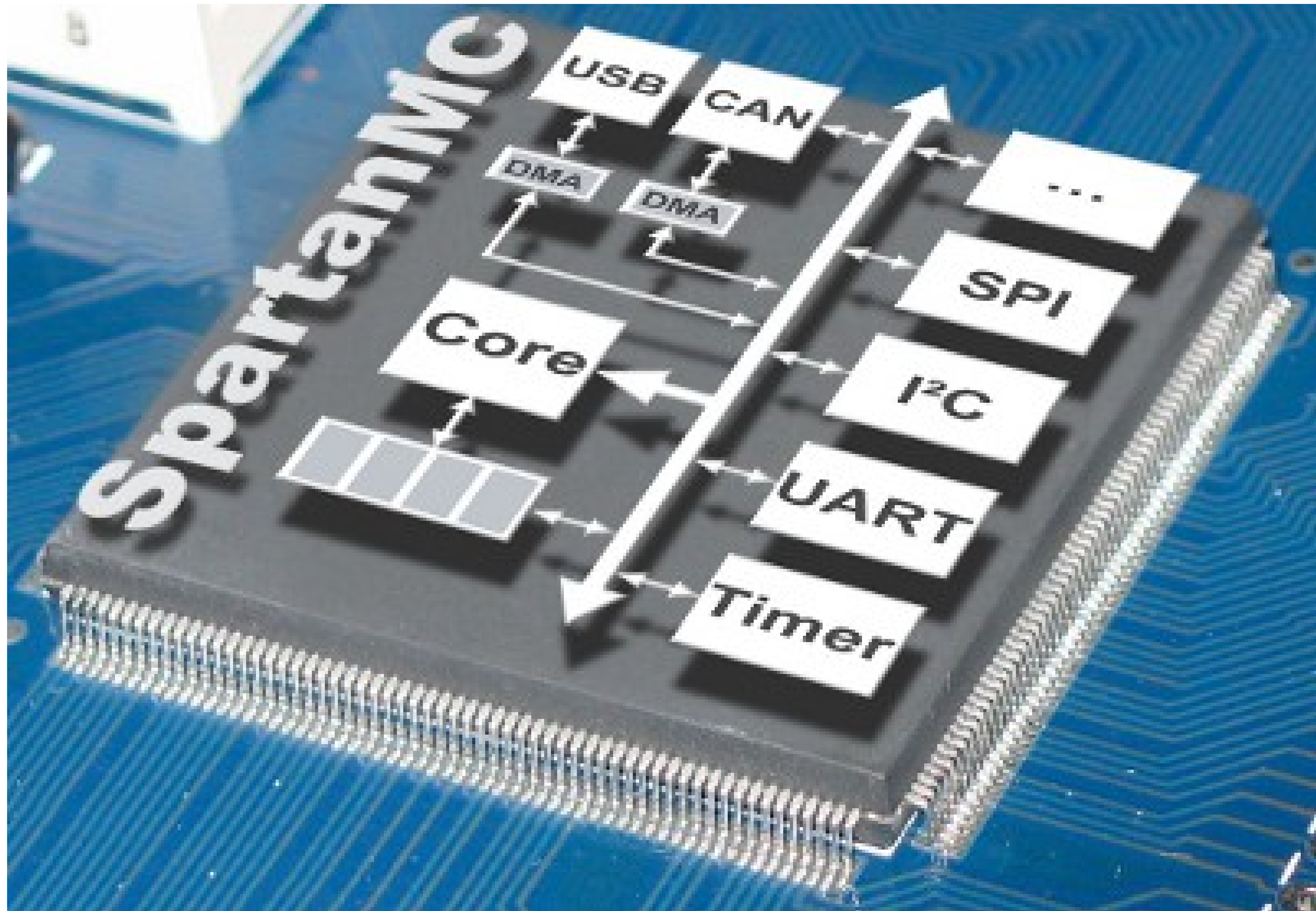
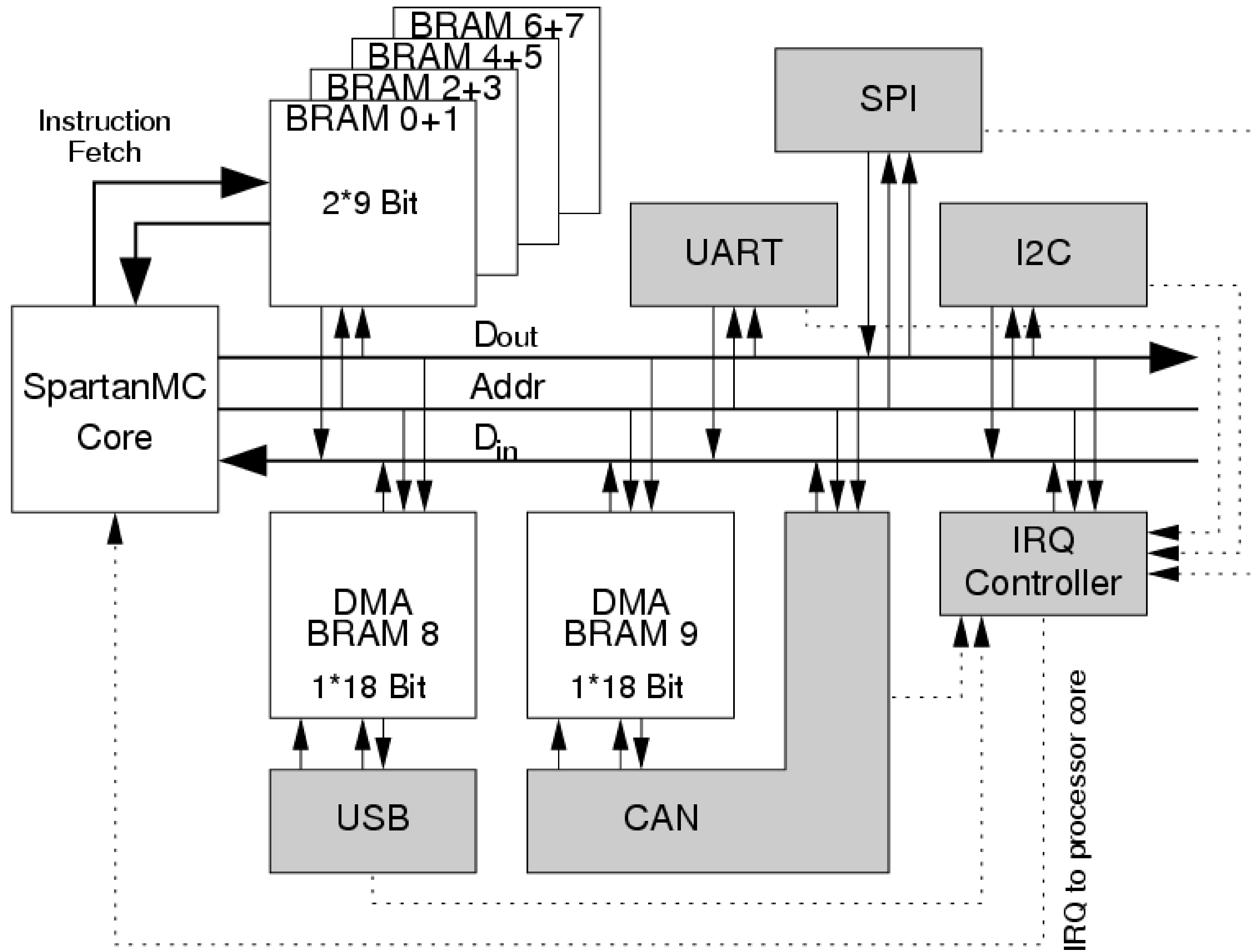
Der SpartanMC SoC



1. Der Mikrokontroller SpartanMC



1. Der Mikrokontroller SpartanMC (2)

- Spartan 3e FPGAs schon ab 2$ erhältlich
- Einsatz in eingebetteten Systemen bei niedrigen Kosten möglich
- Vorhandene SoC Kits mit 8 Bit und 32 Bit Prozessoren kommen sehr schnell an Ihre Grenzen oder benötigen sehr große FPGAs
- SpartanMC soll diese Lücke schließen
  - Einsatz schon auf den kleinen FPGAs möglich
  - 18 Bit um Blockram und Multiplizierer maximal zu nutzen
  - Erstellung einer Konfiguration ohne FPGA Wissen möglich
  - große Vielfalt an Standard und Spezial Peripherie
  - mehrfache Implementierung der Peripherie möglich
  - multiprozessor Systeme sind möglich

