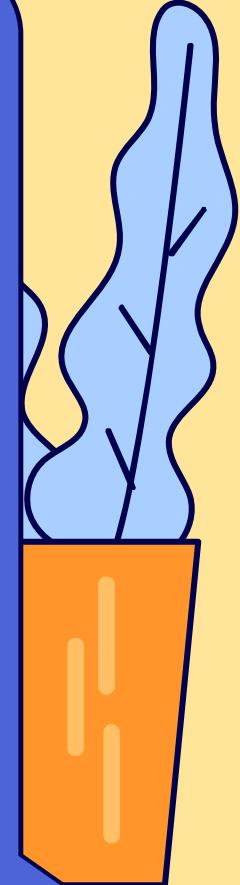
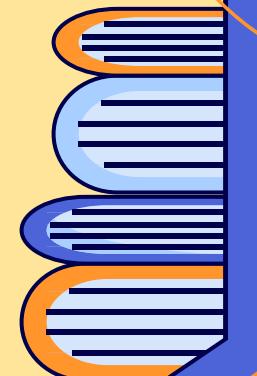
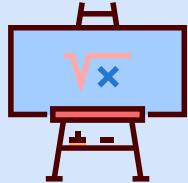




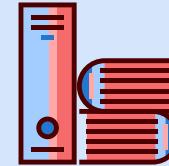
Herzlich willkommen
zum vierten ERA
Tutorium des
Semesters!





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.

- ▶ Varianten mit und ohne Vorzeichen
- ▶ Standard: Verwendung des Zweierkomplements
- ▶ Bits für Zahlen sollte Potenz von 2 sein (aufgrund der Hardware: Register, Speicheradressierung, etc.)

Typname in C	Bits	Vorzeichen	min	max
char	8	ja	-128	127
unsigned char		nein	0	255
short	16	ja	-32.768	32.767
unsigned short		nein	0	65.535
int	32	ja	-2.147.483.648	2.147.483.647
unsigned int		nein	0	4.294.967.295
long long	64	ja	-9.223.372.036.854.775.808	9.223.372.036.854.775.807
unsigned long long		nein	0	18.446.744.073.709.551.615

- ▶ steht **rd** (Register Destination) für ein beliebiges Register als Ziel einer Operation
- ▶ steht **rs** (Register Source) für ein beliebiges Register als Quelle einer Operation
- ▶ steht **ra** (Register Address) für das Register x_1
- ▶ **imm**: 12-bit signed Immediate
- ▶ **uimm**: 12-bit unsigned Immediate
- ▶ **upimm**: Obere 20 bits eines 32-Bit Immediates
- ▶ **Address**: Speicheradresse
- ▶ **[Address]**: Datum an Speicheradresse
- ▶ **{B, J}TA**: {Branch, Jump} Target Address

Load/Store Befehle

- ▶ Load und Store in 8 (b), 16 (h), 32 (w) Bit Varianten
- ▶ Bei 8 und 16 Bit-Load Varianten für Sign- und Zero-Extension des geladenen Datums
- ▶ effektive Adresse = $rs1 + imm$
- ▶ nur Register als Quelle für Adresse (keine Immediates!)

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	-	I	lb rd, imm(rs1)	load byte	$rd = \text{SignExt}([\text{Address}]_{7:0})$
0000011 (3)	001	-	I	lh rd, imm(rs1)	load half	$rd = \text{SignExt}([\text{Address}]_{15:0})$
0000011 (3)	010	-	I	lw rd, imm(rs1)	load word	$rd = [\text{Address}]_{31:0}$
0000011 (3)	100	-	I	lbu rd, imm(rs1)	load byte unsigned	$rd = \text{ZeroExt}([\text{Address}]_{7:0})$
0000011 (3)	101	-	I	lhu rd, imm(rs1)	load half unsigned	$rd = \text{ZeroExt}([\text{Address}]_{15:0})$
0100011 (35)	000	-	S	sb rs2, imm(rs1)	store byte	$[\text{Address}]_{7:0} = rs2_{7:0}$
0100011 (35)	001	-	S	sh rs2, imm(rs1)	store half	$[\text{Address}]_{15:0} = rs2_{15:0}$
0100011 (35)	010	-	S	sw rs2, imm(rs1)	store word	$[\text{Address}]_{31:0} = rs2$

