Aufgabenblatt 2 Zahlensysteme

1 64 Bit Addition

Implementieren Sie eine 64 Bit Addition zweier Zahlen. Die beiden Eingabezahlen liegen im 4 Registern (R0 bis R3) vor. Das Ergebnis soll in den Registern R8 und R9 sowie R10 für den Überlauf der höherwertigen Stellen abgelegt werden. Sie müssen nur einen der drei vorgeschlagenen Lösungsansätze implementieren.

1.1 Lösungsansatz 1 und 2

Wird kein entsprechender Additionsbefehl, der automatisch den Überlauf berücksichtigt, verwendet, so müssen hier drei Additionen durchgeführt werden (Abbildung 1). Zuerst werden die beiden niederwertigen 32 Bit Words zusammenaddiert. Sollte dabei ein Überlauf entstehen (C Flag = 1) so muss dieser in einem separaten Register temporär gespeichert werden. Danach können die beiden höherwertigen 32 Bit Words miteinander addiert werden. Auch der hier unter Umständen entstandene Überlauf muss gesondert gespeichert werden. Als dritte Addition ist das Ergebnis der zweiten Addition mit dem Überlauf aus der ersten Addition zu verrechnen. Auch hierbei kann ein Überlauf entstehen, dieser muss dann entsprechend gespeichert werden.

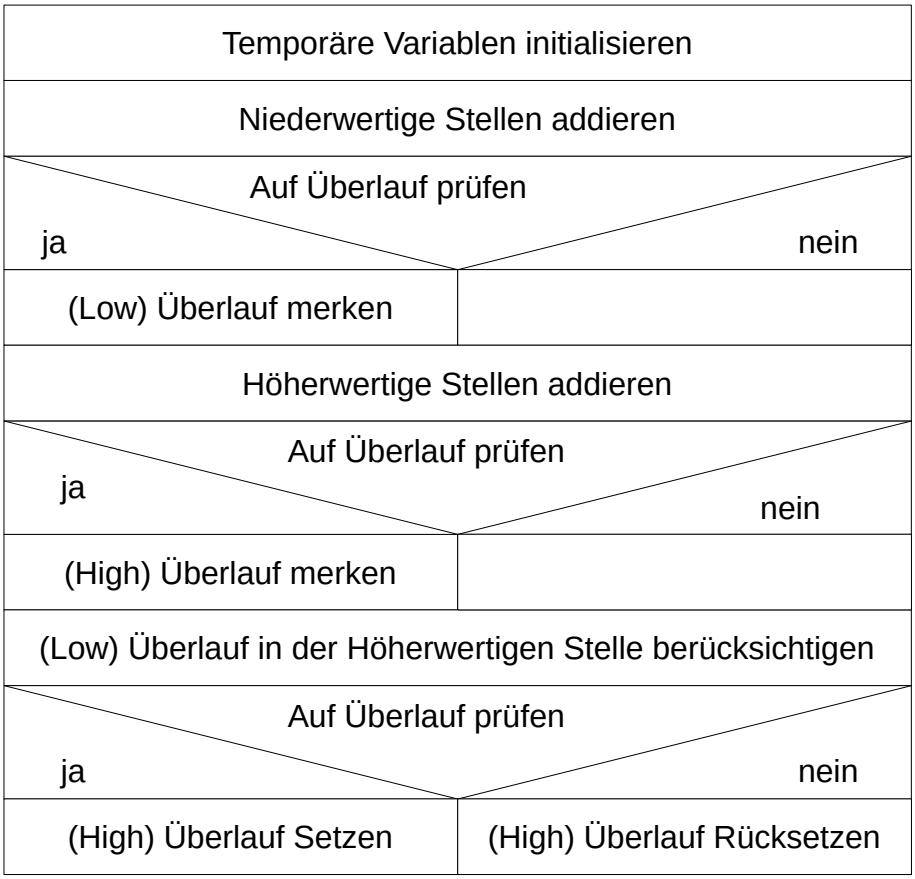


Abbildung 1: 64 Bit Addition, lange Version

Dieser Algorithmus kann entweder mit Sprüngen (lang, Ansatz 1) oder mit bedingter Befehlsausführung (kurzer, Ansatz 2) umgesetzt werden.

