

Anleitung für die Einrichtung der Software-Toolchain

Im Folgendem werden die Schritte aufgeführt, um die Software-Toolchain zu installieren. Als Betriebssystem wird Ubuntu 18.04 genutzt.

Um Software auf den FPGArduino zu nutzen, braucht es einen passenden Cross-Compiler. Für den RISC-V kommt die RISC-V GNU Compiler Toolchain zum Einsatz.

<https://github.com/riscv/riscv-gnu-toolchain>

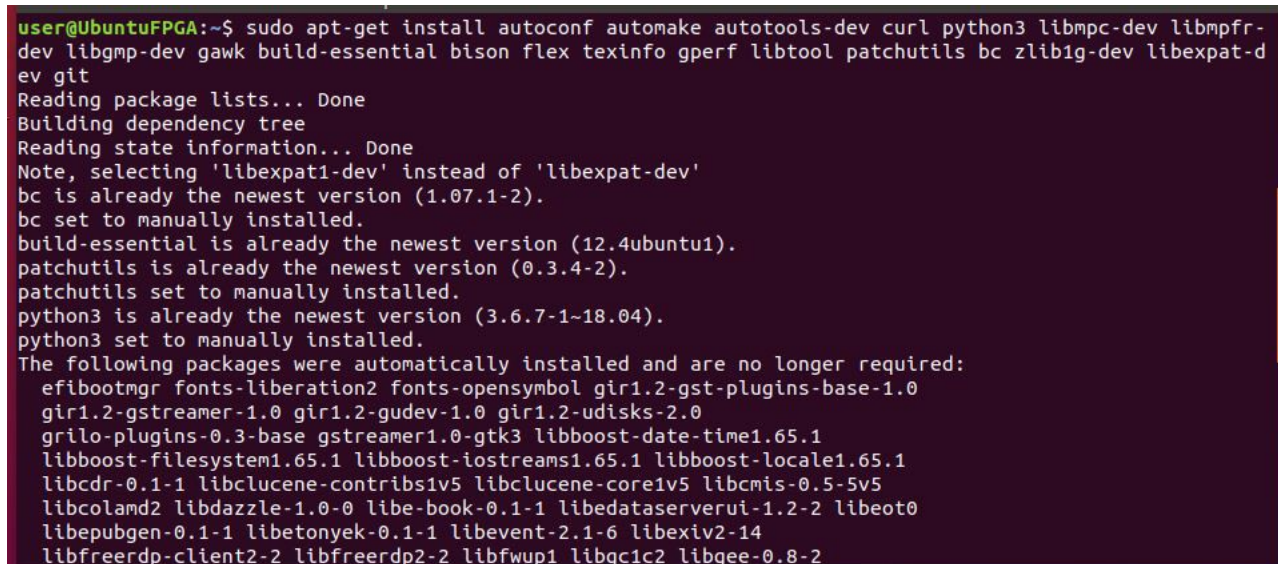
1. Vorbereitung

Für die Software-Toolchain werden einige Pakete vorausgesetzt, die wir installieren müssen. Im Terminal lesen wir die Paketliste neu ein und aktualisieren sie:

```
sudo apt update
```

Anschließend installieren wir alle benötigten Pakete und bestätigen mit *Ja* bzw. *Yes*.

```
sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev  
libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev  
libexpat-dev git
```

