Aufgabenblatt 4: Verwendung des Stacks, BCD-Addition, Arithmetik

1 Aufgabenbeschreibung

In diesem Versuch werden Sie das bisher Gelernte vertiefen. Zunächst werden Sie sich näher mit der Funktion des Stapelspeichers (Stack) befassen. Danach werden Sie anhand einer Routine für die BCD Addition die Parameterübergabe an Subroutinen unter Zuhilfenahme der Prozessorregister üben. Die Übergabe von Daten über Speicherbereiche und bedingte Ausführung von Befehlen ist Gegenstand der letzten Übung.

2 Aufgabenstellung

2.1 Aufgabe 1: Verwendung des Stacks

Beim letzten Labortermin haben wir eine Subroutine (Unterprogramm) verwendet, das an beliebiger Stelle im Programm aufgerufen werden kann. Wie lassen sich diese Unterprogramme wiederum selbst innerhalb von Unterprogrammen aufrufen?

Die Rücksprungadressen müssen dazu nacheinander auf dem Stapelspeicher (= Stack) abgelegt und in umgekehrter Reihenfolge wieder aufgerufen werden. Der Stack-Zeiger (wir verwenden ab jetzt das Register **sp (= r13)** ausschließlich für diesen) darf nun auf keinen Fall mehr manipuliert werden, sonst funktioniert die Rückkehr von den Unterprogrammen nicht mehr. Beim Aufruf eines Unterprogramms muss der Wert des Programmzählers **pc (= r15)** in das Link Register **lr (= r14)** geladen und auf dem Stack abgelegt werden. Vor der Rückkehr wird dieser Wert in den Programmzähler zurückgeschrieben. Daher dürfen Unterprogramme auch nur beim Beenden verlassen werden und nicht über eine Abkürzung (Verzweigung).

Der Aufruf der Subroutinen erfolgt über den Befehl **Branch und Link (bl)**, um den aktuellen Wert des Programmzählers zu speichern.

Das Ablegen eines Wertes auf dem Stack wird auch als Push-, das Holen eines Wertes als Pop-Instruktion bezeichnet. Diese Instruktionen dürfen nur beim Aufruf und Beenden einer Subroutine verwendet werden. Wenn Sie aktuelle Werte auf dem Stack ablegen wollen, dann machen Sie besser einen zweiten auf ;-).

Überlegen Sie immer, welche Registerinhalte Sie nach dem Beenden der Subroutine wiederherstellen wollen und verwenden Sie gegebenenfalls dazu ebenfalls den Stack. Testen Sie den Aufruf der Subroutinen ausgiebig mit dem Debugger, um sicherzustellen, dass alles wie gewünscht funktioniert. Es kann natürlich vorkommen, dass bei vielfachem Aufruf von Subroutinen der Stack irgendwann einmal „überläuft“ oder das Programm überschreibt (das ist noch schlimmer!).

Zur Verwendung der Register wird im sogenannten *ARM Architecture Procedure Call Standard (AAPCS)* vorgeschlagen, die Register **r13, r14** und **r15** ausschließlich wie oben beschrieben zu verwenden. Die Register **r4** bis **r12** dienen der Aufnahme lokaler Variablen innerhalb von Subroutinen. Wenn eine Subroutine also von einer anderen aufgerufen wird, sollten diese Registerwerte auf dem Stack abgelegt werden um später wieder zurückgespeichert werden zu können.

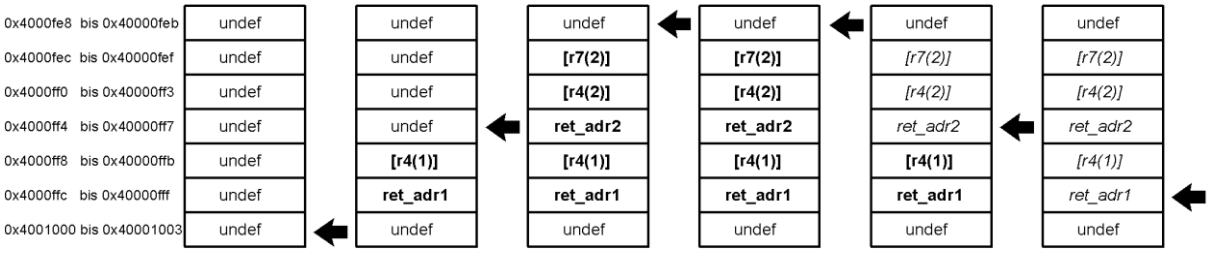
Die Werte der Register **r0** bis **r3** dienen der Parameterübergabe an Subroutinen in dieser Reihenfolge. Wenn mehr als vier Parameter übergeben werden, geschieht dies über den Stack (das ist nur etwas für erfahrene Programmierer). Die Werte der Register **r0** bis **r3** sind flüchtig und werden nicht abgespeichert. Die Rückgabe der Ergebniswerte erfolgt über die selben Register in der gleichen Reihenfolge,

