

Mikroprozessortechnik

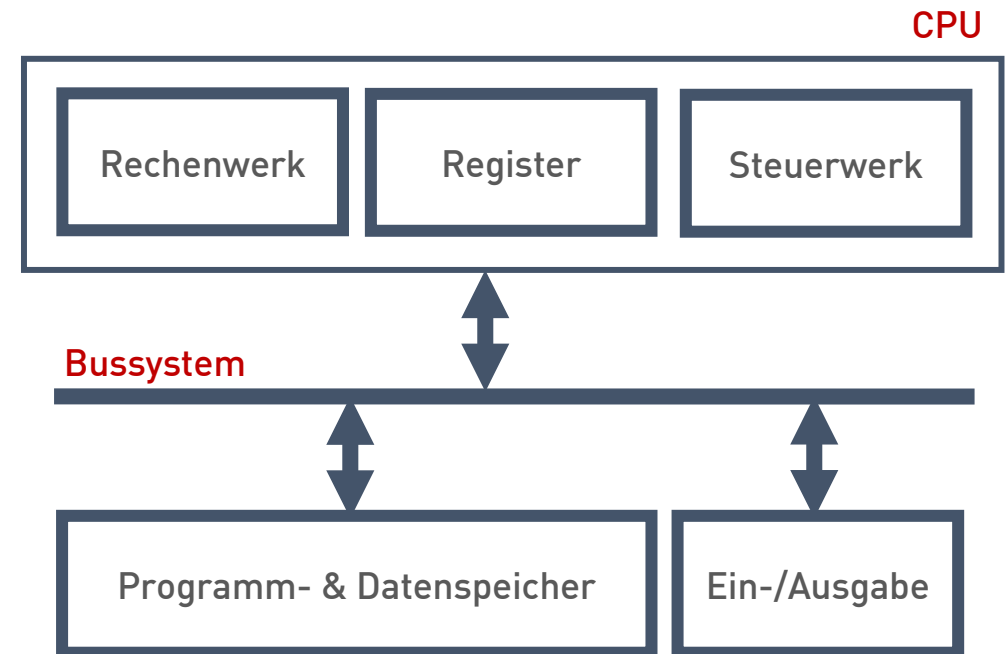
Prof. Dr. Michael Lipp



Programm- und Datenspeicher

Bereits behandelt

- **Aufbau und Struktur des μC**
 - Zentrale Verarbeitungseinheit (CPU)
 - Rechenwerk (ALU) ✓
 - Register (✓)
 - Steuerwerk ✗
 - Speicher (✓)
 - Ein- und Ausgabe-System
 - GPIO ✓
 - ADC ✓
 - ... ✗
 - Bussystem ✗



Programm- und Datenspeicher

- **32-bit Adressraum**

- 4.294.967.296 Byte (4 GiByte) im Embedded Bereich nicht sinnvoll als Speicher nutzbar
- Aufteilung des Adressraums in unterschiedlich genutzte Bereiche

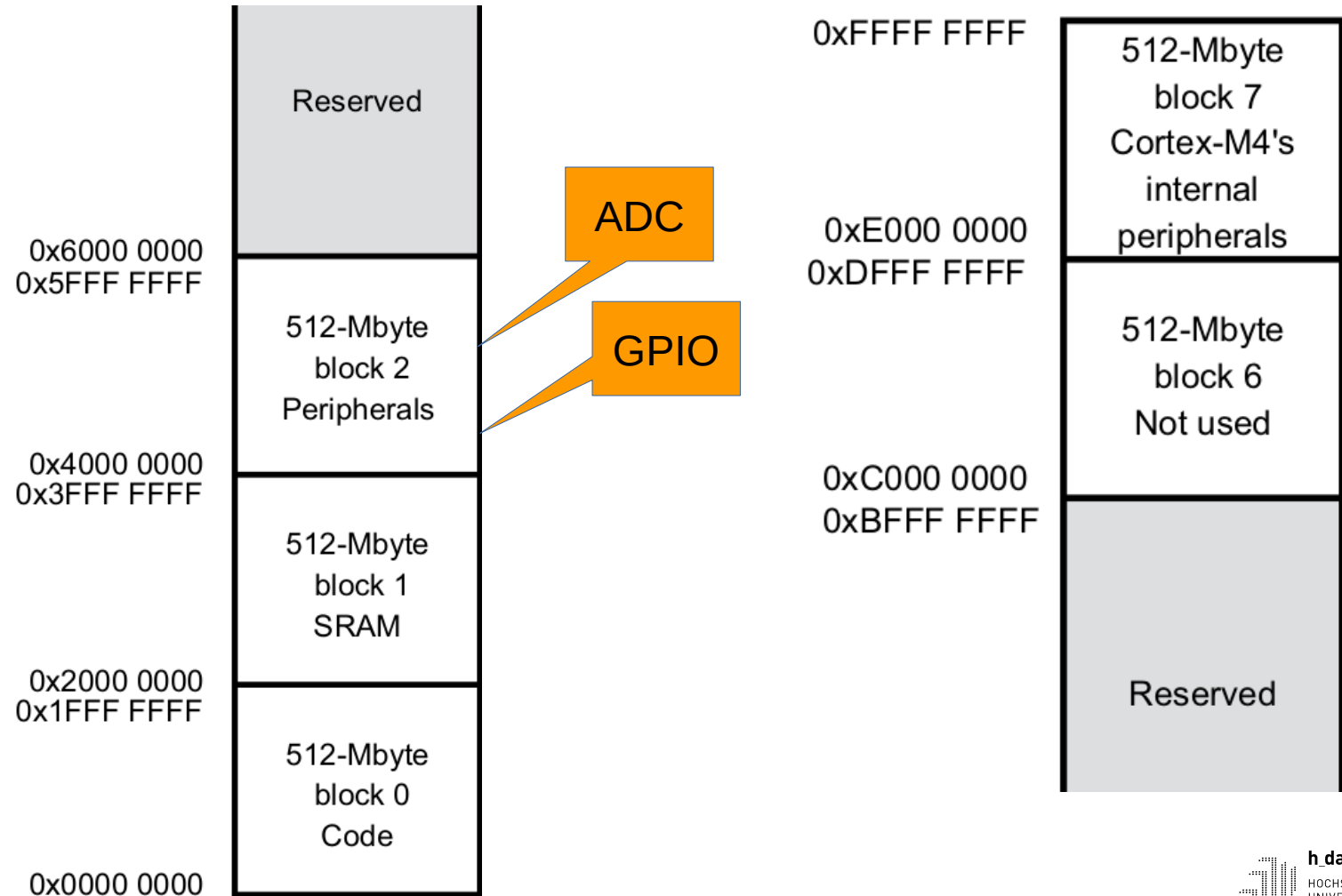
Unsere Mikrokontroller hat 32 bit Adressraum. Das ist für ein verdammt Mikrokontroller viel.