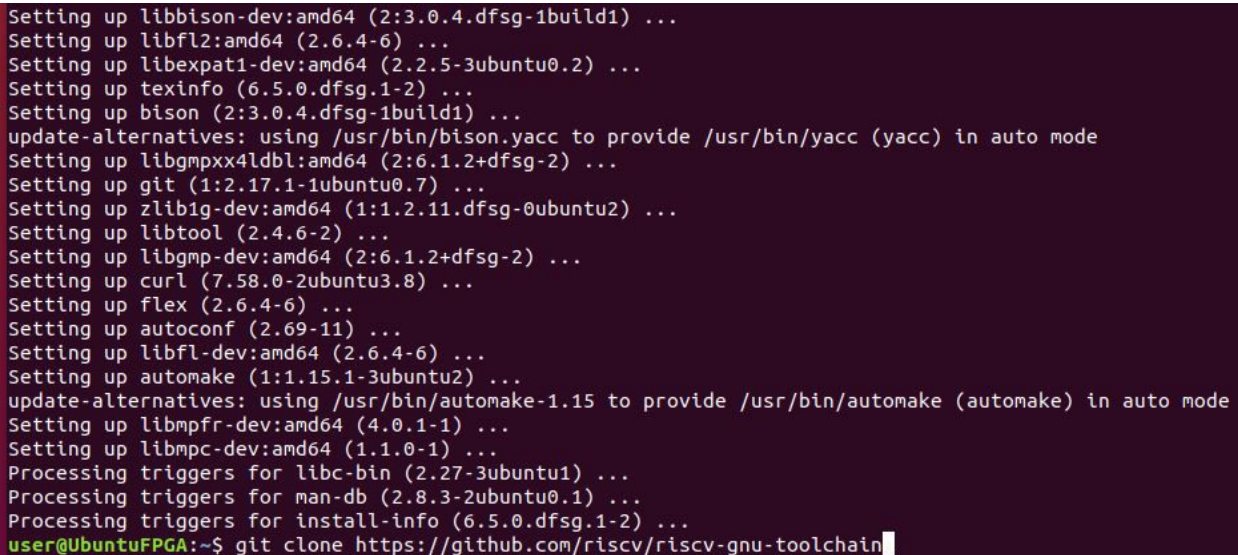


```
user@UbuntuFPGA:~$ sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-  
dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-d  
ev git  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
Note, selecting 'libexpat1-dev' instead of 'libexpat-dev'  
bc is already the newest version (1.07.1-2).  
bc set to manually installed.  
build-essential is already the newest version (12.4ubuntu1).  
patchutils is already the newest version (0.3.4-2).  
patchutils set to manually installed.  
python3 is already the newest version (3.6.7-1~18.04).  
python3 set to manually installed.  
The following packages were automatically installed and are no longer required:  
efibootmgr fonts-liberation2 fonts-opensymbol gir1.2-gst-plugins-base-1.0  
gir1.2-gstreamer-1.0 gir1.2-gudev-1.0 gir1.2-udisks-2.0  
grilo-plugins-0.3-base gstreamer1.0-gtk3 libboost-date-time1.65.1  
libboost-filesystem1.65.1 libboost-iostreams1.65.1 libboost-locale1.65.1  
libcdr-0.1-1 libclucene-contribs1v5 libclucene-core1v5 libcmis-0.5-5v5  
libcolamd2 libdazzle-1.0-0 libe-book-0.1-1 libdataserverui-1.2-2 libeot0  
libepubgen-0.1-1 libetonyek-0.1-1 libevent-2.1-6 libxiv2-14  
libfreerdp-client2-2 libfreerdp2-2 libfwup1 libgc1c2 libgee-0.8-2
```

Abbildung 1: Pakete installieren

Ist die Installation der Pakete abgeschlossen, kopieren wir das RISC-V GNU Compiler Toolchain Respository:

```
git clone https://github.com/riscv/riscv-gnu-toolchain
```

