

TECHNISCHE HOCHSCHULE MITTELHESSEN

Dokumentation der Projektarbeit

SYSTEMROUTINEN FÜR EINEN RISC16-PROZESSOR

vorgelegt von:

Arndt Karger

Elektro- und Informationstechnik

Matr.Nr. 5317057

beleitet durch

Prof. Dr.-Ing. Werner Bonath

Gießen, 19. Oktober 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 | 3 |
| 3.3 | Bitweise Multiplikation | 3 |
| 4 | Zusammenfassung | 4 |
| A | Anhang | 5 |
| A.1 | Verwendung der entworfenen Routinen | 5 |
| A.2 | Multiplikation via Addition: (MULv2) | 6 |
| A.3 | Bitweise Multiplikation: (MULv3) | 8 |
| | Literaturverzeichnis | 11 |

1 Einleitung

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

Der Befehlssatz gliedert sich in zwei Befehle zur Addition (addiere Register mit Register, addiere Register mit Ganzzahl), zwei Befehle für den Speicherzugriff (speichern und laden), einen Logikbefehl (*nand*), einen Befehl zum laden der oberen zehn Bits einer Ganzzahl (*lui*), sowie zwei Sprungbefehlen (bedingter Sprung, Sprung zu Adresse) auf. Zusätzlich zu diesen acht Befehlen stehen im Assembler mehrere Makros zur Verfügung, von welchen zwei verwendet wurden. Das eine (*movi*) lädt die Adresse eines Markers in ein Register, das andere (*halt* $\hat{=}$ *jalr r0, r0*) 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].

Vor Beginn der eigentlichen Projektarbeit wurde die Dokumentation von B. Jacob gesichtet, sowie die von ihm zur Verfügung gestellten Testprogramme ausgeführt.

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 hier beispielhaft an zwei möglichen Multiplikationsalgorithmen durchgeführt. Die beiden Algorithmen sind die Multiplikation via Addition und eine bitweise Multiplikation.

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 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 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 Git getrackt.

3 Erstellte Routinen

3.1 Allgemein

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 Stack (hier $7FFF_{hex}$) beschrieben. Dieses Date steht hart im Speicher und wird über den Marker „Stack_adr“ abgerufen (siehe Abbildung 1). Der Stack wurde „Bottom-Up“ implementiert, was bedeutet, dass er zu kleiner werdenden Adressen hin wächst (von unten nach oben). Das Register r7 zeigt immer auf den Top-Of-Stack (TOS), die Grenzen des Stack werden nicht überwacht.

Man erreicht die Funktion *push* durch einmaliges dekrementieren des SP und speichern eines Registers mit dem Befehl *sw* (store word) (siehe Abbildung 1 line 12 folgende). Analog *pop* durch laden eines Dates aus dem Speicher via *lw* (load word) und anschließend inkrementieren des SP. 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 größere Lesbarkeit beim debuggen zu schaffen, werden in der Initialisierung auch noch der Anfang sowie das Ende des Stacks markiert. Dazu wird in das Register r1 das Date $-1_{dez} \hat{=} FFFF_{hex}$ 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 Stack

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 dafür Sorge zu tragen, dass nach dem Rücksprung die Registerinhalte der beiden Übergaberegister unverändert bleiben. Das Unterprogramm gibt einen Rückgabewert im Register `r3` zurück (siehe A.1).

Der eigentliche Programmaufruf erfolgt über das Makro *movi*. Dieses lädt die Adresse eines Markers in ein angegebenes 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 also $X = 3 + 3 + 3$. Diese Art der Multiplikation ist sehr einfach durch eine Schleife zu realisieren und wurde in der Routine *MULv2* angewendet. Hierbei wird der Multiplikant (`r2`) solange auf das Ergebnisregister (`r3`) addiert, bis der Multiplikator zu Null geworden ist.

