Das Puck-Backend

26.04.2018

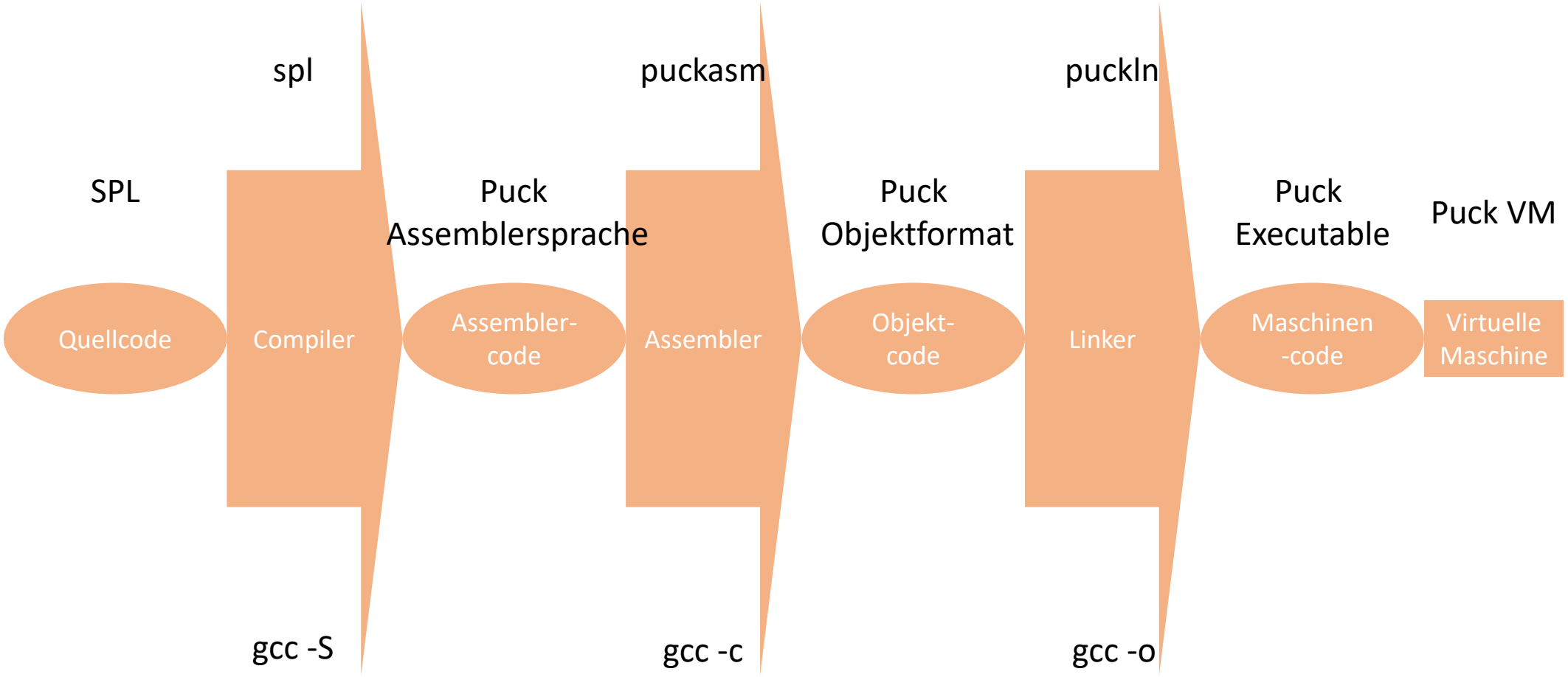
Was ist das Puck Backend?

- Hauptsächliche eine an den Eco32 angelehnte Virtuelle Maschine
  - Für Compilerbau vereinfacht

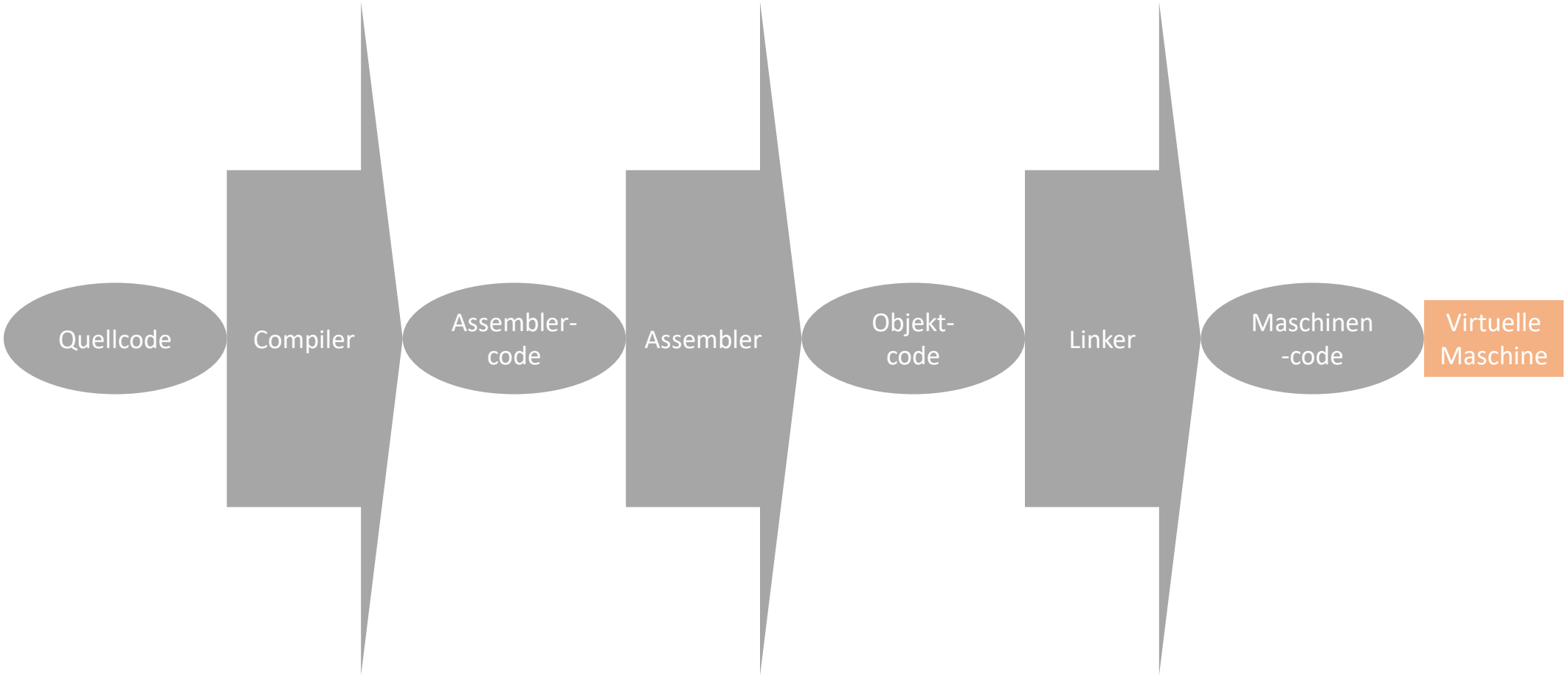
- Im Wintersemester 17/18 bei Prof. Letschert entstanden.

- Dazugehöriger Assembler und Linker

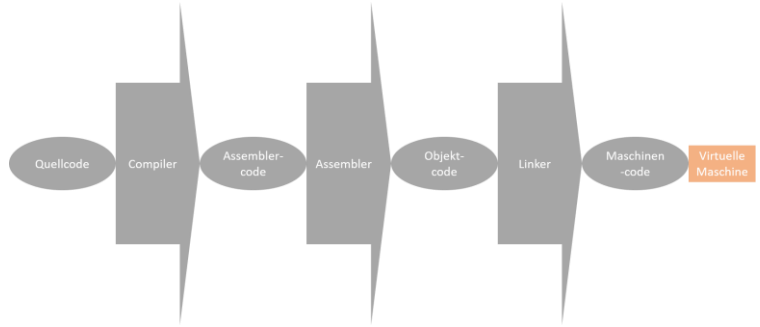
Die Puck-Toolchain



Die Virtuelle Maschine



Die Virtuelle Maschine

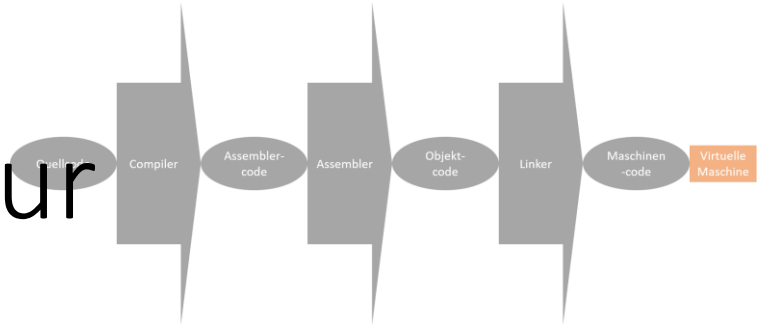


Eine Virtuelle Maschine führt Maschinencode aus

Maschinencode bei Puck

- … besteht aus einer beliebigen Anzahl Instruktionen
- … liegt in einem **nicht menschenlesbaren** Binärformat vor
- ... muss nicht weiter analysiert werden, sondern wird direkt ausgeführt

Die Virtuelle Maschine - Architektur



Die Puck Virtuelle Maschine ist eine **Register-Maschine**

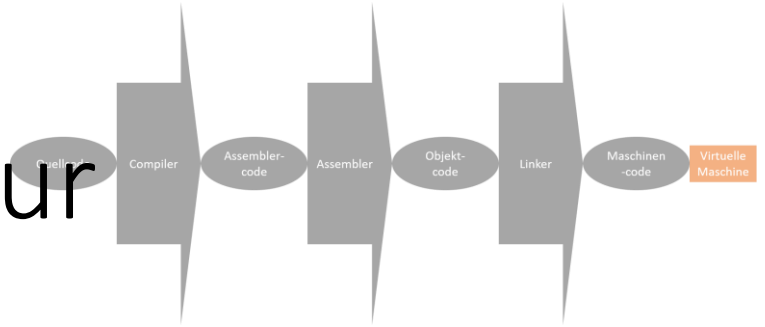
Zum Vergleich: Die Ninja VM aus KSP ist eine **Stack-Maschine**

Rechendaten liegen in Registern, statt auf dem Stack.

