Praktikum Mikrocomputertechnik (EIT - B3, MEC - B5)
Versuch 03

Version: 2024-10-25a

Inhaltsverzeichnis

1 Vorbemerkung

Das Programmieren in Assembler hat viel mit einem Festrausch zu tun:
Es ist eine zähe Sache, man bekommt davon brutal Schädlweh und es kann einem bis zum Erbrechen schlecht werden.
Trotzdem ist der überwiegende Teil der Menschheit der Meinung, man sollte diese Erfahrung einmal gemacht haben; gewährt es einem doch tiefe Einblicke in innere Zusammenhänge.

2 Aufgabenstellung

1. Kennenlernen der Arbeitsweise des Debuggers mit Hilfe eines kleinen Assembler Projekts. Siehe 3.6.
2. Einlesen des Werts eines Tasters; Ein- und Ausschalten von LEDs in Abhängigkeit des Tasters. Siehe 3.7.
3. Erzeugen eines Lauflichts auf der Schrittmotorplatine. Siehe 3.8.

3 Informationssammlung

3.1 Schaltplan der MSP-Platine

Aus dem Schaltplan erkennt man,

- bei Betätigung des Schalters **S2** wird der Pin **P1.3** auf GND gezogen
- **LED1** ist mit Pin **P1.0** über J5-1 und R32 (270 Ohm) verbunden
- **LED2** ist mit Pin **P1.6** über J5-2 und R33 (470 Ohm) verbunden

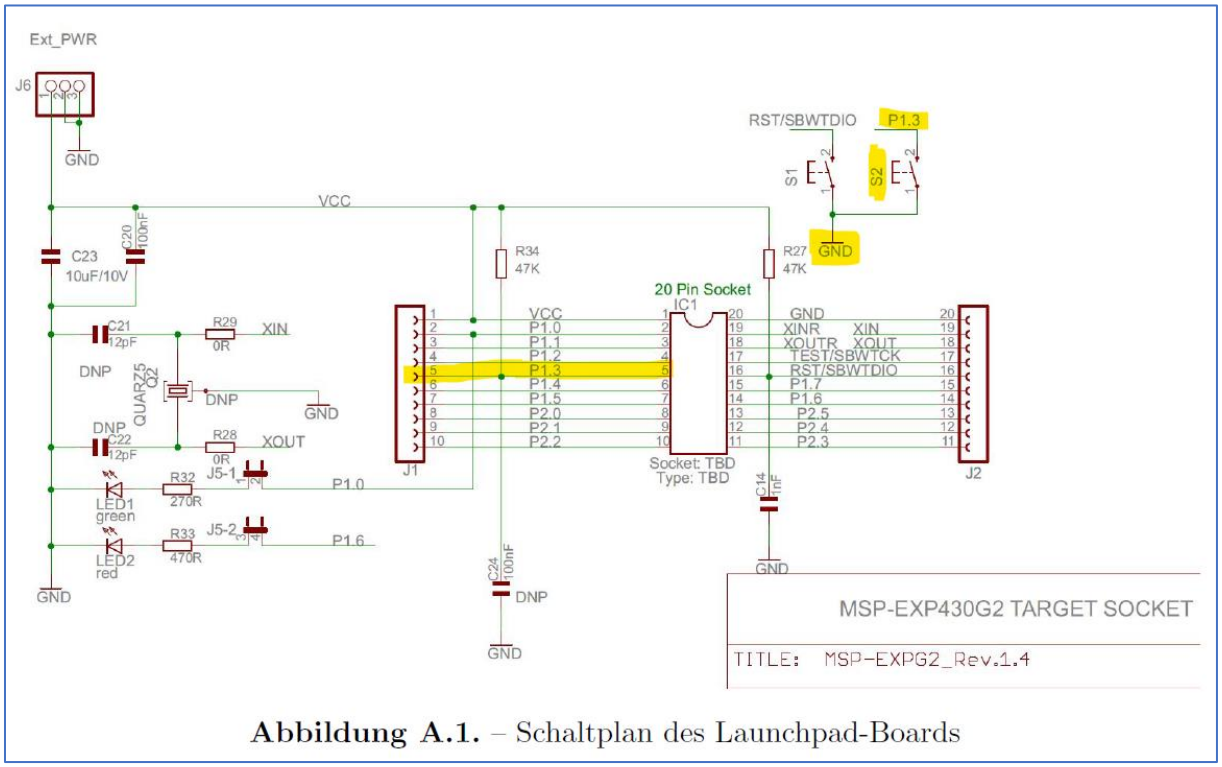


Abbildung A.1. – Schaltplan des Launchpad-Boards

Mikrocomputertechnik_Praktikum_10052022(1).pdf

3.2 Das Statusregister (SR)

Für dieses Praktikum müssen wird die Bedeutung der Bits V, N, Z und C des Statusregisters kennen:

3.2.3 Status Register (SR)

The status register (SR/R2), used as a source or destination register, can be used in the register mode only addressed with word instructions. The remaining combinations of addressing modes are used to support the constant generator. Figure 3-6 shows the SR bits.

Figure 3-6. Status Register Bits

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved							V	SCG1	SCG0	OSC OFF	CPU OFF	GIE	N	Z	C
rw-0							rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0

Table 3-1 describes the status register bits.

Table 3-1. Description of Status Register Bits

Bit	Description	
V	Overflow bit. This bit is set when the result of an arithmetic operation overflows the signed-variable range.	
	ADD(.B), ADDC(.B)	Set when:
		Positive + Positive = Negative
		Negative + Negative = Positive
		Otherwise reset
	SUB(.B), SUBC(.B), CMP(.B)	Set when:
		Positive – Negative = Negative
		Negative – Positive = Positive
		Otherwise reset
SCG1	System clock generator 1. When set, turns off the SMCLK.	
SCG0	System clock generator 0. When set, turns off the DCO dc generator, if DCOCLK is not used for MCLK or SMCLK.	
OSCOFF	Oscillator Off. When set, turns off the LFXT1 crystal oscillator, when LFXT1CLK is not use for MCLK or SMCLK.	
CPUOFF	CPU off. When set, turns off the CPU.	
GIE	General interrupt enable. When set, enables maskable interrupts. When reset, all maskable interrupts are disabled.	
N	Negative bit. Set when the result of a byte or word operation is negative and cleared when the result is not negative.	
	Word operation: N is set to the value of bit 15 of the result.	
	Byte operation: N is set to the value of bit 7 of the result.	
Z	Zero bit. Set when the result of a byte or word operation is 0 and cleared when the result is not 0.	
C	Carry bit. Set when the result of a byte or word operation produced a carry and cleared when no carry occurred.	

Quelle: Slau144k.pdf

3.3 Instruction Set

Quelle: Slau144k.pdf – Kapitel 3.4

Instruction Set

The complete MSP430 instruction set consists of 27 core instructions and 24 emulated instructions. The core instructions are instructions that have unique op-codes decoded by the CPU. The emulated instructions are instructions that make code easier to write and read, but do not have op-codes themselves, instead they are replaced automatically by the assembler with an equivalent core instruction. There is no code or performance penalty for using emulated instruction.

There are three core-instruction formats:

- Dual-operand
- Single-operand
- Jump

All single-operand and dual-operand instructions can be byte or word instructions by using .B or .W extensions. Byte instructions are used to access byte data or byte peripherals. Word instructions are used to access word data or word peripherals. If no extension is used, the instruction is a word instruction.

The source and destination of an instruction are defined by the following fields:

src	The source operand defined by As and S-reg
dst	The destination operand defined by Ad and D-reg
As	The addressing bits responsible for the addressing mode used for the source (src)
S-reg	The working register used for the source (src)
Ad	The addressing bits responsible for the addressing mode used for the destination (dst)
D-reg	The working register used for the destination (dst)
B/W	Byte or word operation: 0: word operation 1: byte operation

NOTE: Destination Address

Destination addresses are valid anywhere in the memory map. However, when using an instruction that modifies the contents of the destination, the user must ensure the destination address is writable. For example, a masked-ROM location would be a valid destination address, but the contents are not modifiable, so the results of the instruction would be lost.

Table 3-17. MSP430 Instruction Set

Mnemonic		Description		V	N	Z	C
ADC(.B) (1)	dst	Add C to destination	dst + C → dst	*	*	*	*
ADD(.B)	src,dst	Add source to destination	src + dst → dst	*	*	*	*
ADDC(.B)	src,dst	Add source and C to destination	src + dst + C → dst	*	*	*	*
AND(.B)	src,dst	AND source and destination	src .and. dst → dst	0	*	*	*
BIC(.B)	src,dst	Clear bits in destination	not.src .and. dst → dst	-	-	-	-
BIS(.B)	src,dst	Set bits in destination	src .or. dst → dst	-	-	-	-
BIT(.B)	src,dst	Test bits in destination	src .and. dst	0	*	*	*
BR (1)	dst	Branch to destination	dst → PC	-	-	-	-
CALL	dst	Call destination	PC+2 → stack, dst → PC	-	-	-	-
CLR(.B) (1)	dst	Clear destination	0 → dst	-	-	-	-
CLRC (1)		Clear C	0 → C	-	-	-	0
CLRN (1)		Clear N	0 → N	-	0	-	-
CLRZ (1)		Clear Z	0 → Z	-	-	0	-
CMP(.B)	src,dst	Compare source and destination	dst - src	*	*	*	*
DADC(.B) (1)	dst	Add C decimally to destination	dst + C → dst (decimally)	*	*	*	*
DADD(.B)	src,dst	Add source and C decimally to dst	src + dst + C → dst (decimally)	*	*	*	*
DEC(.B) (1)	dst	Decrement destination	dst - 1 → dst	*	*	*	*