3.3 Bitweise Multiplikation

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

Betrachten wir beispielhaft die Multiplikation von zwei Acht-Bit-Zahlen ($n = 8$), eine ist der Multiplikant, die andere der Multiplikator. Zuerst untersuchen wir das Most Significant Bit (MSB) des Multiplikators, also das siebte Bit. Ist dieses gesetzt, so schieben wir den Multiplikanten um die Wertigkeit des untersuchten Bits (hier sieben) nach

links und addieren das auf das Ergebnisregister. Ist das MSB nicht gesetzt, so geschieht nichts. [Hol22]

Analog verfahren wir mit allen weiteren Bits. Diesen Prozess müssen wir für alle Bits des Multiplikators durchführen, in unserem Beispiel ergibt das acht Wiederholungen (siehe Abbildung 3). Dieser Prozess ist unabhängig von der Bitbreite.

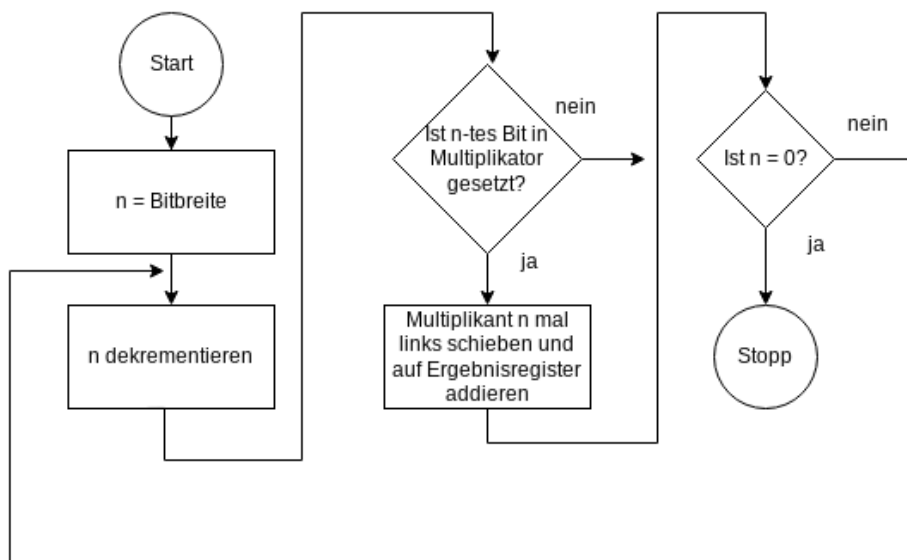


Abbildung 3: Funktionsprinzip der bitweisen Multiplikation

Da die RiSC16-Architektur über keinerlei Schiebebefehle verfügt, wurde hierfür ein kleines 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 $2^2 = 4 = 2 \cdot 2^1$. Schiebt man also ein Wort nach links und füllt rechts mit Nullen auf, so verdoppelt sich die Wertigkeit des Wortes mit jedem Schiebevorgang, kommt also einer Multiplikation mit zwei gleich. Bei einer gewünschten Schrittweite m verdoppelt dieses Unterprogramm das Register also m -mal.

Hier noch Sortieralgo erkl.

4 Zusammenfassung

- Vor- und Nachteile der einzelnen Algorithmen

A Anhang

A.1 Verwendung der entworfenen Routinen

Registerverwendung:

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

Funktionsaufruf: Definiere r5, r6 für Programmaufrufe

1. hole Adresse von UPRO in r1:

movi r5, UPRO

2. springe zu Adresse in r5, speichere Rücksprungadresse in r6:

jalr r6, r5

3. spring wieder zurück: (Hier ist die Rücksprungadresse (r5) unwichtig)

jalr r5, r6

Stack: Definiere SP als r7

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

shift_l:

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

MULv2: (MUL via add)

- Multiplikator in r1, Multiplikant in r2
- Ergebnis wird in r3 zurück gegeben

MULv3: (bitweise MUL)

- Multiplikator in r1, Multiplikant in r2
- Ergebnis wird in r3 zurück gegeben

A.2 Multiplikation via Addition: (MULv2)