Register

- … sind Speicherzellen mit fester Größe
- … sind in realen Rechnern der mit Abstand schnellste Speicherbereich
- … beinhalten Werte, die von Instruktionen genutzt werden

Die Virtuelle Maschine - Architektur



Die Puck Virtuelle Maschine

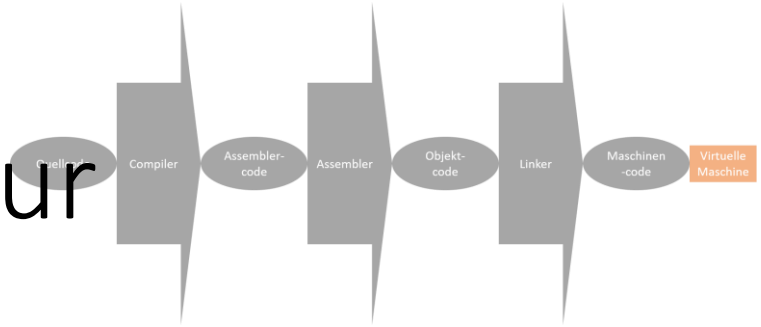
- ist eine 32-bit Maschine, hat also eine Registergröße von 4 Byte
- ist mit 32 solcher Register ausgestattet.
  - Diese sind benannt: $0, $1, $2, …, $30, $31
- ist **big-endian**

Das Register $0 beinhaltet immer den Wert 0!

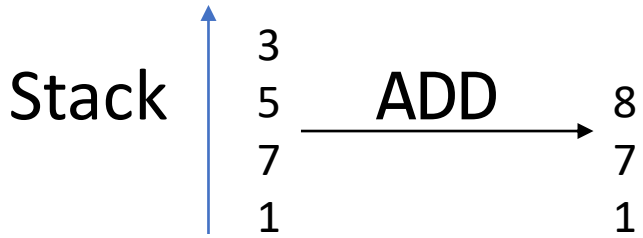
- Der Versuch, dies zu ändern führt zur Beendigung der Programmausführung
- Warum? 0 ist eine häufig benötigte Konstante

Alle anderen stehen theoretisch zur freien Verfügung

Die Virtuelle Maschine - Architektur



Addition Stack-Maschine



Ersetzt die beiden Operanden durch das Ergebnis

Addition Register-Maschine

$1 = 3 | $2 = 5 | $3 = 7 | $4 = 1

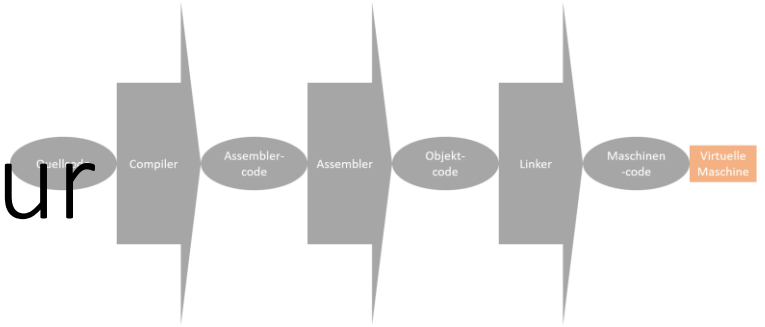
`ADD $1 $2 $3` (Lies: $1 = $2 + $3)

$1 = 12 | $2 = 5 | $3 = 7 | $4 = 1

Werte der Operanden bleiben vorhanden

Überschreibt den Wert des Zielregisters

Die Virtuelle Maschine - Architektur



Eine solche Instruktion kann auch einen seiner Operanden überschreiben: ADD $1 $1 $2 (Lies: $1 = $1 + $2)

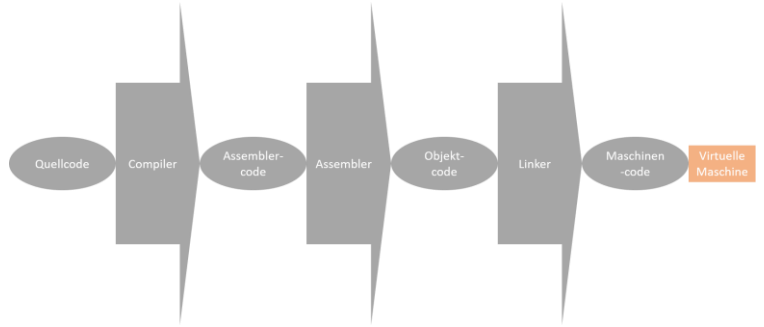
Dies hat den gleichen Effekt, wie der folgende Code:

```
x = x + y;
```

