Technische Hochschule Mittelhessen

# Dokumentation der Projektarbeit

Systemroutinen für einen RiSC16-Prozessor

*vorgelegt von:*
*Arndt Karger*
*Elektro- und Informationstechnik*
*Matrikel-Nr. 5317057*

Betreuer:
Prof. Dr.-Ing. Werner Bonath

Gießen, 9. November 2022

# Selbstständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Hochschule oder Prüfungsstelle vorgelegen.

_____

Ort, Datum, Unterschrift

# Inhaltsverzeichnis

**1 Einleitung** **1**

**2 Aufgabenstellung** **1**

**3 Erstellte Routinen** **2**

3.1 Allgemein . . . . . 2

3.2 Multiplikation via Addition . . . . . 4

3.3 Bitweise Multiplikation . . . . . 4

**4 Zusammenfassung** **6**

**A Anhang** **7**

A.1 Verwendung der entworfenen Routinen . . . . . 7

A.2 Multiplikation via Addition: (MULv2) . . . . . 8

A.3 Bitweise Multiplikation: (MULv3) . . . . . 10

**Literaturverzeichnis** **13**

# 1 Einleitung

Bei dem vorliegenden RiSC16-Prozessor handelt es sich um eine der MIPS-Architektur (Microprocessor without interlocked pipeline stages) [AKT22] ähnlichen Architektur [Bon22], welche über einen 16 Bit Adressraum, acht Register und acht Befehle verfügt. Des Weiteren stehen ein Assembler [Jac22], ein Simulator [Bon21] sowie eine Hardwareimplementierung als FPGA (Field Programmable Gate Array) zur Verfügung.

Wie bei jeder Neuentwicklung ist noch nicht geklärt, ob die Hardwareimplementierung fehlerfrei funktioniert. Unklar ist auch die Leistungsfähigkeit der Prototypenhardware und ob umfangreiche Programme korrekt abgearbeitet werden können. Um das sicherstellen zu können muss der Prozessor einem ersten Test unterzogen werden. Da hier nicht viel Programmcode verfügbar ist, muss dieser erstellt werden. Des Weiteren ist der tatsächliche Programmieraufwand nicht bekannt, welcher zur Implementierung von nicht von der Architektur unterstützen Funktionen nötig ist.

# 2 Aufgabenstellung

Ziel dieser Projektarbeit war das Schreiben von Systemroutinen in Assembler, um die Funktion des Prozessors zu veranschaulichen, sowie die Hardwareimplementierung zu testen. Es sollte deutlich gemacht werden, dass auch mit einem einfachen Befehlssatz komplexe Probleme gelöst werden können. Dies wurde beispielhaft an zwei möglichen Multiplikationsalgorithmen durchgeführt. Die beiden Algorithmen stellen die Multiplikation via Addition und die bitweise Multiplikation dar.

Zuerst wurde ein Stack in Software implementiert, um das Sichern von Registern beim Aufruf der Routinen möglich zu machen. Dann wurde für die Multiplikation via Addition ein Flussdiagramm (siehe Anhang A.2) erstellt und in Assembler implementiert. Nach einer kurzen Literaturrecherche über die genaue Funktion der bitweisen Multiplikation wurde auch hierfür ein Flussdiagramm erstellt (siehe Anhang A.3) und schließlich implementiert. Der Code wurde während der Entwicklungsphase in dem von Herrn Prof. Dr.-Ing. Bonath entwickelten Simulator fortlaufend getestet und mit der Versionsverwaltungssoftware Git getrackt.

# 3 Erstellte Routinen

## 3.1 Allgemein

Der acht Befehle umfassende Befehlssatz der RiSC16-Architektur gliedert sich in:

- zwei Befehle zur Addition (addiere Register mit Register *add* und addiere Register mit Ganzzahl *addi*),
- zwei Befehle für den Speicherzugriff (speichern *sw* und laden *lw*),
- einen Logikbefehl (*nand*),
- einen Befehl zum Laden der oberen zehn Bits einer Ganzzahl (*lui*),
- sowie zwei Sprungbefehlen (bedingter Sprung bei Ungleichheit *bne* und Sprung zu Adresse *jalr*).