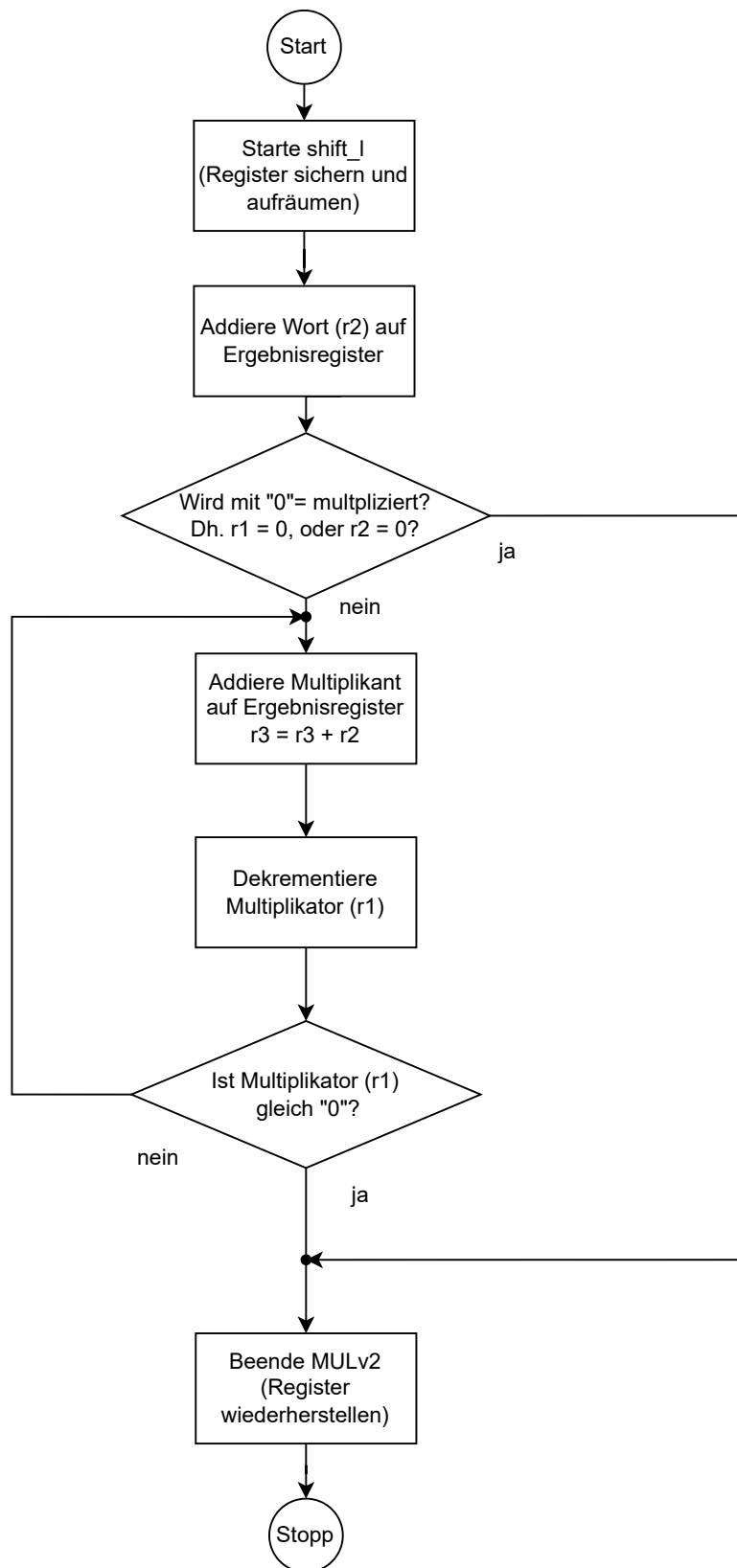
Funktionalität:

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.