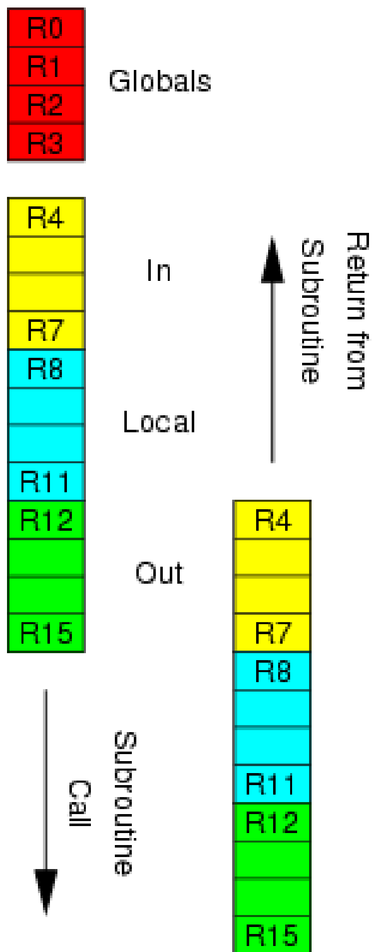
1. Der Mikrokontroller SpartanMC (3)

- Nachteile einer SoC Lösung gegenüber Mikrokontroller
  - kleine Mikrokontroller sind noch billiger als 2$
  - Sie benötigen zur Zeit noch weniger Strom (Batterieeinsatz)
  - ADC und DAC sind vorhanden
- Vorteile einer SoC Lösung
  - Konfiguration kann genau an die Aufgabe angepasst werden
  - Nachträgliche Änderungen und Anpassungen sind möglich
  - unterschiedliche Signalpegel an den Interfaces möglich
  - Skalierung der Leistung durch Wahl des FPGA
  - Leistung steigt mit jeder neuen FPGA Generation
  - keine unterschiedlichen Mikrokontroller in einem Produkt notwendig (in Fernsehern werden viele unterschiedliche Mikrokontroller eingesetzt)

1. Der Mikrokontroller SpartanMC (4)

- Der SpartanMC ist als 18 Bit RISC implementiert
- Die schnellen Blockrams werden für die Register und den Speicher eingesetzt
- Kein Cache Speicher notwendig
- Registerfenster um Registerspeicher voll zu nutzen
- Im Speicher wird ein Port zum lesen der Befehle und das zweite für den Datentransfer eingesetzt (Harvard Architektur)
- Befehle können nur auf 18 Bit Adressen stehen
- Datenzugriff auf obere und untere 9 Bit von den 18 Bit möglich
- Daher ist Befehlsadresse*2 gleich Datenadresse
- 18 Bit Datenzugriff immer nur auf gerade Adressen möglich

1.1 Die Register und der Speicher



- SpartanMC kann 16 Register direkt ansprechen
- r0 bis r3 global immer erreichbar
- r4 bis r15 können bei einem Unterprogramm aufruf oder Interrupt mit einem Offset von 8 umgeschaltet werden
- Rücksprung schaltet das Registerfenster wieder um eine Ebene zurück
- In den unteren 7 Bit des SFR_STATUS Register steht die Nummer des Registerfenster
- 16 + 126*8 = 1024 Register realisiert
- Im Hauptprogramm mit r4 bis r11 arbeiten

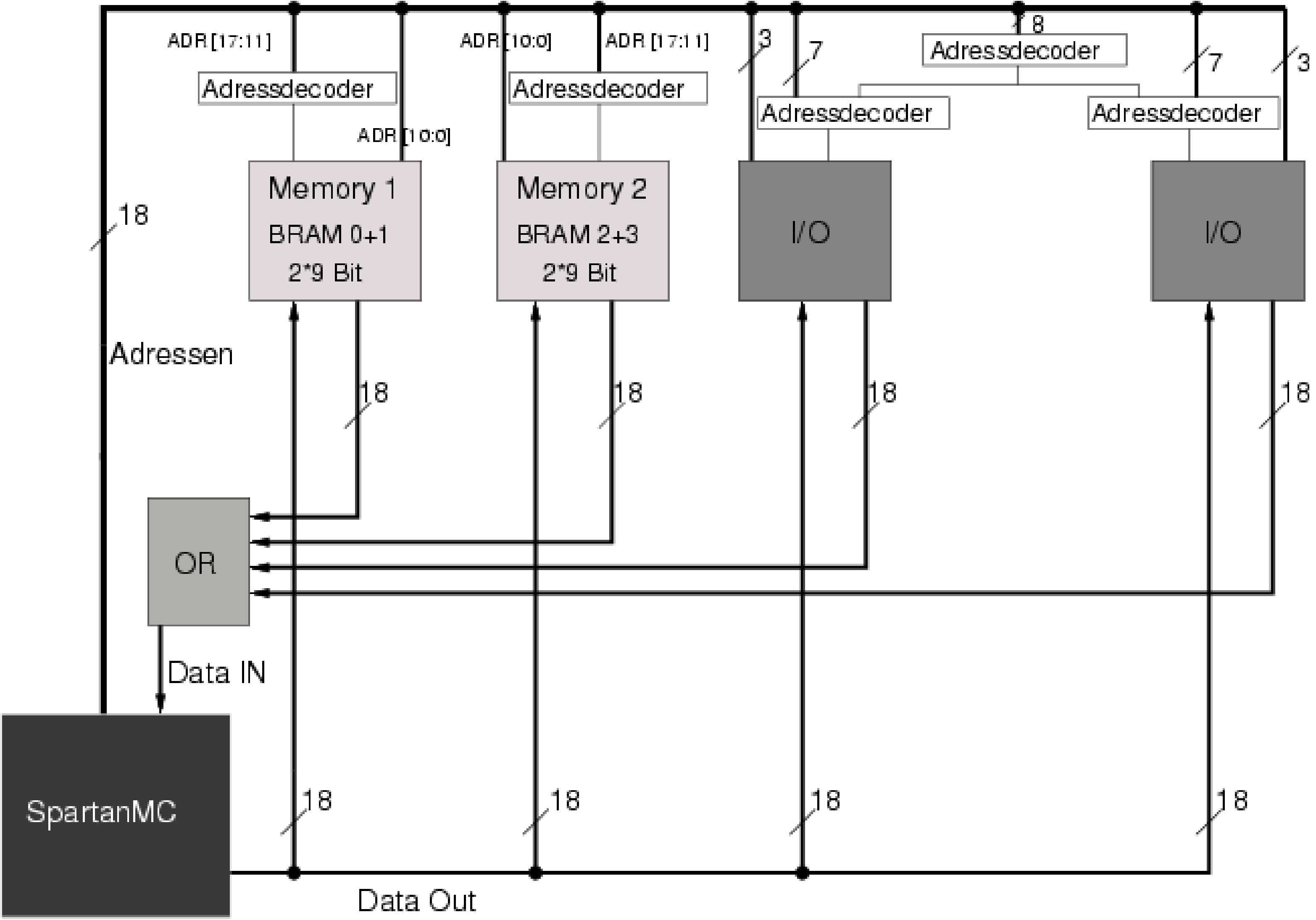
1.1 Die Register und der Speicher (2)

- UP und ISR sollten r8 bis r15 nutzen
- In r11 steht die Rücksprungadresse vom UP oder ISR
- r0 bis r3 für spezielle Operanden (Stackpointer)
- Es können 4 Werte in r12 bis r15 an UP übergeben werden
- Im UP sind diese Werte in r4 bis r7
- Damit auch Rückgabe von 4 Ergebnissen des UP
- 127 geschachtelte UP/ISR Aufrufe ohne Stack möglich
- Register sind in einem 18 Bit Blockram des FPGA realisiert

1.1 Die Register und der Speicher (3)

- Neben den Universalregistern gibt es noch 6 *Special function* Register 0=SFR_STATUS, 1=SFR_LEDS, 2=SFR_MUL, 3=SFR_CC, 4=SFR_IV und 5=SFR_TR
- $SFR_STATUS_{6:0}$ = RegBase – Nummer des Registerfenster
- SFR_STATUS_{7} = MM – ADR_{17} am Speicher für den Datenzugriff
- SFR_STATUS_{8} = EI – Freigabe der Interrupts
- SFR_LEDS – verbunden mit 7 Segment Anzeige oder LEDs
- SFR_MUL – oberen 18 Bit bei der Multiplikation
- SFR_CC_{0} – *Condition code* Bit
- SFR_IV – Startadresse der Interruptverarbeitung
- SFR_TR – Startadresse der TRAP-Tabelle (teilbar durch 256)