Mnemonic		Description		V	N	Z	C
DECD(.B) (1)	dst	Double-decrement destination	dst - 2 → dst	*	*	*	*
DINT (1)		Disable interrupts	0 → GIE	-	-	-	-
EINT (1)		Enable interrupts	1 → GIE	-	-	-	-
INC(.B) (1)	dst	Increment destination	dst +1 → dst	*	*	*	*
INCD(.B) (1)	dst	Double-increment destination	dst+2 → dst	*	*	*	*
INV(.B) (1)	dst	Invert destination	.not.dst → dst	*	*	*	*
JC/JHS	label	Jump if C set/Jump if higher or same		-	-	-	-
JEQ/JZ	label	Jump if equal/Jump if Z set		-	-	-	-
JGE	label	Jump if greater or equal		-	-	-	-
JL	label	Jump if less		-	-	-	-
JMP	label	Jump	PC + 2 × offset → PC	-	-	-	-
JN	label	Jump if N set		-	-	-	-
JNC/JLO	label	Jump if C not set/Jump if lower		-	-	-	-
JNE/JNZ	label	Jump if not equal/Jump if Z not set		-	-	-	-
MOV(.B)	src,dst	Move source to destination	src → dst	-	-	-	-
NOP (2)		No operation		-	-	-	-
POP(.B) (2)	dst	Pop item from stack to destination	@SP → dst, SP+2 → SP	-	-	-	-
PUSH(.B)	src	Push source onto stack	SP - 2 → SP, src → @SP	-	-	-	-
RET (2)		Return from subroutine	@SP → PC, SP + 2 → SP	-	-	-	-
RETI		Return from interrupt		*	*	*	*
RLA(.B) (2)	dst	Rotate left arithmetically		*	*	*	*
RLC(.B) (2)	dst	Rotate left through C		*	*	*	*
RRA(.B)	dst	Rotate right arithmetically		0	*	*	*
RRC(.B)	dst	Rotate right through C		*	*	*	*
SBC(.B) (2)	dst	Subtract not(C) from destination	dst + 0FFFFh + C → dst	*	*	*	*
SETC (2)		Set C	1 → C	-	-	-	1
SETN (2)		Set N	1 → N	-	1	-	-
SETZ (2)		Set Z	1 → Z	-	-	1	-
SUB(.B)	src,dst	Subtract source from destination	dst + .not.src + 1 → dst	*	*	*	*
SUBC(.B)	src,dst	Subtract source and not(C) from dst	dst + .not.src + C → dst	*	*	*	*
SWPB	dst	Swap bytes		-	-	-	-
SXT	dst	Extend sign		0	*	*	*
TST(.B) (2)	dst	Test destination	dst + 0FFFFh + 1	0	*	*	1
XOR(.B)	src,dst	Exclusive OR source and destination	src .xor. dst → dst	*	*	*	*

(2) Emulated Instruction

3.4.1 *Double-Operand (Format I) Instructions*

Figure 3-9 illustrates the double-operand instruction format.

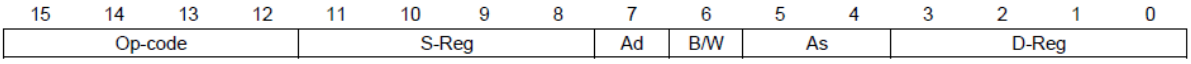


Figure 3-9. Double Operand Instruction Format

Table 3-11 lists and describes the double operand instructions.

Table 3-11. Double Operand Instructions

Mnemonic	S-Reg, D-Reg	Operation	Status Bits			
			V	N	Z	C
MOV(.B)	src,dst	src → dst	-	-	-	-
ADD(.B)	src,dst	src + dst → dst	*	*	*	*
ADDC(.B)	src,dst	src + dst + C → dst	*	*	*	*
SUB(.B)	src,dst	dst + .not.src + 1 → dst	*	*	*	*
SUBC(.B)	src,dst	dst + .not.src + C → dst	*	*	*	*
CMP(.B)	src,dst	dst - src	*	*	*	*
DADD(.B)	src,dst	src + dst + C → dst (decimally)	*	*	*	*
BIT(.B)	src,dst	src .and. dst	0	*	*	*
BIC(.B)	src,dst	not.src .and. dst → dst	-	-	-	-
BIS(.B)	src,dst	src .or. dst → dst	-	-	-	-
XOR(.B)	src,dst	src .xor. dst → dst	*	*	*	*
AND(.B)	src,dst	src .and. dst → dst	0	*	*	*

* The status bit is affected
– The status bit is not affected
0 The status bit is cleared
1 The status bit is set

NOTE: Instructions CMP and SUB

The instructions CMP and SUB are identical except for the storage of the result. The same is true for the BIT and AND instructions.

3.4.2 Single-Operand (Format II) Instructions

Figure 3-10 illustrates the single-operand instruction format.

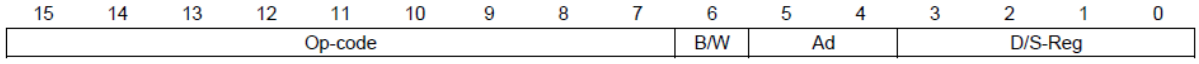


Figure 3-10. Single Operand Instruction Format

Table 3-12 lists and describes the single operand instructions.

Table 3-12. Single Operand Instructions

Mnemonic	S-Reg, D-Reg	Operation	Status Bits			
			V	N	Z	C
RRC(.B)	dst	C → MSB →.....LSB → C	*	*	*	*
RRA(.B)	dst	MSB → MSB →....LSB → C	0	*	*	*
PUSH(.B)	src	SP – 2 → SP, src → @SP	-	-	-	-
SWPB	dst	Swap bytes	-	-	-	-
CALL	dst	SP – 2 → SP, PC+2 → @SP dst → PC	-	-	-	-
RETI		TOS → SR, SP + 2 → SP TOS → PC,SP + 2 → SP	*	*	*	*
SXT	dst	Bit 7 → Bit 8.....Bit 15	0	*	*	*

*	The status bit is affected
–	The status bit is not affected
0	The status bit is cleared
1	The status bit is set

All addressing modes are possible for the CALL instruction. If the symbolic mode (ADDRESS), the immediate mode (#N), the absolute mode (&EDE) or the indexed mode x(RN) is used, the word that follows contains the address information.

3.4.3 Jumps

Figure 3-11 shows the conditional-jump instruction format.

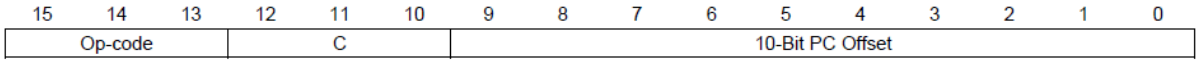


Figure 3-11. Jump Instruction Format

Table 3-13 lists and describes the jump instructions

Table 3-13. Jump Instructions

Mnemonic	S-Reg, D-Reg	Operation
JEQ/JZ	Label	Jump to label if zero bit is set
JNE/JNZ	Label	Jump to label if zero bit is reset
JC	Label	Jump to label if carry bit is set
JNC	Label	Jump to label if carry bit is reset
JN	Label	Jump to label if negative bit is set
JGE	Label	Jump to label if (N .XOR. V) = 0
JL	Label	Jump to label if (N .XOR. V) = 1
JMP	Label	Jump to label unconditionally

Conditional jumps support program branching relative to the PC and do not affect the status bits. The possible jump range is from –511 to +512 words relative to the PC value at the jump instruction. The 10-bit program-counter offset is treated as a signed 10-bit value that is doubled and added to the program counter:

$PC_{new} = PC_{old} + 2 + PC_{offset} \times 2$

3.4 Addressing Modes

Table 3-3. Source/Destination Operand Addressing Modes

As/Ad	Addressing Mode	Syntax	Description
00/0	Register mode	Rn	Register contents are operand
01/1	Indexed mode	X(Rn)	(Rn + X) points to the operand. X is stored in the next word.
01/1	Symbolic mode	ADDR	(PC + X) points to the operand. X is stored in the next word. Indexed mode X(PC) is used.
01/1	Absolute mode	&ADDR	The word following the instruction contains the absolute address. X is stored in the next word. Indexed mode X(SR) is used.
10/-	Indirect register mode	@Rn	Rn is used as a pointer to the operand.
11/-	Indirect autoincrement	@Rn+	Rn is used as a pointer to the operand. Rn is incremented afterwards by 1 for .B instructions and by 2 for .W instructions.
11/-	Immediate mode	#N	The word following the instruction contains the immediate constant N. Indirect autoincrement mode @PC+ is used.

The seven addressing modes are explained in detail in the following sections. Most of the examples show the same addressing mode for the source and destination, but any valid combination of source and destination addressing modes is possible in an instruction.

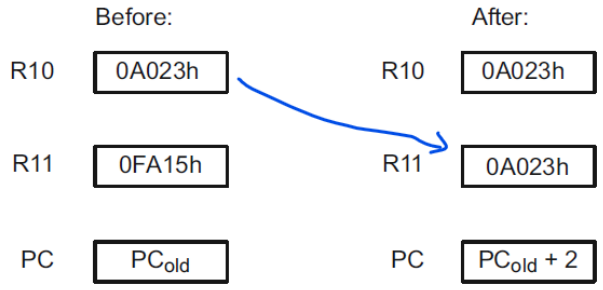
3.3.1 Register Mode

The register mode is described in Table 3-4.