Programm- und Datenspeicher

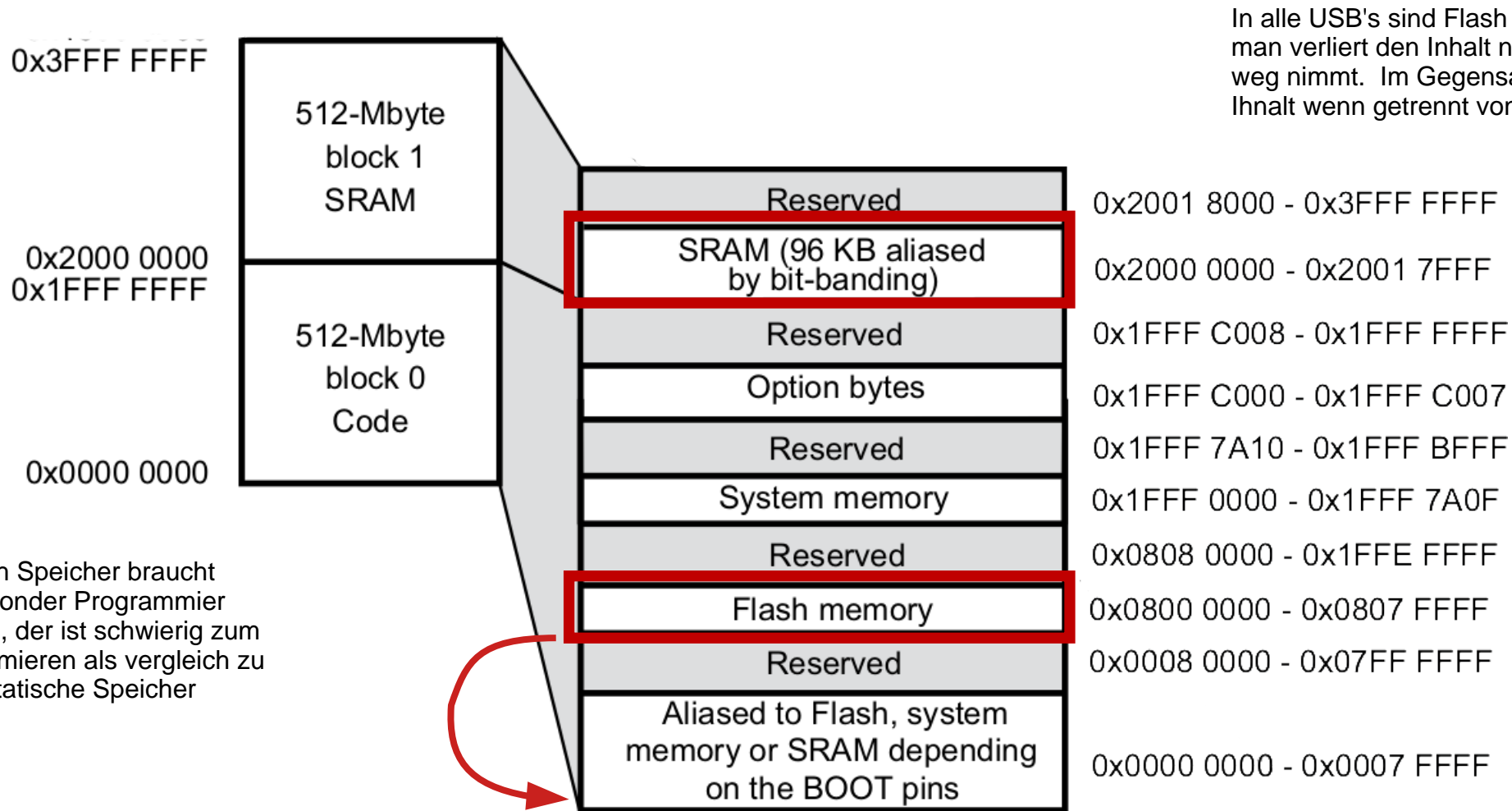
„Grob-Aufteilung“ in 512 MiB Blöcke

Bei unsere Mikrokontroller haben wir
Memory Mapped IO d.h Peripherie
Ausgabe sind dadurch gesteuert dass sie
Register bereit stellen die im Adresse
Bereich des Mikrokontroller liegen

Hinter bestimmte Adresse
verbergen Sie nicht der
Speicher Adresse sondern
auch Register mit IO Einheiten



Programm- und Datenspeicher



In alle USB's sind Flash Speicher drin. Das ist Speicher wo man verliert den Inhalt nicht wenn man obwohl das Strom weg nimmt. Im Gegensatz zu dem SRAM verlieren wir den Inhalt wenn getrennt von der Spannungsversorgung

Für Flash Speicher braucht man besonder Programmier Protokoll, der ist schwierig zum programmieren als vergleich zu einem Statistische Speicher

Es ist wichtig was auf dem Speicher 0000 steht



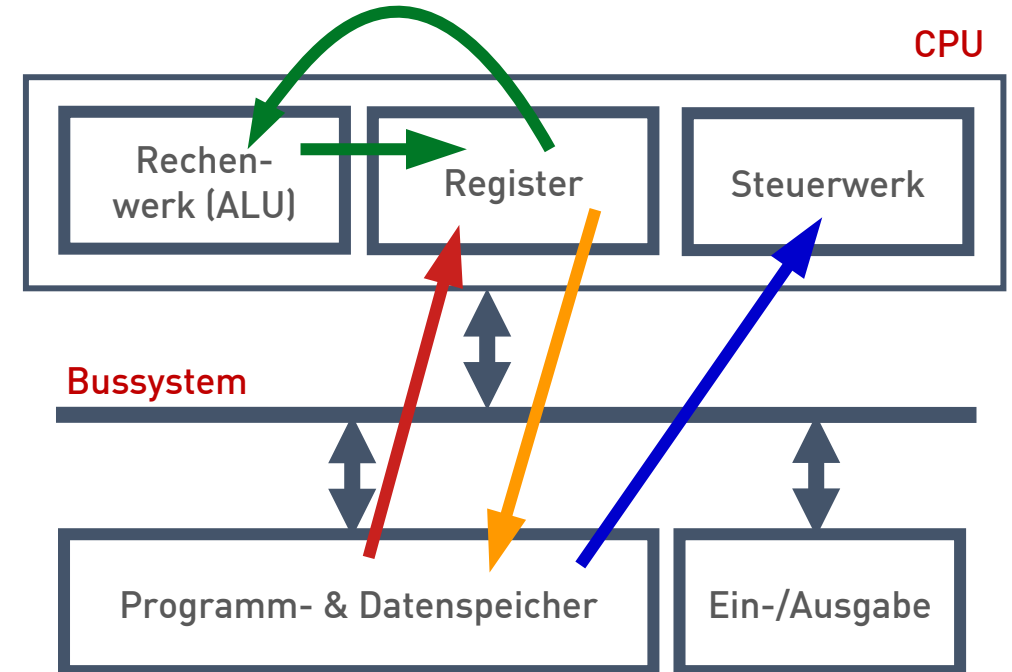
Programmausführung

Wdh.: Arbeitsprinzip

Woher genau wird der Wert aus dem Speicherstelle geholt ?