Zusätzlich zu diesen acht Befehlen stehen im Assembler mehrere Makros zur Verfügung, von denen zwei verwendet wurden. Das eine Makro (*movi*) lädt die Adresse eines Markers in ein Register. Das Andere (*halt*) stoppt den Programmablauf. Von den acht bereitgestellten Registern sind sieben frei verwendbar. Das erste Register (r0) beinhaltet immer den Wert $0000_{hex}$ [Jac00, S. 1].

Die RiSC16-Architektur verfügt über keinerlei Stackoperationen. Daher wird beim Programmstart ein Stack in Software initialisiert. Das Register r7 dient hierbei als Stackpointer (SP). Dieses wird zuerst mit der Startadresse des Stacks (hier $7FFF_{hex}$) beschrieben. Dieser Wert steht hart im Speicher und wird über den Marker „Stack_adr“ abgerufen (siehe Abbildung 1). Der Stack wurde Bottom-Up implementiert. Das bedeutet, dass er zu kleiner werdenden Adressen hin wächst. Das Register r7 zeigt immer auf den Top-Of-Stack (TOS). Das heißt, der SP zeigt immer auf das zuletzt abgelegte Element. Die Größe des Stacks (hier 64 Zeilen) ist rein hardwareabhängig, die Grenzen des Stacks werden nicht überwacht.

Die Funktion *push* wird durch einmaliges dekrementieren des SPs und speichern eines Registers mit dem Befehl *sw* (store word) erreicht (siehe Abbildung 1 line 12 folgende). Analog wird die Funktion *pop* durch das Laden eines Werts aus dem Speicher via *lw* (load word) und anschließendem inkrementieren des SPs erreicht. Eine indirekte Adressierung ist mittels eines Immediate-Offsets möglich (siehe Abbildung 1 line 14).

Prinzipiell ist das ausreichend, um mit dem Stack arbeiten zu können. Um eine bessere Lesbarkeit beim Debuggen zu schaffen, werden in der Initialisierung noch der Anfang sowie das Ende des Stacks markiert. Dazu wird in das Register r1 der Wert $-1_{dez} \mathrel{\widehat{=}} FFFF_{hex}$ geschrieben und an die erste und letzte Adresse im Stack gepusht (siehe Abbildung 1 line 11 fort folgende).

```
8   #initialisiere Stack
9
10              lw r7, r0, Stack_adr    #Stack Adresse in r7 laden, startet bei 7FFF
11              lui r1, 0               #ffff in r1 laden
12              addi r1, r1, -1
13              sw r1, r7, 0            #stack Beginn markieren
14              sw r1, r7, -64          #stack Ende markieren
15              lui r1, 0               #r1 aufräumen
16              bne r7, r0, Start       #übergehe die .fill
17
18  Stack_adr:  .fill 32767
19  debug_counter: .fill 0
20
21
22  Start:      lui r1, 0
```

Abbildung 1: Initialisierung des Stacks

Des Weiteren wurde eine Übergabekonvention erarbeitet, die Folgendes vorsieht: Es können bis zu zwei Werte an Unterprogramme übergeben werden. Diese müssen vor Aufruf des Unterprogramms in die Register r1 und r2 geladen werden (siehe Abbildung 2 line 39 folgende). Das Unterprogramm hat die Aufgabe sicherzustellen, dass nach dem Rücksprung die Registerinhalte der beiden Übergaberegister unverändert bleiben. Der Rückgabewert wird im Register r3 zurückgegeben (siehe Anhang A.1).

Der eigentliche Programmaufruf erfolgt über das Makro *movi*. Dieses lädt die Adresse eines Markers in ein angebenenes Register, hier r5 (siehe Abbildung 2 line 43). Im Anschluss erfolgt ein Sprung durch den Befehl *jalr* zu der Adresse aus r5. Die Rücksprungadresse wird in r6 abgelegt (siehe Abbildung 2 line 44).

```
38  #call MULv3: r3 = r1 * r2
39              addi r1, r1, 1
40              addi r2, r2, -3
41
42  #halt #debug
43              movi r5, MULv3
44              jalr r6, r5
```