Ebenso ist ADD $1 $1 $1 gleichbedeutend mit:

```
x = x + x;
```

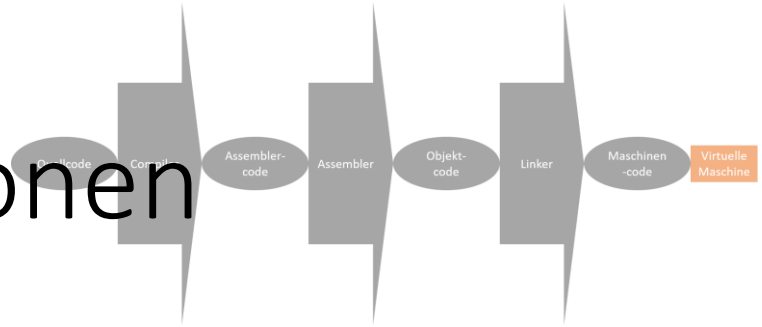
Die Virtuelle Maschine - Speicher



- Die Puck Virtuelle Maschine hat **32 MiB** Haupt-/Arbeitsspeicher

- Dieser beinhaltet sowohl den Code als auch statische Daten sowie den Stack

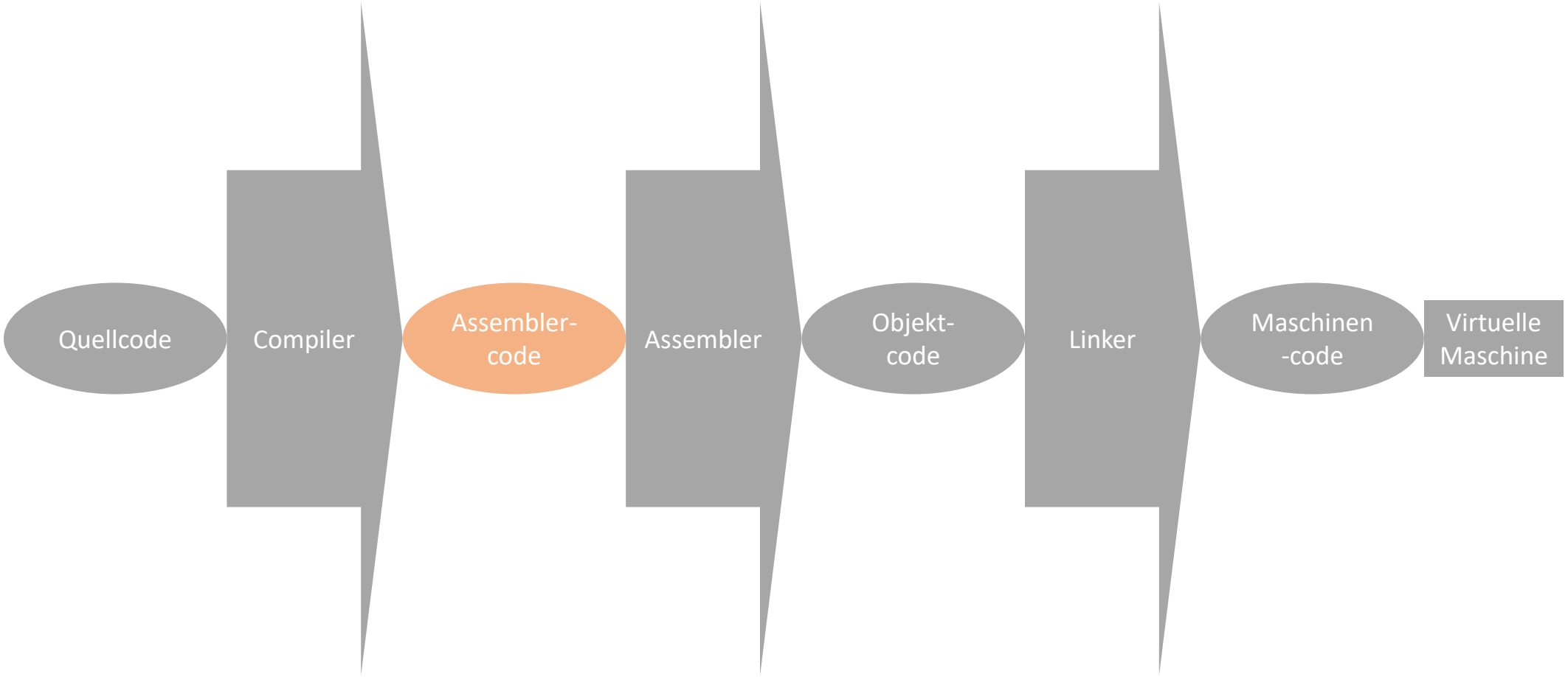
Die Virtuelle Maschine - Instruktionen



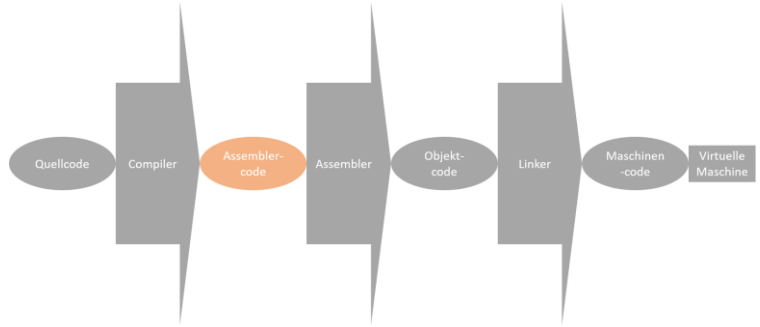
- Die Virtuelle Maschine hat zur Zeit 58 Instruktionen zur Verfügung
- Diese Instruktionen verteilen sich auf:
  - Arithmetische und logische Operatoren (+, - , *, /, %, <, <=, >, >=, ==, !=)
  - Instruktionen zur Flusskontrolle
  - Speicherinstruktionen
  - I/O Instruktionen
  - Instruktionen zum befüllen von Registern mit statischen Daten

- Zur Arbeit mit der Virtuellen Maschine gibt es eine Assemblersprache

Assemblersprache



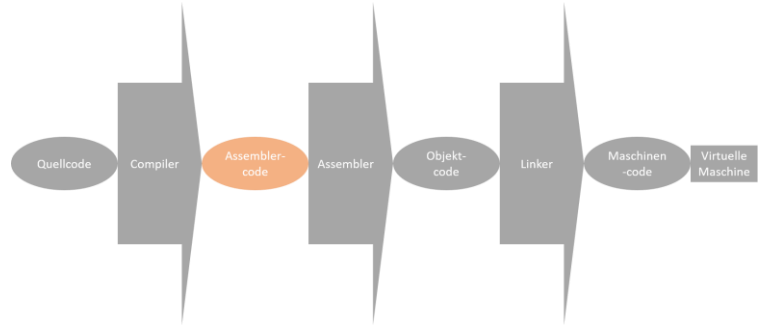
Der Assemblercode



Eine Assemblersprache ist eine für Menschen lesbare Form des Maschinencodes mit kleinen Erweiterungen.

Die Übersetzung in den Maschinencode findet durch den **Assembler** statt.

Der Assemblercode - Syntax




```
INI $1 ;Read operand 1
INI $2 ;Read operand 2
ADD $1 $2 $3
OUTI $1 ;Print result
```

Liest zwei Integer vom Nutzer ein und gibt den Wert der Summe auf der Konsole aus

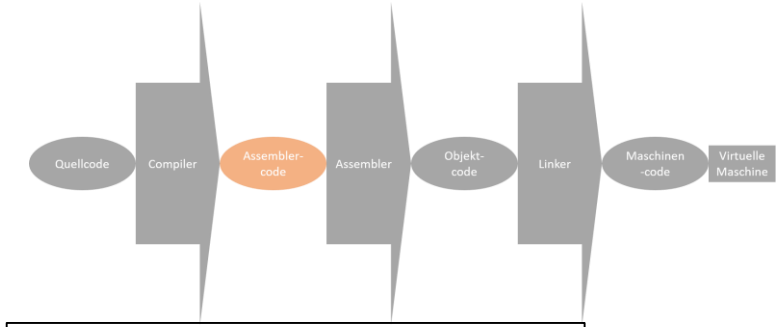
`INI` ⇔ Input Integer

`OUTI` ⇔ Output Integer

An diesem Beispiel sieht man:

- Jede Instruktion bekommt eine eigene Zeile
- Eine Instruktion fordert mit Whitespace getrennte Argumente
- Kommentare werden mit *;* eingeleitet und laufen bis zum Ende der Zeile

Der Assemblercode - Labels




```
INI $1
Loop:
OUTI $1
SUBC $1 $1 1
GEI $2 $1 $0
BRT $2 Loop
```

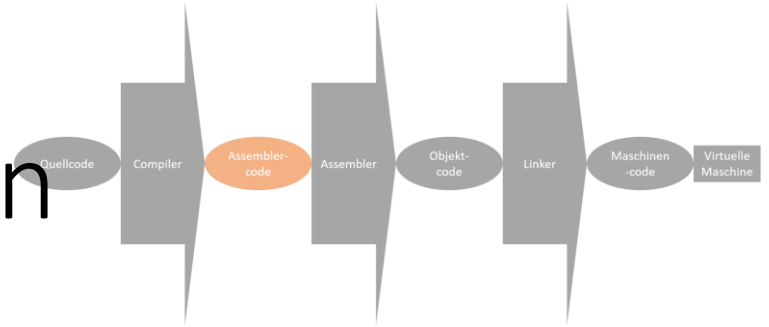
BRT ⇔ Branch if true
GEI ⇔ signed Greater equals

```
x = readInt()
while(x >= 0){
    print(x)
    x = x - 1
}
```

Labels können eingesetzt werden um Adressen im Code einen Namen zu geben.
Diese **symbolischen Adressen** werden vom Assembler und vom Linker in **absolute Adressen** umgewandelt

Labels werden mit einem *Identifier*, gefolgt von einem Doppelpunkt deklariert.
Die Adresse des Symbols bezieht sich auf die **darauffolgende Instruktion**

Der Assemblercode - Instruktionen

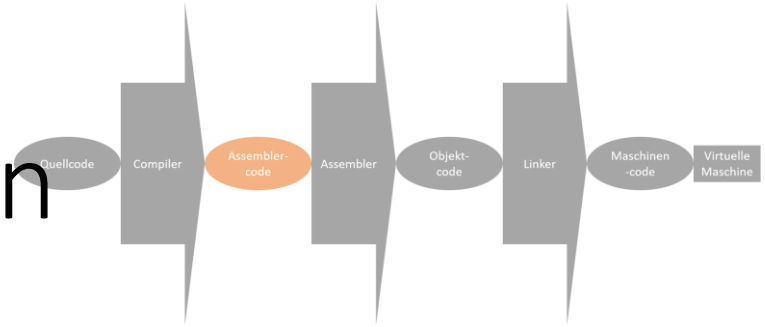


Die meisten Instruktionen sind **Operatoren**.

Operatoren haben drei Argumente: Ein Zielregister, einen linken Operanden, und einen rechten Operanden.

Die Form entspricht immer: ADD `$1 $2 $3` ⇔ `$1 = $2 + $3`

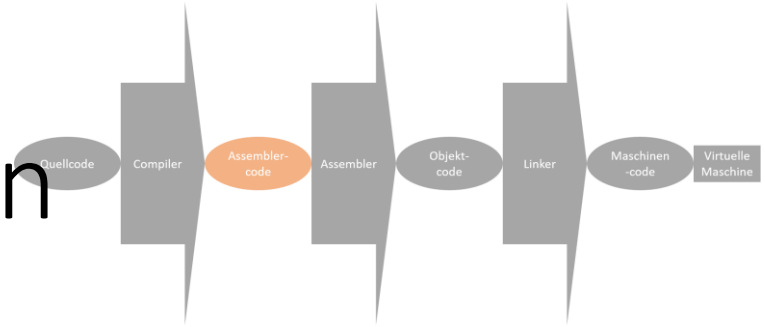
Der Assemblercode - Instruktionen



Integer-Arithmetik-Operatoren (Op-Code 0x00 – 0x07)

ADD	Addition, vorzeichenunabhängig
SUB	Subtraktion, vorzeichenunabhängig
MULU	vorzeichenlose Multiplikation
DIVU	vorzeichenlose Division
MODU	vorzeichenloses Modulo
MULI	vorzeichenbehaftete Multiplikation
DIVI	vorzeichenbehaftete Division
MODI	vorzeichenbehaftetes Modulo

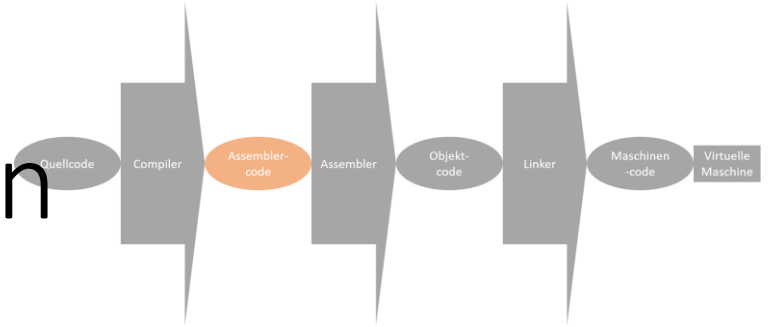
Der Assemblercode - Instruktionen



Bitweise-Operatoren (Op-Code 0x10 – 0x14)

