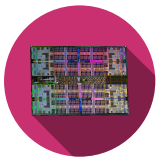


Befehlssatzarchitektur

Übungen Computerarchitektur

1 | Instruction-Set Architecture

1.1 Einfach C-Code zu RISC-V Assembler

Kompilieren Sie den folgenden C-Code in RISC-V Assembler.

a)
```c
a = b + c;
```

b)
```c
a = b + c - d;
```

c)
```c
a = b + 6;
```

d)
```c
// int is a 32-bit signed word
int a = -372;
int b = a + 6;
```

e)
```c
int a = 0xFEDC8765;
```

f)
```c
int a = 0xFEDC8EAB;
```

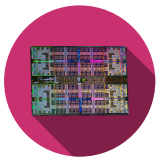
isa/c-to-riscv-01

1.2 Algorithmik C-Code zu RISC-V Assembler

Kompilieren Sie den folgenden C-Code in RISC-V Assembler.

a)
```c
if (i == j){
  f = g + h;
}
f = f - i;
```

b)
```c
if (i == j){
  f = g + h;
}
```



```
else {
  f = f - i;
}
```

c)

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i=i+1){
  sum = sum + i;
}
```

d)

```
// add the powers of 2 from 1 to 100
int sum = 0;
int i;

for (i=1;i<101; i=i*2){
  sum = sum + i;
}
```

e)

```
int array[5];
array[0] = array[0] * 2;
array[1] = array[1] * 2;
```

f)

```
int array[1000];
int i;

for (i=0; i<100; i=i+1){
  array[i] = array[i] * 8;
}
```

g)

```
char str[80] = "CAT";
int len = 0;

// compute length of string
while (str[len]) len++;
```

isa/c-to-riscv-02

1.3 Maschinencode zu RISC-V Assembler

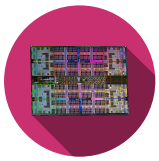
Dekodieren Sie den folgenden Maschinencode in RISC-V-Assembler.

a) **`0x41FE 83B3`**
b) **`0xFDA4 8393`**

isa/machinecode-to-riscv-01

1.4 Logische Operationen mit Registern

Führen Sie den Assember-Code aus und geben Sie den Inhalt des Zielregisters **`rd`** an falls die Quellregister **`rs`** folgende Daten enthalten:



```
s1 = 0x46A1 F1B7
s2 = 0xFFFF 0000
```

a)
```
and s3, s1, s2
```

b)
```
or s4, s1, s2
```

c)
```
xor s5, s1, s2
```

isa/riscv-execution-01

1.5 Logische Operationen mit Werten

Führen Sie den Assember-Code aus und geben Sie den Inhalt des Zielregisters **rd** an falls die Quellregister **rs** folgende Daten enthalten:

```
t3 = 0x3A75 0D6F
```

a)
```
and s5, t3, -1484
```

b)
```
or s6, t3, -1484
```

c)
```
xor s7, t3, -1484
```

isa/riscv-execution-02

1.6 Multiplikationen in RISC-V

Führen Sie den Assember-Code aus und geben Sie den Inhalt des Zielregisters **rd** an falls die Quellregister **rs** folgende Daten enthalten:

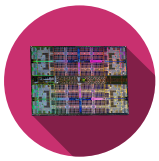
```
s1 = 0x4000 0000
s2 = 0x8000 0000
```

```
mulh s4, s1, s2
mul  s3, s1, s2
```

isa/riscv-execution-03

1.7 Division und Modulo

Führen Sie den Assember-Code aus und geben Sie den Inhalt des Zielregisters **rd** an falls die Quellregister **rs** folgende Daten enthalten:



```
s1 = 0x0000 0011
s2 = 0x0000 0003
```

```
div s3, s1, s2
rem s4, s1, s2
```

isa/riscv-execution-04

1.8 R-Typ zu Maschinencode

Kodieren Sie den folgenden RISC-V-Assembler in Maschinencode.

a)
```
add s2, s3, s4
```

b)
```
sub t0, t1, t2
```

c)
```
sll s7, t0, s1
```

d)
```
xor s8, s9, s10
```

e)
```
srai t1, t2, 29
```

isa/riscv-to-machinecode-01

1.9 I-Typ zu Maschinencode

Kodieren Sie den folgenden RISC-V-Assembler in Maschinencode.

a)
```
addi s0, s1, 12
```

b)
```
addi s2, t1, -14
```

c)
```
lw t2, -6(s3)
```

d)
```
lh s1, 27(zero)
```

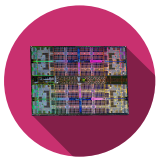
e)
```
lb s4, 0x1F(s4)
```

isa/riscv-to-machinecode-02

1.10 S-Typ zu Maschinencode

Kodieren Sie den folgenden RISC-V-Assembler in Maschinencode.

a)



