Technische Hochschule Augsburg

Projektarbeit

Studienrichtung
Technische Informatik

Implementierung einer Debug-Infrastruktur für einen RISC-V-Prozessor in Simulationsumgebung

Verfasser:
Johannes Hofmann
johannes.hofmann1@tha.de

im Fachgebiet Effiziente Eingebettete Systeme

Technische Hochschule
Augsburg
An der Hochschule 1
86161 Augsburg
Telefon: +49 (0)821-5586-0
Fax: +49 (0)821-5586-3222
info@tha.de

Verfasser: Johannes Hofmann

Prüfer: Prof. Dr. Gundolf Kiefer

Inhaltsverzeichnis

Inhaltsverzeichnis III

Abbildungsverzeichnis V

Tabellenverzeichnis VI

Verzeichnis der Listings VII

1 Einleitung 1

1.1 Motivation . . . 1
1.2 Ziel der Arbeit . . . 1
1.3 Aufbau der Arbeit . . . 2

2 Grundlagen 3

2.1 Einführung in das PicoNut-Projekt . . . 3
2.1.1 Projektziele . . . 3
2.1.2 Gesamtsystem . . . 3
2.1.3 Die „Soft-Peripheral“-Schnittstelle . . . 3
2.2 RISC-V External Debug Support . . . 4
2.3 On-Chip-Debugging . . . 5
2.3.1 GDB (GNU Debugger) . . . 5
2.3.2 OpenOCD . . . 5
2.3.3 Remote Bitbang-Protokoll . . . 5

3 Debug-Infrastruktur im ParaNut-Prozessor 6

4 Soft-Debug-Infrastruktur 6

4.1 Anforderungen . . . 7
4.2 Übersicht . . . 7
4.3 Debug Mode . . . 8
4.4 Remote Bitbang Module . . . 10
4.5 Debug Transport Module (DTM) . . . 10
4.6 Debug Module Interface (DMI) . . . 12
4.7 Debug Module (DM) . . . 12
4.7.1 Anbindung an den DMI-Bus . . . 13

4.7.2 Abstrakte Befehle 14
4.7.3 Anbindung an den Systembus 15
4.7.4 Asynchrones Halten 16

5 Simulator mit Soft-Debug-Infrastruktur 16

6 Fazit 18
6.1 Zusammenfassung 18
6.2 Ausblick 18

Literaturverzeichnis 18

A Anhang a
A.1 Debug Session . . . . . a
A.2 Testprogramm . . . . . a

Abbildungsverzeichnis

2.1 Übersicht PicoNut 4

4.1 Übersicht PicoNut mit Soft-Debug-Infrastruktur 8
4.2 Ablauf im *Debug Mode* 9
4.3 Aufbau des *DMI*-Registers 11
4.4 Aufbau des HARTCONTROL-Registers 16
4.5 Aufbau des HARTSTATUS-Registers 16

A.1 Debug Session mit GDB im Terminal . . . . . a

Tabellenverzeichnis

2.1 Remote Bitbang-Kodierung 6

4.1 Register des *Debug Transport Modules* 11
4.2 System Register im Debug Module 13
4.3 System Register im Debug Module 15

Verzeichnis der Listings

A.1 Code des Testprogrammes . . . . . a

1 Einleitung

1.1 Motivation

Ein Ziel PicoNut-Projektes ist es auf dem PicoNut komplexe Software auszuführen. Das fängt bei dem 3D-Spieleklassiker Doom, Betriebssysteme wie FreeRTOS, Linux und ist nur durch die Gänze der Softwarewelt begrenzt. Bei Softwareentwicklung sind Fehler in der Entwicklung unausweichlich. Um nun Fehler besser analysieren zu können gibt es Werkzeuge wie den Debugger.

Debugger wie der GNU Debugger (GDB) erlauben es Entwickelnden, das Verhalten des Programms zu analysieren, indem sie unter anderem die Möglichkeit erhalten, das Programm anzuhalten, Variablen und Speicher auszulesen und zu verändern. So können Fehler in Programmen effizient gefunden und behoben werden. Das gestaltet die Arbeit der Softwareentwicklenden deutlich effizienter.

Um Eingebettete Systeme ohne Betriebssystem zu Debuggen wie, wie den PicoNut, braucht es von Seiten des Eingebetteten Systems eine entsprechende Hardwareunterstützung.