Table 3-4. Register Mode Description

Assembler Code	Content of ROM
`MOV R10,R11`	`MOV R10,R11`

Length: One or two words
Operation: Move the content of R10 to R11. R10 is not affected.
Comment: Valid for source and destination
Example: `MOV R10,R11`



NOTE: Data in Registers

The data in the register can be accessed using word or byte instructions. If byte instructions are used, the high byte is always 0 in the result. The status bits are handled according to the result of the byte instructions.

Indexed Mode

The indexed mode is described in Table 3-5.

Table 3-5. Indexed Mode Description

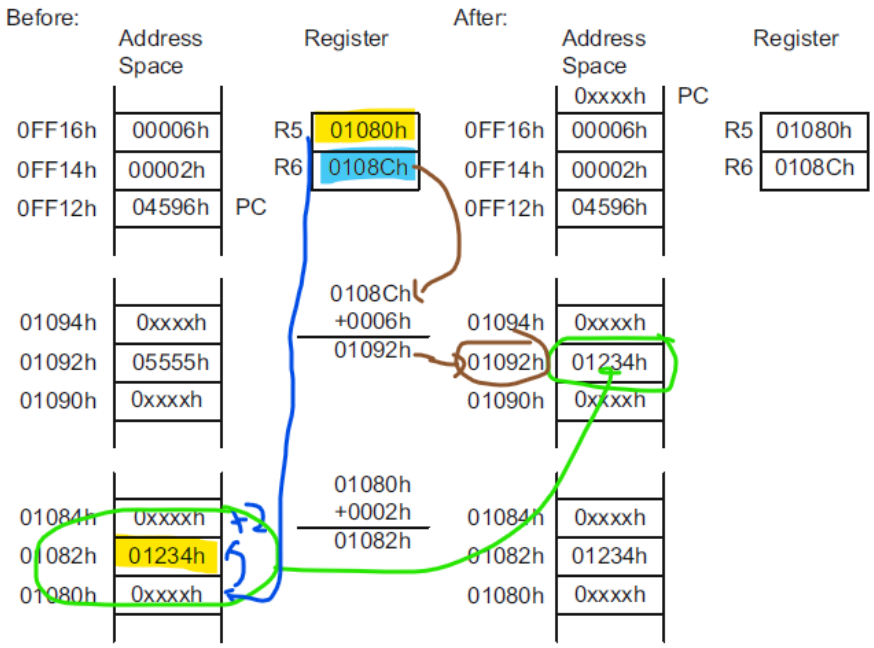
Assembler Code	Content of ROM
`MOV 2(R5),6(R6)`	`MOV X(R5),Y(R6)` X = 2 Y = 6

Length: Two or three words

Operation: Move the contents of the source address (contents of R5 + 2) to the destination address (contents of R6 + 6). The source and destination registers (R5 and R6) are not affected. In indexed mode, the program counter is incremented automatically so that program execution continues with the next instruction.

Comment: Valid for source and destination

Example: `MOV 2(R5),6(R6);`



Symbolic Mode

The symbolic mode is described in Table 3-6.

Table 3-6. Symbolic Mode Description

Assembler Code	Content of ROM
`MOV EDE,TONI`	`MOV X(PC),Y(PC)` X = EDE – PC Y = TONI – PC

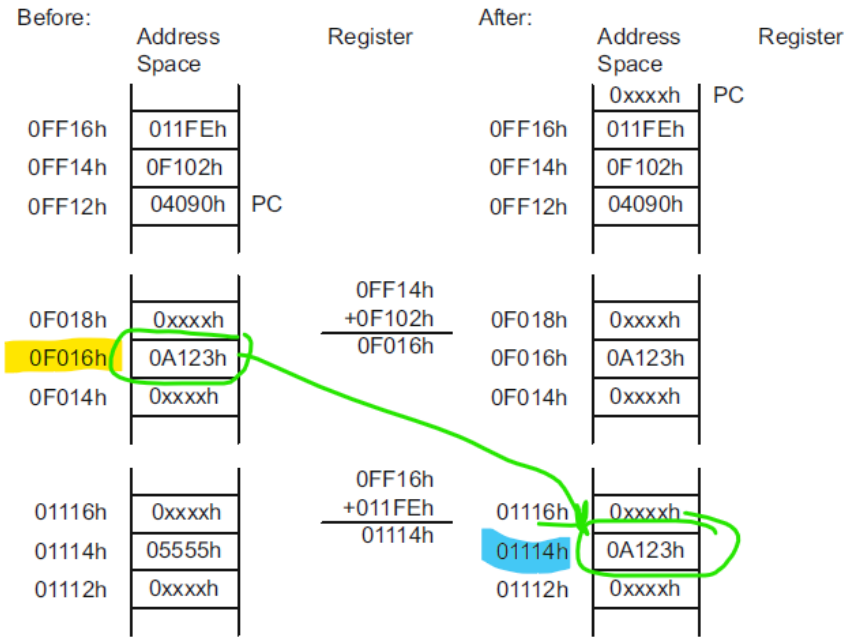
Length: Two or three words

Operation: Move the contents of the source address EDE (contents of PC + X) to the destination address TONI (contents of PC + Y). The words after the instruction contain the differences between the PC and the source or destination addresses. The assembler computes and inserts offsets X and Y automatically. With symbolic mode, the program counter (PC) is incremented automatically so that program execution continues with the next instruction.

Comment: Valid for source and destination

Example:

```
MOV  EDE,TONI  ;Source address EDE = 0F016h
               ;Dest. address TONI = 01114h
```



Absolute Mode

The absolute mode is described in Table 3-7.

Table 3-7. Absolute Mode Description

Assembler Code	Content of ROM
`MOV &EDE,&TONI`	`MOV X(0),Y(0)` X = EDE Y = TONI

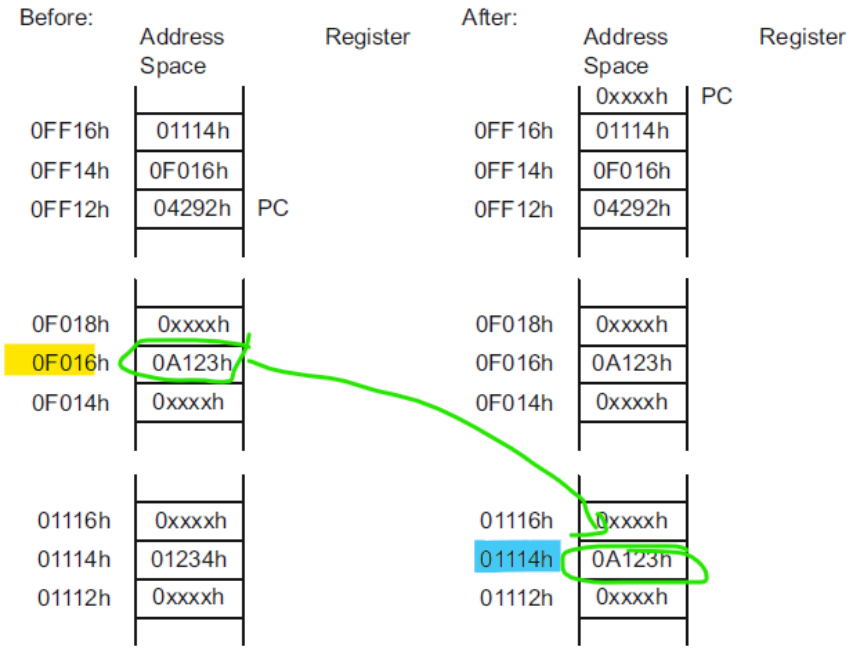
Length: Two or three words

Operation: Move the contents of the source address EDE to the destination address TONI. The words after the instruction contain the absolute address of the source and destination addresses. With absolute mode, the PC is incremented automatically so that program execution continues with the next instruction.

Comment: Valid for source and destination

Example:

```
MOV   &EDE,&TONI   ;Source address EDE = 0F016h
                   ;Dest. address TONI = 01114h
```



Indirect Register Mode

The indirect register mode is described in Table 3-8.

Table 3-8. Indirect Mode Description

Assembler Code	Content of ROM
MOV @R10,0(R11)	MOV @R10,0(R11)

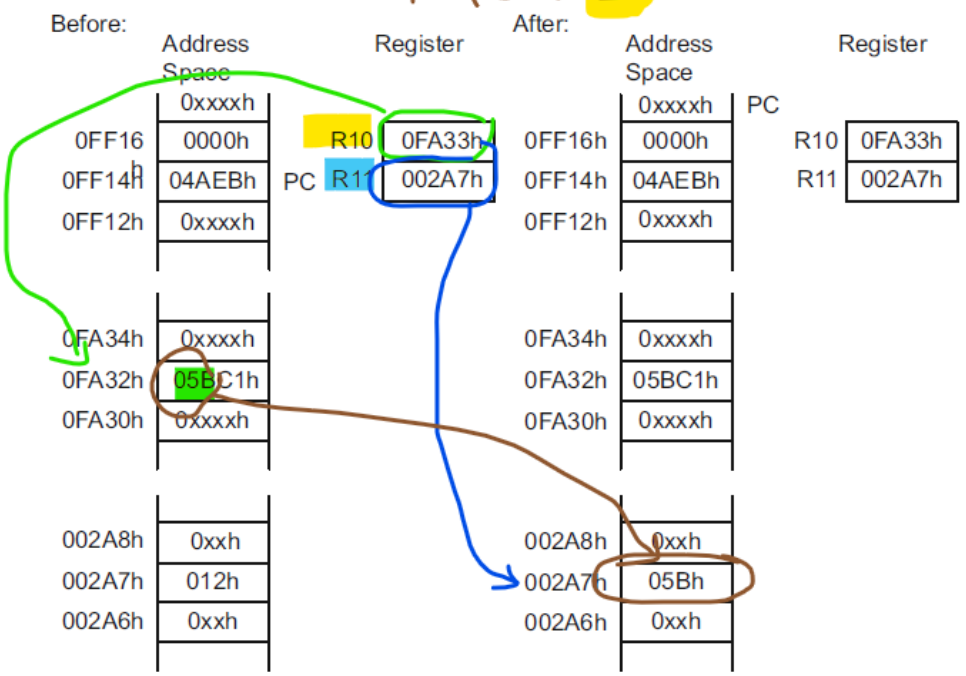
Length: One or two words

Operation: Move the contents of the source address (contents of R10) to the destination address (contents of R11). The registers are not modified.