```
sw t2, -6(s3)
```

b)

```
sh s4, 23(t0)
```

c)

```
sb t5, 0x2D(zero)
```

isa/riscv-to-machinecode-03

1.11 Realzeitsystem

Was ist der Hauptunterschied zwischen einem „harten“ und einem „weichen“ Echtzeitsystem?

- ☐ In einem Hard-Real-Time-System müssen alle Fristen eingehalten werden, während in einem Soft-Real-Time-System gelegentlich einige Fristen überschritten werden können.
- ☐ In einem Soft-Real-Time-System müssen alle Fristen eingehalten werden, während in einem Hard-Real-Time-System gelegentlich einige Fristen überschritten werden können.
- ☐ Eine Spielkonsole muss unter einem „harten“ Echtzeitsystem laufen, da es sonst nicht möglich ist, ein Spiel laufen zu lassen.
- ☐ Der Spritzkopf eines Druckers muss unter einem „harten“ Echtzeitsystem laufen, da die gedruckte Seite sonst fehlerhaft sein kann.
- ☐ Die Geschwindigkeitsmesssonde eines Flugzeugs muss unter einem „harten“ Echtzeitsystem laufen, da sich sonst der Autopilot abschalten kann.
- ☐ Ein „weiches“ Echtzeitsystem schneller ist als ein „hartes“ Echtzeitsystem.
- ☐ Ein „hartes“ Echtzeitsystem schneller ist als ein „weiches“ Echtzeitsystem.
- ☐ Windows 10 ist ein OS, das in der Lage ist, in „harter“ Echtzeit zu arbeiten.
- ☐ Ubuntu Desktop ist ein OS, das in der Lage ist, in „harter“ Echtzeit zu arbeiten.
- ☐ RTLinux ist ein OS, das in der Lage ist, in „harter“ Echtzeit zu arbeiten.

isa/riscv-to-machinecode-04

1.12 U-Typ zu Maschinencode

Kodieren Sie den folgenden RISC-V-Assembler in Maschinencode.

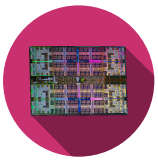
```
lui s5, 0x8CDEF
```

isa/riscv-to-machinecode-05

1.13 J-Typ zu Maschinencode

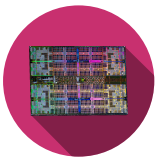
Kodieren Sie den RISC-V-Assembler-Befehl für den **ersten** Sprungbefehl im Maschinencode. Das Programm lautet wie folgt:

```
0x0000540C        jal ra,func1 # <--
0x00005410        add s1, s2, s3
...
```



```
0x001ABC04 func1: add s4, s5, s8
...
```

isa/riscv-to-machinecode-06



2 | Laborergänzung

Um Ihnen zu helfen können Sie gerne der RISC-V-Interpreter auf https://course.hevs.io/car/riscv-interpreter/ sowie Ripes verwenden.

Achten Sie sich auf die Typen der Variablen!

- Der Typ **int** wird als vorzeichenbehaftete 32-Bit-Größe betrachtet.
- Der Typ **unsigned int** wird als unsignierter 32-Bit-Typ betrachtet.
- Wenn eine Zahl dahinter steht (z. B. **int16_t**), bedeutet dies, dass die Variable x-Bit lang ist (hier 16). Wenn ein **u** vorangestellt ist, ist er unsigniert.

uint8_t ist also ein vorzeichenloses Byte, während **int8_t** ein vorzeichenbehaftetes Byte ist.

2.1 Grundrechenarten

a)

```
int b = 1;
int c = 2;
a = b + c;

int b = -1;
int c = 2;
a = b + c;

int b = -12;
int c = 2023;
a = b + c;
```

b)

```
int b = 2;
int c = 3;
int e = -1;
int f = -78;
int g = 2023;
int h = -12;
a = b - c;
d = (e + f) - (g + h);
```

isa/lab-basic-calc

2.2 Speicherzugriff