1.2 Ziel der Arbeit

Das Ziel dieser Arbeit ist die Entwicklung einer Debug-Infrastruktur für den PicoNut. Diese soll vollständig in C++ implementiert werden und ausschließlich in der Simulationsumgebung funktionieren. Das heißt sie muss nicht synthetisierbar sein. Die Umsetzung erfolgt gemäß der *RISC-V External Debug Specification* [9] und dient als Grundlage, um größere Softwareprojekte auf dem PicoNut zu realisieren. Mit einer zuverlässig arbeitenden Debugging-Lösung wird es möglich, Fehler in komplexen Programmen effizient zu finden und zu beheben.

1.3 Aufbau der Arbeit

Zu Beginn der Arbeit, in Abschnitt 2 gibt es eine Einführung in das PicoNut-Projekt und die besondere Soft-Peripheral-Schnittstelle. Gefolgt von einer Vorstellung essenzieller Debugging-Werkzeuge: GDB und OpenOCD. Anschließend werden die grundlegenden Konzepte der RISC-V ISA sowie die *RISC-V External Debug Specification* [9] erläutert.

Folgend wird in Abschnitt 3 ein bestehender RISC-V-Prozessor gezeigt, der schon mit einer entsprechenden Debug-Infrastruktur ausgestattet ist.

Danach wird in Abschnitt 4 die entwickelte Debug-Infrastruktur im Detail betrachtet, insbesondere die drei Hauptkomponenten: das *Remote Bitbang Module*, das *Debug Transport Module* und das *Debug Module*.

In Abschnitt 5 wird die Simulationsumgebung vorgestellt, in die Debug-Infrastruktur integriert wurde um die Funktionalität zu validieren.

Abschließend wird in Abschnitt 6 die Arbeit zusammengefasst und ein Ausblick auf zukünftige Entwicklungen gegeben, wie eine Debug-Infrastruktur für den PicoNut als synthetisierbares Modul.

2 Grundlagen

2.1 Einführung in das PicoNut-Projekt

2.1.1 Projektziele

Das PicoNut-Projekt, der Technischen Hochschule Augsburg im Rahmen der EES-Arbeitsgruppe [5, 13], verfolgt seit Beginn des Jahres 2024 das Ziel, einen minimalen und flexibel erweiterbaren RISC-V-Prozessor zu entwickeln. Der Prozessor soll auf gängiger FPGA-Hardware funktionieren, einen Simulator bereitstellen und über leicht zu verstehende Schnittstellen verfügen. Des weiteren sollen die Teilkomponenten des Prozessors flexibel austauschbar sein, um einen konfigurierbaren Prozessor zu erhalten. Entwicklungsziele des PicoNut-Projektes sind die Unterstützung von Betriebssystemen wie FreeRTOS und Linux, sowie die Integration von Hardwarebeschleunigern für KI-Anwendungen. In diesem Zusammenhang soll der Prozessor für die Lehre und Forschung an der Technischen Hochschule Augsburg eingesetzt werden. [6]

2.1.2 Gesamtsystem

Der PicoNut besteht aus dem dem PicoNut-Kern, dem Systembus und den an den Systembus angeschlossene Peripheriemodulen. Eine Übersicht ist in Abbildung 2.1 dargestellt. Der PicoNut-Kern besteht aus dem *Nucleus*, dem eigentlichen Rechenkern und der *Membrana*. Die Membrana. Diese bildet die Systembusschnittstelle für den *Nucleus*. [4]

2.1.3 Die „Soft-Peripheral“-Schnittstelle

Eine Kerneigenschaft des PicoNuts ist es Module, die mit der Hardwarebeschreibungsbibliothek SystemC [12] implementiert wurden, sowie reine Softwaremodule in einer Simulation zusammen simulieren zu können. Speziell um Soft-Peripheriemodule an das System anzuschließen wurde hierfür eine eigene *Membrana* von Marco Milenkovic