AND	bitweises UND
OR	bitweises ODER
XOR	bitweiser XODER
SL	Linksschieben
SR	Rechtsschieben

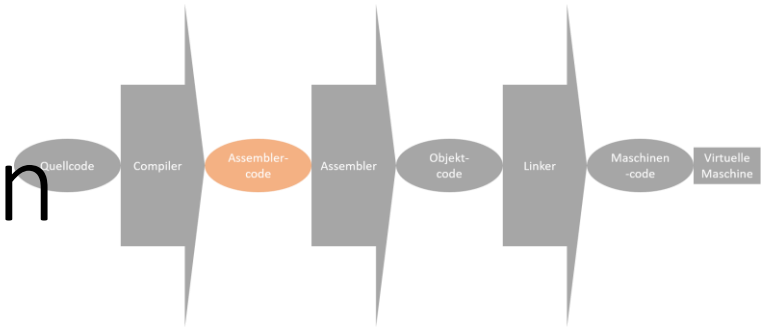
Der Assemblercode - Instruktionen



Vergleichs-Operatoren haben die selbe Argumentstruktur wie arithmetische Operatoren.

Das Ergebnis eines Vergleichs ist 1 wenn die Bedingung erfüllt ist und 0 wenn sie es nicht ist.

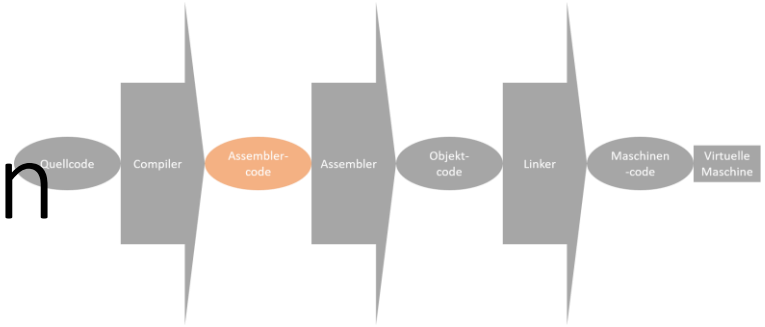
Der Assemblercode - Instruktionen



Integer-Vergleichs-Operatoren (Op-Code 0x20 – 0x29)

EQ	Gleichheit, Vorzeichenunabhängig
NE	Ungleichheit, vorzeichenunabhängig
LTI	Kleiner als, vorzeichenbehaftet
LEI	Kleiner oder gleich, vorzeichenbehaftet
GTI	Größer als, vorzeichenbehaftet
GEI	Größer oder gleich, vorzeichenbehaftet
LTU	Entsprechend für vorzeichenlose Integer
LEU	
GTU	
GEU	

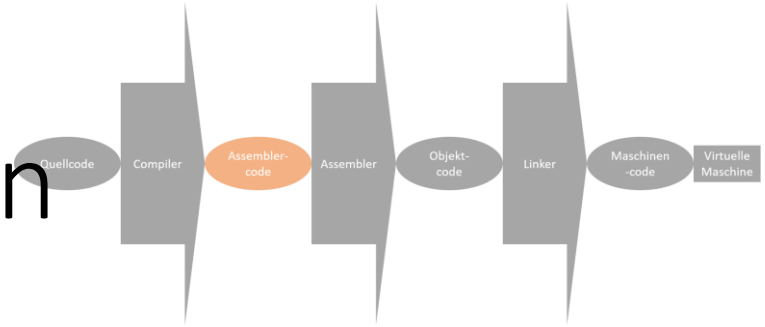
Der Assemblercode - Instruktionen



Instruktionen zur Flusskontrolle steuern die Ausführung des Programms und gehören zu den **wichtigsten Instruktionen**.

JMP	Hat ein Label als Argument, zu dem bedingungslos gesprungen wird
JMPR	Hat ein Register als Argument. Dessen Inhalte werden als Adresse interpretiert, zu der gesprungen wird
BRT	Hat ein Register und ein Label als Argument. Springt zu diesem Label wenn das Register nicht 0 enthält
BRF	Gleiches Layout wie BRT, springt wenn das Register 0 enthält
CALL	Hat ein Register und ein Label als Argument. Springt bedingungslos zu diesem Label und platziert die Rücksprungadresse im angegebenen Register
CALLR	Wie CALL, das Adressargument wird durch ein weiteres Register ersetzt. Nutzt den Inhalt des zweiten Registers als Zieladresse
HALT	Beendet die Ausführung der Maschine.

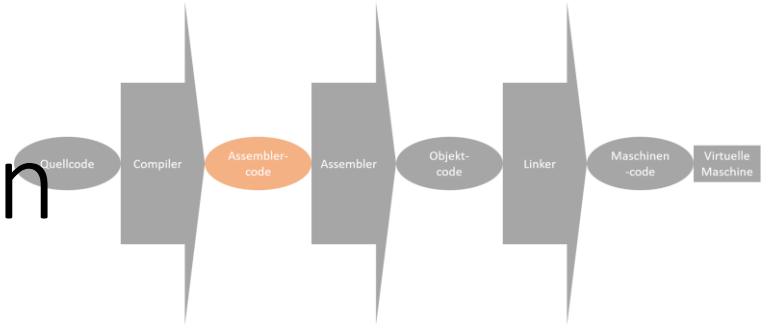
Der Assemblercode - Instruktionen



Beispiele:

```
JMP Loop_begin
JMPR $30
BRF $1 If_end
BRT $1 Do_while_begin
CALL $30 proc_util_gcd
CALLR $30 $5
HALT
```

Der Assemblercode - Instruktionen



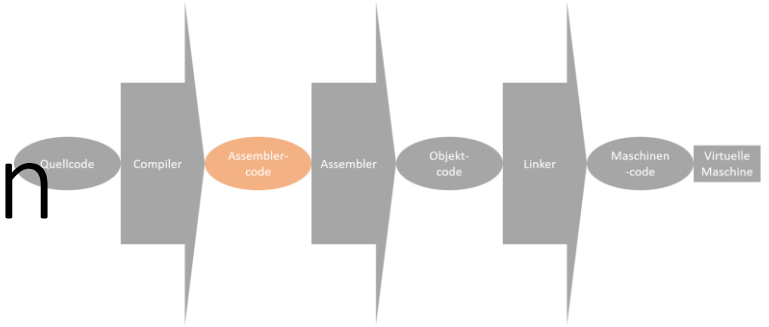
Speicherinstruktionen (Op-Code 0x40 – 0x45) werden genutzt um auf den Hauptspeicher der Maschine zuzugreifen.

`LDW $1 $2 4`

Speicherinstruktionen haben drei Argumente:

- Das erste ist das *Arbeitsregister*, in das der Wert gelesen werden soll/aus dem der Wert in den Speicher geschrieben werden soll.
- Das zweite bezeichnet das *Adressregister*, aus dem die Adresse, von der gelesen wird/an die geschrieben wird, beinhaltet.
- Das dritte Argument ist ein *Offset*, das beim Ausführen der Instruktion auf die Adresse aus dem Adressregister addiert wird (Wertebereich 0 – 255)

Der Assemblercode - Instruktionen

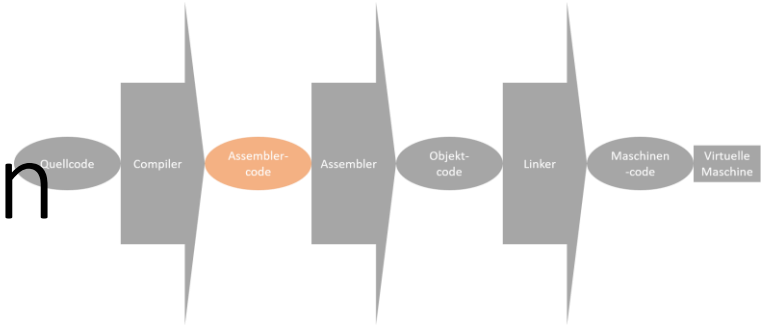



```
LDW $1 $2 4
```

Wenn $2 also den Wert 256 enthält liest diese Instruktion an der Adresse 256 + 4 = 260

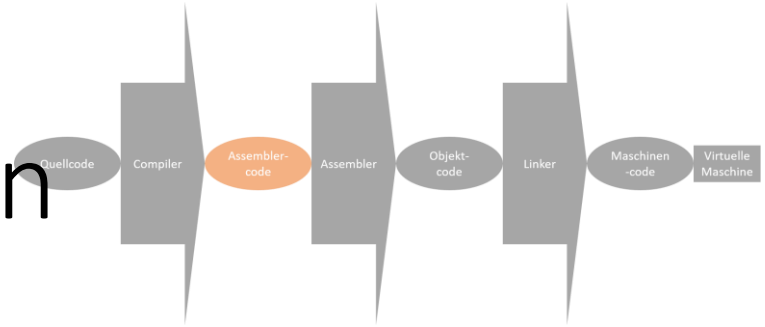
Diese Adresse nennt sich **effektive Adresse**