Comment: Valid only for source operand. The substitute for destination operand is 0(Rd).

Example: MOV.B @R10,0(R11)

MOV.B



Indirect Autoincrement Mode

The indirect autoincrement mode is described in Table 3-9.

Table 3-9. Indirect Autoincrement Mode Description

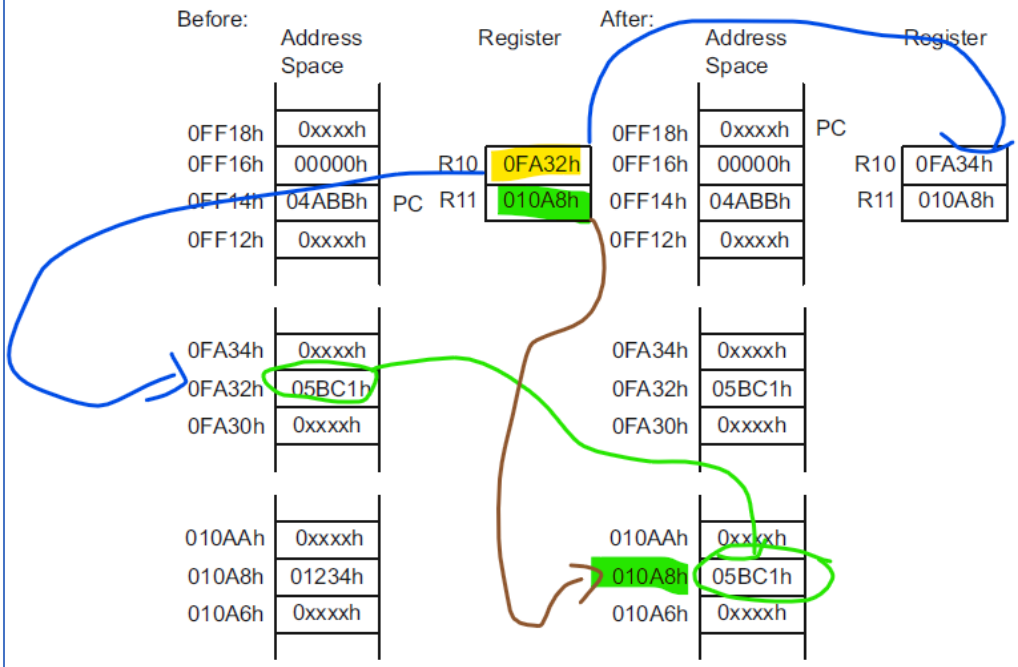
Assembler Code	Content of ROM
`MOV @R10+,0(R11)`	`MOV @R10+,0(R11)`

Length: One or two words

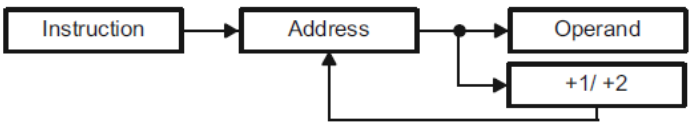
Operation: Move the contents of the source address (contents of R10) to the destination address (contents of R11). Register R10 is incremented by 1 for a byte operation, or 2 for a word operation after the fetch; it points to the next address without any overhead. This is useful for table processing.

Comment: Valid only for source operand. The substitute for destination operand is 0(Rd) plus second instruction INCD Rd.

Example: `MOV @R10+,0(R11)`



The auto-incrementing of the register contents occurs after the operand is fetched. This is shown in Figure 3-8.



Immediate Mode

The immediate mode is described in Table 3-10.

Table 3-10. Immediate Mode Description

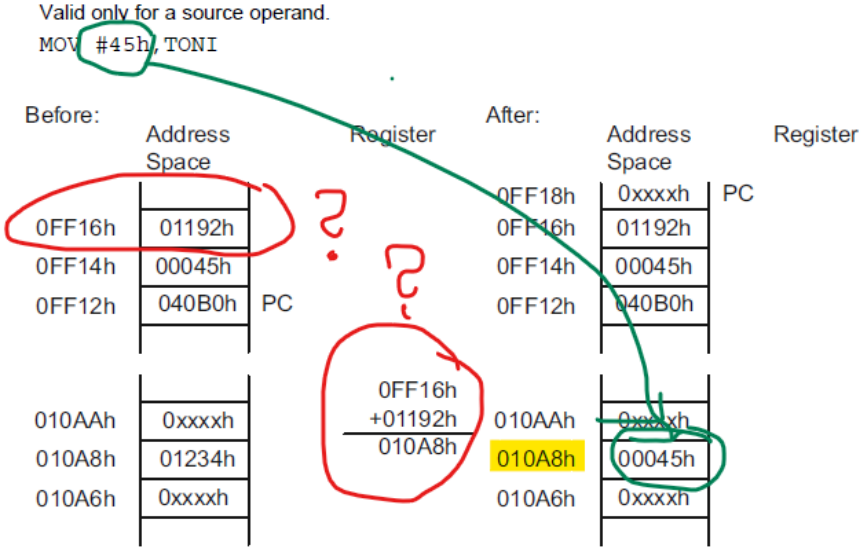
Assembler Code	Content of ROM
MOV #45h,TONI	MOV @PC+,X(PC) 45 X = TONI – PC

Length: Two or three words
It is one word less if a constant of CG1 or CG2 can be used.

Operation: Move the immediate constant 45h, which is contained in the word following the instruction, to destination address TONI. When fetching the source, the program counter points to the word following the instruction and moves the contents to the destination.

Comment: Valid only for a source operand.

Example: MOV #45h,TONI



3.5 BIT (Test Bits in destination)

3.4.6.7 BIT

BIT[.W]	Test bits in destination
BIT.B	Test bits in destination
Syntax	`BIT src,dst  or  BIT.W src,dst`
Operation	src .AND. dst
Description	The source and destination operands are logically ANDed. The result affects only the status bits. The source and destination operands are not affected.
Status Bits	N: Set if MSB of result is set, reset otherwise Z: Set if result is zero, reset otherwise C: Set if result is not zero, reset otherwise (.NOT. Zero) V: Reset
Mode Bits	OSCOFF, CPUOFF, and GIE are not affected.

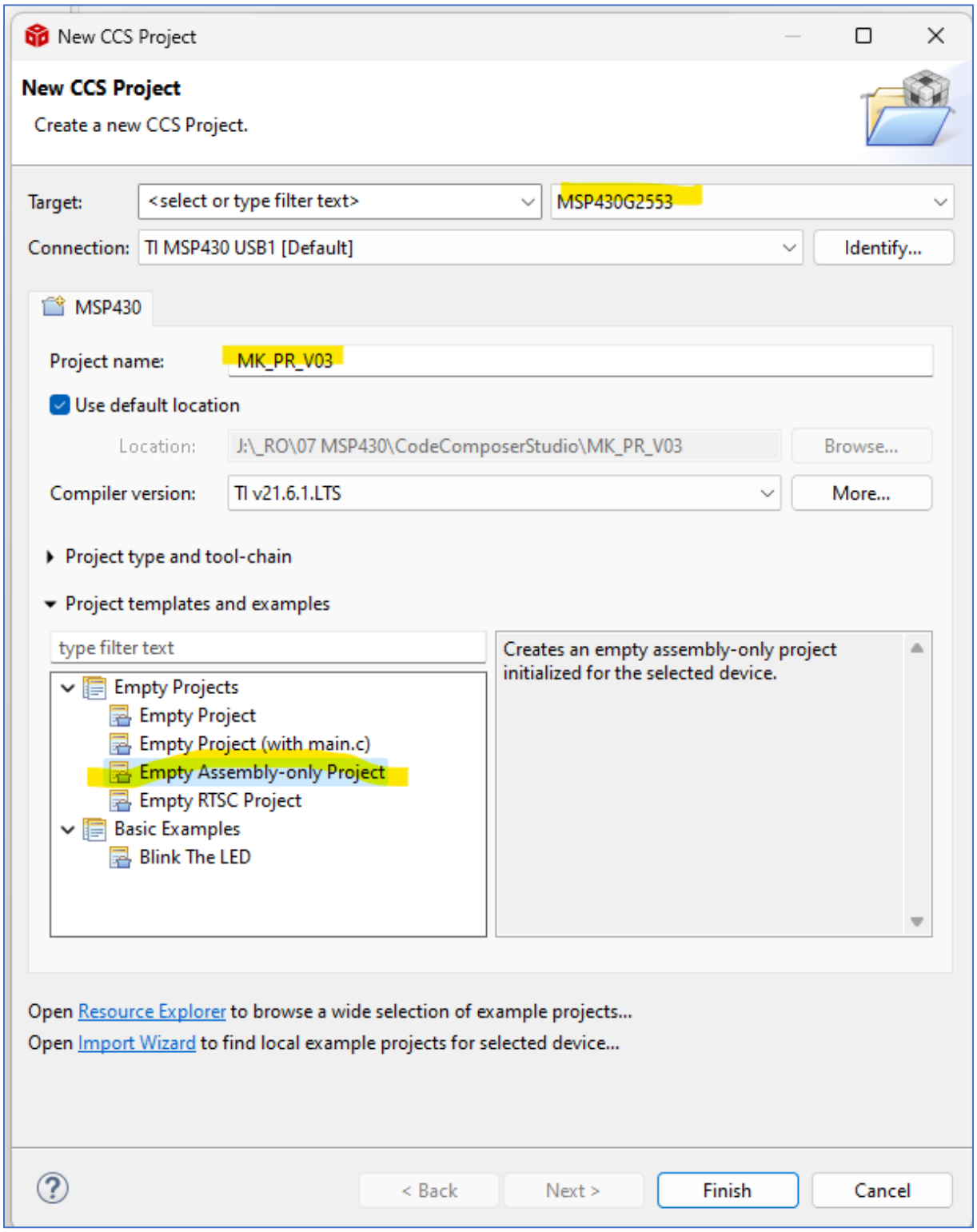
Example	If bit 9 of R8 is set, a branch is taken to label TOM.
	`BIT  #0200h,R8  ; bit 9 of R8 set?`
	`JNZ  TOM  ; Yes, branch to TOM`
	`...  ; No, proceed`