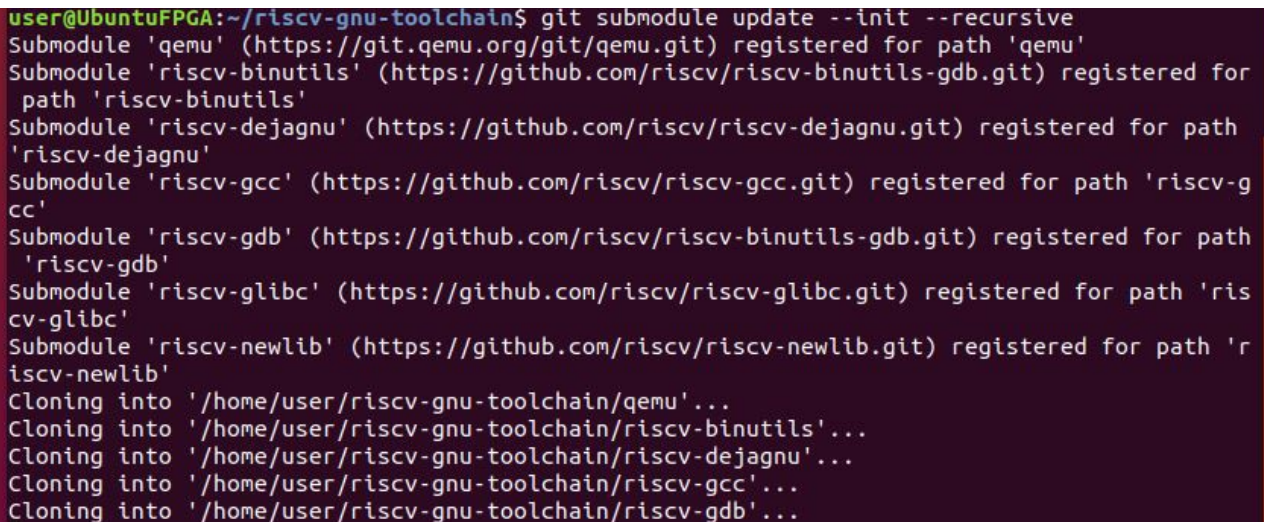


```
Setting up libbison-dev:amd64 (2:3.0.4.dfsg-1build1) ...
Setting up libfl2:amd64 (2.6.4-6) ...
Setting up libexpat1-dev:amd64 (2.2.5-3ubuntu0.2) ...
Setting up texinfo (6.5.0.dfsg.1-2) ...
Setting up bison (2:3.0.4.dfsg-1build1) ...
update-alternatives: using /usr/bin/bison.yacc to provide /usr/bin/yacc (yacc) in auto mode
Setting up libgmpxx4ldbl:amd64 (2:6.1.2+dfsg-2) ...
Setting up git (1:2.17.1-1ubuntu0.7) ...
Setting up zlib1g-dev:amd64 (1:1.2.11.dfsg-0ubuntu2) ...
Setting up libtool (2.4.6-2) ...
Setting up libgmp-dev:amd64 (2:6.1.2+dfsg-2) ...
Setting up curl (7.58.0-2ubuntu3.8) ...
Setting up flex (2.6.4-6) ...
Setting up autoconf (2.69-11) ...
Setting up libfl-dev:amd64 (2.6.4-6) ...
Setting up automake (1:1.15.1-3ubuntu2) ...
update-alternatives: using /usr/bin/automake-1.15 to provide /usr/bin/automake (automake) in auto mode
Setting up libmpfr-dev:amd64 (4.0.1-1) ...
Setting up libmpc-dev:amd64 (1.1.0-1) ...
Processing triggers for libc-bin (2.27-3ubuntu1) ...
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...
Processing triggers for install-info (6.5.0.dfsg.1-2) ...
user@UbuntuFPGA:~$ git clone https://github.com/riscv/riscv-gnu-toolchain
```

Abbildung 2: Respository klonen

Wir wechseln in das Verzeichnis und laden die Submodule:

```
cd riscv-gnu-toolchain
git submodule update --init --recursive
```



```
user@UbuntuFPGA:~/riscv-gnu-toolchain$ git submodule update --init --recursive
Submodule 'qemu' (https://git.qemu.org/git/qemu.git) registered for path 'qemu'
Submodule 'riscv-binutils' (https://github.com/riscv/riscv-binutils-gdb.git) registered for path 'riscv-binutils'
Submodule 'riscv-dejagnum' (https://github.com/riscv/riscv-dejagnum.git) registered for path 'riscv-dejagnum'
Submodule 'riscv-gcc' (https://github.com/riscv/riscv-gcc.git) registered for path 'riscv-gcc'
Submodule 'riscv-gdb' (https://github.com/riscv/riscv-binutils-gdb.git) registered for path 'riscv-gdb'
Submodule 'riscv-glibc' (https://github.com/riscv/riscv-glibc.git) registered for path 'riscv-glibc'
Submodule 'riscv-newlib' (https://github.com/riscv/riscv-newlib.git) registered for path 'riscv-newlib'
Cloning into '/home/user/riscv-gnu-toolchain/qemu'...
Cloning into '/home/user/riscv-gnu-toolchain/riscv-binutils'...
Cloning into '/home/user/riscv-gnu-toolchain/riscv-dejagnum'...
Cloning into '/home/user/riscv-gnu-toolchain/riscv-gcc'...
Cloning into '/home/user/riscv-gnu-toolchain/riscv-gdb'...
```

Abbildung 3: Submodule laden

Nachdem alles runtergeladen wurde, können wir die Toolchain installieren.

2. Installation

Die RISC-V GNU Compiler Toolchain gibt uns die Möglichkeit zwischen der *ELF/Newlib* und der *Linux-ELF/glibc* Toolchain zu wählen. Für den FPGArduino wird der Newlib Cross-Compiler benötigt.