Woher genau?

- **Befehl aus Speicher holen**
- **Je nach Befehl...**
 - Daten aus Speicher oder von I/O-Register in CPU lesen
 - Daten verarbeiten
 - Daten von CPU in Speicher schreiben
- **Neuen Befehl holen usw.**

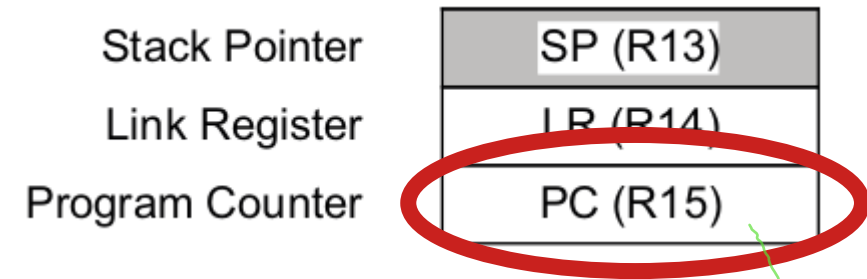


Program Counter

In 16 Bit Register von denen die 13 Bits können als Universal Register Bit verwendet also für die Verarbeitung von der Daten.

Die 3 andere Bits haben besondere Bedeutung

- **32-Bit Register**
- **Zeigt auf den nächsten auszuführenden Befehl**
- **Wird nach jedem Befehl erhöht**
 - Außer bei Sprungbefehlen und Unterprogrammaufrufen



Diese Bit zeigt auf den nächsten auszuführenden Befehl und damit beantworten wir woher wird das Befehl geholt.

Nach jedem Befehl wird der Programm Counter erhöht

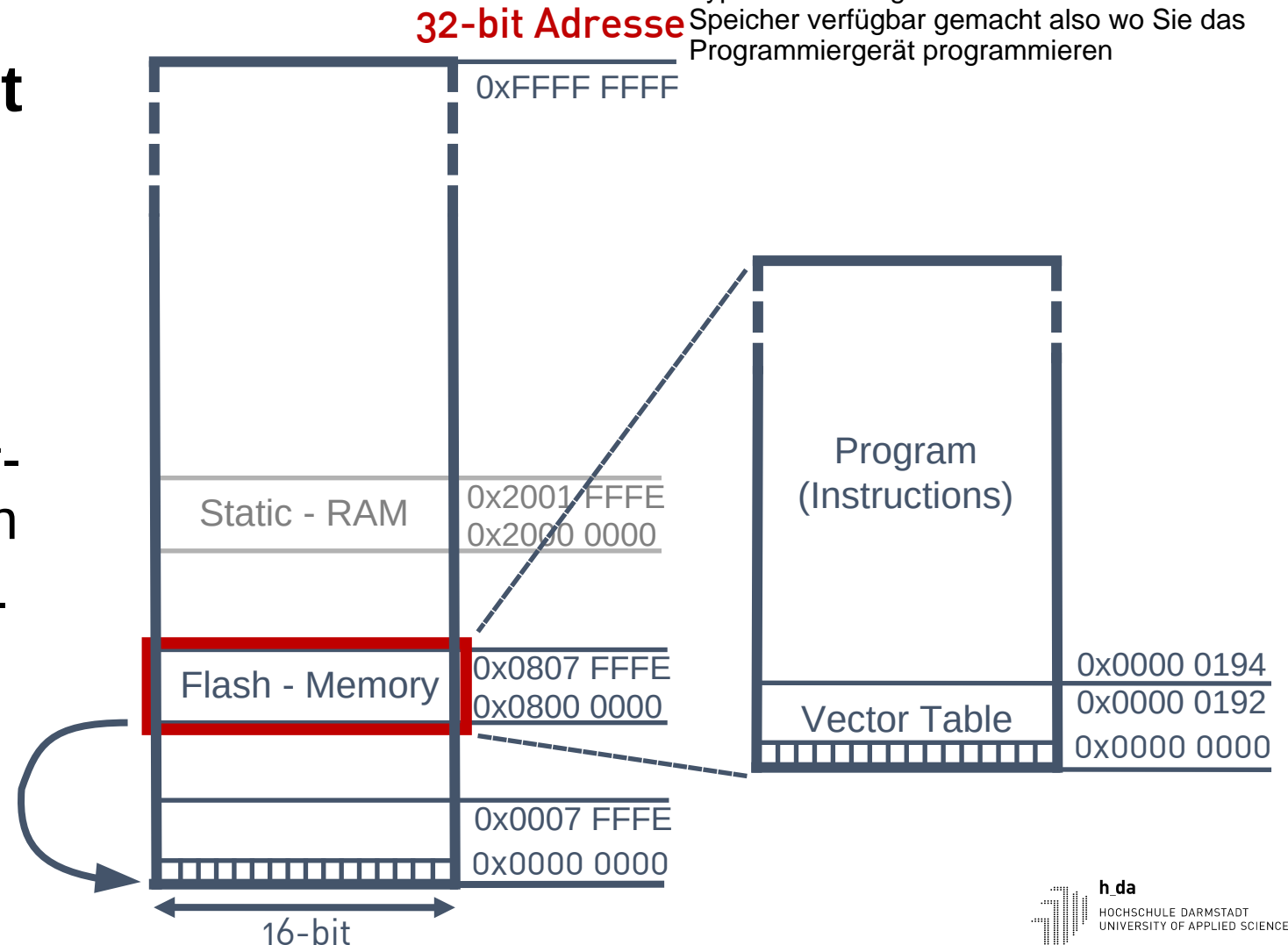
Program Counter

Bei Starten der Mikrokontroller (das schwarze Taste von der Microkontroller drücken) dann wird diese Adresse von der Program Counter 0 und daraus erklärt sich die besondere Bedeutung von dem was steht in der Speicher auf Adresse 0

Typischerweise gibt es dort der Inhalt des Flash Speicher verfügbar gemacht also wo Sie das Programmiergerät programmieren