1.1 Die Register und der Speicher (4)



1.1 Die Register und der Speicher (5)

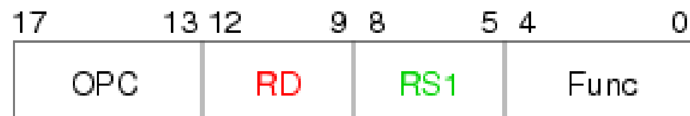
- Ein Speicherblock wird mit zwei 9 Bit Blockrams realisiert
- Speicherblock hat 2k Worte mit 18 Bit
- getrenntes Schreiben in die unteren oder oberen 9 Bit des 18 Bit Speicherwortes möglich
- Anzahl der Speicherblöcke begrenzt durch die Anzahl der Blockrams des verwendeten FPGA
- Blockrams sind immer selektiert, nur Output wird mit (SSR) Signal auf Festwertregister (belegt mit 0) umgeschaltet
- DATA_IN Bus wird durch Zusammenschaltung aller Speicher blöcke und Peripherie Module über OR-GATE realisiert
- Zweites OR-GATE am Befehlslesebus bei mehr als einem Speicherblock notwendig

1.2 Die Adressierungsarten des SpartanMC

- Basis des Befehlssatz: modifizierter DLX-Befehlssatz mit nur 2 Adressen
- Die Register werden direkt adressiert
- Konstanten werden mit 9 Bit direkt angegeben (-256 bis 255 oder 0 bis 511)
- Speicher Zugriff: Register indirekt mit Displacement
  - Displacement nur positiv und nur 5 Bit (0 bis 31)
  - Einschränkung notwendig, da nur 18 Bit/Befehl

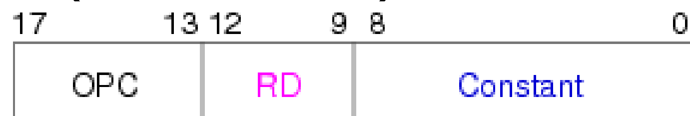
1.3 Die Befehle des SpartanMC

- 4 Formate mit 2 Operanden in 18 Bit
- R (Register) Format



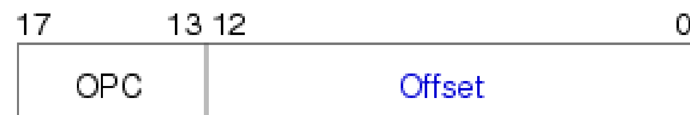

```
add     r4,     r6
subu    r12,    r14
```

- I (Immediate) Format



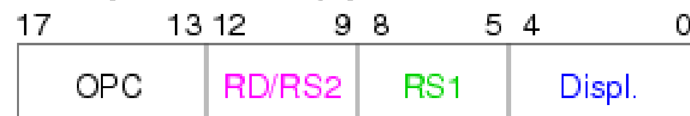

```
addi    r5,     -'0'
bnez    r7,     loop
```

- J (Jump) Format




```
jals    unterprogramm1
beqzc   loop2
```

- M (Memory) Format




```
s18     6(r4),  r3
l9      r10,    0(r12)
```

1.3 Die Befehle des SpartanMC (2)

Hauptmatrix der Befehls Register Bits IR 17 - 13								
IR 17-13	**..000**	**..001**	**..010**	**..011**	**..100**	**..101**	**..110**	**..111**
00...	**Spezial1**	**Spezial2**	J	JALS	BEQZ	BNEZ	BEQZC	BNEZC
01...	ADDI	MOVI	LHI	SIGEX	ANDI	ORI	XORI	MULI
10...	L9	S9	L18	S18	SLLI	--- *)	SRLI	SRAI
11...	SEQI	SNEI	SLTI	SGTI	SLEI	SGEI	IFADDUI	IFSUBUI
Submatrix SPEZIAL 1 der Befehls Register Bits IR 4 - 0								
IR 4-0	**..000**	**..001**	**..010**	**..011**	**..100**	**..101**	**..110**	**..111**
00...	--- *)	--- *)	--- *)	--- *)	SLL	MOV	SRL	SRA
01...	SEQU	SNEU	SLTU	SGTU	SLEU	SGEU	--- *)	--- *)
10...	--- *)	--- *)	--- *)	--- *)	--- *)	--- *)	CBITS	SBITS
11...	--- *)	--- *)	--- *)	--- *)	--- *)	--- *)	--- *)	NOT
Submatrix SPEZIAL 2 der Befehls Register Bits IR 4 - 0								
IR 4-0	**..000**	**..001**	**..010**	**..011**	**..100**	**..101**	**..110**	**..111**
00...	RFE	TRAP	JR	JALR	JRS	JALRS	--- *)	--- *)
01...	--- *)	--- *)	--- *)	--- *)	--- *)	--- *)	--- *)	--- *)
10...	ADD	ADDU	SUB	SUBU	AND	OR	XOR	MUL
11...	SEQ	SNE	SLT	SGT	SLE	SGE	MOVI2S	MOVS2I

- *) Code wird nicht benutzt
- rot geschriebene Befehle werden in der Zukunft nicht unterstützt

1.3.1 Logikbefehle

- Schiebebefehle – 2 Konfigurationsvarianten der CPU
  - es wird immer nur um ein Bit verschoben (wenig hardware)
  - es kann beliebig verschoben werden (*Barrel-shifter*)
- einfachem Schieben
  - Wert in Rs oder im Imm wird ignoriert
  - Mehrfaches Schieben durch Schleifen

1.3.2 Transportbefehle

- **l9** **Rd, disp(Rs)** M Rd <-- 0^9##M[disp+Rs]
- **l18** **Rd, disp(Rs)** M Rd <-- M[disp+Rs]##M[disp+Rs+1]

- **s9** **disp(Rs)**, **Rs2** M M[disp+Rs] <-- Rs2
- **s18** **disp(Rs)**, **Rs2** M M[disp+Rs]##M[disp+Rs+1] <-- Rs2

- **l9** und **s9** zur besseren Speicherauslastung bei Zeichenketten
- In C wird mit **struct** diese Adressierung besonders unterstützt. Programme werden damit deutlich kürzer. Beim SpartanMC kann ein **struct** aber nur über 32 Byte genutzt werden, da **disp** nur 5 Bit hat. Die Verwendung von **struct** ist im lcc18 nur für globale Variable möglich.

1.3.2 Transportbefehle (2)

- **movi2s** **SfrNr, Rs** R SFR_reg <-- Rs
- **movs2i** **Rd, SfrNr** R Rd <-- SFR_reg
  SfrNr wird im R Format immer auf die Bits von Rs1 gespeichert, auch bei MOVI2S
- **sigex** **Rd, BitNr+1** I $Rd \leftarrow Rd_{BitNr}^{17-BitNR}\#\#Rd_{BitNr:0}$
  BitNr+1 darf nur 16, 9 oder 8 sein
- Werden in C die Datentypen ohne den Vorsatz **unsigned** eingesetzt, dann wird beim Laden eines Wertes kleiner 18 Bit immer der Befehl **sigex** eingefügt.

1.3.3 Verzweigebefehle

- Die folgenden 2 Befehle haben einen **Delay-Slot**.
- **beqz** **Rd, label** I PC=PC+1 if(Rd==0) PC <-- label
- **bnez** **Rd, label** I PC=PC+1 if(Rd!=0) PC <-- label

- Das cc darf nicht erst im vorherigen Befehl gesetzt werden!
- **beqzc** **label** J if(cc==0) PC <-- label
- **bnezc** **label** J if(cc!=0) PC <-- label

- **j** **label** J PC <-- label

1.3.3 Verzweigebefehle (2)

- **Unterprogramm und ISR Arbeit**
- **trap** Nummer R RegBase<--RegBase+1;R11<--PC+1;
  PC<--SFR_TR$_{17:8}$##Nummer
- **jals** **label** J RegBase<--RegBase+1;r11<--PC+1;PC<--label
- Die folgenden 5 Befehle haben einen **Delay-Slot**.
- **jalrs** **Rs** R PC=PC+1;RegBase<--RegBase+1;r11<--PC+1;
  PC<--Rs
- **jrs** **Rs** R PC=PC+1;PC<--Rs;RegBase<--RegBase-1
- **rfe** **Rs** R PC=PC+1;PC<--Rs;RegBase<--RegBase-1
  Erzeugt ein Signal für den Interruptkontroller
- **jalr** **Rs** R PC=PC+1;r11<--PC+1;PC<--Rs
- **jr** **Rs** R PC=PC+1;PC<--Rs (Für **CASE** Verzweigung)