```
uint16_t a = mem[3];
mem[4] = a;

int16_t a = mem[3];
mem[4] = a;
```

isa/lab-memory

2.3 Grundlegende Algorithmen

1. Übertrage den 8Bit Wert des Speichers an Addresse 0x0000'1000 seriell Bit für Bit im LSB des Speichers in der Addresse 0x0000'1001. Die restlichen Bits der Speicheraddresse 0x0000'1001 müssen ‚0' betragen. Berechnen sie die BaudRate in $\frac{\text{Instructions}}{\text{Bit}}$ für die gesamte Übertragung?

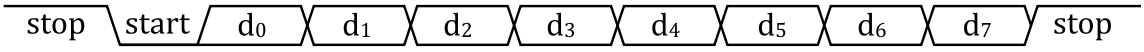


Abbildung 1: UART Serielle Übertragung

2. Multipliziere zwei 4-bit Zahlen zusammen benutzte hierzu zusätzlich einen der Befehle **bne**, **bge**. Der Algorithmus funktioniert folgendermassen: Eine Multiplikation ist das gleiche wie die x-fache Addition der gleichen Zahl. Zum Beispiel: $2 * 9 = 9 + 9 = 18$.

isa/lab-basic-algos

2.4 Branching

2.4.1 If / else

```
int a = 1, b = 2, c;

if(a == b) {
  c = 0;
} else if(a > b) {
  c = 1;
} else {
  c = 2;
}
```

2.4.2 Switch case

```
int a;

switch(mem[2]) {
  case 0:
    a = 17;
    break;
  case 3:
    a = 33;
    break;
  case 8:
  case 12:
    a = 10;
    break;
  default:
    a = 99;
}
```

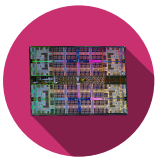
2.4.3 While / Do While

```
// A
int a = 10;

do{a = a - 1;}
while(a != 0);

// B
int a = 10;

do{a = a - 1;}
while(a >= 0);

// C
unsigned int a = 10;

do{a = a - 1;}
while(a >= 0);
```

2.4.4 For

```
int a = 0, i;

for(i = 4; i > mem[0]; i = i - 1) {
  a = a + i;
}
```

isa/lab-branch



2.5 Functions

a)

```
int a = 1, b;
b = doubleIt(a);
b = doubleItOpti(a);

...

// Non-optimized version
// Let's assume a is saved in s0
int doubleIt(int myvar) {
  int a = myvar; // we WANT a in s0 !
  a = a * 2;
  return a;
}

// Optimized version
// Choose your registers freely
// Try having the less possible
//   instructions
int doubleItOpti(int myvar) {
  int a = myvar; //
  a = a * 2;
  return a;
}
```

b)

```
int a=1, b=2, c=3,
 d=4, e=5, f=6, g=7,
 h=8, i=9, j=10, res;
res = sum(a,b,c,d,e,f,
  g,h,i,j);

...