Stacks/Keller

PUSH:

1. sp erniedrigen
2. an Adresse sp schreiben
(= Prädekkrement)

**! Stacks wachsen von „oben nach unten“
→ von hohen zu niedrigen Adressen !**



POP:

1. von Adresse sp lesen
2. sp erhöhen
(= Postinkrement)



Stacks/Keller in RISC-V

Beispiel: Sichern des Registers rs auf dem Stack

PUSH:

1. sp erniedrigen
`addi sp, sp, -4`
2. an Adresse sp schreiben
`sw rs, 0(sp)`
(= Prädekrement)

POP:

1. von Adresse sp lesen
`lw rd, 0(sp)`
 2. sp erhöhen
`addi sp, sp, 4`
(= Postinkrement)
- } Cleanup

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 "Callee-safe"

- 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/>



a) Was ist eine Calling Convention?

- a) Eine Art von Vertrag zwischen allen Entwicklern auf einem System. Sie bestimmt z.B. wie Parameter übergeben werden. Insgesamt ist das Ziel eine einheitliche und zuverlässige Funktionsweise von Programmen. Die Konvention wird nicht explizit überprüft – ihre Einhaltung wird vorausgesetzt. Ansonsten kann eine erfolgreiche Ausführung nicht garantiert werden.

b) Unterteilen Sie die Register in Kategorien und klassifizieren Sie diese nach ihrem Zweck und anderen Unterschieden in der Calling Convention.

b) Die Register und deren Nutzung nach Calling Convention ist in Tabelle 1 gegeben.

Register Name(s)	Usage
x0/zero	Always holds 0
ra	Holds the return address
sp	Holds the address of the boundary of the stack
t0-t6	Holds temporary values that do not persist after function calls
s0-s11	Holds values that persist after function calls
a0-a1	Holds the first two arguments to the function or the return values
a2-a7	Holds any remaining arguments

Abbildung 1: Register und deren übliche Nutzung

Callie oder
Caller-Saved?

- c) Wie können Sie zum Beispiel das Register **s0** trotzdem verwenden?



- c) Wir müssen die Register davor sichern. Dazu können wir den Wert auf den Stack legen. Wir müssen dann nach Verwendung des Registers den Wert wieder vom Stack holen und in das Register zurück schreiben.



- d) Sie schreiben ein Programm in Assembly und rufen ein Unterprogramm auf. Welche Register sind danach garantiert unverändert? Welche müssen Sie vorher sichern um zu garantieren, dass das Programm nicht abstürzt?

- d) Die s-Register und der Stack-Pointer sind garantiert unverändert. Die Rücksprungadresse muss vor einem weiteren Unterprogrammaufruf gesichert werden.

Keep track of ra!!



e) Warum benötigt man eine Calling Convention? Was sind die Vorteile?

- e) Die Calling Convention führt dazu, dass Code einfacher lesbar und erweiterbar ist, da z.B. immer die Rücksprungadresse im gleichen Register steht und der Stack nach dem Unterprogrammaufruf immer wiederhergestellt ist und die Interaktion mit anderen Unterprogrammen einheitlich ist. Zudem führt sie dazu, dass man nicht immer alle Register sichern muss (Speicherzugriffe sind teuer).

f) Wie werden die Parameter und Rückgabewerte in den folgenden Funktionen nach Calling Convention übergeben/zurückgegeben? $R_w \Rightarrow$

- `char add_char(char a, char b)` $a \Rightarrow$ $b \Rightarrow$
- `uint64_t add(uint64_t a, uint64_t b)` $R_w \Rightarrow$ $a \Rightarrow$ $b \Rightarrow$
- `uint64_t add(uint64_t* a, uint64_t b)` $R_w \Rightarrow$ $a \Rightarrow$ $b \Rightarrow$
- `uint128_t copy_and_increment(uint64_t a)` $R_w \Rightarrow$ $a \Rightarrow$

tricky?