1.3.3 Verzweigebefehle (3)

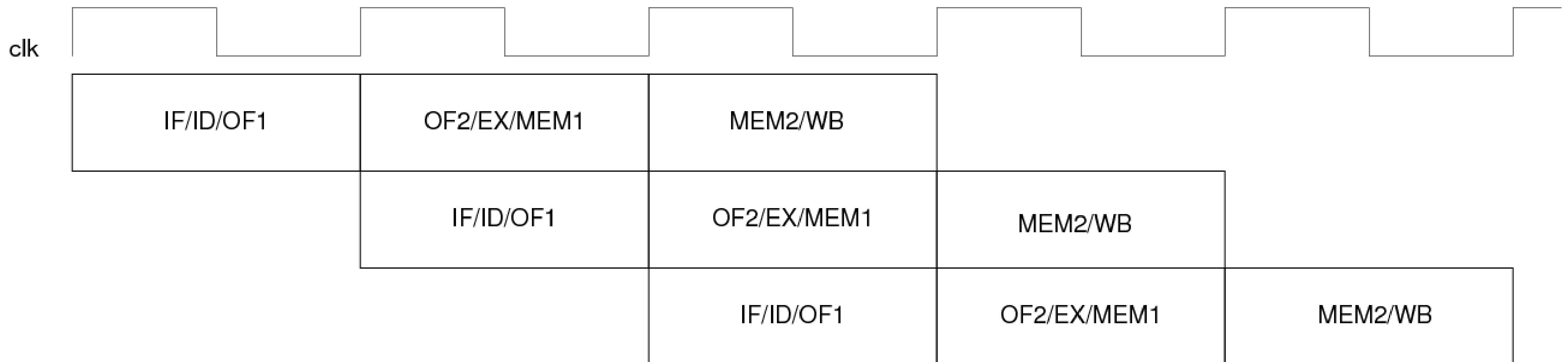
- Der **Delay-Slot** wird bei Befehlen eingesetzt, die den linearen Ablauf der Befehlsabarbeitung verlassen, also nicht mit dem Folgebefehl weiter arbeiten.
- Dies ist meist notwendig, da die Bedingung für den Sprung erst im nächsten Takt vorliegt.
- Um keine Pipeline Verluste zu haben, wird der Befehl hinter dem Sprung noch so abgearbeitet, als ob er davor steht, also auch mit dem alten Register Fenster!
- Der Befehl wird zunächst mit einem NOP (or r0,r0) belegt.
- Bei der Optimierung wird aus der Befehlsfolge vom letzten Einsprung bis zum Sprung ein Befehl gesucht, der die Sprung-bedingung nicht beeinflusst und hinter den Sprung an die Stelle des NOP in den **Delay-Slot** verschoben.
- Der **Delay-Slot** wird in einer Schleife immer mit abgearbeitet.

1.3.4 Steuerbefehle

- Befehle zum ein- und ausschalten von Statusbits
  - CC – BitNr = 0 = SFR_CC_0
  - MM – BitNr = 1 = SFR_STATUS_7
  - INT – BitNr = 2 = SFR_STATUS_8

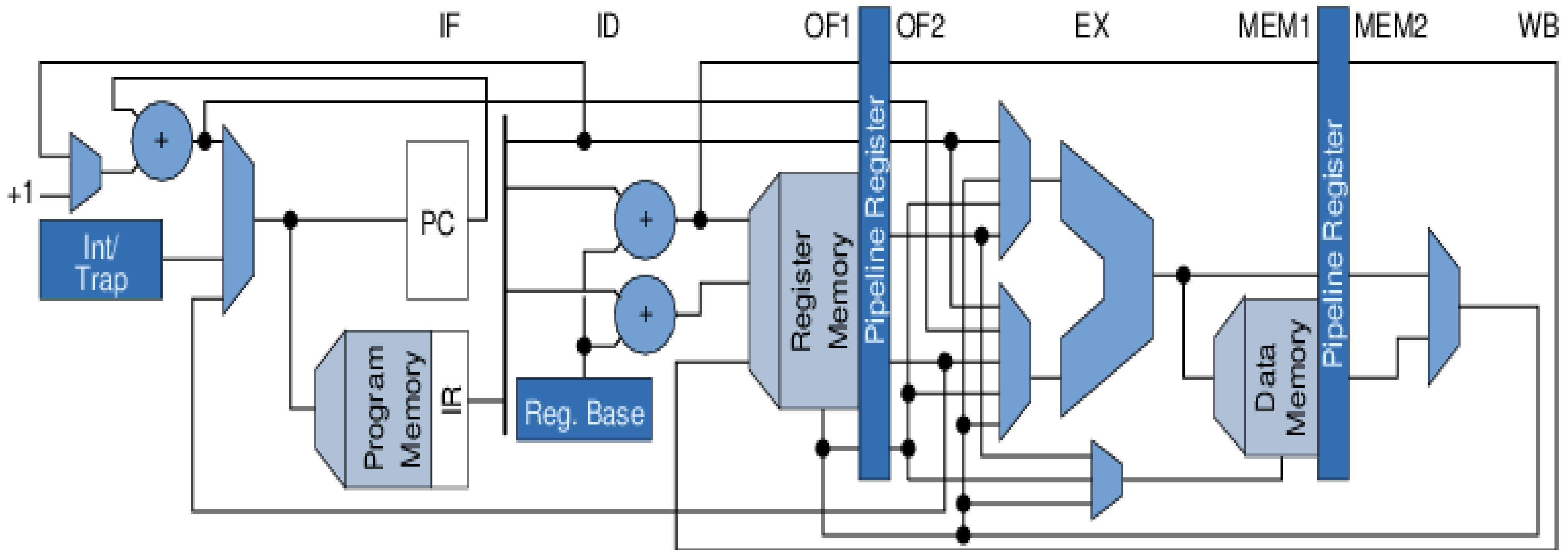
- **cbits** **BitNr** R SFR_bit <-- 0
- **sbits** **BitNr** R SFR_bit <-- 1

2. Die Pipeline des SpartanMC



IF	-	Instruction Fetch
ID	-	Instruction Decode
OF1	-	Operanden Fetch 1 (Konstanten und Bypass1 vorbereiten)
OF2	-	Operanden Fetch 2 (Auswahl Register, Bypass1, Bypass2 oder Konstante)
EX	-	Execute Instruction (Ausführung der Befehle oder Berechnung von Speicheradressen)
MEM1	-	Memory 1 (Adressierung des Speichers (der I/O) und schreiben der Daten im Store Befehl)
MEM2	-	Memory 2 (Daten lesen im Load Befehl)
WB	-	Write Back (Zurückschreiben der Ergebnisse in das Register)

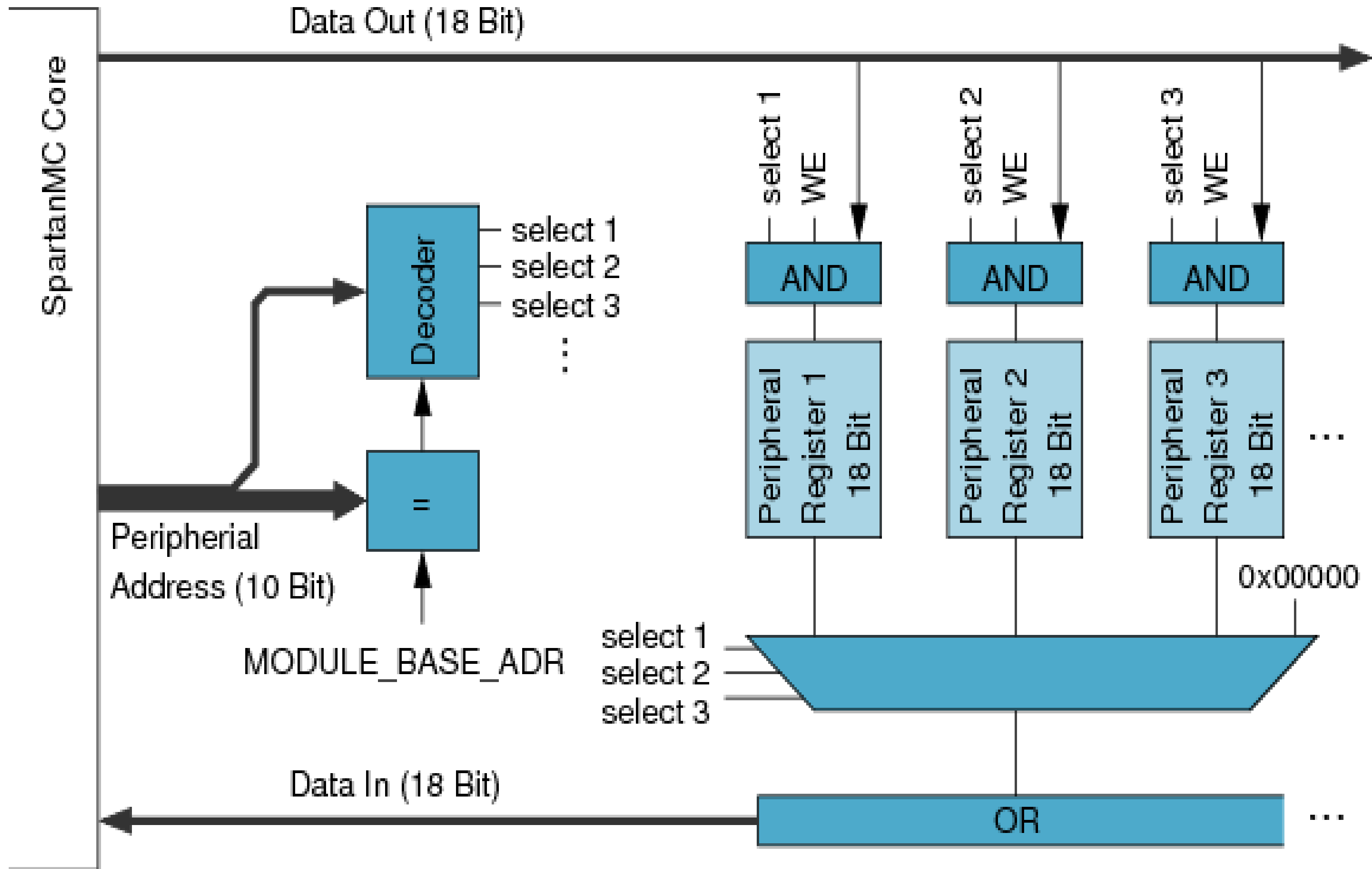
2. Die Pipeline des SpartanMC (2)