Skript:

2.6.1 Addition
2.6.2 Addition ohne Carry
2.8.2.5 Anwendungsfall – if...else

1.2 Lösungsansatz 3 (Level - Streber)

Wird ein entsprechender Additionsbefehl, der den Überlauf automatisch berücksichtigt, verwendet, so kann die kürzere und einfachere Version implementiert werden. Hier sind keine Sprünge mehr notwendig und die Verwendung von bedingter Ausführung beschränkt sich auf das Speichern des Überlaufs des höherwertigen 32 Bit Wortes. Dieser Algorithmus ist der kürzeste und der schnellste der drei.

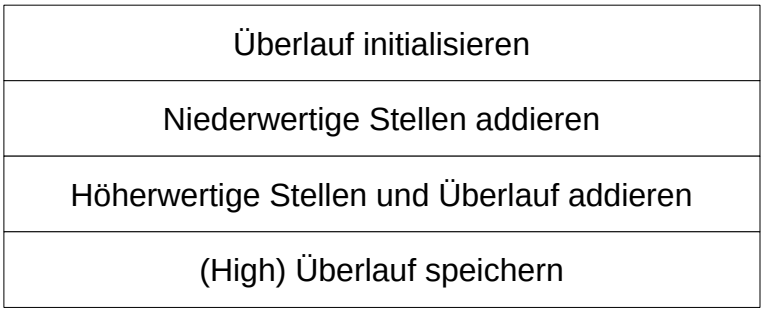


Abbildung 2: 64 Bit Addition, kurze Version

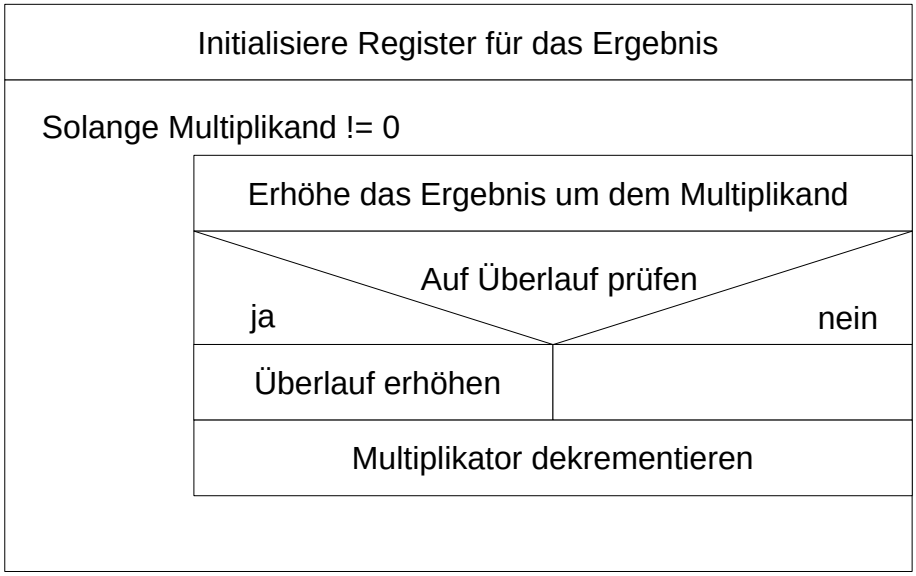
Skript:
2.6.1 Addition
2.6.2 Addition ohne Carry
2.6.3 Addition mit Überlauf
2.8.2.5 Anwendungsfall – if...else

2 Multiplikation

Da manche einfachen Mikrocontroller keine Multiplikationsbefehle besitzen müssen diese bei Bedarf selbst implementiert werden. In dieser Aufgabe wollen wir lernen wie das funktioniert. Im Unterschied zu einer Addition bzw. Subtraktion ist die Implementierung einer Multiplikation oder Division nicht für vorzeichenlose und vorzeichenbehaftete Zahlen gleich. Einfachheitshalber beschäftigen wir uns hier nur mit der vorzeichenlosen Multiplikation. Es muss nur einer der beiden hier vorgestellten Algorithmen umgesetzt werden.