die Rückgabe von mehr als vier Werten erfolgt wieder über den Stack. Wenn die Parameterwerte mehr als 32 Bit verwenden, müssen sie auf mehrere Register aufgeteilt werden. Der AAPCS erlaubt sogar das Mischen von Assembler und C-Programmen!

Zu Beginn des Programms wird der Stack-Zeiger auf die Adresse 0x40001000 gelegt

ldr sp, =0x40001000

Die nachfolgende Abbildung zeigt die Verwendung eines full descending (absteigenden) Stacks, der beginnend mit der höchsten Adresse verwendet wird. Fett gedruckte Werte wurden gerade auf dem Stack abgelegt und überschreiben alte Werte. Bereits gelesene Werte verbleiben auf dem Stack und sind kursiv gedruckt. Durch „undef“ gekennzeichnete Werte wurden noch nicht beschrieben.

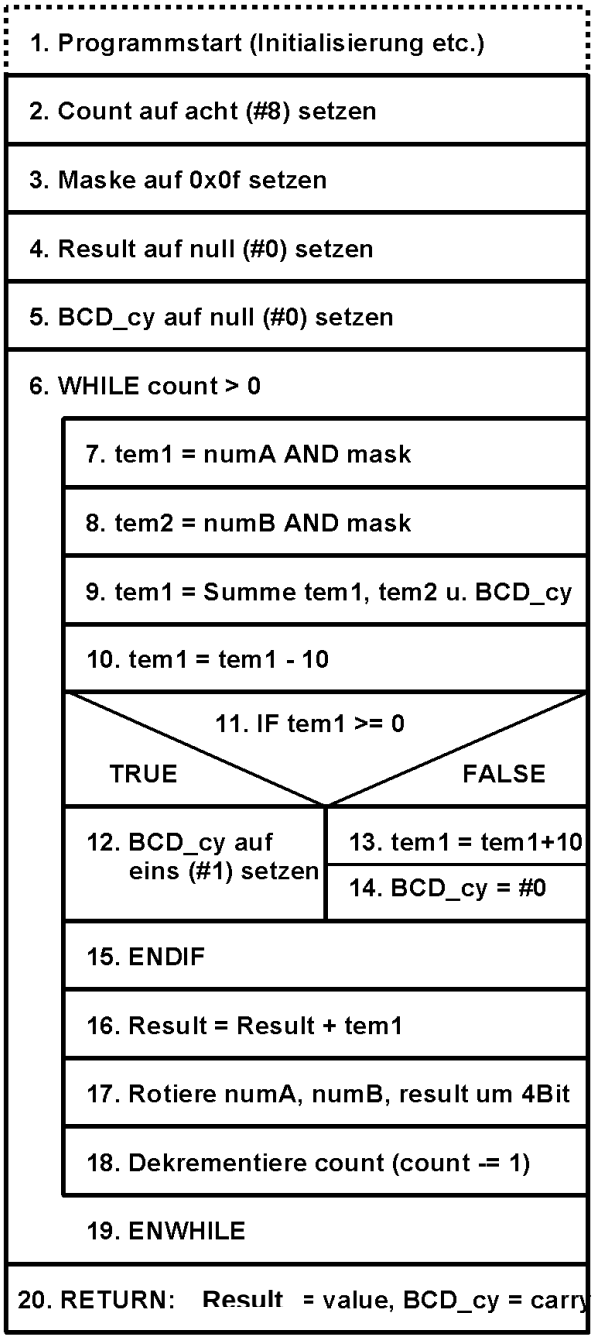


Aufgabe: Legen Sie ein Rahmenprogramm zum verschachtelten mehrfachen Aufrufen von Subroutinen an. Diese Aufrufe erfolgen immer über den Branch und Link (bl) Befehl. Schreiben Sie mehrere Subroutinen mit Test Labels zum Rücksprung. Gehen Sie dazu folgendermaßen vor:

1. **In den Subroutinen verwenden Sie nach dem Einsprung den Befehl stmfd sp!, {rx, lr}, wobei rx für die zu sichernden Register steht – probieren Sie einfach die Verwendung einiger Registerwerte aus.**
2. **Innerhalb der Subroutine springen Sie eine weitere an (Einsprung s. o.).**
3. **Die Rücksprünge erfolgen durch den Befehl ldmfd sp!, {rx, pc}. Vergessen sie nicht die Ausrufezeichen, um den Wert des Stack-Zeigers zu aktualisieren.**
4. **Zum Testen setzen Sie einen Breakpoint auf die Befehlszeile des Aufrufs der Subroutine und merken sich die Registerinhalte, vor allem den Wert des Programmzählers und des Stack-Zeigers. Führen sie den Aufruf der Subroutine in Einzelschritten aus. Vergewissern Sie sich, dass der Wert des Stack-Zeigers nach Aufruf des Befehls STMFD (Store Memory Full Descending) um vier (4!) Adressen dekrementiert wird (warum eigentlich?).**
5. **Setzen Sie einen weiteren Breakpoint auf die Programmzeile des Befehlsaufrufs LDMFD (Load Memory Full Descending) und beobachten Sie ebenfalls den Wert des Stack-Zeigers und des Programmzählers. Überprüfen Sie die Programmzeile, die nach dem Aufruf dieses Befehls angesprungen wird und stellen Sie sicher, dass dieser Rücksprung an die richtige Programmadresse erfolgt und alle zurückgeschriebenen Registerwerte korrekt sind. Überprüfen Sie ebenfalls die Korrektheit der durch die Subroutine zurückgelieferten Werte.**

2.2 Aufgabe 2: BCD-Addition

Aus der Vorlesung Elektronik 2 ist noch das BCD-Zahlenformat (Binary Coded Decimal) bekannt, in dem nur die Werte $0_{10}..9_{10}$ (0000_2 ..1001_2) verwendet werden. In diesem Zahlenformat kann natürlich auch die Addition verwendet werden. Die Umwandlung des BCD- in das Binärformat ist sehr einfach, da jedes Nibble eine Dezimalstelle speichert, z.B.: 95276_{10} = 0000 0000 0000 1001 0101 0010 0111 0110_2.



Das Nassi-Shneiderman-Struktogramm der BCD-Addition ist nachfolgend abgebildet:

Zum Packen der Dezimalstellen (= Nibbles) in die 32bit-Worte müssen diese Nibbles an die entsprechenden Stellen verschoben werden. Dazu dienen sogenannte „rotates“, die parallel zu mov-Anweisungen ausgeführt werden, z.B.:
mov r1, r1, ror #4,
die den Inhalt von **r1** um vier Bitstellen nach rechts verschiebt.

Die einzelnen Nibbles können beginnend mit der niedrigsten Stelle einfach addiert werden, nachdem sie an die richtigen Positionen verschoben worden.

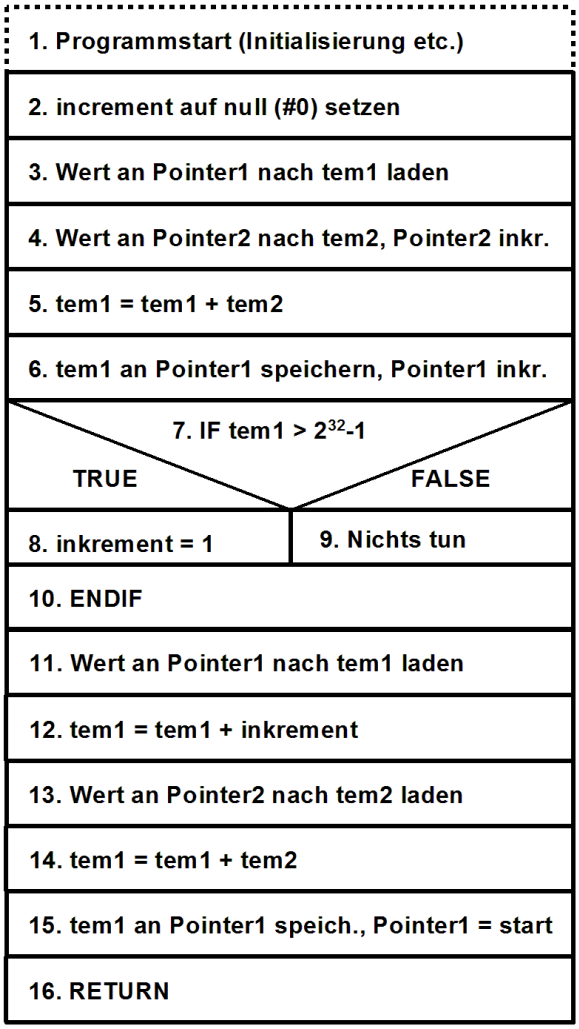
Vor der Verwendung eines Operanden kann dieser an die richtige Position geschoben werden, die Verwendung bedingter Befehle kann das Programm stark verkürzen.