Example	If bit 3 of R8 is set, a branch is taken to label TOM.
	`BIT.B  #8,R8`
	`JC  TOM`

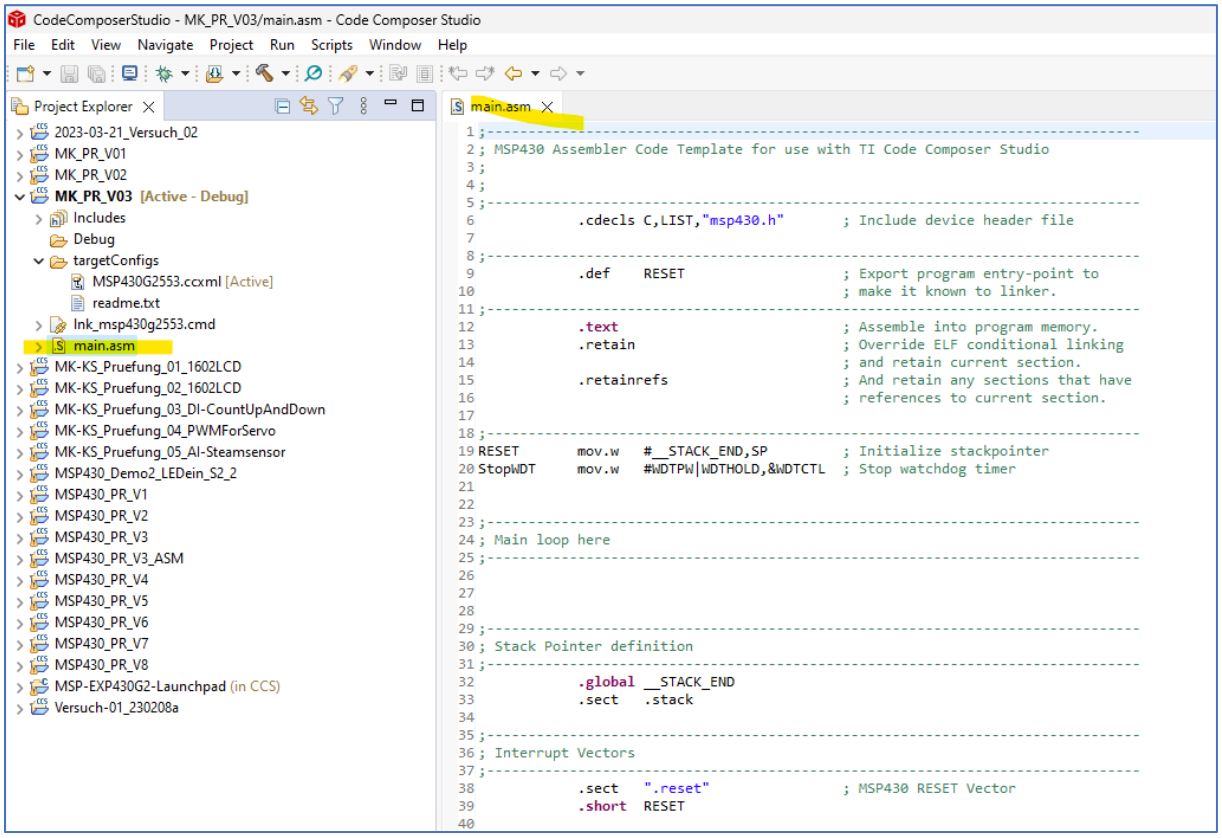
3.6 Aufgabe 01: Kennenlernen des Debuggers

3.6.1 Erzeugen eines Assembler Projekt in CCS

Über das Dialogfeld New CCS Project …



.. erzeugen wir uns ein vorkonfiguriertes Projekt:



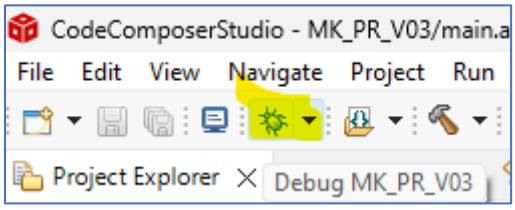
3.6.2 Erweitern des Quellcode in main.asm

Fügen Sie im Main loop Code zur Ansteuerung der beiden LEDs ein:

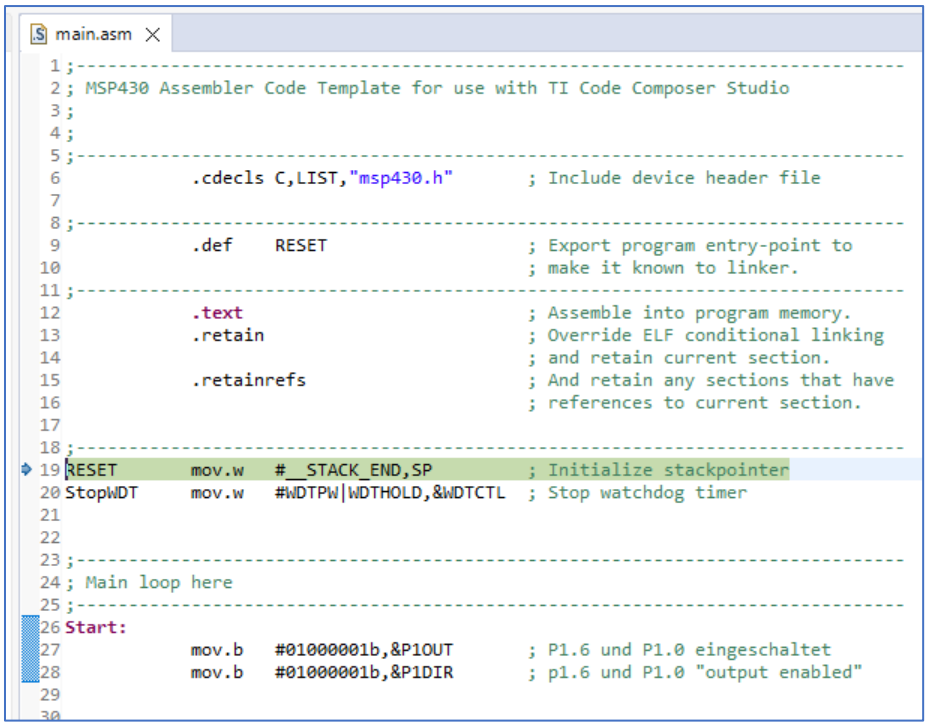
```
;-------------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------------
Start:
            mov.b   #01000001b,&P1OUT       ; P1.6 und P1.0 eingeschaltet
            mov.b   #01000001b,&P1DIR       ; p1.6 und P1.0 "output enabled"