int sum(int v1, int v2,
  int v3, int v4, int v5,
  int v6, int v7, int v8,
  int v9, int v10){
  int c;
  c = v1 + v2 + v3 + v4 +
    v5 + v6 + v7 + v8 +
    + v9 + v10;
  return c;
}
```

isa/lab-fcts

2.6 Advanced Algorithmus

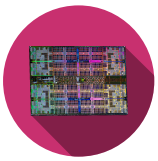
2.6.1 Modulo

Der Modulo % ist eine Operation, die auf zwei positive ganze Zahlen angewendet wird und ist nichts anderes als der Rest der Division. Zum Beispiel ergibt 5 geteilt durch 3 ergibt 1 (man kann 3 in 5 einmal hineinlegen), **Rest 2**.
Das Modulo einer Zahl durch 0 ist nicht definiert.
Die Definition für vorzeichenbehaftete Zahlen weicht je nach Sprache ab. Wir behandeln hier nur die unsigned Version.
Modulo ist vielseitig einsetzbar und ermöglicht es, Werte zu begrenzen, Informationen zu extrahieren, eine X- und Y-Position aus einem X*Y-Wert zu berechnen in einem Array mit bekannter Grösse ...

- Geben Sie einen Code an, mit dem diese Operation für jede positive ganze Zahl unter Verwendung des Sets RV32IM durchgeführt werden kann.
- Wie kann das Gleiche in RV32I implementiert werden? Beschreiben Sie das/die Konzept(e).

Die Aufgabe des Compilers ist es, den Code so gut wie möglich zu optimieren. Wenn die erkannte Operation ein Modulo ist, bei dem eine Konstante eine Potenz von 2 ist (z. B. x % 2, y % 8 ...), ist eine Variante möglich, die keine Division enthält.

- Geben Sie diese Variante an.



Das Konzept des Modulo für reelle Zahlen kam mit der Entwicklung der Rechenleistung auf und das Ergebnis weicht auch je nach Sprache ab. In C ist eine spezielle Funktion der std-Bibliotheken erforderlich, **fmod()**. In Python ist diese Operation nativ. In jedem Fall sind sie ressourcenintensiver und erfordern die Behandlung einiger Sonderfälle (NaN, infinity).

2.6.2 °F -> °C

Wir möchten eine Funktion erstellen, die Fahrenheitgrade in Celsius umwandeln kann, da die Fahrenheitswerte zwischen 32 und 1000 schwanken, ist eine Genauigkeit auf das Grad ausreichend.
Die Formel ist einfach: $C = (F - 32) * \frac{5}{9}$.
Es wäre praktisch wenn eine FPU zur Verfügung stünde, aber es wird nur den RV32I-Basisbefehlssatz unterstützt.
Um dieses Problem zu umgehen, können Sie einige Tricks anwenden, der Algorithmus lautet wiefolgt:

A. Berechnen Sie $C = F - 32$
B. Mit 5 multiplizieren
C. Durch 9 teilen
   - $\frac{1}{9}$ kann durch eine spezielle Binäre Darstellung gespeichert werden

     ```
     +-----+-----+-----+-----+-----+-----+------------+------------+
     | b31 | b30 | b29 | b28 | b27 | ... |         b1 |         b0 |
     +-----+-----+-----+-----+-----+-----+------------+------------+
     | 2^0 | 2^-1| 2^-2| 2^-3| 2^-4| ... |      2^-30 |      2^-31 |
     +-----+-----+-----+-----+-----+-----+------------+------------+
     | 1   | 1/2 | 1/4 | 1/8 | 1/16| ... |1/1737418240|1/2147483648|
     +-----+-----+-----+-----+-----+-----+------------+------------+
     ```

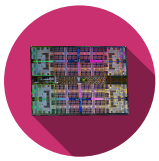
     ins diesem Fall ist dies $0000\_1110\_0011\_1000\_1110\_0011\_1000\_1110_2$.
   - Die Konstante kann vorberechnet werden und beträgt $\frac{2^n}{9} + 1$. Je grösser n ist, desto höher ist die Genauigkeit. Die Anzahl Bits definieren die maximale Grösse von n. Die $+1$ ist eine Aufrundung für die verlorene Genauigkeit.
   - Nehmen wir $n = 16$. Unsere magische Zahl lautet daher magic $= \frac{2^n}{9} + 1 = \frac{65536}{9} + 1 = 7282$.
   - Wir müssen den Wert mit dieser magischen Zahl multiplizieren
D. und danach durch 2^n dividieren. Im Fall $n = 16 \rightarrow \frac{1}{65536}$

Um die Arbeit zu vereinfachen, werden mehrere Annahmen getroffen:
- Die magische Zahl und die Temperatur in Fahrenheit sind immer positiv.
- Die Grösse der grössten Multiplikation beträgt:

$$\begin{aligned} \text{nbBits}_{\max_{\text{fahrenheit}}} + \text{nbBits}_{\text{mult5}} + \text{nbBits}_{\text{magicNumber}} &= \\ 10(\text{max. } 1000\text{-}32) + 3 + (n - \text{nbBits}_{\text{div9}} + 1) &= \\ 10 + 3 + (16 - 4 + 1) &= \\ = 26 \text{ bits} & \end{aligned} \tag{1}$$

  - Sie überschreitet nie 32 Bit für $n < 23$.

Testen und optimieren Sie die Funktion:



- Schreiben Sie den entsprechenden Code.
- Testen Sie mit verschiedenen Werten von Fahrenheit.
- Testen mit mehreren Werten für n (16, 18, 20). *Vergessen Sie nicht die magische Zahl neu zu berechnen.*
- Testen Sie die Funktion mit $n = [22, 23]$, für $°F = [100, 400, 1000]$.

isa/lab-adv-algos