Als Installationspfad wählen wir */opt/riscv32*.

Anlegen des Installationsverzeichnis:

```
sudo mkdir /opt/riscv32
sudo chown $USER /opt/riscv32
```

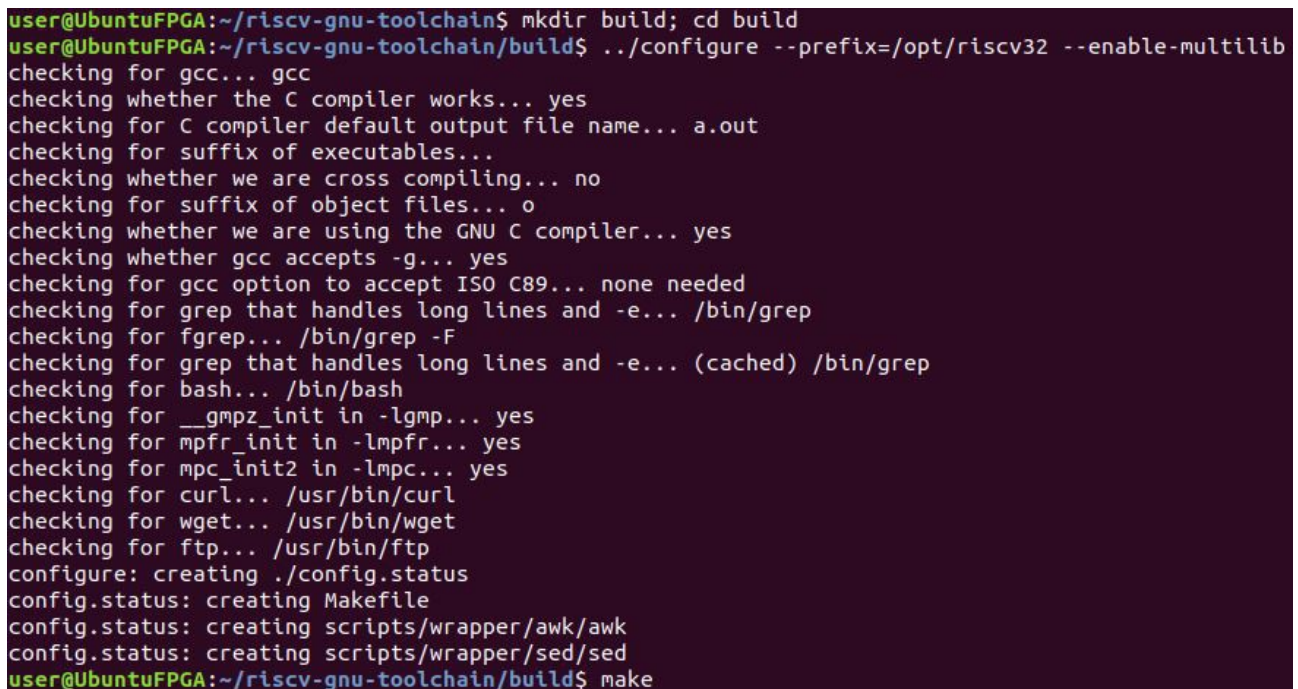


```
Submodule path 'riscv-binutils': checked out 'd7f734bc7e9e5fb6c33b433973b57e1eed3a7e9f'
Submodule path 'riscv-dejagnu': checked out '4ea498a8e1fafeb568530d84db1880066478c86b'
Submodule path 'riscv-gcc': checked out '54945eb8ad5a066da2d4e6a62ffeb513d341eb41'
Submodule path 'riscv-gdb': checked out 'fec47beb8a1f0a6c4a6b0c548cded5711d0c27da'
Submodule path 'riscv-glibc': checked out '06983fe52cfe8e4779035c27e8cc5d2caab31531'
Submodule path 'riscv-newlib': checked out 'f289cef6be67da67b2d97a47d6576fa7e6b4c858'
user@UbuntuFPGA:~/riscv-gnu-toolchain$ sudo mkdir /opt/riscv32
[sudo] password for user:
user@UbuntuFPGA:~/riscv-gnu-toolchain$ sudo chown $USER /opt/riscv32
user@UbuntuFPGA:~/riscv-gnu-toolchain$
```