2.1 Multiplikation durch Addition

Dieser Algorithmus an sich ist sehr einfach. In einer Schleife wird der Multiplikand so oft akkumuliert wie die Höhe des Wertes des Multiplikators ist. Da bei einer 32 Bit Multiplikation das Ergebnis größer als 32 Bit werden kann (max. 64 Bit), so muss dabei die Anzahl der Überläufe mitgezählt werden. Nachdem Sie diesen Algorithmus umgesetzt haben, überlegen Sie sich wie Sie ihn von einer ***while*** in eine ***do...while Schleife*** konvertieren können. Was müssen Sie beachten, damit der Algorithmus auch mit ***do...while*** immer korrekt funktioniert? Präsentieren Sie die konvertierte Version.



tAbbildung 3: Multiplikation durch wiederholte Addition

Tipp:
Die optimale ***do..while*** Version ist nicht kürzer (code), aber dafür deutlich schneller in der Ausführung.

Skript
2.6.4 Subtraktion
2.8.2.2 Anwendungsfall – Die while{} Schleife
2.8.2.3 Anwendungsfall – Die do..while{} Schleife

2.2 Multiplikation durch Faktorisierung (Level Streber)

Die Multiplikation durch Addition ist sehr zeitaufwendig, da bei einer Multiplikation mit n die Addition auch n mal ausgeführt werden muss. Jede CPU hat aber eine einfache Multiplikations-/Divisionseinheit die sehr schnelle Multiplikationen/Divisionen mit 2er Potenzen ausführen kann. Also müssen wir die Zahl in Ihre 2er Potenzen Faktoren „zerlegen“.
Schauen wir uns das mal an einem Beispiel an, wie wollen die Zahl 4 mit 5 multiplizieren. Die 5 entspricht binär 0b0101 also $0*2^3+ 1* 2^2 + 0*2^1 + 1*2^0$, die gelöschten Bits im Multiplikator werden nicht berücksichtigt, die gesetzten Bits nehmen den Wert des Multiplikator an somit :
$4 * 5 = 0 * 2^3 + 4 * 2^2 + 0 * 2^1 + 4 * 2^0$
$4 * 5 = 0 + 16 + 0 + 4 = 20$

Die Multiplikation des Multiplikanden mit den Zweierpotenzen erreicht man über die Schiebearithmetik. So ist:
$4 * 2^0$ => 4 « 0 = 4
$4 * 2^1$ => 4 « 1 = 8
$4 * 2^2$ => 4 « 2 = 16
$4 * 2^3$ => 4 « 3 = 32

also können wir es auch folgendermaßen aufschreiben:
$4 * 5 = 4 * 2^2 + 4 * 2^0$ <=> 4 « 2 + 4 « 0

Wie wir sehen ist der Algorithmus an sich nicht sonderlich kompliziert. Wir müssen nur feststellen, welche der Schiebeoperationen wir für das Endergebnis berücksichtigen müssen und die Ergebnisse dieser Operationen zusammen-summieren. Die Feststellung, ob eine Potenz berücksichtigt werden muss, kann mit einer entsprechenden Schiebeoperation (LOGISCH!) erfolgen (Abbildung 4). Diese kann kontinuierlich in jedem Schleifendurchlauf erfolgen. Falls nach einer Schiebung um 1 ein Bit in den Carry „reingefallen“ ist, so muss der Faktor berücksichtigt werden, ist der Carry hingegen 0 so wird die Addition mit dem Faktor ignoriert. Zum Schluss der Schleife muss der Multiplikand noch mit 2 multipliziert werden, um den folgenden Faktor zu generieren. Auch dies kann mit einem Schiebebefehl erfolgen, diesmal nach rechts. Bleiben im Multiplikator keine weiteren Faktoren mehr vorhanden, so ist die Multiplikation abgeschlossen. Den Verlauf der Zwischenschritte solch einer Multiplikation können Sie in der Abbildung 5 sehen. Dort können Sie beobachten wie der Multiplikand mit jedem Schritt wächst und der Multiplikator immer kleiner wird, bis dieser zwangsläufig den Wert 0 erreicht. Nach dem der Multiplikator den Wert 0 erreicht hat, ist das Ergebnis vollständig und die Multiplikation damit abgeschlossen.