Abbildung 2: Aufruf des Unterprogramms MULv3

## 3.2 Multiplikation via Addition

Eine Möglichkeit eine Multiplikation zu implementieren besteht darin, den Multiplikanten so oft zu sich selbst dazu zu addieren, wie es der Multiplikator vorschreibt. Aus $X = 3 \cdot 3$ wird $X = 3 + 3 + 3$. Diese Art der Multiplikation ist sehr einfach durch eine Schleife zu realisieren und wurde in der Routine *MULv2* angewendet (siehe Anhang A.2). Hierbei wird der Multiplikant (r2) solange auf das Ergebnisregister (r3) addiert, bis der Multiplikator (r1) zu Null geworden ist (siehe Abbildung 3 line 85 fort folgende). Vor Ausführung der Routine wird überprüft, ob Multiplikator oder Multiplikant gleich Null sind (siehe Abbildung 3 line 79 fort folgende) und gegebenenfalls wird Null zurück gegeben. Eine vorzeichenbehaftete Multiplikation ist nicht vorgesehen, ist jedoch prinzipiell möglich solange nur der Multiplikant negativ ist.

```
78  #wird mit 0 multipliziert?
79              bne r1, r0, r1_OK       #ist r1=0?
80              bne r7, r0, MUL_finish  #fertig
81
82  r1_OK:      bne r2, r0, MUL_loop    #ist r2=0?
83              bne r7, r0, MUL_finish  #fertig
84
85  MUL_loop:   add r3, r3, r2          #addiere r3 mit r2
86              addi r1, r1, -1         #r1 = r1 - 1
87              bne r1, r0, MUL_loop    #ist r1=0?
```

Abbildung 3: Multiplikation via Addition

## 3.3 Bitweise Multiplikation

Eine andere Möglichkeit der Multiplikation besteht darin, zwei Zahlen bitweise zu multiplizieren. Der entwickelte Multiplikationsalgorithmus funktioniert nach folgendem Prinzip:

Beispielhaft wird im Nachfolgenden die Multiplikation von zwei Acht-Bit-Zahlen ($n = 8$) betrachtet. Davon ist eine Zahl der Multiplikant und die Andere der Multiplikator. Zuerst wird das Most Significant Bit (MSB), das siebte Bit, des Multiplikators (r2) untersucht. Ist dieses gesetzt, so wird der Multiplikant (r1) um die Wertigkeit des untersuchten Bits, im ersten Schritt siebenmal, nach links geschoben und auf das Ergebnisregister (r3) addiert. Ist das MSB nicht gesetzt, so wird die Addition nicht ausgeführt [Hol22].

Diesen Prozess müssen wir für alle Bits des Multiplikators durchführen, in unserem Beispiel ergibt das acht Wiederholungen (siehe Abbildung 4). Dieser Prozess ist unabhängig von der Bitbreite, da diese lediglich die Anzahl der Wiederholungen des Algorithmus vorschreibt.