A.3 Bitweise Multiplikation: (MULv3)

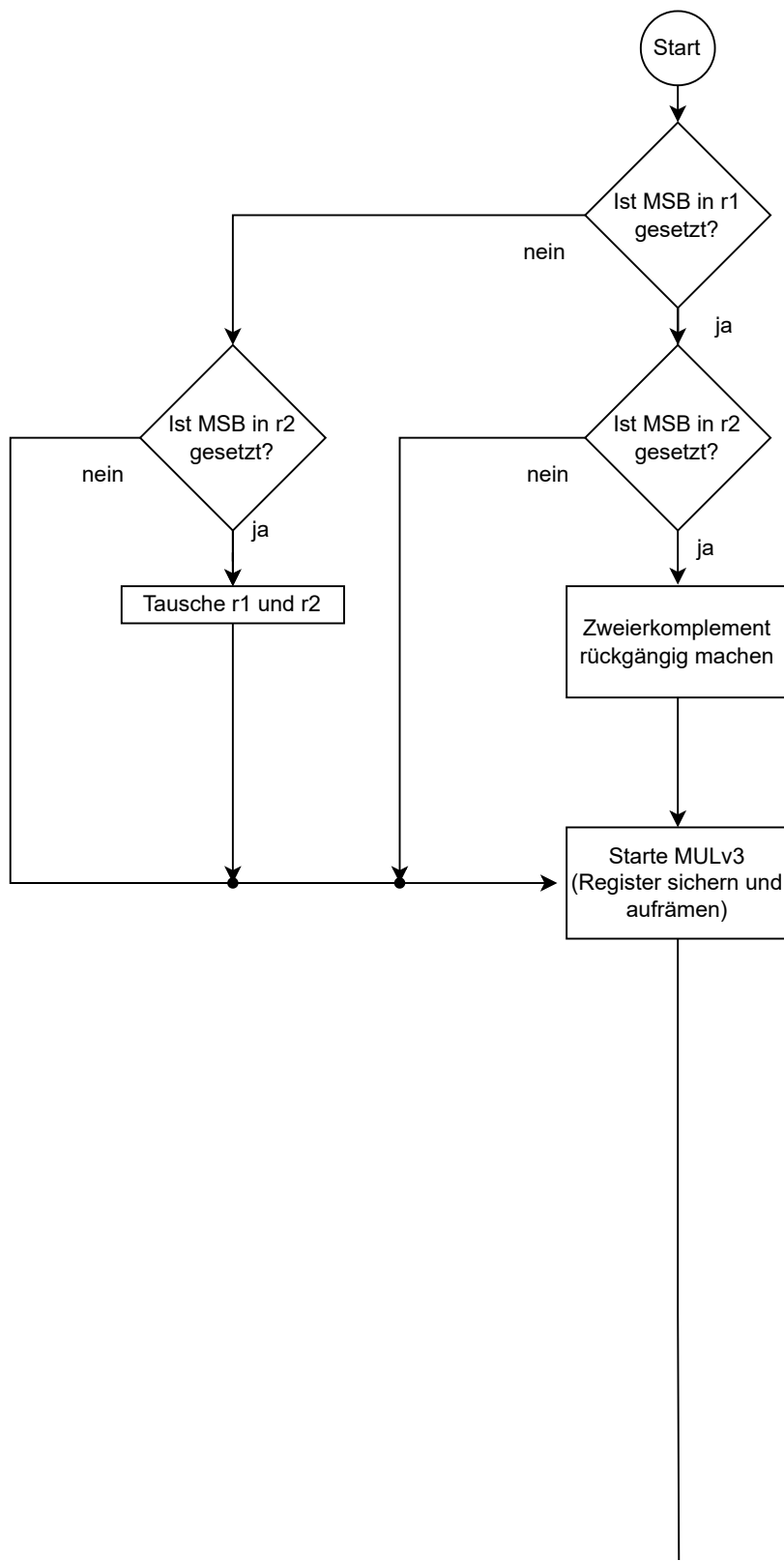
Funktionalität:

Multipliziere zwei vorzeichenbehaftete Acht-Bit-Zahlen.