Funktionsparameter -> a0-a7

Für nur lokal wichtige Parameter -> t0-t6

Rückgabewert -> Lieber a0, wenn das nicht reicht, dann a1...

- f) • a wird in a0 übergeben, b in a1 und der Rückgabewert steht in a0
- Die niederwertigen 32 Bit von a/b werden in a0/a2 übergeben, die höherwertigen 32 Bit von a/b in a1/a3. Der Rückgabewert wird über a0/a1 zurückgegeben.
 - Der Pointer a wird über a0 übergeben. Der 64 Bit Wert b über a1 und a2. Der Rückgabewert wird über a0 und a1 zurückgegeben.
 - Da Datentypen größer als 64 Bit als Referenz zurückgegeben werden, wird in a0 ein Pointer für das Ergebnis übergeben. a wird über a1 und a2 übergeben.

Gegeben sei folgendes Programm, das ab der Startadresse 0x200 im Speicher liegt. Das heißt die Bitfolge der Instruktion `addi a0, zero, 4` steht an 0x200.

- a) Annotieren Sie alle Zeilen mit den entsprechenden Adressen der Instruktionen. Sie dürfen davon ausgehen, dass sie aufeinanderfolgend im Speicher liegen und keine komprimierten Instruktionen verwendet werden.

Wie viele
Bytes ist
jeder Instruktion?

Sind Labels
Instruktionen?

```
0x200    _start:  
          addi a0, zero, 4  
          jal ra, magic  
          ebreak  
  
magic:  
        beq a0, zero, break  
        addi sp, sp, -8  
        sw ra, 0(sp)  
        sw a0, 4(sp)  
        addi a0, a0, -1  
        jal ra, magic  
        lw t0, 4(sp)  
        mul a0, a0, t0  
        lw ra, 0(sp)  
        addi sp, sp, 8  
        jalr zero, 0(ra)  
  
break:  
      addi a0, zero, 1  
      jalr zero, 0(ra)
```

```
_start:  
0x200      addi a0, zero, 4  
0x204      jal ra, magic  
0x208      ebreak  
  
magic:  
0x20c      beq a0, zero, break  
0x210      addi sp, sp, -8  
0x214      sw ra, 0(sp)  
0x218      sw a0, 4(sp)  
0x21c      addi a0, a0, -1  
0x220      jal ra, magic  
0x224      lw t0, 4(sp)  
0x228      mul a0, a0, t0  
0x22c      lw ra, 0(sp)  
0x230      addi sp, sp, 8  
0x234      jalr zero, 0(ra)  
  
break:  
0x238      addi a0, zero, 1  
0x23c      jalr zero, 0(ra)
```



- b) Gehen Sie Schritt für Schritt durch die ausgeführten Befehle. Notieren Sie sich jeden Schritt, in dem sich der Stack ändert, sowie dessen Zustand zu diesem Zeitpunkt.

```
_start:  
0x200    addi a0, zero, 4  
0x204    jal ra, magic  
0x208    ebreak  
  
magic:  
0x20c    beq a0, zero, break  
0x210    addi sp, sp, -8  
0x214    sw ra, 0(sp)           sw speichert Werte vom Register im Hauptspeicher!  
0x218    sw a0, 4(sp)  
0x21c    addi a0, a0, -1  
0x220    jal ra, magic  
0x224    lw t0, 4(sp)           lw speichert Werte vom Hauptspeicher im Register!  
0x228    mul a0, a0, t0  
0x22c    lw ra, 0(sp)  
0x230    addi sp, sp, 8  
0x234    jalr zero, 0(ra)  
  
break:  
0x238    addi a0, zero, 1  
0x23c    jalr zero, 0(ra)
```

jal ra speichert den nächsten Befehl in ra!