Der Assemblercode - Instruktionen



LDB	Liest das Byte an der effektiven Adresse in das *least significant byte* des Arbeitsregisters. Der Rest des Arbeitsregisters wird auf 0 gesetzt
LDHW	Liest zwei Bytes (ein *Halfword*) von der effektiven Adresse in die unteren beiden Byte des Arbeitsregisters. Die andere Hälfte des Registers wird auf 0 gesetzt
LDW	Liest vier Byte (ein *Word*) von der effektiven Adresse und überschreibt die Inhalte des Arbeitsregisters damit
STB	Speichert das *least significant byte* des Arbeitsregisters an die effektive Adresse
STHW	Speichert die unteren beiden Byte des Arbeitsregisters an der effektiven Adresse
STW	Speichert den kompletten Inhalt des Arbeitsregisters an der effektiven Adresse

Der Assemblercode - Instruktionen

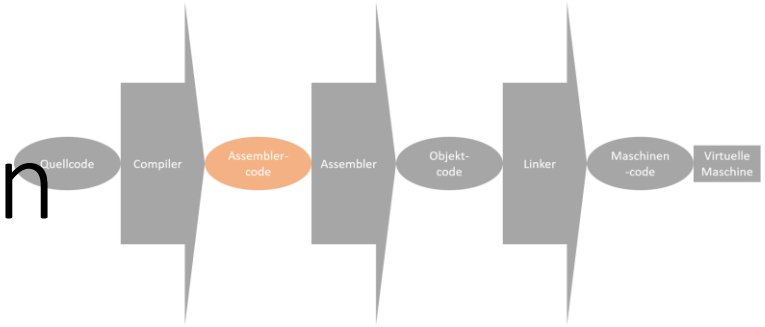


I/O-Instruktionen lesen/schreiben formatierte Daten von Standard Input/auf Standard Output

Sie haben ein Argument: Das Arbeitsregister, in das geschrieben/aus dem gelesen wird

`INC`	Liest ein Byte
`OUTC`	Schreibt das *least significant byte* als ASCII-Character formatiert
`INU`	Liest einen vorzeichenlosen Integer
`OUTU`	Schreibt die Inhalte des Arbeitsregisters als vorzeichenlosen Integer
`INI`	Liest einen vorzeichenbehafteten Integer
`OUTI`	Schreibt die Inhalte des Arbeitsregisters als vorzeichenbehafteten Integer

Der Assemblercode - Instruktionen



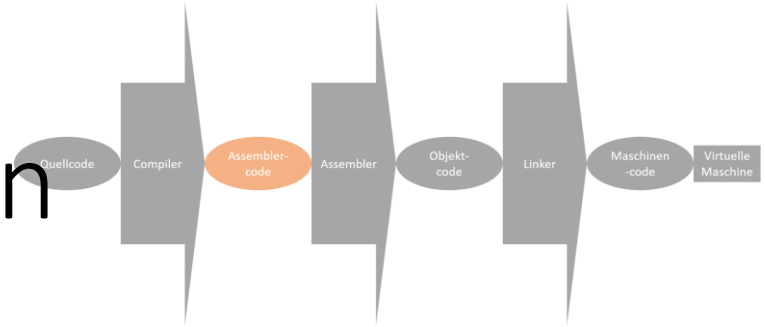
Diese Instruktionen (Op-Code 0x60 – 0x62) werden genutzt um konstante Daten in Register zu laden

Sie haben zwei Argumente:

- Das erste ist das Register, in das geschrieben wird
- Das zweite ist der Wert, der geladen werden soll

`SETB`	Setzt das *least significant byte* des Zielregisters. Der Rest wird auf 0 gesetzt
`SETHW`	Setzt die unteren beiden Byte des Zielregisters. Der Rest wird auf 0 gesetzt
`SETW`	Überschreibt den kompletten Wert des Registers Kann statt eines Zahlenwertes auch eine symbolische Adresse/ein Label als Wert haben Kann alles was SETB und SETHW auch können, ist allerdings größer.

Der Assemblercode - Instruktionen



Die restlichen Instruktionen wurden hinzugefügt, um bestimmte Aufgaben zu vereinfachen

ADDC und SUBC sind Operatoren, deren zweiter Operand kein Register, sondern ein konstanter Wert ist, der das zweite Register ersetzt

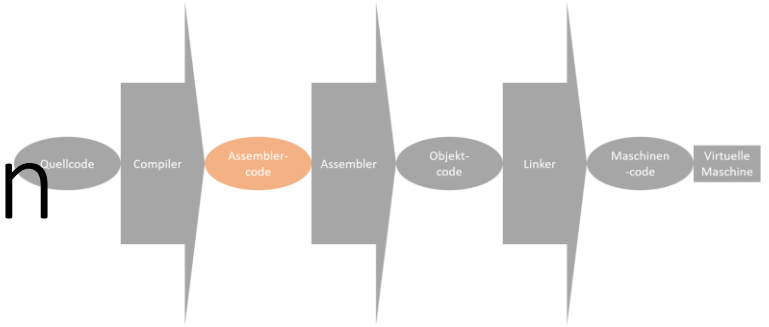
Beispiel: ADDC **$31 $31 32** $31 = $31 + 32 x = x + 32

CP hat zwei Register als Argumente

Die Inhalte des zweiten Registers werden komplett in das erste kopiert

CP $1 $2 $1 = $2 x = y

Der Assemblercode - Instruktionen

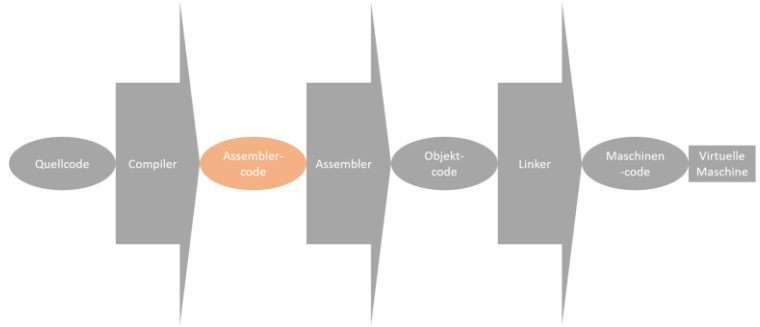


Detaillierte Informationen über den Instruktionssatz und die genaue Syntax erhalten Sie in der offiziellen Dokumentation und im Praktikum

Anmerkung: In der offiziellen Dokumentation sind Instruktionen enthalten, die sich auf Fließkommazahlen beziehen.

Diese werden von der VM zur Zeit **nicht unterstützt**, sie sind für SPL aber auch nicht nötig

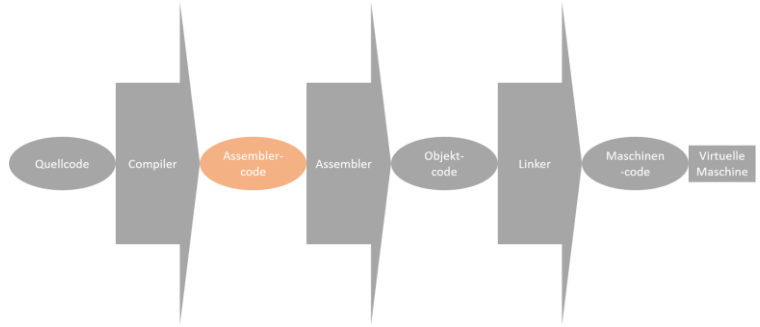
Der Assemblercode - Direktiven



Direktiven

- spezielle Anweisungen an den Assembler, die sich nicht direkt in eine Maschineninstruktion übersetzen lassen
- erhöhen die Nutzbarkeit der Assemblersprache **enorm**

Der Assemblercode - Direktiven



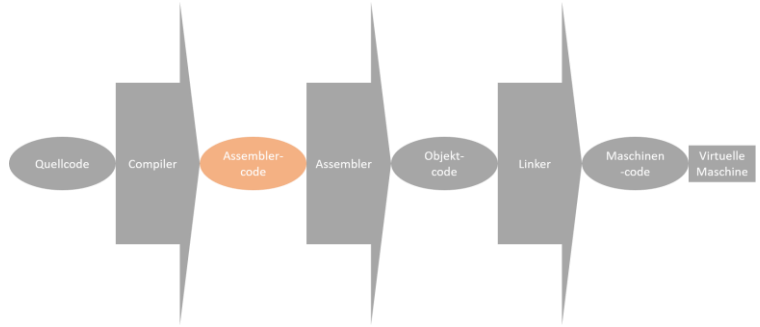
Direktiven werden in einer eigenen Zeile angegeben.

Direktiven werden mit einem Punkt (.) eingeleitet, gefolgt vom *Direktivennamen* und eventuellen Argumenten

Beispiel:

```
.executable main; Declares main to be executable
```

Der Assemblercode - Direktiven

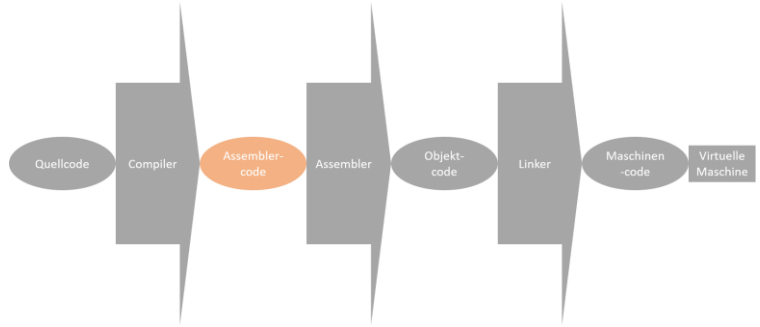


Frage: Wo startet die Ausführung eines Puck Assembler Programms?