- **(Start-)Programm steht in nicht-flüchtigem Speicher**

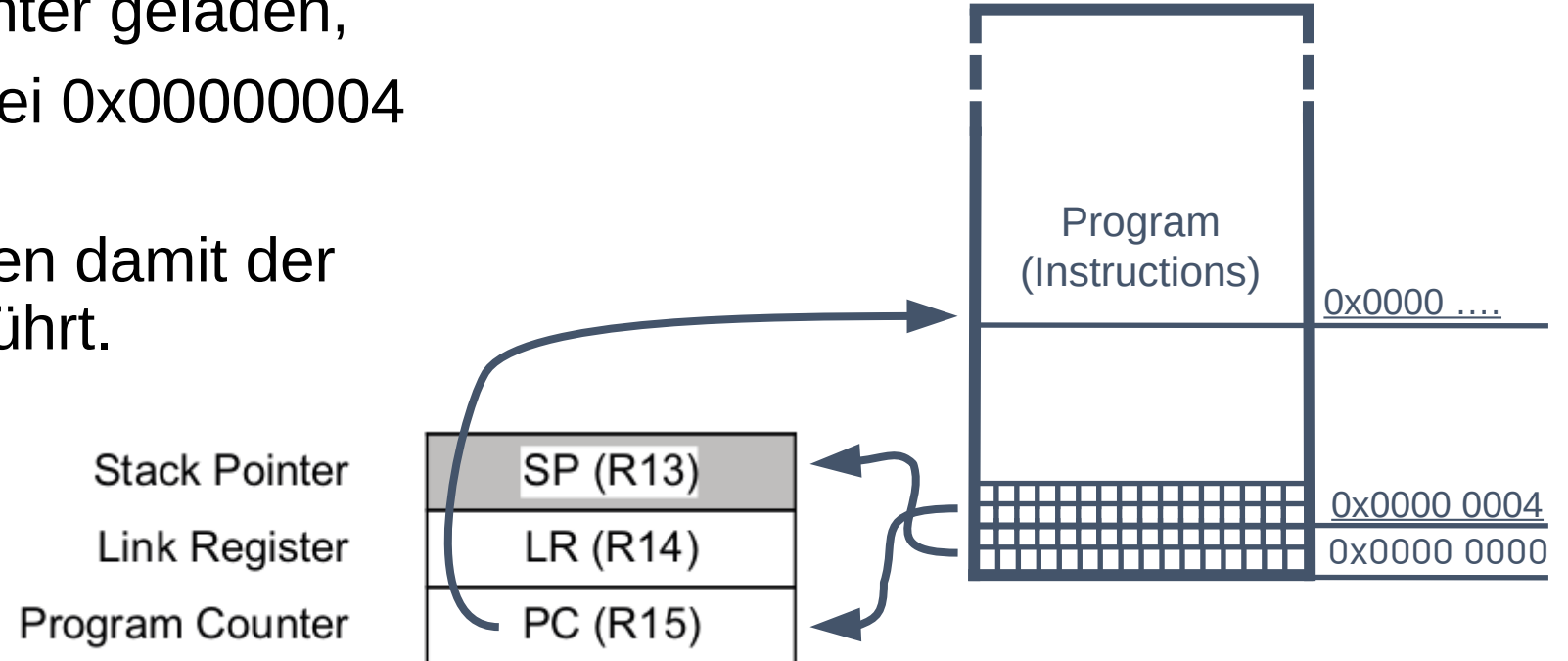
- Typisch: Flash-Speicher (wie USB-Stick)
- Wird durch Programmiergeräte in μC geschrieben
- Beginnt mit einer Vektortabelle (nächste Folie)
- Wird beim Booten an Adresse 0 „gemappt“



Boot-Vorgang

- **Beim Booten wird**

- der 32-bit Wert bei 0x00000000 in den Stack-Pointer geladen,
- der 32-bit Wert bei 0x00000004 in den PC und
- der Befehl, auf den damit der PC zeigt, ausgeführt.



Minimales Programm

CubeIDE - MinimalProgram/myCode/main.s - STM32CubeIDE

File Edit Navigate Search Project Run Window Help

Debug Project Explorer

MinimalProgram Debug [STM32 Cortex-M C/C++ Appli...]
 MinimalProgram.elf [cores: 0]
 Thread #1 [main] 1 [core: 0] (Suspended : Signal : ...)
 Reset_Handler() at main.s:18 0x8000194
 arm-none-eabi-gdb (8.1.0.20180315)
 ST-LINK (ST-LINK GDB server)

```

1 /*
2  * See https://www.silabs.com/community/mcu/32-bit/knowledge-base.entry.html/2018/11/11/unde=
3  */
4
5 .syntax unified
6 .cpu cortex-m4
7 .fpu softvfp
8 .thumb
9
10 .section .text
11
12 Reset_Handler:
13 /* Beim Booten nicht erforderlich, aber der
14  Reset_Handler kann auch angesprungen werden! */
15 // ldr sp, _estack // Lade Register SP mit Konstante
16
17 Main_Loop:
18 b Main_Loop // Springe zum Befehl nach dem Label "Main_Loop"
19
20 .global Default_Handler
21 Default_Handler:
22 Infinite_Loop:
23 b Infinite_Loop
24
25 // A minimal vector table for a Cortex M3, from CubeMX generated code.
26 .section .isr_vector, "a", %progbits
27
28 .word _estack
29 .word Reset_Handler
30

```

Name	Value	Description
r0	0	
r7	0	
r8	0	
r9	0	
r10	0	
r11	0	
r12	0	
sp	0x20018000	
lr	-1	
pc	0x8000194 <Re...	
xpsr	16777216	
d0	0	
d1	0	
d2	0	
d3	0	
d4	0	

Name : pc
 Hex:0x8000194
 Decimal:134218132
 Octal:01000000624
 Binary:100000000000000000000000110010100
 Default:0x8000194 <Reset_Handler>

Console Problems Executables Debugger Console Memory Call Hierarchy

Monitors

000 : 0x0 <Hex> New Renderings...

Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	00	80	01	20	94	01	00	08	97	01	00	08	97	01	00	08
00000010	97	01	00	08	97	01	00	08	97	01	00	08	00	00	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	97	01	00	08
00000030	97	01	00	08	00	00	00	00	97	01	00	08	97	01	00	08
00000040	97	01	00	08	97	01	00	08	97	01	00	08	97	01	00	08

760M of 1536M

Live Coding

Minimales Programm

Zahldarstellung im Speicher: „Little Endian“, d.h. das niederwertigste Byte einer Zahl liegt an der ersten der benötigten Speicherstellen (an der Speicherstelle mit der kleinsten Adresse)

Debugger Console | Memory | Call Hierarchy

000 : 0x0 <Hex> | + New Renderings...