*sw speichert Werte vom Register im Hauptspeicher!
(nicht Register)*

lw speichert Werte vom Hauptspeicher im Register!

```

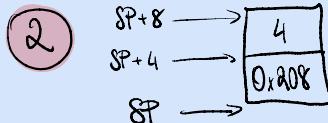
_start:
0x200    addi a0, zero, 4
          jal ra, magic
0x204    ebreak

magic:
0x20c    beq a0, zero, break
0x210    addi sp, sp, -8
0x214    sw ra, 0(sp)
0x218    sw a0, 4(sp)
0x21c    addi a0, a0, -1
0x220    jal ra, magic
0x224    lw t0, 4(sp)
0x228    mul a0, a0, t0
0x22c    lw ra, 0(sp)
0x230    addi sp, sp, 8
0x234    jalr zero, 0(ra)

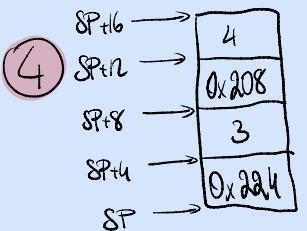
break:
0x238    addi a0, zero, 1
0x23c    jalr zero, 0(ra)

```

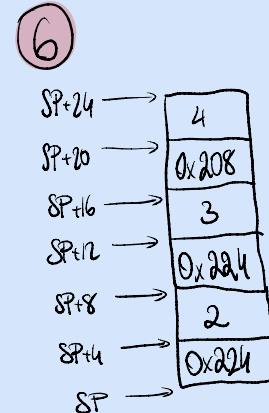
① $a0 = (4)_{10}$
 $ra = 0x208$



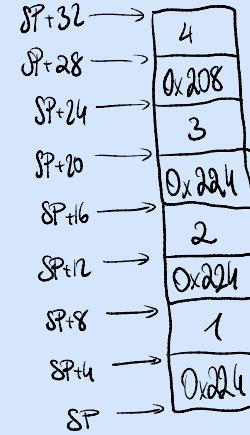
③ $a0 = (3)_{10}$
 $ra = 0x204$



⑤ $a0 = (2)_{10}$
 $ra = 0x204$



⑦ $a0 = (1)_{10}$
 $sp = 0x204$



⑨ $a0 = (0)_{10}$
 $ra = 0x204$

```

_start:
0x200    addi a0, zero, 4
0x204    jal ra, magic
0x208    ebreak

```

magic:

```

0x20c    beq a0, zero, break
0x210    addi sp, sp, -8
0x214    sw ra, 0(sp)
0x218    sw a0, 4(sp)
0x21c    addi a0, a0, -1
0x220    jal ra, magic
0x224    lw t0, 4(sp)
0x228    mul a0, a0, t0
0x22c    lw ra, 0(sp)
0x230    addi sp, sp, 8
0x234    jalr zero, 0(ra)

```

break:

```

0x238    addi a0, zero, 1
0x23c    jalr zero, 0(ra)