Antwort: Beim Assemblieren des Programms ist dies nicht festgelegt!
Dies findet während des Linkings statt.

Doch der Linker muss wissen, welche Stellen in Frage kommen!

Der Assemblercode - Direktiven



Ausführbare Adressen sind Codestellen, die als Einstiegspunkt in das Programm dienen können

Anders als bei den meisten Sprachen muss dieser nicht *main* heißen

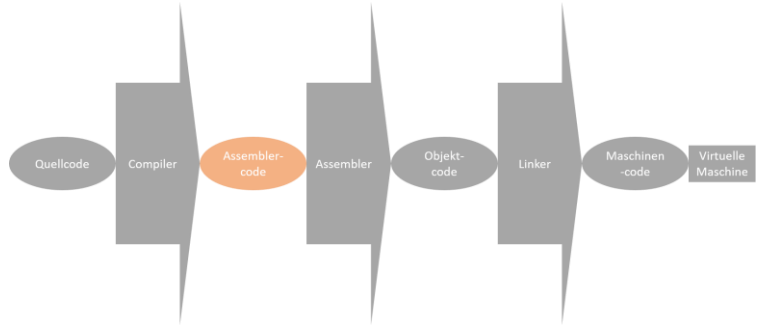
```
.executable possible_entrypoint
```

…deklariert die symbolische Adresse „possible_entrypoint“ als ausführbar.

Importierte Symbole können nicht als ausführbar markiert werden!

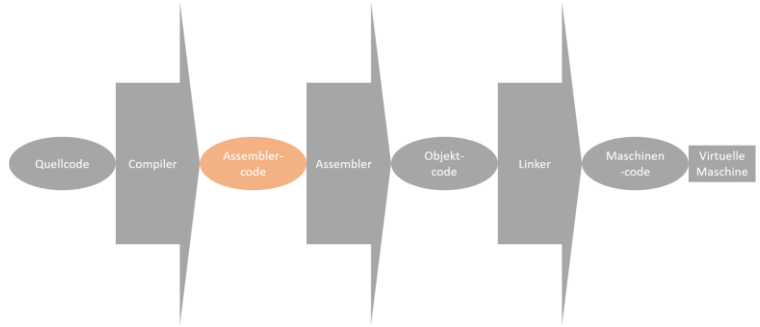
Wichtig: In SPL muss die „main“-Prozedur als ausführbar markiert werden!

Der Assemblercode - Direktiven



Die Realisierung des Einstiegspunkts und der Initialisierungen nimmt der **Linker** vor!

Der Assemblercode - Sonstiges



Die Assemblersprache definiert **Aliase für Register**.

Diese können an jeder Stelle statt eines Registernamens verwendet werden

Definiert sind:

NULL ⇔ $0

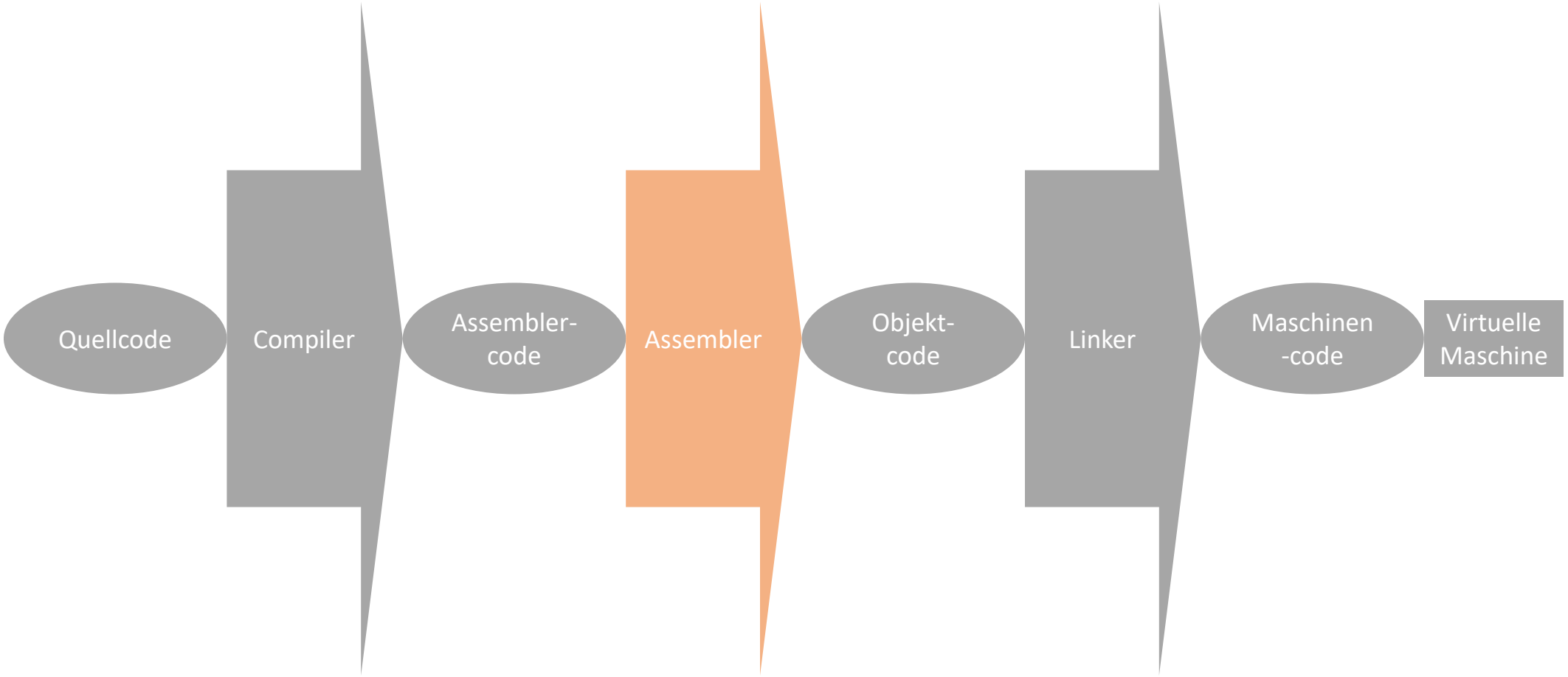
RETURN ⇔ $30

STACK ⇔ $31

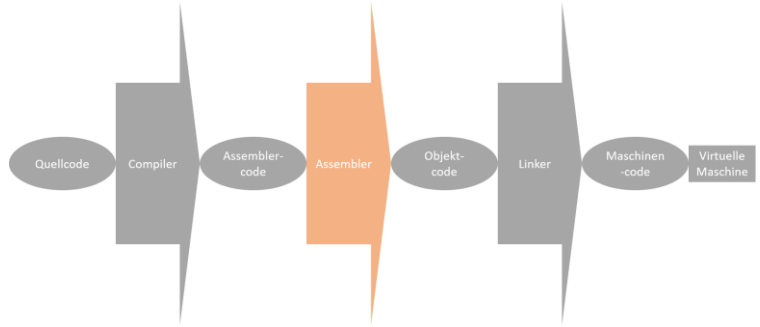
Die letzten beiden kommen durch den Linker zustande, der diese Register auf entsprechende Art und Weise nutzt.

Das NULL-Alias dient nur zur leichteren Lesbarkeit

Assembler


Quellcode
Compiler
Assembler-
code
Assembler
Objekt-
code
Linker
Maschinen
-code
Virtuelle
Maschine