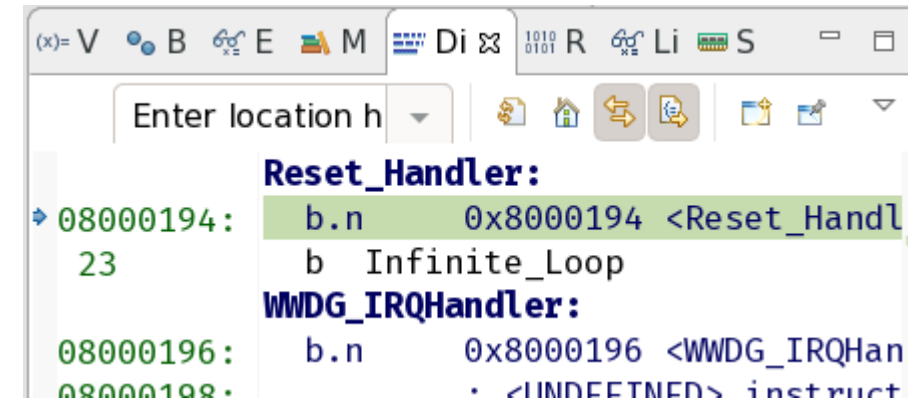
Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	00	80	01	20	94	01	00	08	97	01	00	08	97	01	00	08
00000010	97	01	00	08	97	01	00	08	97	01	00	08	00	00	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	97	01	00	08

Name	Value	Description
r6	0	
r7	0	
r8	0	
r9	0	
r10	0	
r11	0	
r12	0	
sp	0x20018000	
lr	-1	
pc	0x8000194 <Re	

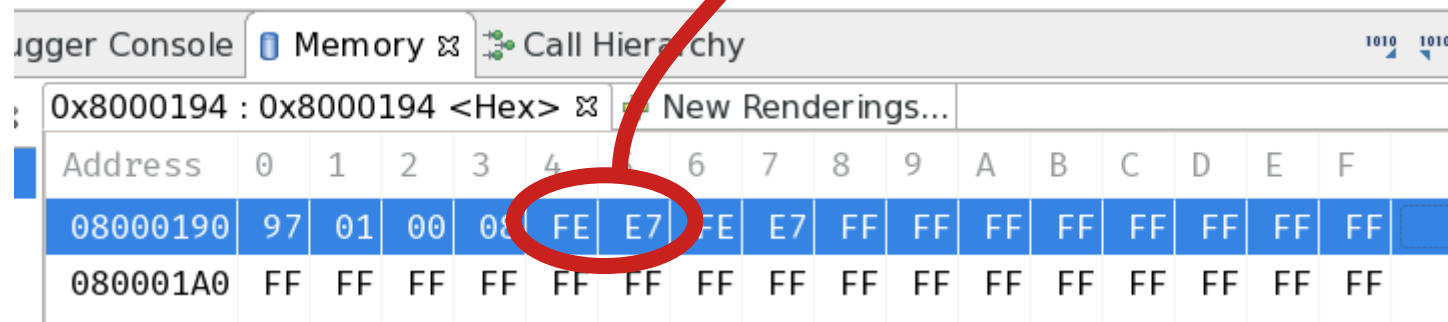
Minimales Programm

Befehle (Instructions) liegen im Speicher als 16-bit Werte (16-bit halfword). Manche Befehle bestehen auch aus mehr als einem Wort.

Im Disassembler-Fenster werden die Befehle für Menschen leichter lesbar aufbereitet (dekodiert).



```
(x)= V B E M Di 1010 R Li S
Enter location h
Reset_Handler:
08000194: b.n 0x8000194 <Reset_Handler>
23 b Infinite_Loop
WWDG_IRQHandler:
08000196: b.n 0x8000196 <WWDG_IRQHandler>
08000198: . <UNDEFINED> instruct
```



Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
08000190	97	01	00	00	FE	E7	FE	E7	FF	FF	FF	FF	FF	FF	FF	FF
080001A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF

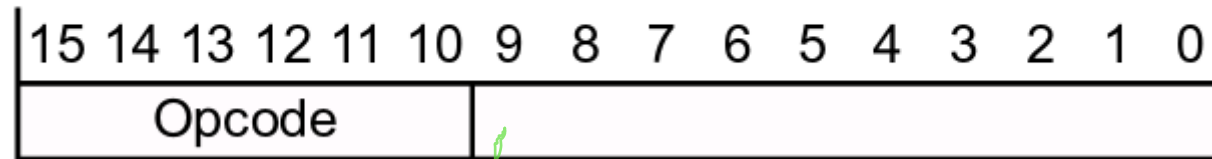
Es wird immer ausgeführt und nicht abhängig von der Bedingung und das ist ein sogenannte relativ Sprung Befehl und weist wohin gesprungen wird . Das steht nicht in der Sprung Argument dabei sondern es ist ein Wert der zum Programm Counter hin addiert wird. Der Sprung Befehl springt zu der Adresse die sich ergibt aus dem aktuelle Wert des Programm Counters plus diesem Offset

Minimales Programm

Die 6 Bits machen den Opcode aus. Diese Steuerwerk sagt das es handelt sich um einen Sprung Befehl und die verbleibende 10 Bits wird benutzt der Offset darzustellen. Das ist ein 10 stellige zweier Komplement Zahl.

• Aufbau der Befehlswörter (Thumb instruction set)

The encoding of a 16-bit Thumb instruction is:



Das ist der Variable Teile

Erstes „Schnuppern“, mehr zu Opcodes und Assembler-Programmierung folgt in späteren Vorlesungen

• Beispiel: 0xE7FE = 0b1110 0111 1111 1110

Opcode	Instruction or instruction class
11100x	Unconditional Branch, see B on page A8-332

-2
Zurück zum gleichen Befehl

Minimales Programm

- **Opcodes sind von Menschen nur schwer erstellbar**
 - (Haben Menschen aber zu Beginn der Computer-Ära gemacht)
- **Hilfsprogramm Assembler**
 - Mnemonische Kürzel für Opcodes (z.B. „b“ für branch)
 - Berechnung z.B. der Offset-Werte
 - Kodierung von Argumenten (Register etc.)
 - Definition von Konstanten
 - Instruktionen zur Steuerung des Übersetzungsvorgangs

Beispiel 32-Bit-Rechnung

CubeIDE - AssemblerExample1/myCode/main.s - STM32CubeIDE

File Edit Navigate Search Project Run Window Help

Debug Project Explorer

AssemblerExample1 Debug [STM32 Cortex-M C/C++ A]
 AssemblerExample1.elf [cores: 0]
 Thread #1 [main] 1 [core: 0] (Suspended : Step)
 Main_Loop() at main.s:23 0x80001a0
 0xffffffff

arm-none-eabi-gdb (8.1.0.20180315)
ST-LINK (ST-LINK GDB server)