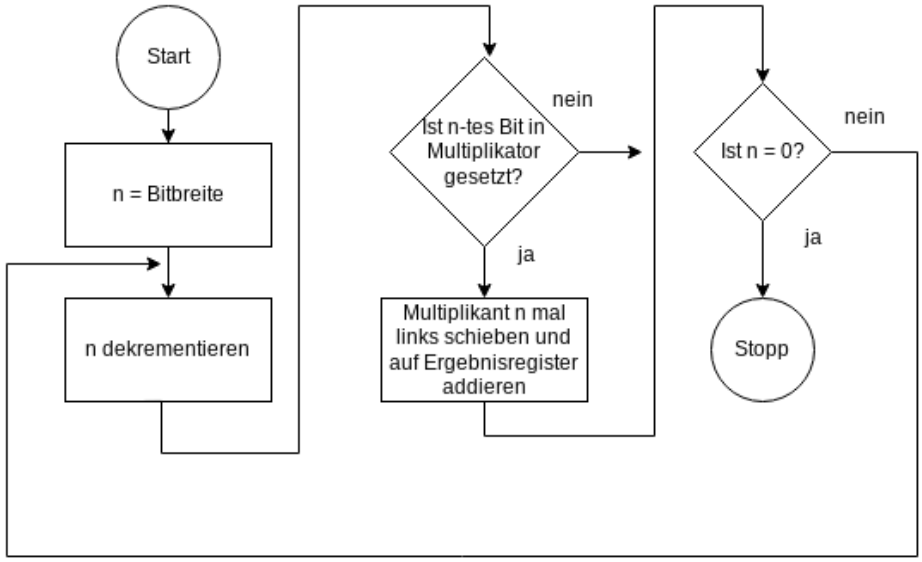



Abbildung 4: Funktionsprinzip der bitweisen Multiplikation

Da die RiSC16-Architektur über keinerlei Schiebebefehle verfügt, wurde hierfür ein Unterprogramm entworfen, welches als Rückgabewert das um eine gewünschte Schrittweite geschobenes Register hat und nach folgendem Prinzip arbeitet:

Im einfachen Binärsystem hat die Stelle $n + 1$ immer genau die doppelte Wertigkeit der Stelle $n$, zum Beispiel $2^2 = 4 = 2 \cdot 2^1$. Wird ein Wort nach links geschoben und rechts mit Nullen aufgefüllt, so verdoppelt sich die Wertigkeit des Wortes mit jedem Schiebevorgang. Dies kommt einer Multiplikation mit Zwei gleich. Bei einer gewünschten Schrittweite $m$ verdoppelt das Unterprogramm das Register $m$-mal.

Zuletzt wurde der bitweisen Multiplikation noch ein Sortieralgorithmus vorgeschaltet, welcher automatisch die richtige Registerkonfiguration für vorzeichenbehaftete Zahlen vornimmt. Der Algorithmus untersucht die Vorzeichen (MSBs) von Multiplikator und Multiplikant. Ist genau einer der Beiden negativ, stellt der Algorithmus sicher, dass dieser im Register r1 steht (siehe Anhang A.3).

Ein Grund für die Notwendigkeit dieser Konfiguration ist, dass ein gesetztes MSB keine Wertigkeit in einer negativen Zahl repräsentiert, sondern nur ein Vorzeichen. Die Bits des Multiplikators (r2) werden jedoch zur Entscheidung, ob der Multiplikant (r1) geschoben werden muss, verwendet. Das bedeutet, sie werden alle als Bits mit einer Wertigkeit interpretiert.

Ein Vorzeichenbit führt hier zu falschen Ergebnissen. Mathematisch sind Multiplikator und Multiplikant vertauschbar.

Analog dazu macht der Sortieralgorithmus bei der Multiplikation von zwei vorzeichenbehafteten Zahlen (beide MSBs gesetzt) die Zweierkomplementumwandlung, welche hier zur Darstellung von negativen Zahlen verwendet wird, rückgängig. Es entstehen zwei positive Zahlen (siehe Anhang A.3). Grundlage dafür ist dieselbe, oben erklärte Problematik. Mathematisch betrachtet entsteht hier wieder kein Unterschied.

# 4 Zusammenfassung

Der Befehlssatz der RiSC16-Architektur umfasst nur acht Befehle. Damit sind die Operationen prinzipiell auf Additionen, *NAND*-Logikoperationen und Speicherzugriffe beschänkt. Das direkte Arbeiten mit Speicherzugriffen und die Möglichkeit mittels Logikoperationen einzelne Bits zu manipulieren, ergibt bei der Programmierung viele Freiheitsgrade und erlaubt abstrakte Lösungsansätze. Das zeigt der Algorithmus zur bitweisen Multiplikation besonders deutlich. Der Programmumfang schwankt dabei stark zwischen etwa 30 Zeilen für die Multiplikation via Addition und 150 Zeilen für die bitweise Multiplikation.

Tatsächliche Hardwaretests konnten in dieser Arbeit nicht durchgeführt werden. Daher stehen keinerlei Messergebnisse zu Durchlaufzeiten verschiedener Programme zur Verfügung.

Alles in allem wird in dieser Projektarbeit deutlich, dass auch mit einem minimalen Befehlssatz praktische Problemstellungen durch die Verflechtung verschiedener Algorithmen lösbar sind. Nicht von der Architektur gegebene Funktionen können in Software implementiert und zur Bearbeitung komplexer Problematiken kombiniert werden.