Abbildung 4: Installationsverzeichnis anlegen

Im *riscv-gnu-toolchain* Verzeichnis legen wir das Verzeichnis *build* an und wechseln in dieses für die Konfiguration der Toolchain. Mit *make* starten wir die Installation.

```
mkdir build; cd build
../configure --prefix=/opt/riscv32 --enable-multilib
make
```



```
user@UbuntuFPGA:~/riscv-gnu-toolchain$ mkdir build; cd build
user@UbuntuFPGA:~/riscv-gnu-toolchain/build$ ../configure --prefix=/opt/riscv32 --enable-multilib
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking for grep that handles long lines and -e... /bin/grep
checking for fgrep... /bin/grep -F
checking for grep that handles long lines and -e... (cached) /bin/grep
checking for bash... /bin/bash
checking for __gmpz_init in -lgmp... yes
checking for mpfr_init in -lmpfr... yes
checking for mpc_init2 in -lmpc... yes
checking for curl... /usr/bin/curl
checking for wget... /usr/bin/wget
checking for ftp... /usr/bin/ftp
configure: creating ./config.status
config.status: creating Makefile
config.status: creating scripts/wrapper/awk/awk
config.status: creating scripts/wrapper/sed/sed
user@UbuntuFPGA:~/riscv-gnu-toolchain/build$ make
```

Abbildung 5: Installation der Toolchain

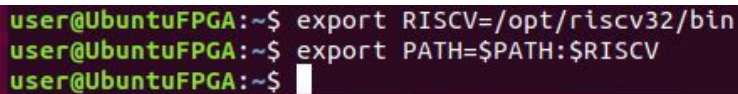
Um nach der Installation die Toolchain nutzen zu können muss `/opt/riscv32/bin` zu `PATH` hinzugefügt werden.

```
export RISCV=/opt/riscv32/bin
```

```
export PATH=$PATH:$RISCV
```

Alternativ: `.profile` öffnen z.B mit `nano ~/.profile` und `RISCV=/opt/riscv32/bin` und

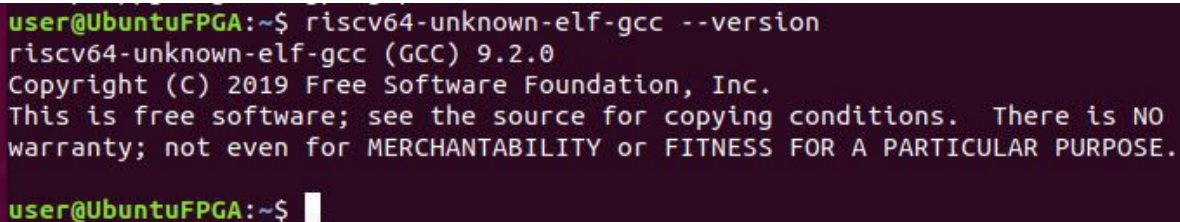
`PATH=$PATH:$RISCV` am Ende hinzufügen



```
user@UbuntuFPGA:~$ export RISCV=/opt/riscv32/bin
user@UbuntuFPGA:~$ export PATH=$PATH:$RISCV
user@UbuntuFPGA:~$
```

Abbildung 6: Pfad zu `PATH` hinzufügen

Nun sollte `riscv64-unknown-elf-gcc` zur Verfügung stehen. Bei erfolgreicher Installation der Software-Toolchain wird mit `riscv64-unknown-elf-gcc --version` folgendes ausgegeben:



```
user@UbuntuFPGA:~$ riscv64-unknown-elf-gcc --version
riscv64-unknown-elf-gcc (GCC) 9.2.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
user@UbuntuFPGA:~$
```

Abbildung 7: `gcc-Version` anzeigen

3. Software-Toolchain nutzen

Um Programme auf den Softcore-Prozessor mit dem FPGArduino laufen zu lassen, wird dem Compiler ein Linkerskript und den startup-Code in Assembler übergeben.

Linkerskript:

Die Aufgabe eines Linker-Skript ist zu beschreiben wie die Segmente im Speicher abgebildet werden und definiert wo der Programmstart `ENTRY(_start)` ist.