Git Staging main.s

```
6 .cpu cortex-m4
7 .fpu softvfp
8 .thumb
9
10 .section .text
11
12 Reset_Handler:
13 /* Beim Booten nicht erforderlich, aber der
14  Reset_Handler kann auch angesprungen werden! */
15 // ldr sp, _estack // Lade Register SP mit Konstante
16
17 /* Laden von Werten aus dem Speicher ist nur über ein
18  Register möglich, das als Pointer-Variable eingesetzt
19  wird. Das C-Äquivalent wäre: int32_t* r0 = 0x20000000; */
20 mov r0, 0x20000000
21 /* Mit Hilfe des Pointers kann man jetzt auf den Speicher
22  zugreifen. C-Äquivalent: int32_t r1 = *r0; */
23 ldr r1, [r0]
24 /* Es ist möglich, beim Zugriff Offsets zu dem Wert in einem
25  Register zu addieren. C-Äquivalent: int32_t r2 = *(r0 + 1); */
26 ldr r2, [r0, 4]
27 add r3, r1, r2
28
29 Main_Loop:
30 b Main_Loop // Springe zum Befehl nach dem Label "Main_Loop"
31
32 .global Default_Handler
33 Default_Handler:
34 Infinite_Loop:
35 b Infinite_Loop
```

Quick Access

Name	Value	Description
r0	0x20000000 (Hex)	General Purpose
r1	1	
r2	2	
r3	3	
r4	0	
r5	0	
r6	0	
r7	0	
r8	0	
r9	0	
r10	0	
r11	0	
r12	0	
sp	0x20018000	

Name : r0
Hex:0x20000000
Decimal:536870912
Octal:0400000000
Binary:10000000000000000000000000000000
Default:536870912

Console Problems Executables Debugger Console Memory Call Hierarchy

Monitors

Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x8000194																
0x20000000	01	00	00	00	02	00	00	00	00	F3	F9	F3	20	00	F0	91
20000010	D1	F0	00	03	00	00	00	00	09	D4	DF	F8	94	03	45	F2
20000020	55	51	01	60	06	21	41	60	40	F6	FF	71	81	60	01	20
20000030	02	BD	2D	E9	F0	41	80	46	0D	46	16	46	1F	46	00	24
20000040	37	B1	02	2E	08	00	05	D3	03	2E	08	FE	08	24	08	FA

Writable Smart Insert 26 : 18 : 877 721M of 1536M

Live Coding

Link-Register

Wir brauchen ein Speicherplatz zum Speichern von wo wir hin gekommen sind in der Unterprogrammaufruf

Wenn ich in den Unterprogramm springe dann es bedeutet ich einen neue Wert in meinem Programm Counter lade nämlich den Wert die Adresse wo die Befehle des Unterprogramm ausmachen. Das heißt den aktuelle Wert von meinem Programm gehen verloren und wir wissen nicht wo wir waren auf unsere Hauptprogramm. Damit ich in meinem Hauptprogramm zurück kann

- **Unterprogrammaufruf**

- Beispiel:

```
void main(void){
```

```
    int i,k;
```

```
    i = sub(13);
```

```
    k = sub(14);
```

```
    // ...
```

```
}
```

```
int sub(int i){
```

```
    return i*2;
```

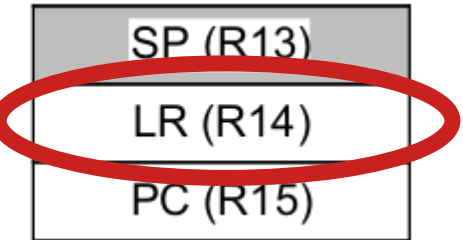
```
}
```

- Spring aus main nach sub (PC ← sub)
- Welchen Wert lädt return in den PC?

Stack Pointer

Link Register

Program Counter



Bevor wir ein Sprung machen wird der aktuelle Wert von der Programm Counter in den Link Register geschrieben. Dann wird der Start Adresse von der erste Befehle von Unterprogramm in Program Counter geschrieben. Und dann wird die Ausführung in der Unterprogramm weiter gemacht. Wenn es zum Ende ist wird der Wert wieder von der Link Register in der Programm Counter geladen.

Link-Register

- **Lösung**

- Sprung zu sub mit „branch with link“ („bl“)
 - Speichert die Adresse des nächsten Befehls (nach „bl“) im Link-Register
 - Lädt PC mit Adresse des Sprungziels
- Rücksprung
 - Lade den Wert von lr in pc („mov pc, lr“)

- **Neues Problem**

- Was passiert, wenn aus sub heraus wiederum ein Unterprogramm aufgerufen wird? Ein erneutes „bl“ würde den Wert in lr überschreiben...

Das bedeutet den aktuelle Wert des Programm Counter der in meinem erste Unterprogramm ist, der würde überschrieben und wenn ich dann zurücksprunge kann ich gut in der erste Unterprogramm aber nicht mehr in Hauptprogramme. Das Register **kann nur einen Wert speichern** . Um dieses Problem zum lösen wird der Stack Pointer benutzt.

Stack-Pointer

Wenn ich in einem Unterprogramm springe dann darf dieses Unterprogramm die Werte in den Register nicht verändern und wenn es Register verändern will dann muss es die Werte in den Register sichern. Und diese sichern passiert genau dadurch, dass er die Werte auf diese Stapelspeicher legt und vor dem Rücksprung die Werte wieder von der Stapelspeicher holt

- **Stack-Pointer realisiert einen “Stapelspeicher”**

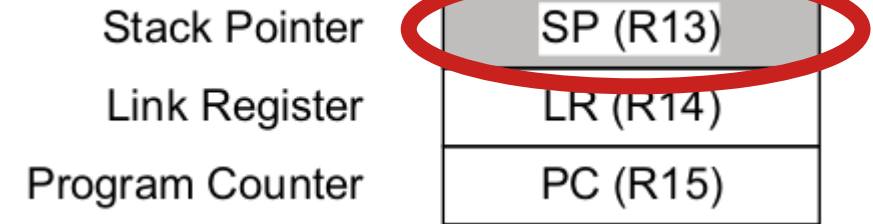
- Auch LIFO („last in first out“)

- **Funktionsweise**

- Werte in Registern, die gesichert werden müssen, werden auf den Stapel gelegt (kopiert)
- Zurückholen vom Stapel in die Register stellt die Werte wieder her

- **Faustregel:**

- Unterprogramme dürfen die Werte in Registern nicht verändern (außer sie nutzen Register für die Rückgabe eines Wertes). (Gilt auch für lr!)



Diese Methode realisiert mit dem Stack Pointer sp und den Speicher.