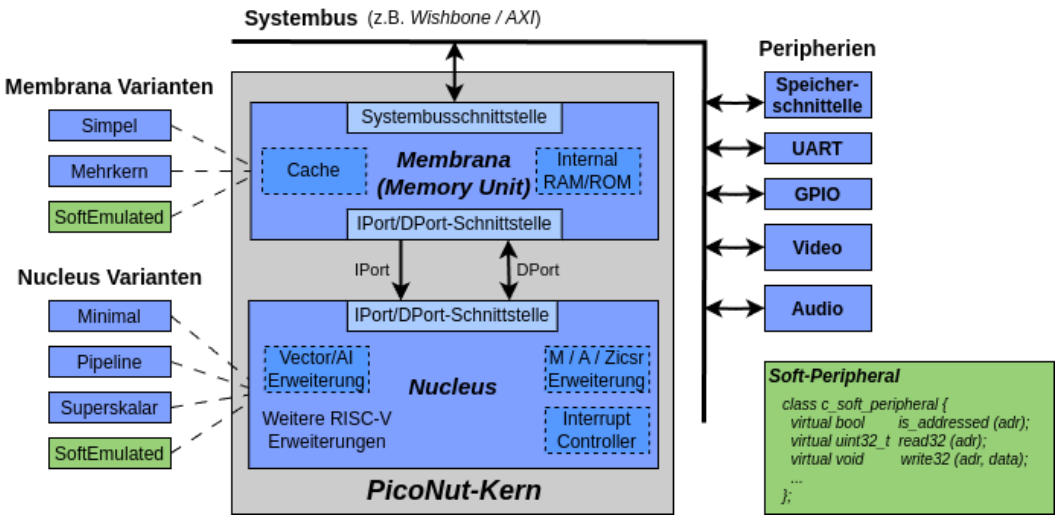

Abbildung 2.1: Übersicht PicoNut

<Marco.Milenkovic@tha.de> entwickelt. Die sogenannte *Soft-Memory Unit* verwaltet Speicherzugriffe des *Nucleus* auf die angebundenen sogenannten *Soft-Peripherals*. Die *Soft-Memory Unit* ist selbst ebenfalls *nicht* synthetisierbar. [4]

Teile der Debug-Infrastruktur sind mit einer Registerschnittstelle an den Systembus angeschlossen. Das wird benötigt, damit der Nucleus Daten mit dem *Debug Host* austauschen kann, sowie für Ablaufsteuerungsmechanismen. Da in dieser Arbeit eine Debug-Infrastruktur für eine Simulationsumgebung entwickelt wurde, ist diese ebenfalls als reines Softwaremodul implementiert. Die Teilmodule mit einer Registerschnittstelle sind als *Soft-Peripheral* implementiert, siehe 4.7.3.

2.2 RISC-V External Debug Support

Die *RISC-V External Debug Specification* [9] ermöglicht das Debugging von RISC-V-Prozessoren über externe Debug-Schnittstellen wie JTAG oder SWD. Es umfasst Funktionen wie das Setzen von Breakpoints, das Auslesen und Schreiben von Registern und Speicher, sowie die Steuerung der Programmausführung. Der Debug-Mechanismus basiert auf einem *Debug Module*, siehe 4.7, das über ein standardisiertes Interface mit einem Debug-Übersetzer wie OpenOCD kommuniziert. Der Debug-Übersetzer dient als Schnittstelle zwischen dem eingebetteten Systems und dem Debugger. [9]

2.3 On-Chip-Debugging

2.3.1 GDB (GNU Debugger)

GDB (GNU Debugger) ist ein weit verbreitetes Open-Source-Tool für Debugging, auch Debugger genannt. Es ermöglicht Entwickelnden, Programme schrittweise auszuführen, Speicherinhalte zu lesen und zu verändern, Breakpoints zu setzen und Variablen zu manipulieren. Für das entwickeln komplexerer Programme ist das Debuggen ein hilfreiches Tool um Fehler in Programmen effizient zu finden. [7]

GDB kann in Kombination mit OpenOCD für Embedded-Entwicklung genutzt werden um Programme zu debuggen, die nicht lokal auf dem Host Computer ausgeführt werden. [7]

2.3.2 OpenOCD

OpenOCD (Open On-Chip Debugger) ist ein Open-Source-Tool zur Hardware-Debugging und Programmierung von Mikrocontrollern oder selbst entwickelter Hardware. OpenOCD wird in Kombination mit GDB genutzt um die Funktionalitäten von GDB beim debuggen nativer Programme auf Programme zu übertragen, die auf der echten oder auch simulierten Hardware ausgeführt werden. Das OpenOCD-Projekt wurde von Dominic Rath an der Technischen Hochschule Augsburg gestartet und ist mittlerweile weltweit im Einsatz. [11]