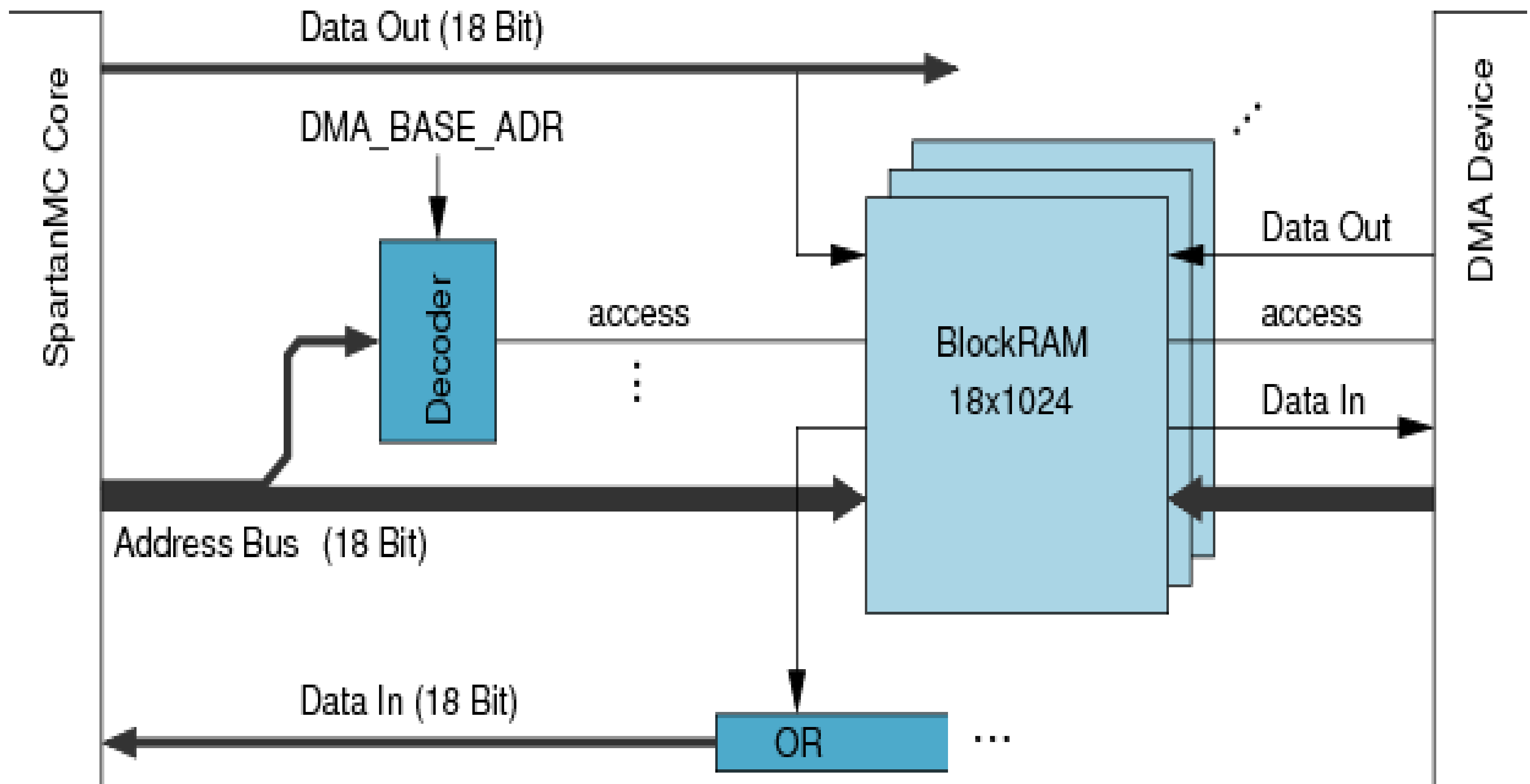
3.1 Interface des SpartanMC

- Interface werden wie Speicher angesprochen (Memory Maped)
- Jedes Interface kann ein oder mehrere Ports am OR_GATE belegen
- Geräteregister reservieren immer 18 Bit
- I/O Basisadresse befindet sich hinter dem letzten Speicherblock
- Von Adresse 0x00000 beginnt der Speicher
- DMA Puffer liegen unterhalb der I/O Basisadresse und oberhalb des Speichers
- Interruptsignale müssen im Gerät bis zur Annahme gespeichert werden
- Interruptsignale müssen unterdrückbar sein (nach RESET)