`.text` – Code

`.data` – Initialisierte Variablen

`.rodata` _ Read-Only-Variablen

Beispiel: `link.ld` (von `spu32-Softcore`):

```
OUTPUT_ARCH( "riscv" )
ENTRY( _start )
SECTIONS
{
    /* text: test code section */
    . = 0x00000000
    .text : { *(.text) }
    /* data: Initialized data segment */
    .data : { *(.data) }
    /* End of uninitialized data segment */
    _end = .;
}
```

crt0:

Die crt0 enthält die Programminitialisierungsfunktionen eines Programms. Dazu gehören Aufgaben wie initialisieren globaler Variablen, Anlegen des Stacks und den Sprung auf main.

Beispiel: crt0.s (von spu32-Softcore)

```
.section .text

.global _start
_start:
    # reset vector at 0x0
    . = 0x0
    j _init

# interrupt handler
. = 0x10
_interrupt:
    # for now just do an endless loop
    j _interrupt

_init:
    # set up stack pointer
    li sp, 4096

    # call main function
    jal ra, main

    # back to start
    j _start
```

4. C-Programm für den SPU32-Softcore compilieren und einbinden

led.c:

```
#include <stdint.h>
#include <stdbool.h>

#define reg_leds (*(volatile uint8_t*)0xFFFFFFFF0)

void main() {
    uint32_t led = 0;

    while (1) {
        reg_leds = (led >> 16) & 0xFF;
        led = led + 1;
    }
}
```

Generieren der Objektdatei:

```
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -static -nostdlib -fno-builtin-printf -Os -fPIC -c led.c
```

Einbinden von crt0.s und link.ld:

```
riscv64-unknown-elf-gcc -march=rv32i -mno-div -mabi=ilp32 -static -nostdlib -fno-builtin-printf -Os -fPIC -fdata-sections -ffunction-sections -o led.elf crt0.s led.o -Tlink.ld -Xlinker --gc-sections
```

Generieren der bin Datei:

```
riscv64-unknown-elf-objcopy -O binary led.elf led.bin
```

Generieren der dat Datei:

```
hexdump -v -e '1/1 "%02x" "\n"' led.bin > led.dat
```

Anmerkung: -march: Gibt durch Angabe der Architektur vor, welche Befehle und Register zur Verfügung stehen.

-mabi: Gibt mit der ABI die Aufrufkonventionen vor.

Einbinden in den SPU32:

In der *top.v* unter *boards/fpgarduino* wird der Pfad zum Programm angegeben.

```
rom_wb8 #(
    .ROMINITFILE("./software/asm/led.dat")
) rom_inst (
    .CLK_I(clk),
    .STB_I(rom_stb),
    .ADR_I(cpu_adr[8:0]),
    .DAT_I(cpu_dat),
    .DAT_O(rom_dat),
    .ACK_O(rom_ack)
);
```

Abbildung 8: Einbinden der dat Datei in der top.v

Erklärungen zu Argumenten:

- march=rv32im:32, 32-bit general-purpose integer registers + M extension
- mabi=ilp32:int, long, and pointers are all 32-bits long. long long is a 64-bit type, char is 8-bit, and short is 16-bit
- static: Do not link against shared libraries.
- nostdlib, -nostartfiles: Do not use the standard system startup files or libraries when linking.
- fno-builtin-printf: Disables special handling and optimizations of standard C library function printf
- Os: Specifies the level of optimization to use when compiling source files
- fPIC: Generate position-independent code (PIC)
- d: disassemble
- O binary: Write the output file using the object format binary
- x assembler-with-cpp: assembly code contains C directives
- fdata-sections -ffunction-sections: Place each function or data item into its own section in the output file
- Xlinker -gc-sections: Enable garbage collection and remove all unused code

5. Quellen

RISC-V GNU Compiler Toolchain

URL: <https://github.com/riscv/riscv-gnu-toolchain>

The -march, -mabi, and -mtune arguments to RISC-V Compilers URL:

<https://www.sifive.com/blog/all-aboard-part-1-compiler-args>

GCC Command Options URL:

URL: <https://gcc.gnu.org/onlinedocs/gcc/Invoking-GCC.html>

Using LD, the GNU linker - Linker Scripts URL:

https://scgberlin.de/content/media/http/informatik/gcc_docs/ld_3.html

crt0

URL: <https://de.wikipedia.org/wiki/Crt0>

spu32-Softcore crt.s und link.ld URL:

<https://github.com/maikmerten/spu32/blob/master/software/c-firmware/crt0.s>

<https://github.com/maikmerten/spu32/blob/master/software/asm/link.ld>