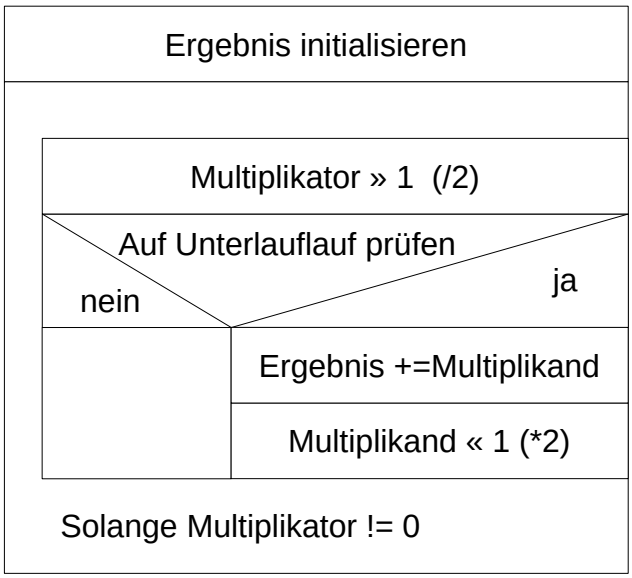


Abbildung 4: Multiplikation durch Faktorisierung

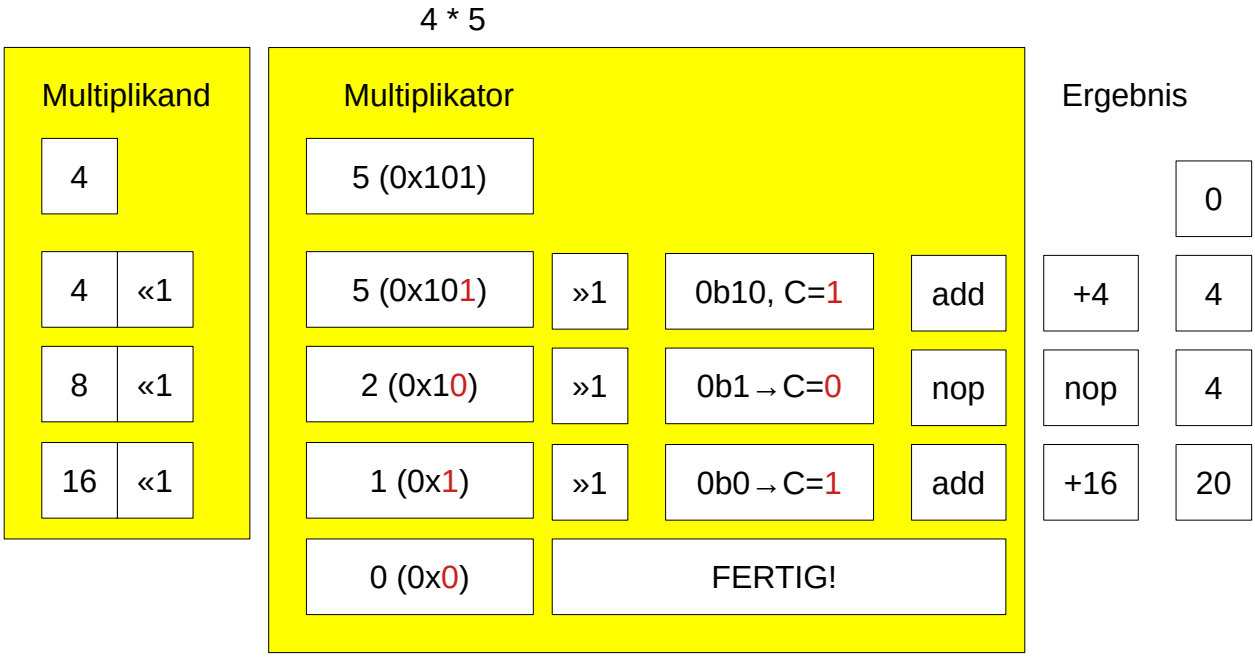


*Abbildung 5: Verlauf der Faktorisierung bei einer Multiplikation mit 4 * 5*

Das ganze hat allerdings einen bzw. zwei Hacken, da sowohl das Ergebnis, als auch der Multiplikand überlaufen können, so dass beide als 64 Bit Werte implementiert werden müssen. Der nächste Schritt wäre dann den 32 Multiplikanden auf 64 Bit zu erweitern (Abbildung 6). Da dieser immer mit 2 multipliziert wird, kann das durch Schiebung der beiden Teile jeweils um 1 Bit erreicht werden. Es entsteht dabei allerdings ein weiteres Problem, der (Schiebe-) Überlauf der niederwertigen Stellen kann nicht automatisch übernommen werden. Da der niederwertigste Bit nach einer Schiebung nach links immer 0 ist, kann der Überlauf mittels eines Logisch-binären Befehls einkopiert werden.

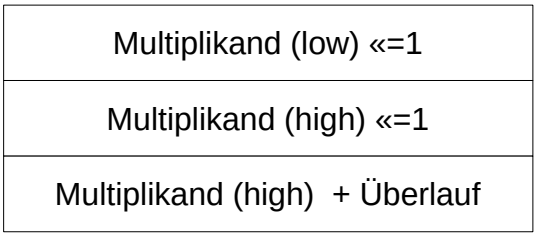


Abbildung 6: 64Bit Schiebung

Das zweite Problem, das es zu lösen gilt, ist, dass das Ergebnis ein höheres Wert als mit 32 Bit darstellbar sind. Dieses Problem haben wir aber bereits in der Aufgabe 2.1 gelöst. Hauptsächlich der dritte Ansatz bietet sich wegen seiner Kompaktheit hier an. Da die Multiplikation nicht über 64 Bit hinausgehen kann, ist hier die Beachtung des Überlaufs im 64ten Bit nicht notwendig. Dafür gibt es hier ein anderes Problem, da die Addition selbst durch den Carry-Bit des Schiebens getriggert wird. Die Lösung ist sehr einfach, nämlich...