3.1 Interface des SpartanMC



3.2 DMA Interfaces des SpartanMC

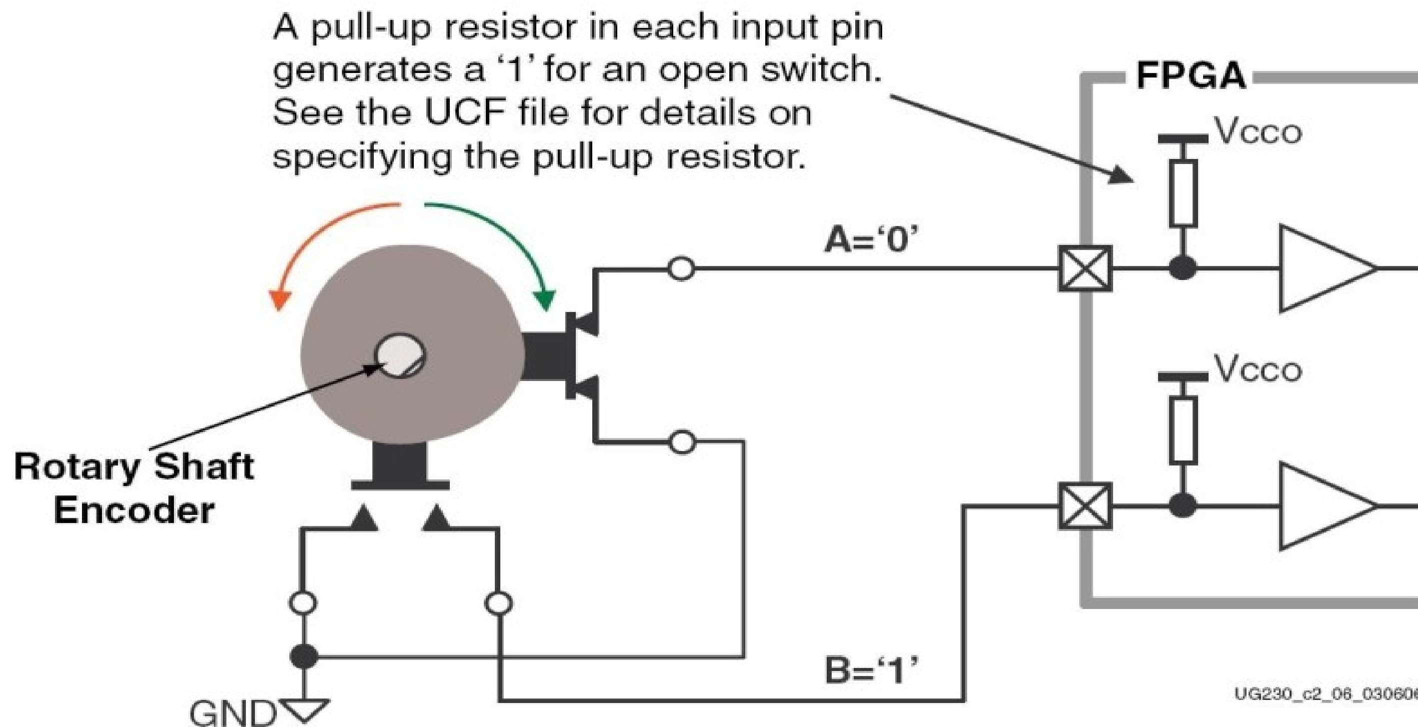


3.3 Spezial Interfaces für den SpartanMC

- In SoC Lösungen ist es möglich Spezial Interface mit Hardware zu unterstützen ohne dadurch zusätzliche Kosten zu verursachen.
- Die Verwendung in der Software wird dadurch einfacher.
- Es braucht keine Systemleistung für so einfache Dinge wie Endprellen verschwendet werden.
- Für den SpartanMC wird ein immer breiteres Spektrum solcher Interface zur Verfügung gestellt.
- Die meisten Lösungen unterstützen den Roboter Einsatz.

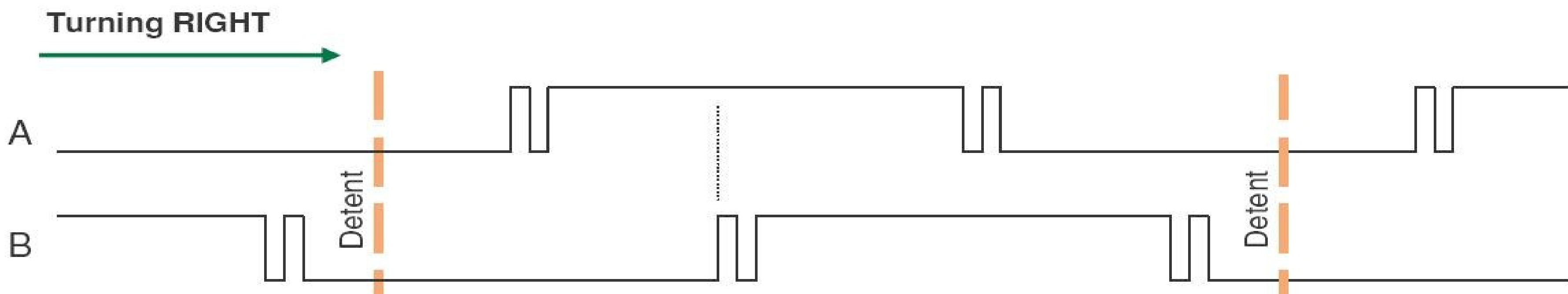
3.3.1 Rotationstaster (Sensor)

- Zur Erkennung der Drehrichtung und Drehzahl
- Zur Auswahl in „Ein stell Menüs“
- Zur Erkennung von Bewegungen mechanischer Teile



3.3.1 Rotationstaster (Sensor) 2

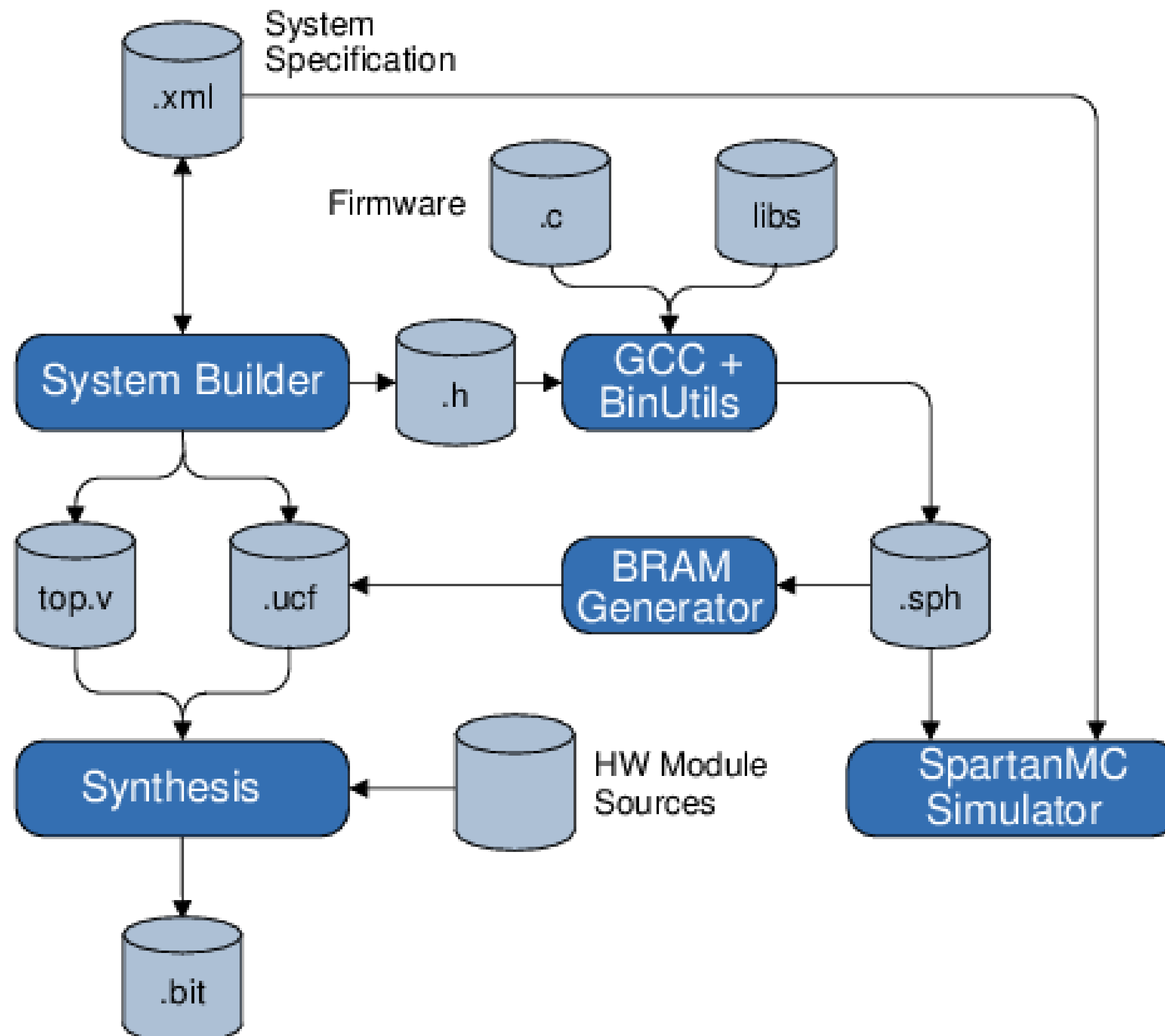
- Das Interface entprellt die Kontakte A und B und erzeugt daraus je ein Signal für rechts oder links Drehung.
- Alle diese Signale und das Drücken der Taste können Interrupt auslösen.
- Ein 18 Bit Zähler wird beim rechts drehen inkrementiert und beim links drehen decrementiert.
- Der Zähler kann zu jeder Zeit mit jedem beliebigen 18 Bit Wert geschrieben werden.



3.3.2 Schrittmotoren Interface

- Das Interface übernimmt das Beschleunigen und Bremsen des Motors
- Die Anzahl der Beschleunigungsphasen (typisch 5) ist generierbar
- Nach der Initialisierung muss nur die Anzahl der Schritte und die Richtung eingestellt werden
- Es werden 3 Interrupts ausgelöst für:
  - Topspeed erreicht
  - Abbremsung beginnt
  - Stopp Zustand erreicht

4 Werkzeuge zur Programmierung und Konfiguration des SpartanMC



4.1 Der GCC
4.1.1 Konsolen Funktionen

- 1. **uart_wait_idle(UART_MONITOR);**

  Die Funktion wartet auf den Ruhezustand einer UART, deren Adresse mit dem Namen der Uart im jConfig als Argument (hier UART_MONITOR) in den Klammern angegeben wird. Die Funktion wartet, bis die Schieberegister von Rx und Tx sowie RESET Funktionen abgeschlossen sind. Danach kann dann bei unidirektionalen Datenübertragungen die Datenrichtung geändert werden oder eine andere Datenrate, oder andere um Initialisierungen vorgenommen werden. Bei Verwendung der UART_LIGHT darf diese Funktion nicht aufgerufen werden. Wird die Funktion beim Systemstart vor der 1. Verwendung der Uart nicht aufgerufen, dann werden die Daten verfälscht, wenn der Ruhezustand noch nicht eingenommen wurde!