Aufgabe: Schreiben Sie die Subroutine zur BCD-Addition. Beachten Sie dabei folgendes:

1. **Die beiden Operanden müssen im Hexadezimalformat eingegeben werden, obwohl sie als Dezimalzahlen interpretiert werden.**
2. **Verwenden Sie r4 für die Variable „Count“, r5 für die „Maske“, r6 für „BCD_cy“ (Carry) und r7 für „Result“.**
3. **Speichern Sie die Inhalte der Register r4 bis r9 und des Linkregisters auf den Stack: stmfd sp!, {r4-r9, lr}**
4. **Lesen Sie die Inhalte der Register r4 bis r9 und des pc beim Rücksprung vom Stack: ldmfd sp!, {r4-r9, pc}.**

2.3 Aufgabe 3: Arithmetik – Addition zweier 64bit-Zahlen

Wir haben die 32bit-Addition bereits zum zweiten Labortermin durchgeführt. Dabei haben wir die Operanden aus Registern geladen, normalerweise sind die Werte jedoch an fortlaufenden Speicheradressen abgelegt, auf die über Zeiger zugegriffen wird. Im „Little-Endian-Modus“ wird der niedrig signifikante Teil an der niedrigen Adresse, im „Big-Endian-Modus“ an der höheren Adresse abgelegt. Mögliche Überläufe, die bei den Additionen auftreten können, werden durch „Carries“ aufgefangen, die natürlich weiterverarbeitet werden müssen.



Das nebenstehende Struktogramm zeigt den Algorithmus einer Subroutine zur 64bit-Addition unter Verwendung von Zeigern. Die Subroutine bekommt als Parameter die Zeiger auf die Zahlen übergeben und überschreibt den ersten Parameterwert mit der Summe.

Aufgabe: Implementieren Sie die Subroutine zunächst ohne Abspeichern des „Carries“ für den Summenwert. Verwenden Sie statt der Vorgehensweise zur Lösung der Aufgabe des zweiten Labortermins durch verschachtelte Sprünge die bedingte Ausführung von Befehlen (conditional execution). Der ADC-Befehl (Add with Carry):

adc{<cond>}{s} <rd>, <rn>, <shifter_operand>

wertet gleichzeitig das Carry-Flag aus.

Gehen Sie folgendermaßen vor:

1. **Beim Aufruf der Subroutine speichern Sie r5 bis r7 und das lr auf den Stack. r5 dient der Aufnahme der niederwertigen Bits des ersten Operanden, auf die über den Zeiger r0 zugegriffen wird, r6 dient der Aufnahme der niederwertigen Bits des zweiten Operanden, auf die über den Zeiger r1 zugegriffen wird.**
2. **Der Zeiger wird danach um vier Adressen (32bit) inkrementiert: ldr r6, [r1], #4 .**
3. **Nach der Addition werden die niederwertigen Bits wieder über den Zeiger r0 im Register r5 abgespeichert, der anschließend ebenfalls um vier Adressen inkrementiert wird.**
4. **Nach der Addition wird das Carry-Flag im Register r7 gespeichert. Alternativ können Sie den adc Befehl verwenden, siehe Punkt 7.**
5. **Anschließend werden die höherwertigen Bits addiert, auf die über die selben Zeiger zugegriffen wird. Zur Summe wird noch gegebenenfalls das Carry-Bit addiert, das Ergebnis wird über den Zeiger im Register r5 abgelegt (in dem zuvor die signifikanten Bits des ersten Operanden gespeichert waren).**
6. **Zum Schluss wird der Zeiger r0 wieder um vier Adressen dekrementiert (#-4) und die Register r5 bis r7 und der pc vom Stack zurückgelesen.**
7. **Zur Addition der signifikanten Bits lässt sich der Befehl adcs (add carry with set flags) verwenden, der das Carry-Bit der vorangehenden Addition der niederwertigen Bits gleich mit addiert. Gleichzeitig kann ein Überlauf wieder erneut im Carry-Bit abgespeichert werden!**