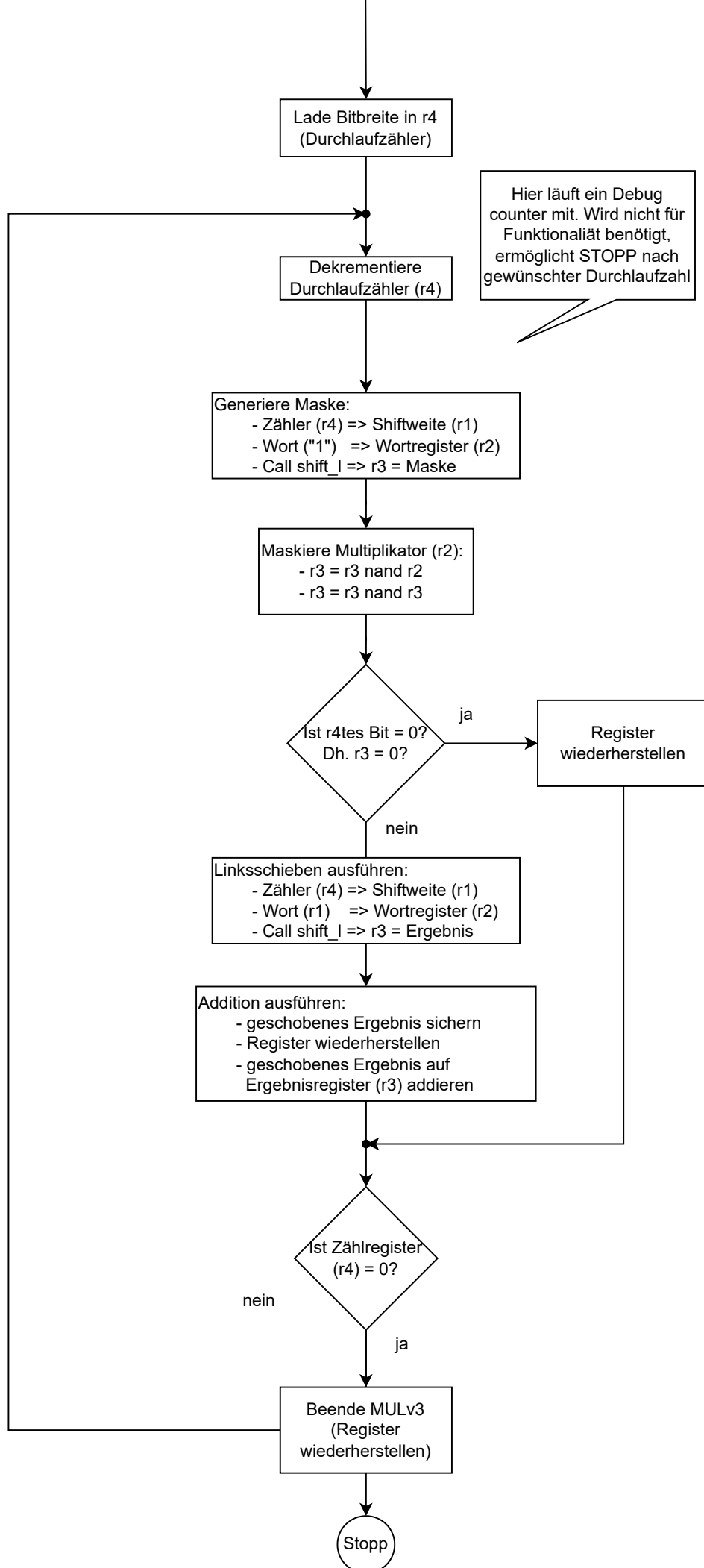
Verwendung:

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

Flussdiagramm siehe nächste Seite.



Konfigurationsalgorithmus für
negative Zahlen



Literatur

- [AKT22] HARDWARE AKTUELL. *MIPS-Architektur*. 2022. URL: <https://www.hardware-aktuell.com/lexikon/MIPS-Architektur> (besucht am 18.10.2022).
- [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).