# A Anhang

## A.1 Verwendung der entworfenen Routinen

**Registerverwendung:**

- r1: zur Wertübergabe, nach Unterprogrammaufruf unverändert
- r2: zur Wertübergabe, nach Unterprogrammaufruf unverändert
- r3: Ergebnisregister
- r4: Zählregister innerhalb von Funktionen
- r5: don't-care
- r6: Rücksprungadressen
- r7: Stackpointer

**Funktionsaufruf:** Definiere r5, r6 für Programmaufrufe

1. Adresse von Unterprogramm in r5 laden:
   *movi r5, Unterprogramm*
2. Zur Adresse in r5 springen, Rücksprungadresse in r6 speichern:
   *jalr r6, r5*
3. zurückspringen: (Hier ist die Rücksprungadresse (r5) unwichtig)
   *jalr r5, r6*

**Stack:** Definiere SP als r7

- push:
  - *addi r7, r7, -1* (SP dekrementieren)
  - *sw r1, r7, 0* (r1 pushen)
- pop:
  - *lw r3, r7, 0* (in r3 poppen)
  - *addi r7, r7, 1* (SP inkrementieren, damit SP immer auf letztes Ereignis zeigt)

**shift_l:**

- shiftweite (n) in r1, zu shiftendes Wort (a) in r2
- Ergebnis wird in r3 zurückgegeben

**MULv2:** (MUL via add)

- Multiplikator in r1, Multiplikant in r2
- Ergebnis wird in r3 zurückgegeben

**MULv3:** (bitweise MUL)

- Multiplikant in r1, Multiplikator in r2
- Ergebnis wird in r3 zurückgegeben

## A.2 Multiplikation via Addition: (MULv2)

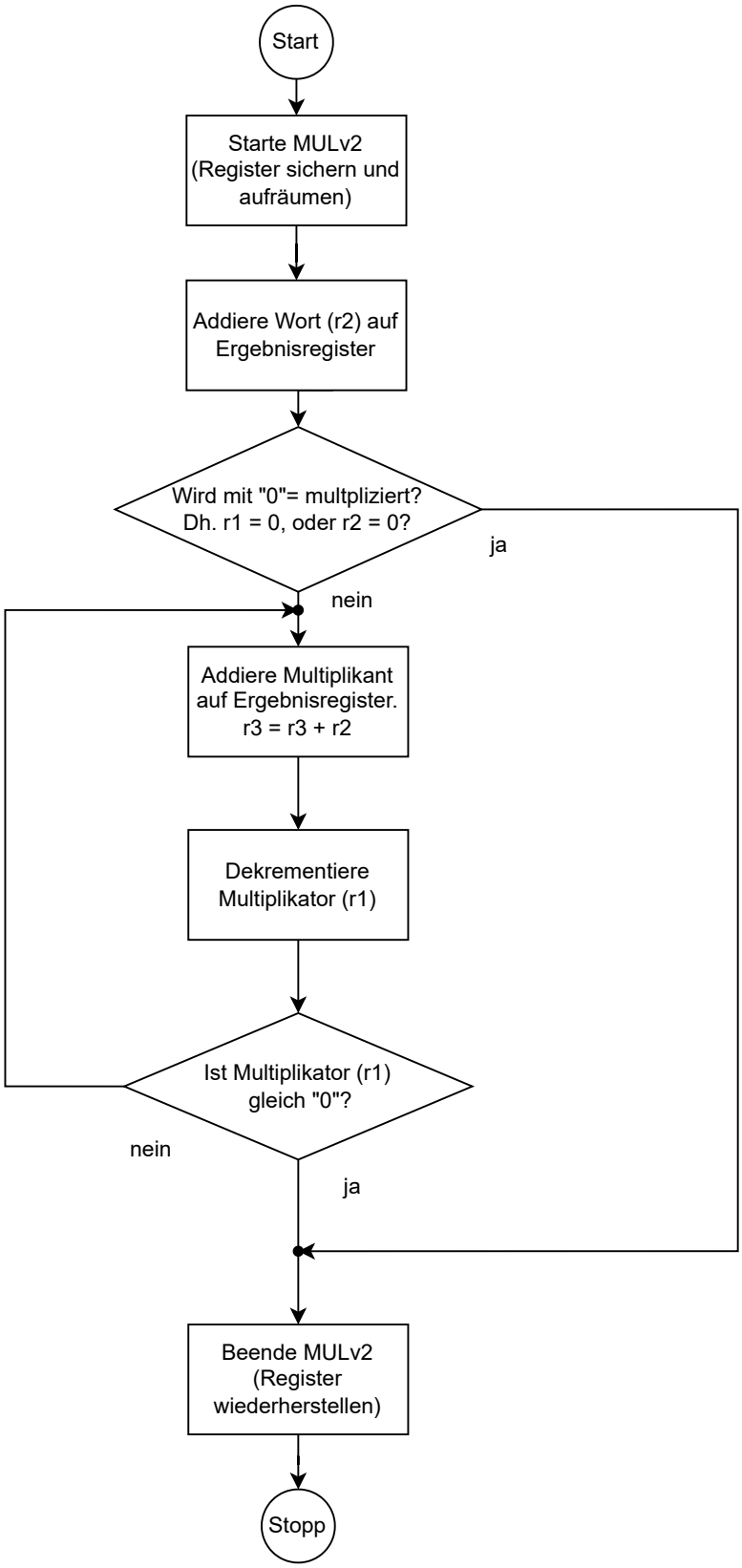
Funktionalität:

Multipliziere zwei vorzeichenlose Zahlen.

Verwendung:

- $r3 = r1 \cdot r2$
- Multiplikator und Multiplikant vor Programmaufruf in r1 und r2 laden.
- Diese stehen auch nach Programmablauf unverändert dort.

Flussdiagramm siehe nächste Seite.


Start
Starte MULv2 (Register sichern und aufräumen)
Addiere Wort (r2) auf Ergebnisregister
Wird mit "0"= multpliziert? Dh. r1 = 0, oder r2 = 0?
ja
nein
Addiere Multiplikant auf Ergebnisregister. r3 = r3 + r2
Dekrementiere Multiplikator (r1)
Ist Multiplikator (r1) gleich "0"?
nein
ja
Beende MULv2 (Register wiederherstellen)
Stopp

### A.3 Bitweise Multiplikation: (MULv3)

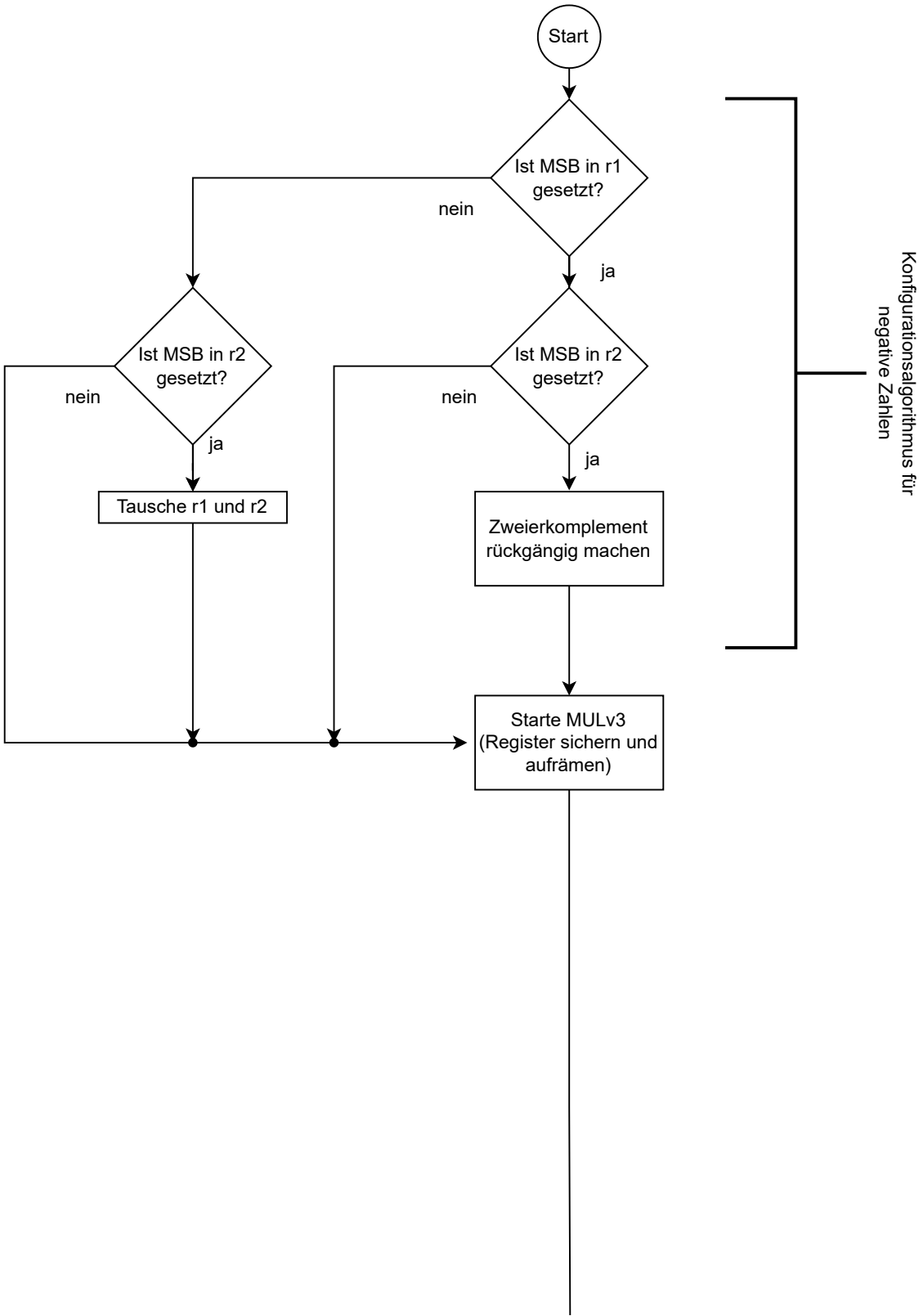
Funktionalität:

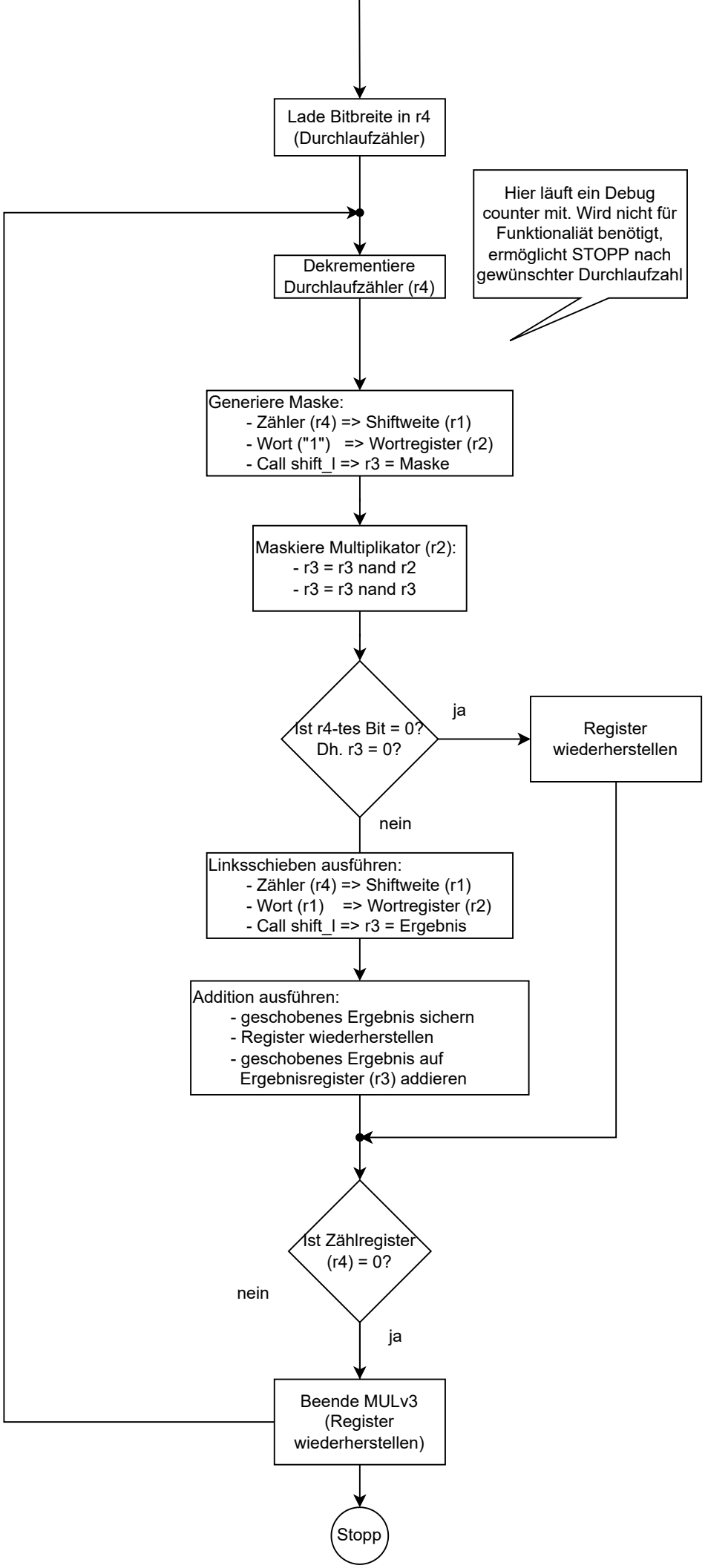
Multipliziere zwei vorzeichenbehaftete Acht-Bit-Zahlen.