OpenOCD bietet dabei eine Schnittstelle für GDB sowie diverse Schnittstellen für die Hardware. Darunter zählen zum Beispiel JTAG, SWD sowie die Remote Bitbang-Schnittstelle, die speziell für die Kommunikation mit softwaregesteuerten JTAG-Adaptern gedacht ist. Diese Schnittstelle ermöglicht es, JTAG-Signale über eine serielle oder TCP/IP-Verbindung zu steuern. Damit ist es möglich Simulationsmodelle mit OpenOCD zu verbinden. [11]

2.3.3 Remote Bitbang-Protokoll

Das Remote Bitbang-Protokoll ist eine einfache, textbasierte Schnittstelle zur Steuerung von JTAG-Pins über eine TCP/IP Verbindung. Das ist nützlich bei Debug-Hardware, die keine native JTAG-Unterstützung bietet zum Beispiel bei simulierter Hardware. [2]

Das Protokoll kodiert verschiedene Befehle als ASCII Zeichen. Es ist möglich mittels der Kodierung in Tabelle 2.1 die Pegel der JTAG-Eingangssignale *tms*, *tdi*, *tck* zu

schreiben, sowie den Pegel des Ausgangssignal *tdo* zu lesen. Zudem kann dem remote mitgeteilt werden, den TCP/IP Socket zu schließen. Außerdem lässt sich das *Debug Transport Module* damit zurücksetzen.[2, 3]

Zeichen	Beschreibung
R	Lese Pegel von *tdo*
Q	TCP/IP Socket Verbindung schließen
0	Schreibe Pegel *tms=0*, *tdi=0*, *tck=0*
1	Schreibe Pegel *tms=0*, *tdi=0*, *tck=1*
2	Schreibe Pegel *tms=0*, *tdi=1*, *tck=0*
3	Schreibe Pegel *tms=0*, *tdi=1*, *tck=1*
4	Schreibe Pegel *tms=1*, *tdi=0*, *tck=0*
5	Schreibe Pegel *tms=1*, *tdi=0*, *tck=1*
6	Schreibe Pegel *tms=1*, *tdi=1*, *tck=0*
7	Schreibe Pegel *tms=1*, *tdi=1*, *tck=1*
r	Reset des *Debug Transport Modules*

Tabelle 2.1: Remote Bitbang-Kodierung

3 Debug-Infrastruktur im ParaNut-Prozessor

Der ParaNut-Prozessor ist ein an der Technischen Hochschule Augsburg von Studierenden entwickelter Prozessor mit RISC-V-Architektur. Er verfügt über eine synthetisierbare Debug-Infrastruktur, die auf der *RISC-V External Debug Specification* [9] basiert. Obwohl der ParaNut als Parallelprozessor konzipiert wurde, konnte in der bisherigen Implementierung der Debug-Infrastruktur nur ein Kern aktiv Debugged werden. In dieser Hinsicht zeigt das Projekt eine starke Ähnlichkeit zum PicoNut, da auch hier die Debugging-Funktionalität auf einen einzelnen Kern beschränkt ist. [1]

4 Soft-Debug-Infrastruktur

4.1 Anforderungen

Es soll eine Debug-Infrastruktur für den PicoNut entwickelt werden. Diese soll *RISC-V External Debug Specification* [9] Version 0.13.2 konform sein. Ziel soll sein Debugging in Simulationsumgebung mit Hilfe von OpenOCD und GDB zu ermöglichen. Dafür müssen folgende Funktionalitäten unterstützt werden:

- Halt- und Resume-Funktion
- Breakpoints setzen
- Single Step-Funktion
- GPRs lesen und schreiben
- CSRs lesen und schreiben
- Speicher lesen und schreiben

4.2 Übersicht

Die Debug-Infrastruktur besteht aus drei Modulen: das *Remote Bitbang Module*, das *Debug Transport Module* und das Herzstück der Infrastruktur bildet das *Debug Module*. In Abbildung 4.1 ist eine Übersicht der Infrastruktur mit den genannten Modulen zu sehen.