Der Assembler



Der Puck Assembler wird als ausführbare .jar Datei ausgeliefert

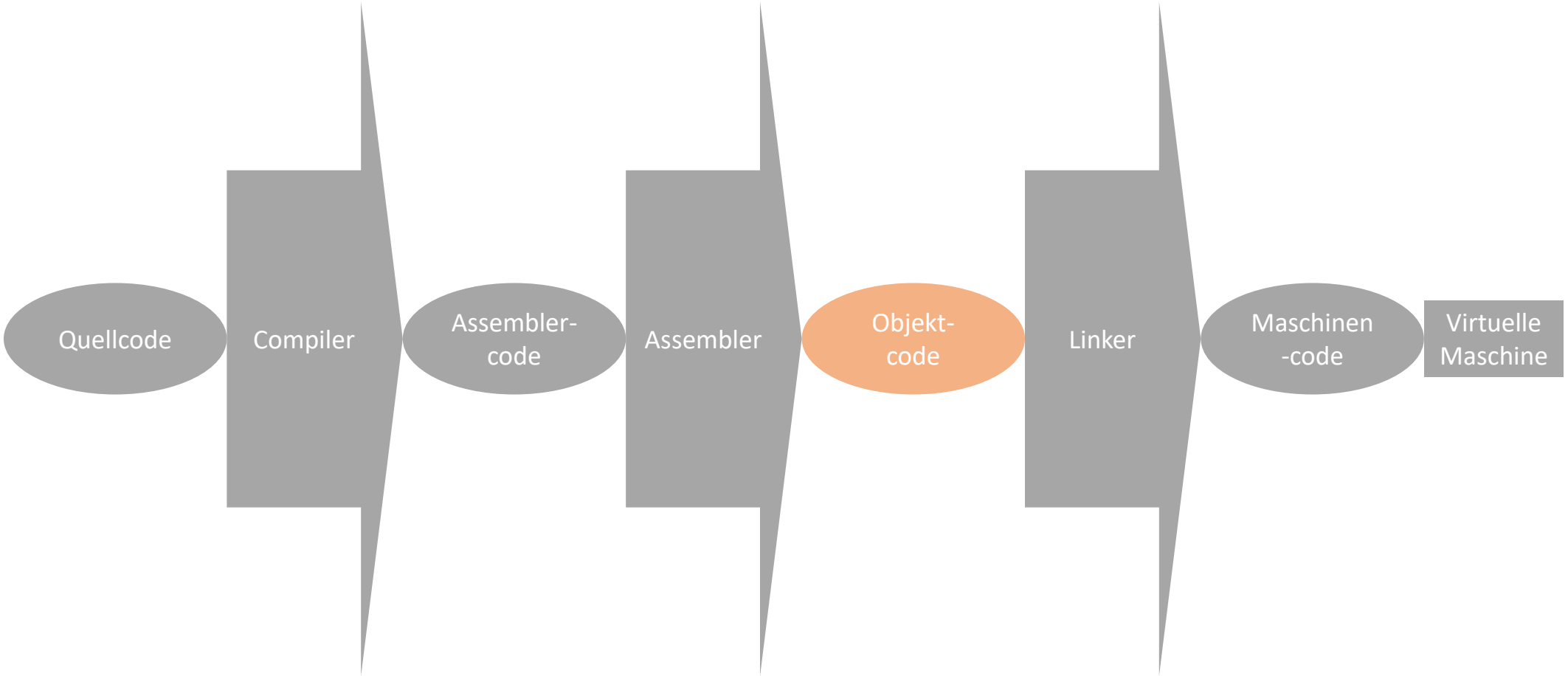
Er erwartet als einziges Argument den Pfad der zu übersetzenden Assemblercode-Datei.

Beispiel: **`java -jar puckasm.jar ./test.a`**

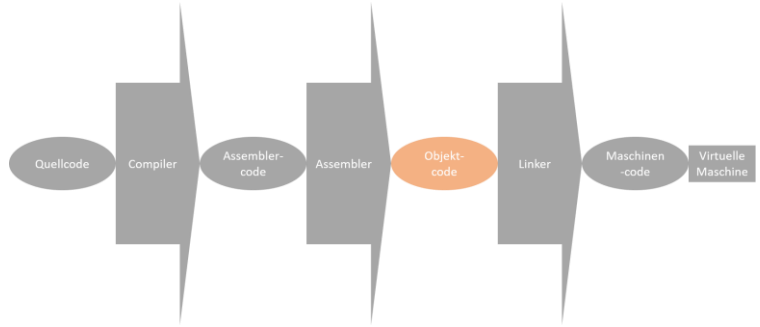
Dies entspricht dem gcc Aufruf **`gcc -c ./test.a`**

Der Assembler übersetzt Assemblercode in Objektcode für den Linker.

Objektcode


Quellcode
Compiler
Assembler-code
Assembler
Objekt-code
Linker
Maschinen-code
Virtuelle Maschine

Objektcode



Objektcode ist Maschinencode mit zusätzlichen Informationen

… über den Objektnamen

… über exportierte Symbole

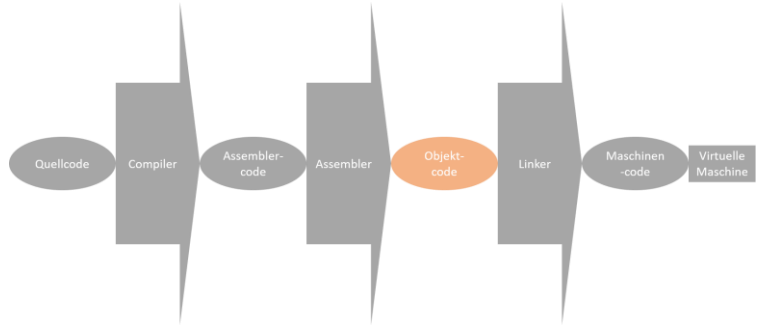
… über importierte Symbole

… über ausführbare Symbole

… über Initialisierungs-Symbole

… über Sprungadressen

Objektcode

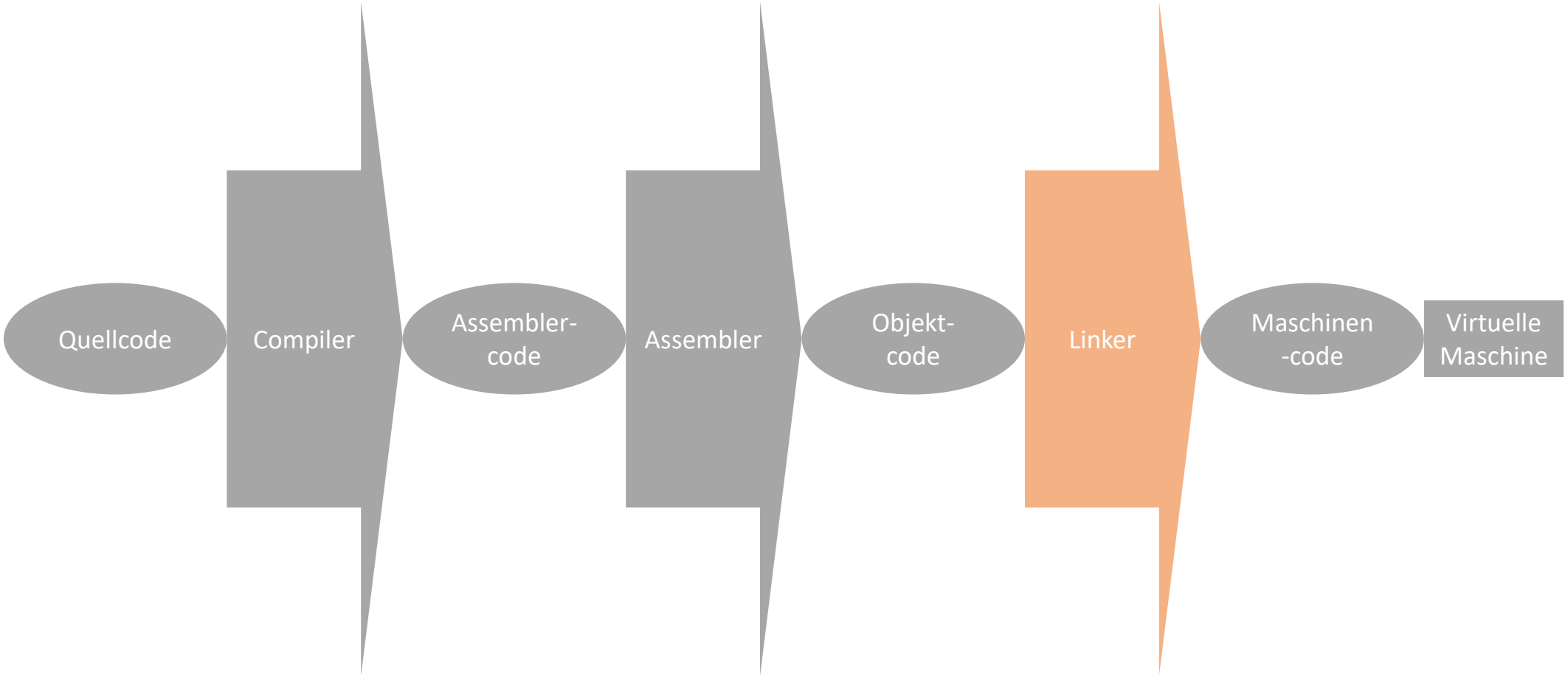


Diese Informationen liegen vollständig im Header der Objektdatei

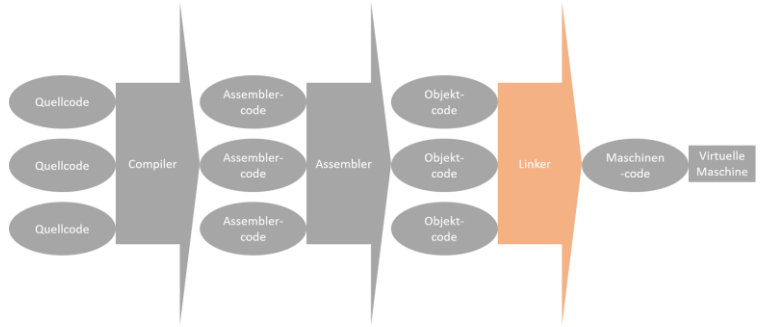
Für Interessierte: Das genaue Format des Objektcodes ist in der Dokumentation einzusehen

Der Linker nutzt diese Informationen um aus einer oder mehreren Objektdateien ein ausführbares Programm zu erzeugen

Linker


Quellcode
Compiler
Assembler-
code
Assembler
Objekt-
code
Linker
Maschinen
-code
Virtuelle
Maschine