Stack Pointer

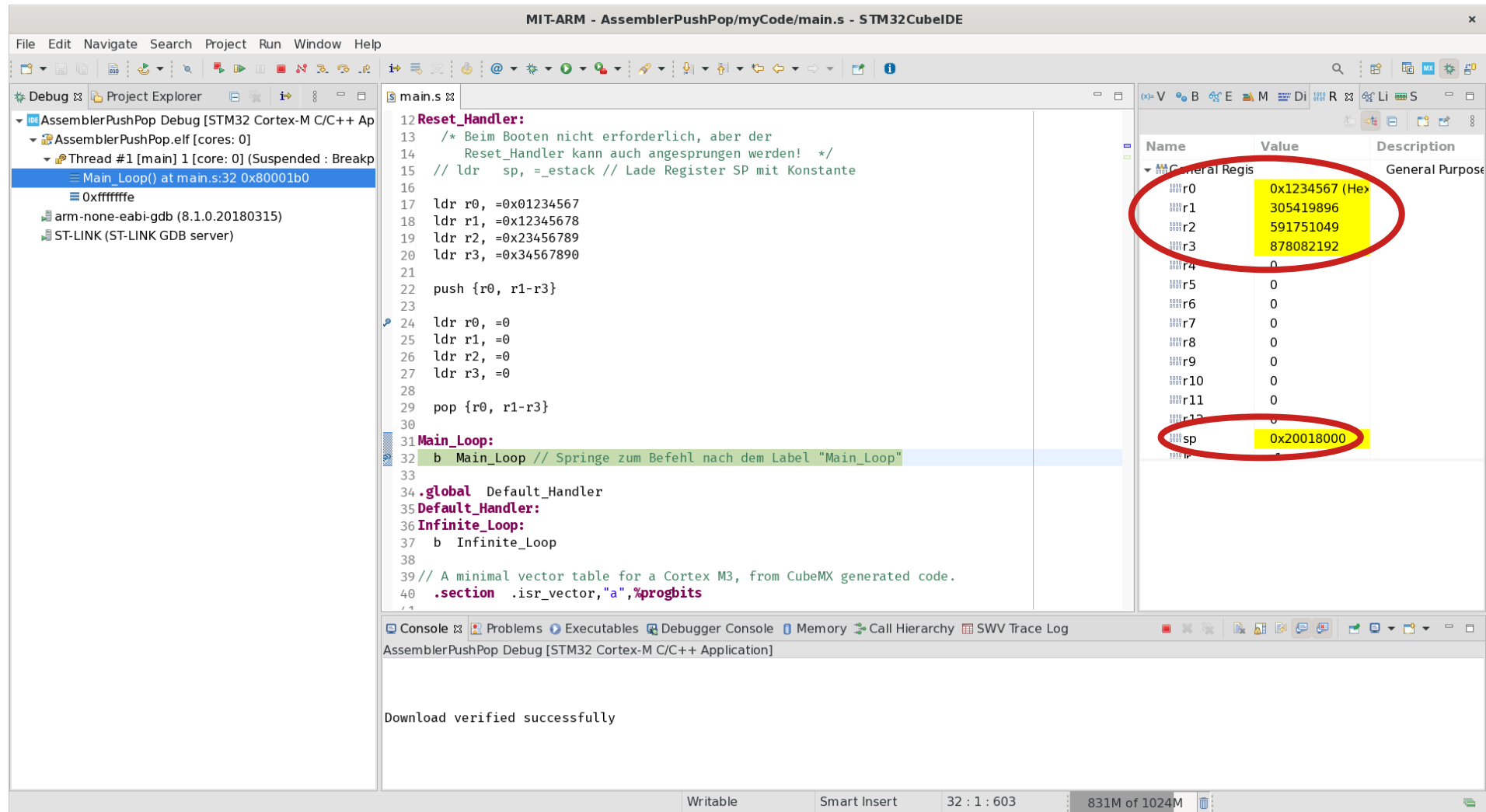
The screenshot shows the MIT-ARM IDE interface. The main window displays assembly code for a file named `main.s`. The code includes a `.thumb` directive, a `.section .text` directive, and a `Reset_Handler` function. The `Reset_Handler` function initializes registers `r0`, `r1`, `r2`, and `r3` with specific values and then pushes them onto the stack. The `Main_Loop` function is also defined, which branches to itself. The `Default_Handler` and `Infinite_Loop` are also shown.

On the right side, the 'General registers' table is visible. It lists registers `r0` through `r11` and the stack pointer `sp`. The values for `r0` and `sp` are circled in red. The value for `r0` is `0x1234567` (Hex) and the value for `sp` is `0x20017ff0`.

At the bottom, the 'Monitors' window shows memory addresses and their corresponding values. The address `0x20017ff0` is circled in red, and its value is `00000000`.

An orange oval with the text 'Live Coding' is positioned in the bottom right corner of the IDE window.

Stack Pointer



The screenshot shows the MIT-ARM IDE interface. The main window displays assembly code for a file named `main.s`. The code includes a `Reset_Handler` and a `Main_Loop`. The `Reset_Handler` initializes registers `r0`, `r1`, `r2`, and `r3` with specific values and then pushes them onto the stack. The `Main_Loop` is currently selected, showing a branch instruction `b Main_Loop`. On the right side, the 'General Register' window is open, displaying the current values of the registers. The values for `r0`, `r1`, `r2`, and `r3` are highlighted in yellow and circled in red. The stack pointer (`sp`) is also highlighted in yellow and circled in red.

Name	Value	Description
r0	0x1234567 (Hex)	General Purpose
r1	305419896	General Purpose
r2	591751049	General Purpose
r3	878082192	General Purpose
r4	0	General Purpose
r5	0	General Purpose
r6	0	General Purpose
r7	0	General Purpose
r8	0	General Purpose
r9	0	General Purpose
r10	0	General Purpose
r11	0	General Purpose
r12	0	General Purpose
sp	0x20018000	Stack Pointer

Stack Pointer

- **Push-Befehl**

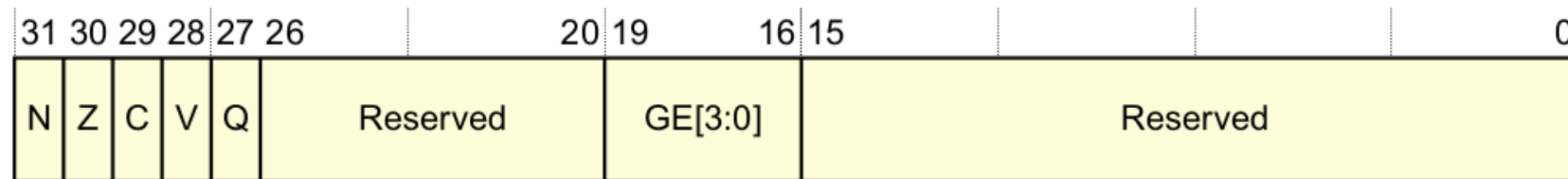
- bekommt als Argument die zu speichernden Register
- SP wird um die Anzahl benötigter Bytes vermindert
- Registerinhalte werden ab (vermindertem) SP in Speicher kopiert