4.1 Der GCC
4.1.1 Konsolen Funktionen (2)

- 2. **stdio_uart_open(UART_MONITOR);**

  Die Funktion verknüpft die Uart mit der Adresse UART_MONITOR mit den STDIO Operationen, die in den folgenden Funktionen beschrieben werden.

- 3. **putchar(zeichen);**

  Ausgabe eines 8 Bit Wertes an das STDIO Gerät. Der folgende Code sendet eine ASCII "0".

```
unsigned char   zeichen;
zeichen = 0x30;
putchar(zeichen);
```

4.1 Der GCC
4.1.1 Konsolen Funktionen (3)

- 4. **zeichen = getchar();**

  Warten auf die Eingabe eines 8 Bit Wertes vom STDIO Gerät. Der folgende Code übergibt den eingegebenen Wert der Variablen "zeichen".

```
unsigned char   zeichen;
zeichen = getchar();
```

4.1 Der GCC
4.1.1 Konsolen Funktionen (4)

- 5. **getchar_nb(&zeichen);**

  Testen, ob am STDIO Gerät ein Zeichen eingegeben wurde. Im folgenden Beispiel liefert die definierte Funktion eine 0x0 wenn nichts eingegeben wurde oder den Code des gesendeten Zeichens.

```
// z=uart_getstat() // z=0 keine Eingabe
unsigned char uart_getstat(void){
        unsigned char   zeichen = 0;
        getchar_nb(&zeichen);
        return zeichen;
}
```

4.1 Der GCC
4.1.1 Konsolen Funktionen (5)

- 6. **printf("\r\n Wert = 0x%6x\n",out_value);**

  Senden von Zeichenketten und Anzeige von Werten auf dem STDIO Gerät. Es wird die übliche C-Funktion realisiert, es können aber nur maximal 2 Werte pro Aufruf angezeigt werden. Die Anzeige von Werten im LONG Format ist nicht möglich!

```
unsigned int out_value = 0x12345;
printf("\r\n\t  Hallo ich bin SpartanMC 18   \r");
printf("\r\n\t  SpartanMC 18 TU-Dresden InfMR\r");
printf("\r\n\t  Wert = 0x%6x\n",out_value);
```

4.1 Der GCC
4.1.1 Konsolen Funktionen (6)

- 7. **print_lx(**unsigned long wert, unsigned int anz**);**

  Anzeigen von long Werten (36 Bit) hexadezimal auf dem STDIO Gerät. Mit dem 2. Parameter wird die Anzahl der Anzeigestellen festgelegt. Es sind Werte von 0 bis 9 möglich. Bei 9 werden immer alle Stellen angezeigt und bei 0 nur die Stellen ungleich 0. Bei 1 wird nur die Stelle 16^0 immer angezeigt.

```
#include <io_funktionen.h>
unsigned long out_value = 0x123456789;
printf("\r\n 36 Bit Wert = 0x");
print_lx(out_value, 1);
```

4.1 Der GCC
4.1.1 Konsolen Funktionen (7)

- 8. **print_ld(**unsigned long wert, unsigned int sig, unsigned int anz**);**

  Anzeigen von long Werten (36 Bit) dezimal auf dem STDIO Gerät. Mit dem 3. Parameter wird die Anzahl der Anzeigestellen festgelegt. Es sind Werte von 0 bis 11 möglich. Bei 11 werden immer alle Stellen angezeigt und bei 0 nur die Stellen ungleich 0. Bei 1 wird nur die Stelle 10^0 immer angezeigt. Mit sig=0 wird wert ohne Vorzeichen angezeigt. Für ungleich 0 wird das Vorzeichen und der Betrag angezeigt.

```
#include <io_funktionen.h>
unsigned long out_value = -34359738368;
printf("\r\n 36 Bit Wert = ");
print_ld(out_value, 1, 1);
```

4.1 Der GCC
4.1.1 Konsolen Funktionen (8)

- 9. **get_lx(**unsigned int pos**);**

  Eingabe von long Werten (36 Bit) hexadezimal mit dem STDIO Gerät. Mit dem Parameter pos wird die Anzahl der Eingabestellen festgelegt, bis zu der nach ENTER ein TAB auf die Konsole ausgegeben wird. Mit BS kann die Eingabe korrigiert werden.

```
#include <io_funktionen.h>
unsigned long in_value;
printf("\r\n 36 Bit Wert = 0x");
in_value = get_lx(4);
```

4.1 Der GCC
4.1.1 Konsolen Funktionen (9)

- 10. **get_ld(**unsigned int sig, unsigned int pos**);**

  Eingabe von long Werten (36 Bit) dezimal mit dem STDIO Gerät. Mit dem Parameter pos wird die Anzahl der Eingabestellen festgelegt, bis zu der nach ENTER ein TAB auf die Konsole ausgegeben wird. Mit BS kann die Eingabe korrigiert werden. Mit sig = 0 wird ein Wert ohne Vorzeichen beginnen mit einer Ziffer erwartet. Bei ungleich 0 muss die Eingabe mit „+“ oder „-“ anfangen.

```
#include <io_funktionen.h>
unsigned long in_value;
printf("\r\n 36 Bit Wert = ");
in_value = get_ld(1, 4);
```

4.1 Der GCC
4.1.1 Konsolen Funktionen (10)

- 11. **get_18(**char *meld, unsigned int pos**);**

  Eingabe von Werten (18 Bit) hexadezimal mit dem STDIO Gerät. Mit dem Parameter pos wird die Anzahl der Eingabestellen festgelegt, bis zu der nach ENTER ein TAB auf die Konsole ausgegeben wird. Mit BS kann die Eingabe korrigiert werden. Der 1. Parameter ist eine Zeichenkette zur Eingabeaufforderung. Bei der Eingabe von Werten > 0x3ffff wird eine Fehlermeldung angezeigt und erneut zur Eingabe aufgefordert.

```
#include <io_funktionen.h>

unsigned long in_value;

in_value = get_18(„\r\n 18 Bit = 0x“, 4);
```

4.1 Der GCC
4.1.1 Konsolen Funktionen (11)

- 12. **get_18d(**char *meld, unsigned int sig, unsigned int pos**);**

  Eingabe von Werten (18 Bit) dezimal mit dem STDIO Gerät. Mit dem Parameter pos wird die Anzahl der Eingabestellen festgelegt, bis zu der nach ENTER ein TAB auf die Konsole ausgegeben wird. Mit BS kann die Eingabe korrigiert werden. Der 1. Parameter ist eine Zeichenkette zur Eingabeaufforderung. Bei der Eingabe von Werten die mehr als 18 Bit benötigen wird eine Fehler-meldung angezeigt und erneut zur Eingabe aufgefordert. Mit sig = 0 wird ein Wert ohne Vorzeichen beginnen mit einer Ziffer erwartet. Bei ungleich 0 muss die Eingabe mit „+“ oder „-“ anfangen.

```
#include <io_funktionen.h>

unsigned long in_value;

in_value = get_18d(„\r\n 18 Bit = “, 1, 4);
```

```
printf("\r\n\n 36 Bit Summen berechnen\
    \r\n   i \tSummand 1 \tSummand 2 \tSumme 9 Stellen\t Summe mit\
    \r\n\t\t\t\t\t\t\teiner Stelle min.\r\n");

unsigned int   z   = 0;       // Index fuer Sumanden
unsigned long  sum = 0;       // Summe     36 Bit
unsigned long  su1 = 0;       // Summand 1 36 Bit
unsigned long  su2 = 0;       // Summand 2 36 Bit
while  (1) {
    printf("\r\n %3d\t0x",z);
    su1 = get_lx(6);       // hex Eigabe
    printf("\t+ 0x");
    su2 = get_lx(4);       // hex Eingabe
    printf("\t= 0x");
    sum = su1 + su2;
    print_lx(sum, 9);      // hex Ausgabe mit 9 Stellen
    printf("\t = 0x");
    print_lx(sum, 1);      // hex Ausgabe mit mindestens einer Stelle
    z++;
}
```