```

3.6.3 Untersuchungen mit dem Debugger

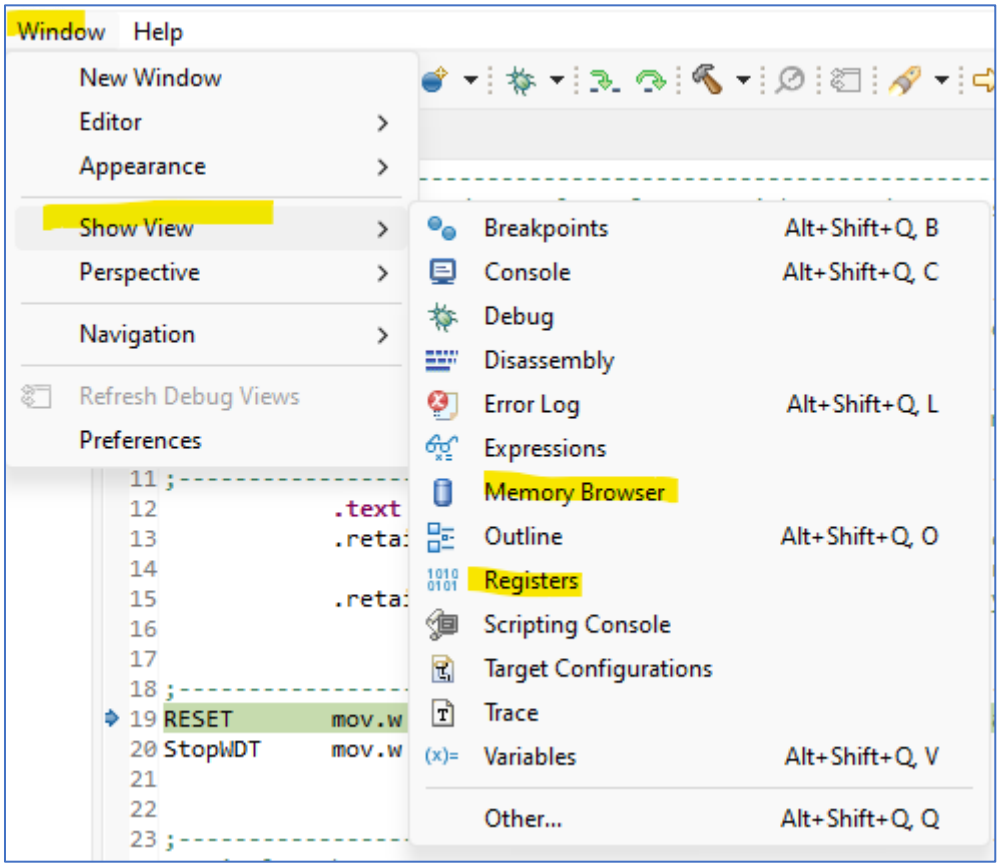
Starten Sie den Debugger durch Klicken auf die Laus:



Das Programm wird jetzt kompiliert und auf des LaunchPad geladen. Die Programmausführung steht in Zeile 19:

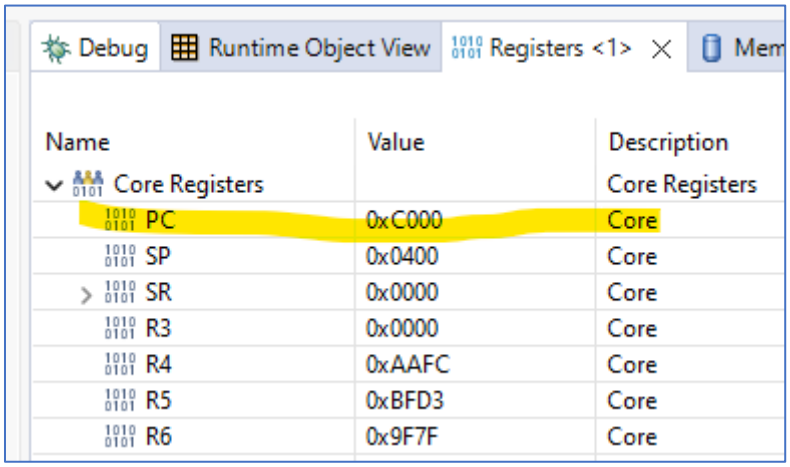


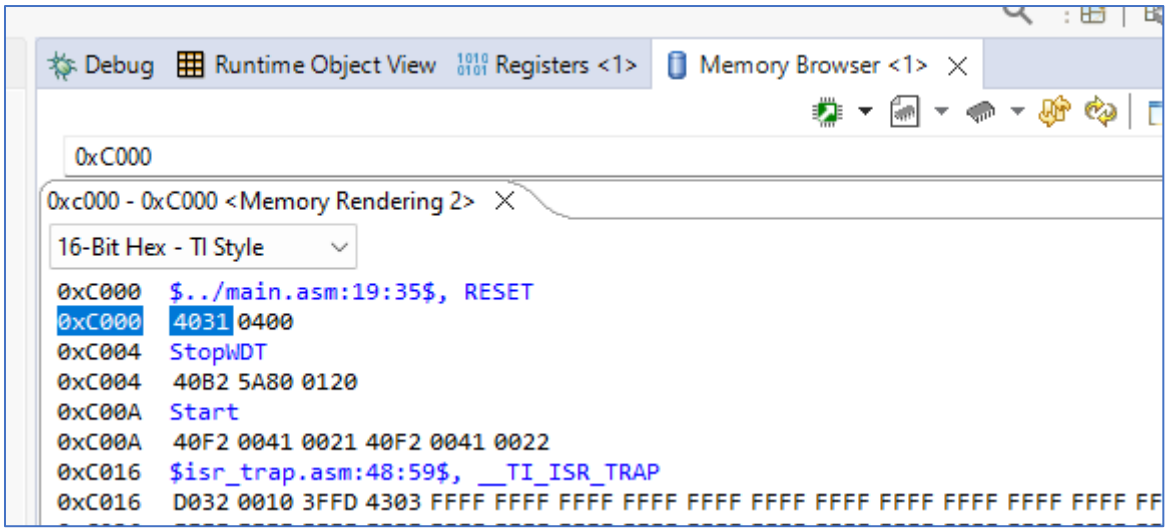
Der Debugger stellt verschiedene Fenster zu Verfügung.
Zum Öffnen von Fenstern wählen Sie im Menü **Window – Show View**:



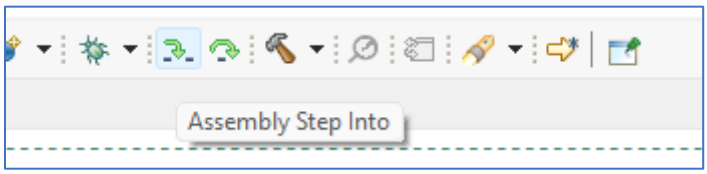
Voll lässig: Mit Hilfe des **Memory Browsers** können Sie den Inhalt des Speichers des MSPs ansehen, und zum Teil sogar manipulieren.
Starten Sie bei **Adresse 0** und sehen sie sich ein bißchen im Speicher um:

Der Inhalt des Registers **PC** verrät uns, ab welcher Stelle im Speicher der Programmcode beginnt:

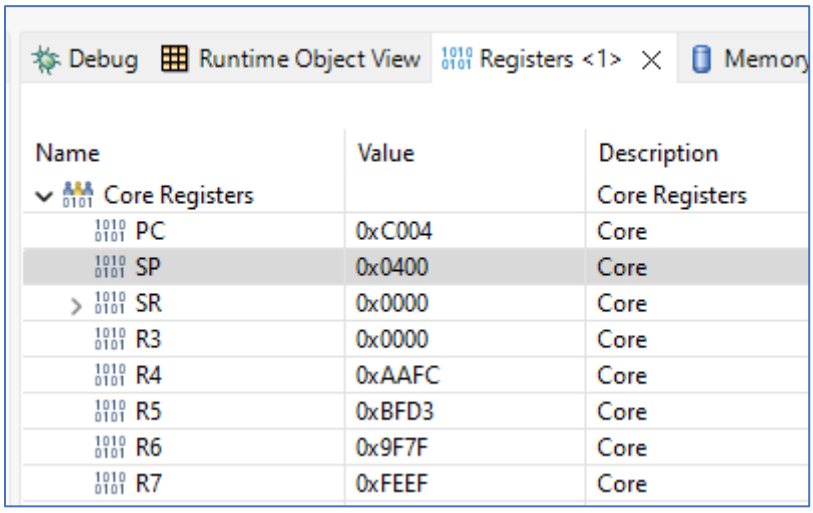




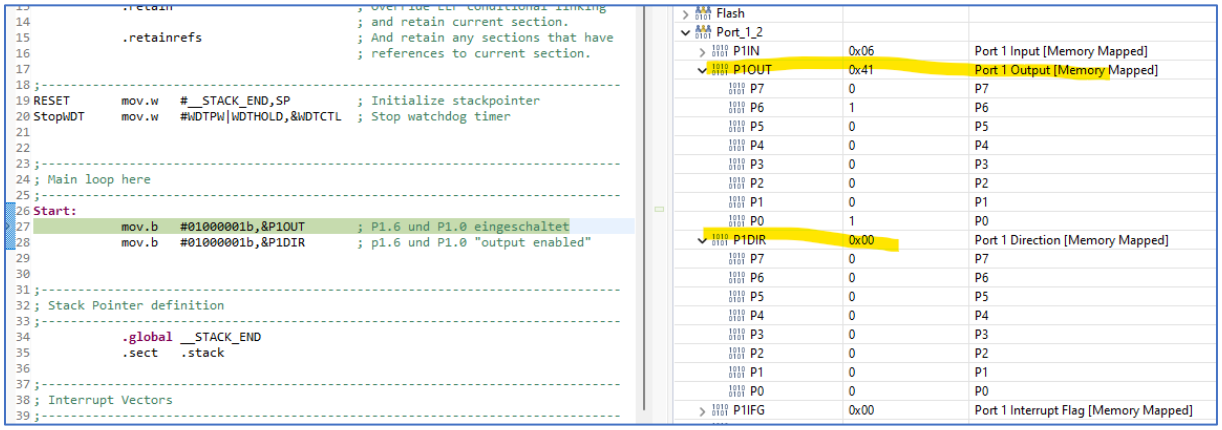
Mit Hilfe von **Assembly Step Into** können Sie den Code Zeile für Zeile ausführen.