- **Pop-Befehl**

- bekommt als Argument die wiederherzustellenden Register
- Werte im Speicher werden ab (aktuellem) SP in Register kopiert
- SP wird um die Anzahl ausgelesener Bytes erhöht

Statusregister

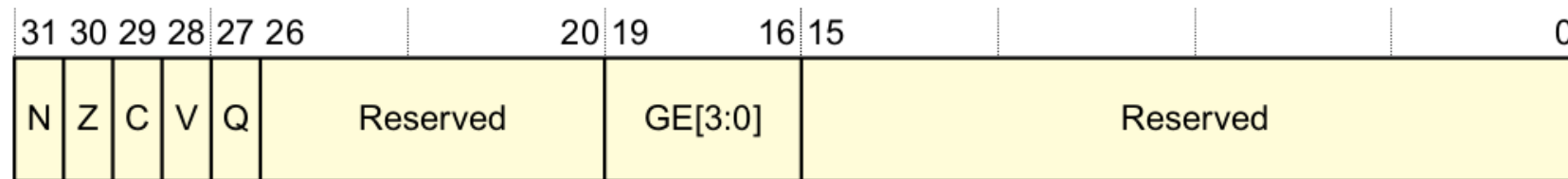
- **Präzise: Application Program Status Register (APSR)**



- N, bit[31]** Negative condition flag. Set to bit[31] of the result of the instruction. If the result is regarded as a two's complement signed integer, then $N == 1$ if the result is negative and $N == 0$ if it is positive or zero.
- Z, bit[30]** Zero condition flag. Set to 1 if the result of the instruction is zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.
- C, bit[29]** Carry condition flag. Set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition.

Statusregister

- **Präzise: Application Program Status Register (APSR)**



V, bit[28] Overflow condition flag. Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.

Q, bit[27] Set to 1 if a SSAT or USAT instruction changes the input value for the signed or unsigned range of the result. In a processor that implements the DSP extension, the processor sets this bit to 1 to indicate an overflow on some multiplies. Setting this bit to 1 is called saturation.

GE[3:0], bits[19:16], DSP extension only

Greater than or Equal flags. SIMD instructions update these flags to indicate the results from individual bytes or halfwords of the operation. Software can use these flags to control a later SEL instruction. For more information, see [SEL on page A7-351](#).

In a processor that does not implement the DSP extension these bits are reserved.

Statusregister

Der Werte in Status Register kann in General Purpose Register kopiert werden aber normalerweise machen wir das nicht sondern sie verwenden diese Bits in Branch Befehl

- **Aktualisierung der Flags im APSR ist für viele Befehle optional**
 - ADD ... bildet die Summe (zweier Register) ohne Aktualisierung
 - ADDS ... bildet die Summe mit Aktualisierung der Flags
- **APSR kann in ein General Purpose Register kopiert werden**
 - Es kann auch von einem GPR in das APSR kopiert werden
- **Üblicher Weise erfolgt die „Abfrage“ der Bits über die bedingten Sprunganweisungen „b<c>“**
 - Spezialität der ARM-Architektur: auch andere Anweisungen können bedingt, d.h. abhängig von den Bits im APSR ausgeführt werden

Statusregister

cond	Mnemonic extension	Meaning, integer arithmetic	Meaning, floating-point arithmetic ^a	Condition flags
0000	EQ	Equal	Equal	$Z == 1$
0001	NE	Not equal	Not equal, or unordered	$Z == 0$
0010	CS ^b	Carry set	Greater than, equal, or unordered	$C == 1$
0011	CC ^c	Carry clear	Less than	$C == 0$
0100	MI	Minus, negative	Less than	$N == 1$
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	$N == 0$
0110	VS	Overflow	Unordered	$V == 1$
0111	VC	No overflow	Not unordered	$V == 0$
1000	HI	Unsigned higher	Greater than, or unordered	$C == 1$ and $Z == 0$
1001	LS	Unsigned lower or same	Less than or equal	$C == 0$ or $Z == 1$
1010	GE	Signed greater than or equal	Greater than or equal	$N == V$
1011	LT	Signed less than	Less than, or unordered	$N != V$
1100	GT	Signed greater than	Greater than	$Z == 0$ and $N == V$
1101	LE	Signed less than or equal	Less than, equal, or unordered	$Z == 1$ or $N != V$
1110	None (AL) ^d	Always (unconditional)	Always (unconditional)	Any

Live Coding

Statusregister

Beispiel: Schleife

Hier sehen wir bei Register r0 wird der Wert 3 geladen

Dann wird es addiert und dann SubS gemacht, d.h. Subtrahiert mit setzen des Status Flag. Von der register r0 ist 1 subtrahiert und wieder in das r0 gelegt.

Dann kommt der Befehl bne -> branch if not equal d.h. branch wenn das Z(Zero) Flag nicht gesetzt ist d.h. wenn das subtrahieren ein andere Wert als 0 geliefert hat dann wird gebrancht. Das bedeutet ich führe das so oft aus bis in den Register r0 nach diese Subtraktion von 1 den Wert 0 steht. Wenn das subS 0 liefert dann ist der Zero Flag gesetzt und dann wird der Conditional Branch nicht wieder ausgeführt. Diese Branch hat nur ausgeführt bei NotEqual

The screenshot shows the STM32CubeIDE interface. The main editor displays assembly code for a file named `main.s`. The code includes a reset handler and a loop labeled `countDownLoop`. The `countDownLoop` starts by moving the value 3 into register `r0` and then enters a loop where it subtracts 1 from `r0` and branches back to the start of the loop if not equal (bne). The register window on the right shows the current state of the registers. Register `r0` contains the value 3 (0x3), while all other registers are 0. The stack pointer (`sp`) is at 0x20018000.

```

8  .thumb
9
10 .section .text
11
12 Reset_Handler:
13  /* Beim Booten nicht erforderlich, aber der
14   Reset_Handler kann auch angesprungen werden! */
15  // ldr sp, =_estack // Lade Register SP mit Konstante
16
17  mov r0, 3
18  mov r1, 0
19
20 countDownLoop:
21  add r1, r1, 1
22  subS r0, r0, 1
23  bne countDownLoop
24
25 Main_Loop:
26  b Main_Loop // Springe zum Befehl nach dem Label "Main_Loop"
27
28 .global Default_Handler
29 Default_Handler:
30 Infinite_Loop:
31  b Infinite_Loop
32
33 // A minimal vector table for a Cortex M3, from CubeMX generated code.
34 .section .isr_vector,"a",%progbits
35 .word _estack
36 .word Reset_Handler
  
```

Name	Value	Description
General Regis		
r0	0x3 (Hex)	General Purpose
r1	0	
r2	0	
r3	0	
r4	0	
r5	0	
r6	0	
r7	0	
r8	0	
r9	0	
r10	0	
r11	0	
r12	0	
sp	0x20018000	

Console output: Download verified successfully