Vorbereitung 2.7:

DSP verfügt über eine Hardwareunterstützung zur Schleifenprogrammierung

↳ Vorteil: Schleifen können ohne aufwendigen Softwareaufwand programmiert werden

Es stehen 2 Registersätze zur Verfügung mit denen jeweils eine Schleife programmiert werden kann.

Satz 0: LC0, LT0, LB0

Satz 1: LC1, LT1, LB1

LC = Loop Counter
LT = Loop Top
LB = Loop Bottom

LC muss auf gewünschte Zahl der Durchläufe gesetzt werden
LT enthält die Adresse des ersten Befehls, LB die des letzten.

a) Bestimmen Sie den Inhalt von I0 nach folgenden Operationen

```
I0.L=0xCCA0;        // Inhalt Indexregister I=0xFF800CA0
I0.H=0xFF80;
LC0 =0;             // zyklische Adressierung für I0 deaktiviert
P1=2; M3=4;
LSETUP(_LOOP_START, _LOOP_END) LC0=P1; // Schleife erstellen, Startadresse _LOOP_START
                                       Endadresse _LOOP_END
                                       Anzahl Schleifendurchläufe =2
_LOOP_START: R1=[I0++M3]; // Speicher Inhalt der Speicherstelle
                          auf die das Indexregister zeigt in R1. Erhöhe I0 um 4
_LOOP_END:   R1=[I0++M3]; // -"-  Erhöhe I0 um 4
```

↳ Schleife wird 2mal durchlaufen, I0 wird pro Schleifendurchlauf um 8 erhöht. $8 \cdot 2 = 16 = 0x10$

$0xFF800CA0 + 0x10 = 0xFF800CB0$

Vorbereitung 2.8: Vektorbefehle (SIMD Instructions)

Bei Vektorbefehlen wird der Inhalt eines 32 Bit Datenregisters als mehrere individuelle Datenworte (z.B. 2 16Bit Worte) behandelt.
→ Vorteil: Dadurch lassen sich mit einem Befehl beispielsweise zwei Multiplikationen und oder zwei Shiftbefehle gleichzeitig durchführen.

a) In einem 32 Bit Register R4 stehe der Wert 0x0007 0400
In 16 Bit Register R3.L stehe 0xFFFF. Gesucht ist Inhalt R0.

```
R0 = ASHIFT R4 BY R3.L (V); // Betrachtet Inhalte von R4.L und
                            // R4.H als unabhängige 16 Bit Operanden
```

ist Inhalt von R3.L positiv → logischer Linksshift
negativ → arithm. Rechtsshift

R3.L = FFFF ≙ −1 → einmal arithm. rechtsshiften

R4.H = 0x0007 → 0b0000000000000111 »»1 = 0000000000000111 = 0x0007

R4.L = 0x0400 → 0b0000010000000000 »»1 = 0000001000000000 = 0x0200

R0 = 0x00070200

b) In den 32 Bit-Registern R2 und R3 stehen die Werte 0x10000800 bzw. 0x08001000
Gesucht: Inhalte der Akkumulatoren.

```
A0 = R2.L * R3.L, A1 = R2.H * R3.H; // zwei skalare Operationen welche gleich-
                                    // zeitig ausgeführt werden. Eine im
                                    // MAC0 eine im MAC1
```

A0 = A1 = 0800 . 1000 = 0x00800000

left shift correction → 0x01000000

A0 [A0.X | A0.W] → 0x00 01000000
A1

A0.X: extension bits

2.9 Vorbereitung: Parallelbefehle

- Bis zu 3 kompatible Befehle parallel
- Erstes Befehl muss 32 Bit Befehlswortlänge haben, zweiter und dritter jeweils 16 Bit
- Syntaktisch getrennt durch ||

Speicher wird wie folgt initialisiert:

Adresse	0xFF800CA4	0xFF800CA5	0xFF800CA6	0xFF800CA7
Daten	0x00	0x10	0x00	0x08

Bestimmen Sie den Inhalt des Akkumulators A0 und der Register R0.L und R1.L nach folgenden Operationen

```
I0.L = 0xCA4;
I0.H = 0xFF80;
I1   = I0;
```

// Inhalt des Indexregisters beschreiben

```
A0 = 0 || R0.L = W[I0++] || R1.L = W[I1++];
```

- A0 = 0: Schreibe 0 in den Akku 0
- R0.L = W[I0++]: R0.L = 0x1000, I0 = 0xFF800CA6
- R1.L = W[I1++]: R1.L = 0x1000, I1 = 0xFF800CA6

```
A0 += R0.L * R1.L || R0.L = W[I0] || R1.L = W[I1];
```

- A0 += R0.L * R1.L: 0x1000 · 0x1000
  = 0x01000000 (2.30)
  = 0x02000000 (1.31)
- R0.L = W[I0]: R0.L = 0x0800
- R1.L = W[I1]: R1.L = 0x0800

A0 + 0x02000000 = 0x00 02000000

A0.x

Vorbereitung 2.10 Mixed C / Assembler-Programmierung

- Assembler Code wird aus dem C-Code aufgerufen mit function(...) → Sprungbefehl zu _function
- Die ersten drei 32 Bit Wörter der Parameterliste werden über die Register R0 - R2 übergeben
- weitere Parameter werden über den Stack übergeben
- Rückgabewerte die höchstens 32 Bit benötigen werden in R0 zurückgegeben
- Rückgabewerte zwischen 32 und 64 Bit über R0 und R1

a) In einem C-Programm finden Variablendefinition und ein Funktionsaufruf wie folgt statt. Bestimmen Sie den Inhalt der Variablen c nach Aufruf des Assembler Programms.

```
struct a {short x; short y;} = {100, 200};   // 16 Bit
int b = 300, c;
c = fct(&a, b);   // &a Pointer der auf a zeigt
```

```
_fct:
    I0 = R0 // kopiere Adresse unter der a gespeichert ist
    R2.L = W[I0++];
    R2.H = W[I0];
    R3 = R2.L * R2.H (IS);
    R0 = R3 + R1;
    RTS;
```

R0 enthält Adresse unter der a gespeichert ist.

R1 = b = 300 = 0x012C

→ I0 zeigt nun auf a.x

→ a.x = 100 = 0x0064 nach R2.L kopieren, dann I0 um 2 erhöhen → a.y

→ a.y = 200 = 0x00C8 nach R2.H kopieren

→ R3 = 0x0064 · 0x00C8 = 0x4E20

→ R0 = 0x4E20 + 0x012C = 0x4F4C

→ Inhalt von R0 wird als c zurückgegeben → c = 0x4F4C = 20300