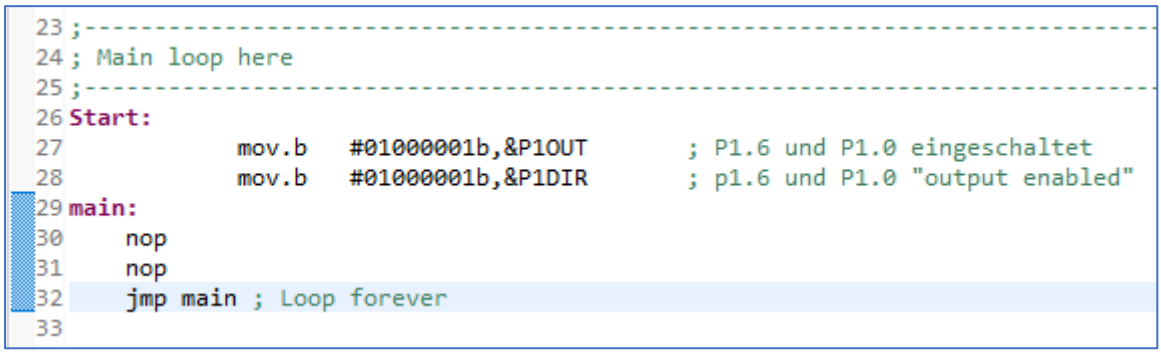
Der Inhalt des Registers PC verrät uns, welche Programmzeile als nächstes abgearbeitet werden wird:



Gehen Sie jetzt schrittweise durch Ihr Programm. Beachten Sie, wie sich dabei die Inhalte im Speicher ändern und dann auch die LEDs eingeschaltet werden.

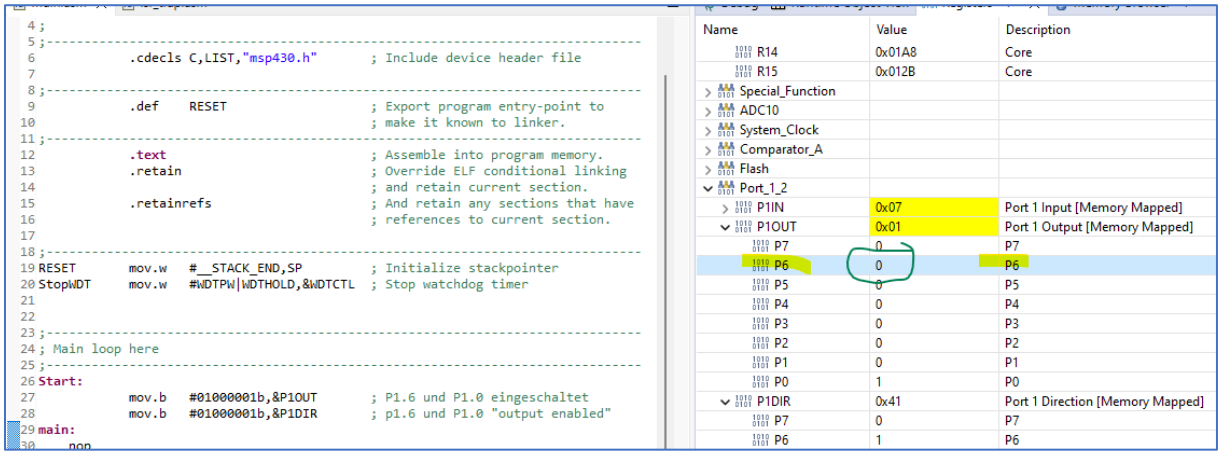


Jetzt wollen wir noch ein Endlosschleife in unser Programm einbauen.
Erweitern Sie Ihr Programm wie folgt:



Verifizieren Sie die Funktion des Kommandos **jmp,** indem Sie durch den Programmablauf steppen.

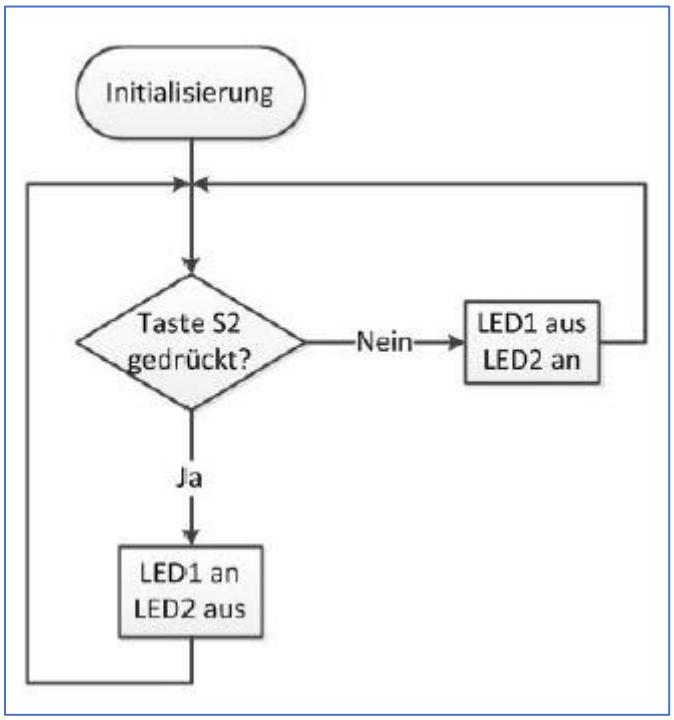
Schalten Sie eine LED aus, indem Sie über den **Debugger** den Inhalt von **P1OUT** ändern:



3.7 Aufgabe 02: Auswahl der leuchtenden LED mit dem Taster

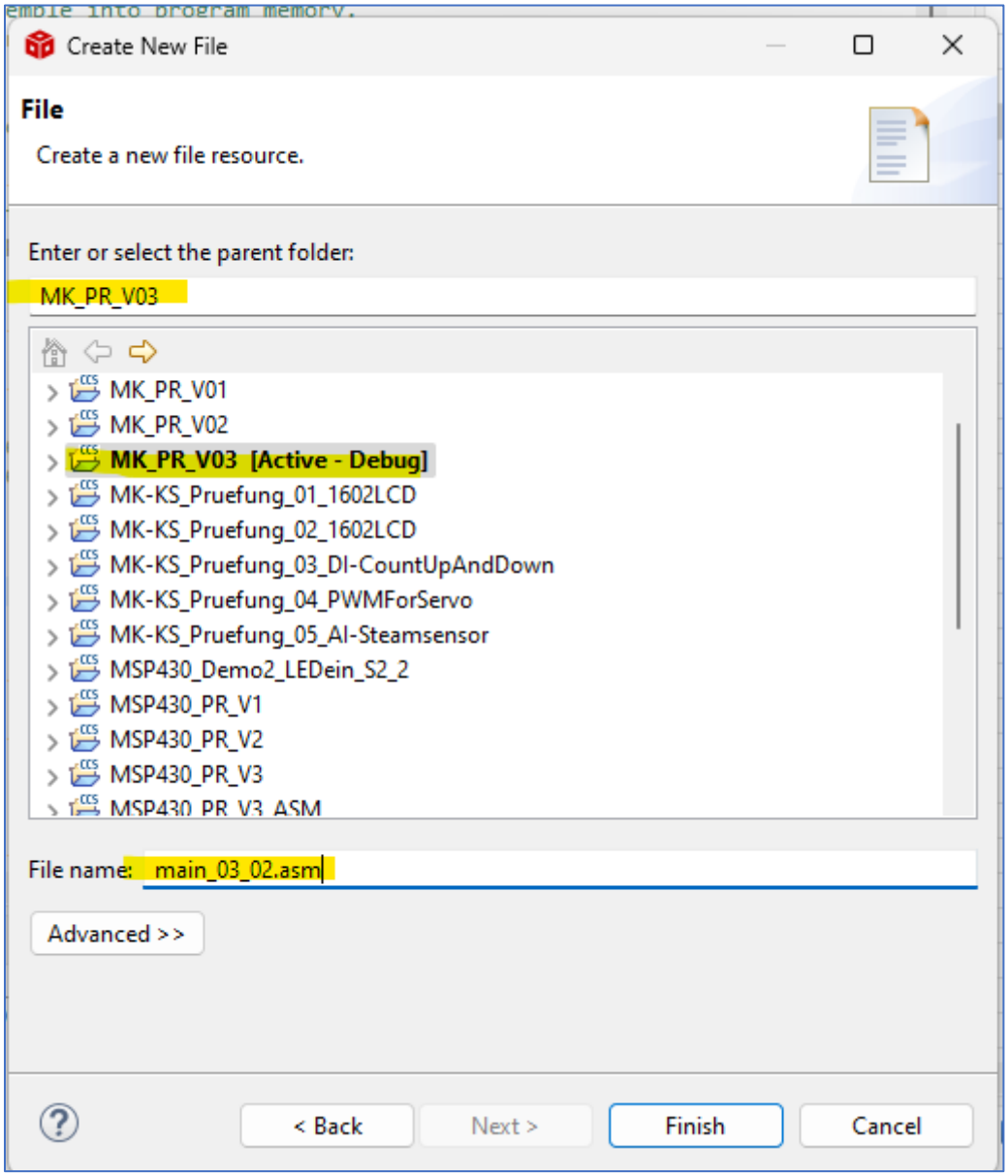
3.7.1 Aufgabenstellung

Bei **gedrücktem** Schalter S2 soll **LED1 ein**- und LED2 ausgeschaltet sein.
Bei **geöffnetem** Schalter S2 soll **LED2 ein**- und LED1 ausgeschaltet sein.



3.7.2 Erzeugen eine neuen .asm Datei

Zunächst erzeugen Sie eine neue .asm Datei mit dem Namen **main_03_02.asm** in Ihrem aktuellen Projekt:



Fügen Sie hier den gesamten Quellcode der alten Datei ein.