Tipp: Implementieren Sie diese Aufgabe in drei Stufen

1. Zuerst eine reine 32 Bit Multiplikation (testen!)
2. Dann erweitern Sie den Multiplikanden auf 64 Bit, inklusive Test, ob dieser wirklich komplett in jedem Zyklus verschoben wird.
3. Im letzten Schritt erfolgt die Erweiterung des Ergebnisses auf 64 Bit.

Skript:
2.8.2.1 Anwendungsfall – Die for Schleife
2.8.2.2 Anwendungsfall – Die while{} Schleife
2.8.2.3 Anwendungsfall – Die do..while{} Schleife
2.1.2.1 Logisches Schieben nach links
2.1.2.2 Logisches Schieben nach rechts

3 Werte Binarisieren

Binarisieren Sie 8 nacheinander im Speicher liegende 32 Bit Zahlen. Nach dieser Operation hat jeder der Werte nur noch einen Wert von entweder 0 oder 1. Die Ergebnisse speichern Sie nacheinander in einem Register so, dass der erste Wert in das 7 Bit im Register aufgenommen wird, der zweite in das sechste etc. (Abbildung 7). Bitte beachten Sie, dass die Speicheradressen im Beispiel nur zum besseren Verständnis angegeben sind und nicht mit Ihren eigenen übereinstimmen werden. Den Zugriff auf die Werte selbst erhalten Sie über die indirekte Adressierung einer ***load*** Anweisung.