36 Bit Summen berechnen

i	Summand 1	Summand 2	Summe 9 Stellen	Summe mit einer Stelle min.
0	0x0	+ 0x0	= 0x000000000	= 0x0
1	0x1	+ 0x1	= 0x000000002	= 0x2
2	0x12	+ 0x12	= 0x000000024	= 0x24
3	0x123	+ 0x123	= 0x000000246	= 0x246
4	0x1234	+ 0x1234	= 0x000002468	= 0x2468
5	0x12345	+ 0x12345	= 0x00002468A	= 0x2468A
6	0x123456	+ 0x123456	= 0x0002468AC	= 0x2468AC
7	0x1234567	+ 0x1234567	= 0x002468ACE	= 0x2468ACE
8	0x123456789	+ 0x123456789	= 0x2468ACF12	= 0x2468ACF12
9	0x23456789A	+ 0x23456789A	= 0x468ACF134	= 0x468ACF134
10	0x3456789AB	+ 0x3456789AB	= 0x68ACF1356	= 0x68ACF1356
11	0x456789ABC	+ 0x456789ABC	= 0x8ACF13578	= 0x8ACF13578
12	0x56789ABCD	+ 0x56789ABCD	= 0xACF13579A	= 0xACF13579A
13	0x6789ABCDE	+ 0x6789ABCDE	= 0xCF13579BC	= 0xCF13579BC
14	0x789ABCDEF	+ 0x789ABCDEF	= 0xF13579BDE	= 0xF13579BDE
15	0x89ABCDEF0	+ 0x89ABCDEF0	= 0x13579BDE0	= 0x13579BDE0
16	0x			

```
    printf("\r\n\n 36 Bit Summen berechnen\
        \r\n   i \tSummand 1 \tSummand 2 \tSumme 9 Stellen\t Summe
dezimal\r\n");

    unsigned int    z   = 0;        // Index fuer Sumanden
    unsigned long   sum = 0;        // Summe     36 Bit
    unsigned long   su1 = 0;        // Summand 1 36 Bit
    unsigned long   su2 = 0;        // Summand 2 36 Bit
    while   (1) {
        printf("\r\n %3d\t",z);
        su1 = get_ld(1, 7);         // dez Eingabe
        printf("\t+ ");
        su2 = get_ld(1, 5);         // dez Eingabe
        printf("\t= 0x");
        sum = su1 + su2;
        print_lx(sum, 9);           // hex Ausgabe
        printf("\t = ");
        print_ld(sum, 1, 11);       // dez Ausgabe
        z++;
    }
```

```
36 Bit Summen berechnen
  i     Summand 1       Summand 2       Summe 9 Stellen  Summe dezimal

  0     -9              + +123456789    = 0x0075BCD0C    = +00123456780
  1     -89             + +123456789    = 0x0075BCCBC    = +00123456700
  2     -789            + +123456789    = 0x0075BCA00    = +00123456000
  3     -6789           + +123456789    = 0x0075BB290    = +00123450000
  4     -56789          + +123456789    = 0x0075AEF40    = +00123400000
  5     -456789         + +123456789    = 0x00754D4C0    = +00123000000
  6     -3456789        + +123456789    = 0x007270E00    = +00120000000
  7     -23456789       + +123456789    = 0x005F5E100    = +00100000000
  8     -123456789      + +123456789    = 0x000000000    = +00000000000
  9     +123456789      + -9            = 0x0075BCD0C    = +00123456780
 10     +123456789      + -89           = 0x0075BCCBC    = +00123456700
 11     +123456789      + -789          = 0x0075BCA00    = +00123456000
 12     +123456789      + -6789         = 0x0075BB290    = +00123450000
 13     +123456789      + -56789        = 0x0075AEF40    = +00123400000
 14     +123456789      + -456789       = 0x00754D4C0    = +00123000000
 15     +123456789      + -3456789      = 0x007270E00    = +00120000000
 16     +123456789      + -23456789     = 0x005F5E100    = +00100000000
 17     +123456789      + -123456789    = 0x000000000    = +00000000000
 18
```

36 Bit Summen berechnen

i	Summand 1	Summand 2	Summe 9 Stellen	Summe dezimal
0	+5	+ +5	= 0x00000000A	= +00000000010
1	-5	+ +10	= 0x000000005	= +00000000005
2	+20	+ +20	= 0x000000028	= +00000000040
3	+34359738360	+ +7	= 0x7FFFFFFFF	= +34359738367
4	+34359738360	+ +8	= 0x800000000	= -34359738368
5	-34359738367	+ +7	= 0x800000008	= -34359738360
6	+20	+ +0	= 0x000000014	= +00000000020
7				

4.1 Der GCC
4.1.2 Funktion für 36 Bit Ergebnis vom MUL

```
__asm__("\n.include \"const.s\"");
/*
 * Multiplikation mit 36 Bit Produkt aus 18 Bit * 18 Bit
 */

long mul36(int fa1, int fa2){
             long prod;
   unsigned int  prodl;
   prodl = fa1;
   __asm__("MUL     %0,      %1":"+r"(prodl):"r"(fa2));
   __asm__("MOVS2I  %0,      SFR_MUL":"=r"(prod));
   prod  = prod << 18;     // SFR Inhalt nach HIGH schieben
   prod  = prod | prodl;   // LOW Einblenden
   return  prod;           // 36 Bit Produkt
}
```

```
MUL 18 Bit * 18 Bit = 36 Bit

Faktor 1 18 Bit = 0x2
Faktor 2 18 Bit = 0x4
Produkt  36 Bit = 0x8

Faktor 1 18 Bit = 0x20
Faktor 2 18 Bit = 0x20
Produkt  36 Bit = 0x400

Faktor 1 18 Bit = 0x222
Faktor 2 18 Bit = 0x222
Produkt  36 Bit = 0x48C84

Faktor 1 18 Bit = 0x4444
Faktor 2 18 Bit = 0x4444
Produkt  36 Bit = 0x12343210

Faktor 1 18 Bit = 0x
```

```
void main() {
   interrupt_disable();
   uart_wait_idle(UART_MONITOR);
   UART_MONITOR->ctrl_stat = UART_RX_EN|UART_TX_EN|
            UART_TWO_STOP|UART_DATA_LEN_8|UART_BPS_115200;
   stdio_uart_open(UART_MONITOR);
   interrupt_enable();
   printf("\r\n\n MUL 18 Bit * 18 Bit = 36 Bit Dezimal\r\n");
   while (1) {
      unsigned long  prod  = 0;  // Produkt   36 Bit
      unsigned int   fa1   = 0;  // Faktor  1 18 Bit
      unsigned int   fa2   = 0;  // Faktor  2 18 Bit
      fa1   = get_18d("\r\n Faktor 1 18 Bit = ", 1, 6);
      fa2   = get_18d("\r\n Faktor 2 18 Bit = ", 1, 3);
      prod  = mul36(fa1, fa2);
      printf("\r\n Produkt  36 Bit = ");
      print_ld(prod, 1, 1);
      printf("\r\n");
   }
}
```

```
MUL 18 Bit * 18 Bit = 36 Bit Dezimal

Faktor 1 18 Bit = +3
Faktor 2 18 Bit = -4
Produkt  36 Bit = -12

Faktor 1 18 Bit = +131071
Faktor 2 18 Bit = +131071
Produkt  36 Bit = +17179607041

Faktor 1 18 Bit = +131072      Wert > 18 Bit!
Faktor 1 18 Bit = +131073      Wert > 18 Bit!
Faktor 1 18 Bit = -131072
Faktor 2 18 Bit = -131072
Produkt  36 Bit = +17179869184

Faktor 1 18 Bit = -131073      Wert > 18 Bit!
Faktor 1 18 Bit = +1024
Faktor 2 18 Bit = +2
Produkt  36 Bit = +2048

Faktor 1 18 Bit =
```

4.1 Der GCC
4.1.3 Funktionen für das SFR_LED Register

```
#include <led7.h>

/*
 * Werte an 7 LEDs oder 7 Segmentanzeige am SFR_LED ausgeben.
 */
unsigned charwert1  = 0x7f;
unsigned charwert2  = 0;
unsigned charwert3  = 0xd;

led7_set(wert1);    // Alle 7 LEDs einschalten

wert2  = led7_get();// Wert2 ist jetzt auch 0x7f

led7_hex(wert3);    // „d“ wird auf 7 Segmentanzeige angezeigt
led7_hex(16);       // Bei Werten größer 15 ist die Anzeige aus-
                    // geschaltet
```

;