```

$$⑩ \quad a0 = (1)_{10}$$

SP+32	4
SP+28	0x208
SP+24	3
SP+20	0x224
SP+16	2
SP+12	0x224
SP+8	1
SP+4	0x224
SP	

$$⑪ \quad t0 = 1$$

$$\begin{aligned} a0 &= 1 \\ ra &= 0x224 \end{aligned}$$

$$⑫ \quad t0 = 2$$

$$\begin{aligned} a0 &= 2 \\ ra &= 0x224 \end{aligned}$$

SP+16	4
SP+12	0x208
SP+8	3
SP+4	0x224
SP	2

$$⑬ \quad t0 = 3$$

$$\begin{aligned} a0 &= 6 \\ ra &= 0x224 \end{aligned}$$

SP+8	4
SP+4	0x208
SP	

$$⑭ \quad t0 = 4$$

$$\begin{aligned} a0 &= 24 \\ ra &= 0x208 \end{aligned}$$

ebreak



b) Vor erstem jal ra, magic: 0x208, 0x4

Vor zweitem jal ra, magic: 0x224, 0x3, 0x208, 0x4

Vor drittem jal ra, magic: 0x224, 0x2, 0x224, 0x3, 0x208, 0x4

Vor viertem jal ra, magic: 0x224, 0x1, 0x224, 0x2, 0x224, 0x3, 0x208, 0x4

Nach return von Unterprogramm für $n = 1$: 0x224, 0x2, 0x224, 0x3, 0x208, 0x4

Nach return von Unterprogramm für $n = 2$: 0x224, 0x3, 0x208, 0x4

Nach return von Unterprogramm für $n = 3$: 0x208, 0x4

Nach return von Unterprogramm für $n = 4$: empty



c) Was berechnet das Unterprogramm `magic`?



c) Die Fakultät der übergebenen Zahl

d) Hält das Programm die Calling Convention ein? Wenn nein, was müsste man ändern?

```
        _start:  
0x200      addi a0, zero, 4  
0x204      jal ra, magic  
0x208      ebreak  
  
        magic:  
0x20c      beq a0, zero, break  
0x210      addi sp, sp, -8  
0x214      sw ra, 0(sp)  
0x218      sw a0, 4(sp)  
0x21c      addi a0, a0, -1  
0x220      jal ra, magic  
0x224      lw t0, 4(sp)  
0x228      mul a0, a0, t0  
0x22c      lw ra, 0(sp)  
0x230      addi sp, sp, 8  
0x234      jalr zero, 0(ra)  
  
        break:  
0x238      addi a0, zero, 1  
0x23c      jalr zero, 0(ra)
```

- d) Nein, das 16-Byte Stack-Alignment wird nicht eingehalten. Lösung: Künstlich mehr Stack allozieren (Stackpointer erniedrigen).

3 Größter gemeinsamer Teiler

Erstellen Sie ein Unterprogramm ggT, welches den größten gemeinsamen Teiler von zwei in a0 und a1 übergebenen Zahlen rekursiv berechnet. Das Ergebnis soll in a0 zurückgegeben werden. Als Hilfestellung ist unten C-Code zur rekursiven Berechnung des ggT gegeben. Achten Sie auch darauf die Calling Convention einzuhalten.

Hinweis: unsigned steht für unsigned int.

```
a0          a0          a1
unsigned ggT(unsigned a, unsigned b) {
    if (a==b)
        return a;
    else if (a < b)
        return ggT(a, b-a);
    else
        return ggT(a-b, b);
}
```

beq r1, r2, label
r1>r2 → bge(u) r1 , r2, label
r1<r2 → blt(u) r1 , r2, label

```
ggT:
```

```
    beq a0, a1, finished
    bltu a0, a1, lt
    sub a0, a0, a1
    j cont
```

```
lt:
```

```
    sub a1, a1, a0
```

```
cont:
```

```
    addi sp, sp, -16
    sw ra, 0(sp)
    jal ra, ggT
    lw ra, 0(sp)
    addi sp, sp, 16
```

```
finished:
```

```
    jalr zero, 0(ra)
```



Sehr hilfreich für die HA!

4 optional: Rekursive Folge

Schreiben Sie ein Unterprogramm, welches diese Folge rekursiv berechnet:

$$a_n = 2 \cdot a_{n-1} + n, \quad a_0 = 10$$

Verwenden Sie nur Befehle aus Tabelle 1.

Hinweis: Sie können das Ergebnis mithilfe dieser Formel überprüfen: $a_n = -n + 3 \cdot 2^{n+2} - 2$.