3.7.3 Konfigurieren der benötigten IOs und Testen der Konfiguration

Um uns das Leben leichter zu machen, wollen wir zunächst ein paar Definitionen festlegen. Die LED1 ist mit P1.0 des MSP verbunden. Dieser Pin wird in der Port-Registern in der Regel über Bit0 angesprochen.
Erzeugen wir also folgende Definition:

```
18 ;-------------------------------
19 ; Definitionen
20 ;-------------------------------
21 LED1         .equ    BIT0
```

Mit dieser Definition können wir im Code mit folgendem Kommando den Pin von LED1 als Ausgang festlegen:

```
Start:
            bis.b   #LED1,&P1DIR            ; Pin von LED1 als Ausgang festlegen
```

Informieren Sie sich mit Ihren Vorlesungsunterlagen oder über das Datenblatt slau144 im Kapitel zum Instruction-Set (3.4).

Beantworten Sie sich folgende Fragen, bevor sie weitermachen:

Was bedeutet der Ausdruck „**.b**“ im Kommando **bis.b**?

Was bewirkt das Kommando **bis.b**?

Wozu dient die Raute (**#**) vor dem Ausdruck LED1?

Wozu dient das Kaufmannsund (**&**) vor dem Ausdruck **P1OUT**?

Erzeugen Sie ähnliche Definitionen für
- LED2 an P1.6 und den
- Schalter S2 an P1.3

Erzeugen Sie Code in ihrem Programm, der die LED2 als Ausgang festlegt.

Erzeugen Sie Code in ihrem Programm der LED1 und LED2 zum Leuchten bringt.
Welche Register müssen hierzu beschrieben werden?

Enablen Sie für den Pin an S2 den internen Pull-Up Widerstand und stellen Sie ihn auf Pull-Up
Welche beiden Register müssen hierzu beschrieben werden?

Testen Sie die Funktionalität des bisher erzeugten Programms, indem Sie das Programm im Debug-Modus starten.

Sehen sie sich die Information im Debugger an. Überprüfen Sie, ob sie den Zustand von S2 im entsprechenden Register finden können.
Welcher Pin in welchem Register muß betrachtet werden?

Hinweis zum Überwachen der Schalterstellungen:
Schalten Sie den **Refresh** über den entsprechenden Button ein:

Manchmal funktioniert die Interpretation der Bits erst nach einigen Sekunden, oder wenn man ein manuelles Refresh durchführt

3.7.4 Programmieren der Funktionalität

Jetzt wollen wir dem MSP beibringen, daß er die Aufgabenstellung aus 2 befolgt.

Zum Testen, ob ein Bit gesetzt ist, kann man die Instruktion **bit** verwenden.
Informieren Sie sich im Datenblatt über die Funktion dieser Instruktion.

Das Kommando **bit.b** bewirkt:

Zum Springen innerhalb des Programms verwenden wir die Instruktionen **jz** und **jmp**.

Informieren Sie sich im Datenblatt über die Funktion dieser Instruktionen.
Das Kommando **jmp** bewirkt:

Das Kommando **jz** bewirkt:

Verwenden Sie die Labels `main:`, `LED1AUS_LED2AN:` und `LED1AN_LED2AUS:` um Sprungadressen für Ihr Programm zu definieren.

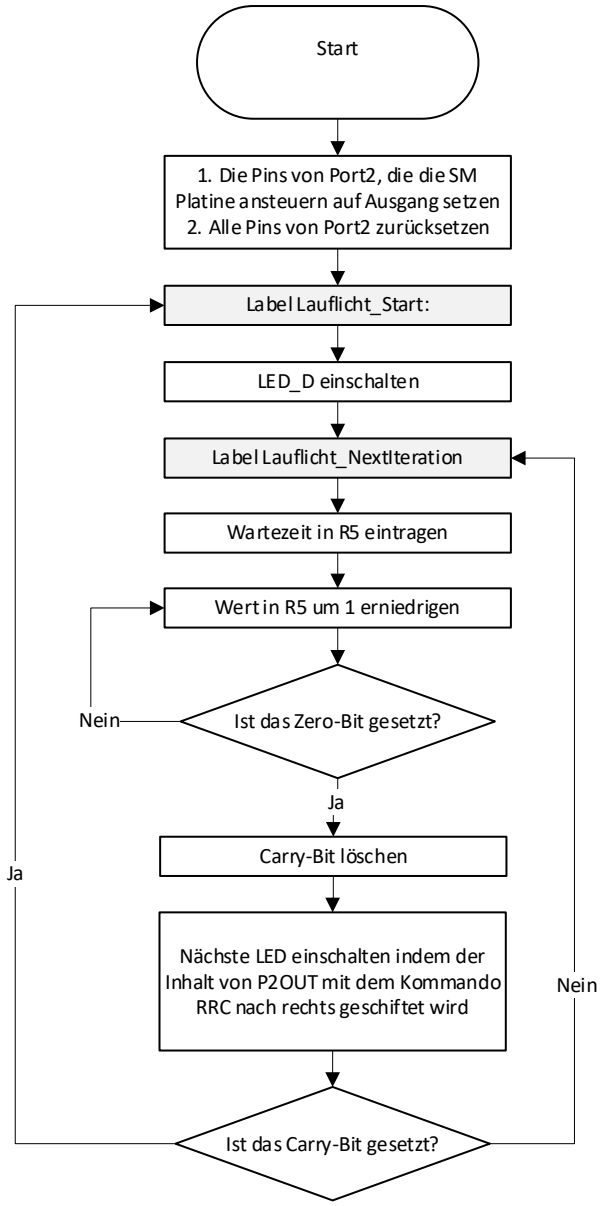
Schreiben Sie ein Programm, das die Aufgabenstellung erfüllt.
Hier ein Hinweis zum Anfangen:

```
; Drücken von S2 schaltet LED1 an, LED2 aus
main:       bit.b   #S2,&P1IN
            jz      LED1AN_LED2AUS
LED1AUS_LED2AN:
```

3.8 Aufgabe 03: Erzeugen eines Lauflichts auf der Schrittmotorplatine

Zunächst erzeugen Sie eine neue .asm Datei mit dem Namen **main_03.03.asm** in Ihrem aktuellen Projekt.

3.8.1 Ablaufdiagramm



J:_RO\07 MSP430\Praktikum\Praktikum MKolb\Bilder\PR_V03_main_03_03_a.vsdx

3.8.2 Benötigte Kommandos

Für Ihr Programm werden Sie folgende Kommandos benötigen.

Informieren sie ich im Datenblatt über diese Kommandos:

Kommando	Bedeutung
BIS	
MOV	
DEC	
JNZ	
CLRC	
RRC	
JC	
JMP	

3.4.6.17 DEC

***DEC[.W]**	Decrement destination
***DEC.B**	Decrement destination
Syntax	DEC dst or DEC.W dst DEC.B dst
Operation	dst - 1 → dst
Emulation	SUB #1,dst SUB.B #1,dst
Description	The destination operand is decremented by one. The original contents are lost.
Status Bits	N: Set if result is negative, reset if positive Z: Set if dst contained 1, reset otherwise C: Reset if dst contained 0, set otherwise V: Set if an arithmetic overflow occurs, otherwise reset. Set if initial value of destination was 08000h, otherwise reset. Set if initial value of destination was 080h, otherwise reset.
Mode Bits	OSCOFF, CPUOFF,and GIE are not affected.
Example	R10 is decremented by 1.

```
     DEC     R10     ; Decrement R10
; Move a block of 255 bytes from memory location starting with EDE to memory
location starting with
; TONI. Tables should not overlap: start of destination address TONI must not
be within the range EDE
; to EDE+0FEh
     MOV     #EDE,R6
     MOV     #255,R10
L$1  MOV.B   @R6+,TONI-EDE-1(R6)
     DEC     R10
     JNZ     L$1
```

Do not transfer tables using the routine above with the overlap shown in Figure 3-13.

Quelle: slau144k

4.6.2.41 RRC

RRC[.W]	Rotate right through carry destination word
RRC.B	Rotate right through carry destination byte
Syntax	`RRC dst` or `RRC.W dst` `RRC.B dst`
Operation	C → MSB → MSB–1 → ... LSB+1 → LSB → C
Description	The destination operand is shifted right by one bit position as shown in Figure 4-40. The carry bit C is shifted into the MSB and the LSB is shifted into the carry bit C.
Status Bits	N: Set if result is negative (MSB = 1), reset otherwise (MSB = 0) Z: Set if result is zero, reset otherwise C: Loaded from the LSB V: Reset
Mode Bits	OSCOFF, CPUOFF, and GIE are not affected.
Example	RAM word EDE is shifted right one bit position. The MSB is loaded with 1.

```
SETC            ; Prepare carry for MSB
RRC     EDE     ; EDE = EDE >> 1 + 8000h
```

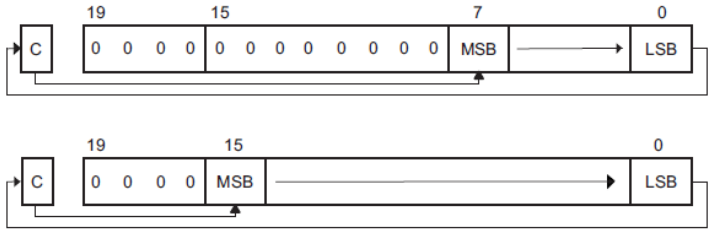


Figure 4-40. Rotate Right Through Carry RRC.B and RRC.W

Quelle: slau144k