Verwendung:

- $r3 = r1 \cdot r2$
- Multiplikant und Multiplikator vor Programmaufruf in r1 und r2 laden.
- Diese stehen auch nach Programmablauf unverändert dort.

Flussdiagramm siehe nächste Seite.


Start
Ist MSB in r1 gesetzt?
nein
ja
Ist MSB in r2 gesetzt?
nein
ja
Tausche r1 und r2
Ist MSB in r2 gesetzt?
nein
ja
Zweierkomplement rückgängig machen
Starte MULv3 (Register sichern und aufrämen)
Konfigurationsalgorithmus für negative Zahlen


Lade Bitbreite in r4
(Durchlaufzähler)
Hier läuft ein Debug counter mit. Wird nicht für Funktionaliät benötigt, ermöglicht STOPP nach gewünschter Durchlaufzahl
Dekrementiere Durchlaufzähler (r4)
Generiere Maske:
- Zähler (r4) => Shiftweite (r1)
- Wort ("1") => Wortregister (r2)
- Call shift_l => r3 = Maske
Maskiere Multiplikator (r2):
- r3 = r3 nand r2
- r3 = r3 nand r3
Ist r4-tes Bit = 0?
Dh. r3 = 0?
ja
Register wiederherstellen
nein
Linksschieben ausführen:
- Zähler (r4) => Shiftweite (r1)
- Wort (r1) => Wortregister (r2)
- Call shift_l => r3 = Ergebnis
Addition ausführen:
- geschobenes Ergebnis sichern
- Register wiederherstellen
- geschobenes Ergebnis auf Ergebnisregister (r3) addieren
Ist Zählregister (r4) = 0?
nein
ja
Beende MULv3 (Register wiederherstellen)
Stopp

## Literatur


[AKT22] HARDAWARE AKTUELL. *MIPS-Architektur*. 2022. URL: https://www.hardware-aktuell.com/lexikon/MIPS-Architektur (besucht am 18. 10. 2022).

[Bon21] Prof. Dr.-Ing. Werner Bonath. *RiSC16-Simulator — Benutzerhinweise*. 2021.

[Bon22] Prof. Dr.-Ing. Werner Bonath. *Projektarbeiten W. Bonath - Assemblerprogrammierung eines RiSC16-Prozessors*. 2022.

[Hol22] Roland Holzer. *Multiplizierwerk*. 2022. URL: http://www.holzers-familie.de/schule/book/Multiplikation.html#:~:text=Multiplizierwerk,es%20wieder%20die%20erste%20Zahl. (besucht am 18. 10. 2022).

[Jac00] Bruce Jacob. *The RiSC-16 Instruction-Set Architecture*. 2000. URL: https://user.eng.umd.edu/~blj/RiSC/RiSC-isa.pdf (besucht am 18. 10. 2022).

[Jac22] Bruce Jacob. *The RiSC-16 Architecture*. 2022. URL: https://user.eng.umd.edu/~blj/RiSC/ (besucht am 19. 10. 2022).