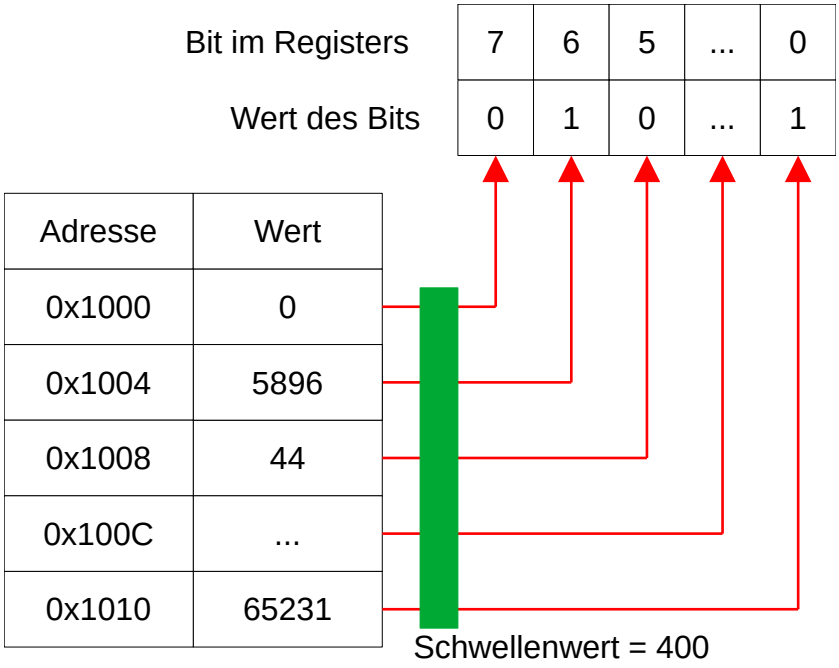


Abbildung 7: Werte Binarisieren

Skript:
1.13 Globale Variablen im Speicher
2.1.2.1 Logisches Schieben nach links
2.1.2.2 Logisches Schieben nach rechts
2.2.1 Werte aus Speicherstellen (Variablen) laden
2.2.4.2 Indirekte Adressierung mit Postinkrement / Postdekrement
2.7.4 Binäres ODER (orr)

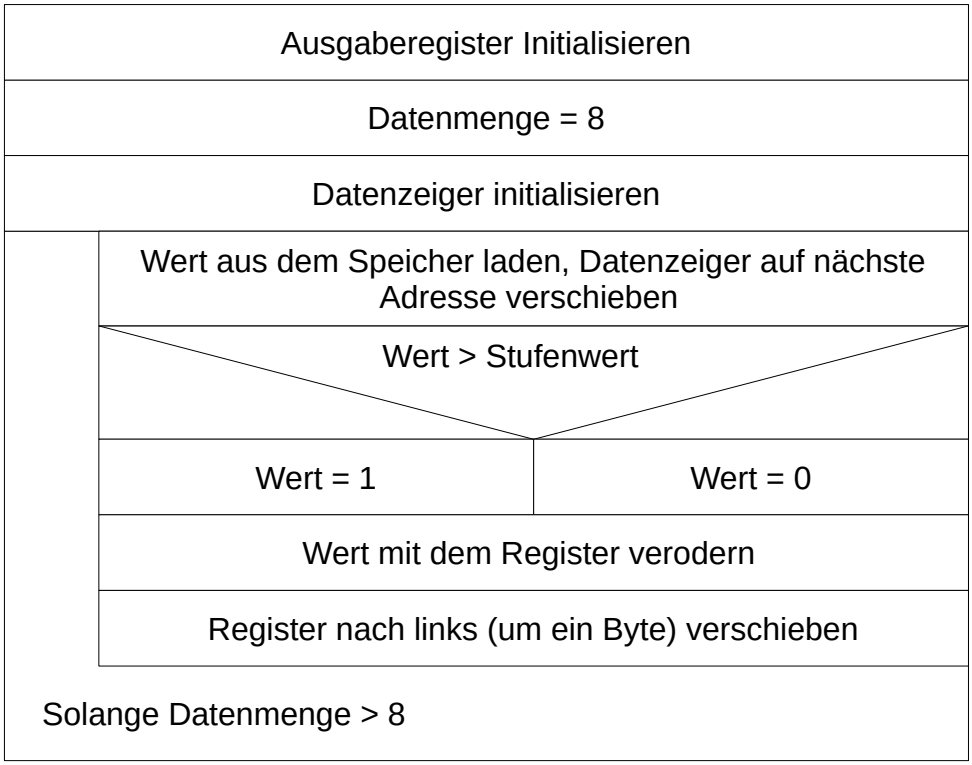


Abbildung 8: Datenbinarisierung