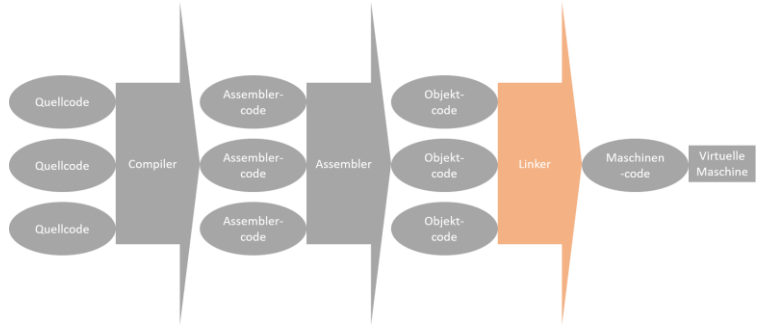
Linker



Der Linker erfüllt mehrere wichtige Aufgaben:

- Er bindet Objektcode-Dateien zu einem Executable
- Er löst symbolische Adressen auf und ersetzt sie durch absolute Adressen
- Er generiert *Startup-Code*, der den gewählten Einstiegspunkt aufruft

Linker - Konventionen



Der Linker generiert für die Initialisierung und den Einstiegspunkt Code

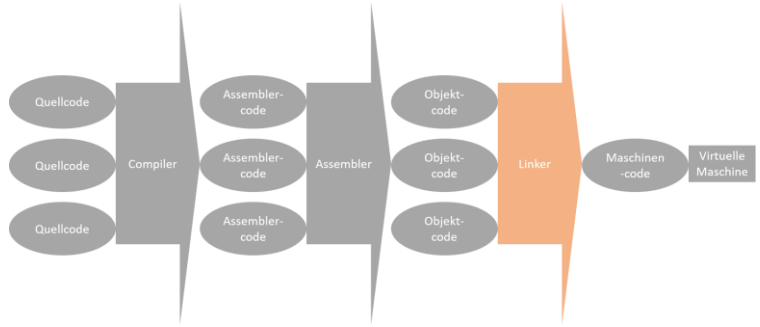
=> muss den Stack definieren und ein Register für Rücksprungadressen festlegen!

Der Linker legt den Stackpointer in `$31` (alias: STACK)

Der Stack wächst **nach unten!**

Der Linker legt alle Rücksprungadressen in `$30` (alias: RETURN)

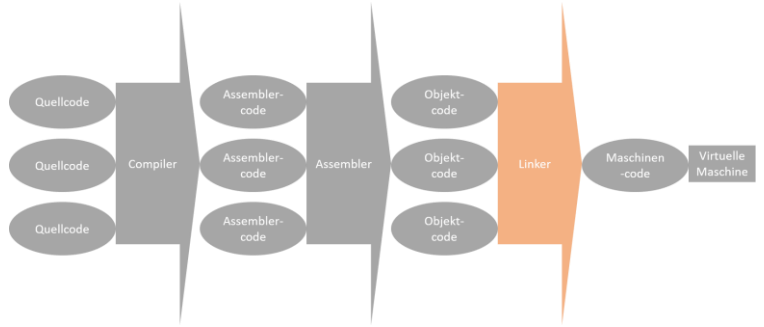
Linker



Erwartet den Namen **einer** Objektdatei als Commandline-Argument

```
java -jar puckln.jar test.o
```

Linker



Es können mehrere Objekte gemeinsam zu einem Executable gebunden werden.

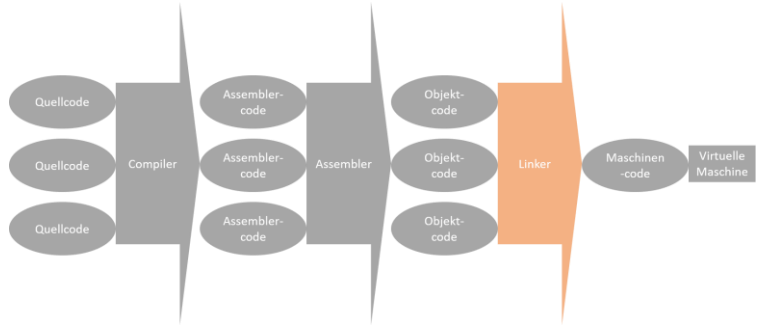
Beispiel: Die SPL Standardlibrary wird als separates Objekt ausgeliefert.

Damit der Linker alle Objekte findet muss ein Objektpfad angegeben werden

```
java -jar puckln.jar test.o -op path/to/objects
```

Dieser Objektpfad beinhaltet test.o sowie alle Abhängigkeiten

Linker

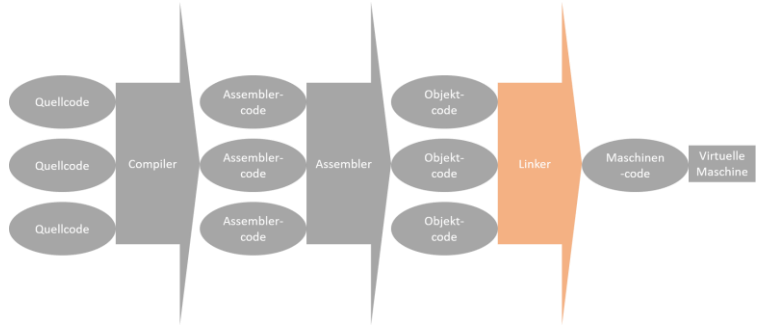


Es kann außerdem ein Ausgabepfad angegeben werden.

Dies geschieht mit der Linker Option –out

```
java –jar puckln.jar
     test.o
     –op path/to/objects
     –out path/to/executables
```

Linker



Eine wichtige Aufgabe des Linkers ist den Programmeinstiegspunkt zu ermitteln. Dieser heißt bei SPL **immer** „main“

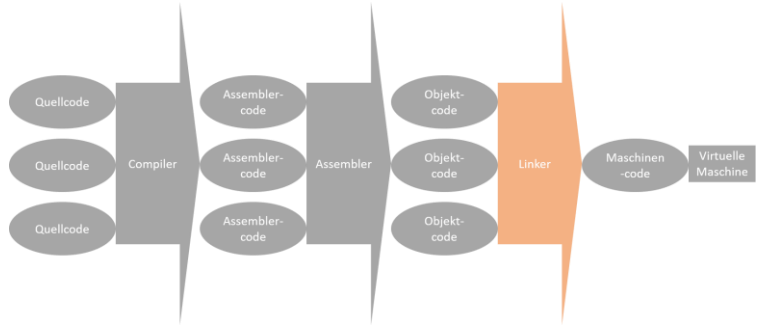
Dafür wird die Commandline-Option –proc verwendet.

```
java –jar puckln.jar
      test.o
      –op path/to/objects
      –out path/to/executables
      –proc main
```

Dies entspricht dem gcc Aufruf

```
gcc test.o –o path/to/executables/test.main.x
```

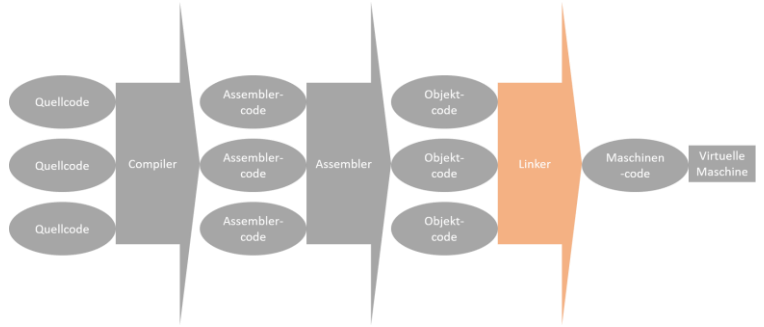
Linker




```
java –jar puckln.jar
     test.o
     –op path/to/objects
     –out path/to/executables
     –proc main
```

- Der Linker verwendet mit diesem Befehl die symbolische Adresse „main“ als Einstiegspunkt.
- Wird keine Einstiegsprozedur festgelegt wird nach einer Prozedur namens „main“ als Standardeinstellung gesucht
- Die Einstiegsadresse **muss** im Assemblercode als executable markiert werden!

Linker

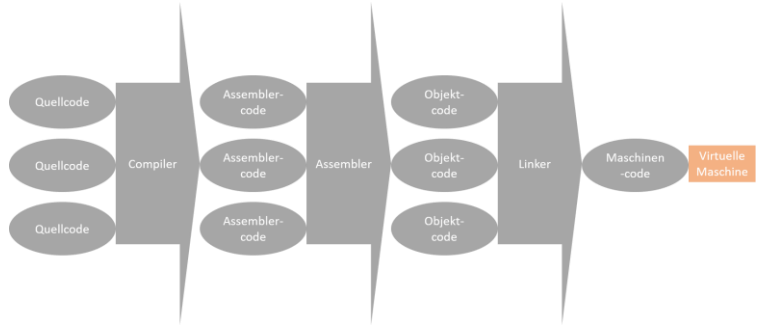


Die Ausgabe des Linkers ist ein Executable mit dem Namen ***object.procedure*.x**

Im Beispiel von eben: **test.main.x**

Dies ist nun mit der Virtuellen Maschine ausführbar.

Virtuelle Maschine



`Windows:` **`.\puck.exe test.entry.x`**

`Mac/Linux:` **`./puck test.entry.x`**

Der Debugger kann mit der Option --debug aktiviert werden.

Er ist bisher ziemlich primitiv und erlaubt nur das Schrittweise durchlaufen des Programms.

Im Laufe des Semesters wird er um weitere Funktionalität erweitert.