Der Benutzer interagiert mit dem *Debug Host* (z.B. ein Laptop), auf dem ein Debugger (z.B. GDB) ausgeführt wird. Der Debugger kommuniziert mit einem Debug-Übersetzer (z.B. OpenOCD). Dieser kann mit dem *Remote Bitbang Module* des PicoNuts kommunizieren. Das *Remote Bitbang Module* verbindet via JTAG-Schnittstelle den *Debug Host* mit dem *Debug Transport Module*. Das *Debug Transport Module* ermöglicht den Zugriff auf ein *Debug Module* über den *Debug Module Interface*-Bus. [9]

In Abschnitt 4.3 wird das Verhalten des Nucleus beschrieben, nachdem er angehalten wurde.

In Abschnitt 4.4 wird die Implementierung des *Remote Bitbang Modules* beschrieben. Es wird die Schnittstelle gezeigt mit der sich OpenOCD mit dem PicoNut verbindet.

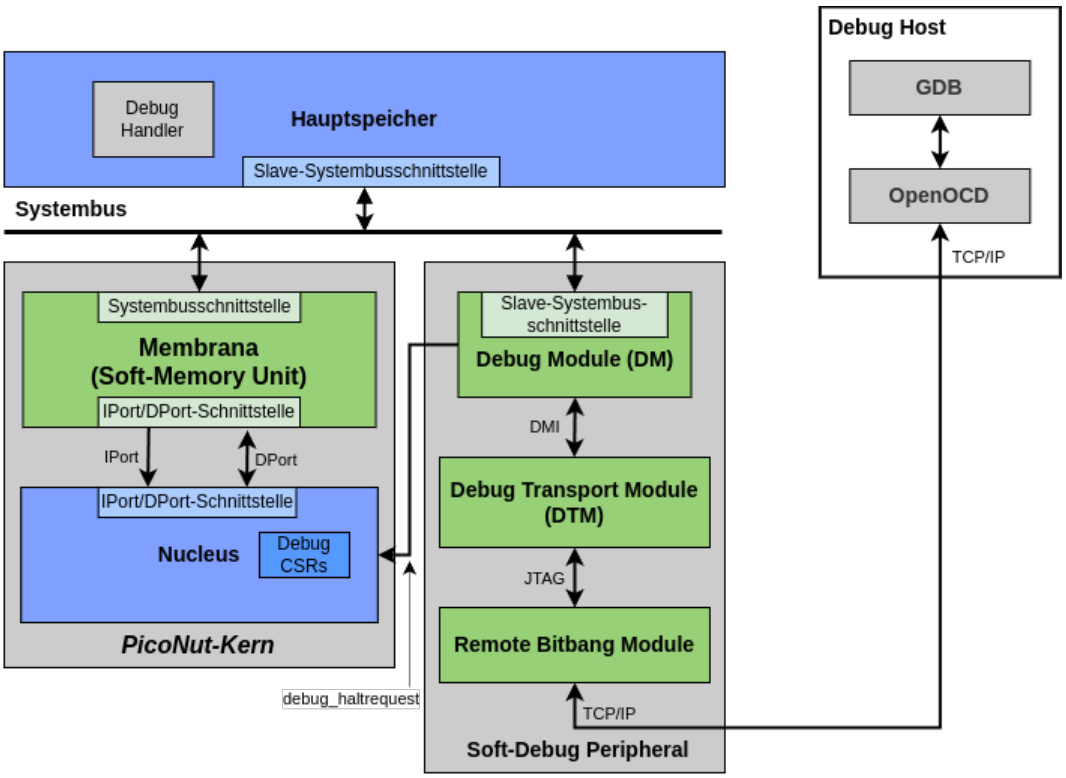

Abbildung 4.1: Übersicht PicoNut mit Soft-Debug-Infrastruktur

In Abschnitt 4.5 wird die Implementierung des *Debug Transport Modules* beschrieben.

In Abschnitt 4.6 wird der *Debug Module Interface*-Bus beschrieben. Es wird gezeigt wie es für eine Simulationsumgebung effizient gestaltet werden konnte.

In Abschnitt 4.7 wird die Implementierung des *Debug Modules* beschrieben. Außerdem wird auf die Registerschnittstellen des Moduls eingegangen. Zudem wird das Konzept der abstrakten Befehle um effizient Maschinenbefehle auf dem PicoNut ausführen zu können erläutert. Abschließend wird beschrieben wie der Nucleus asynchron angehalten werden kann.

4.3 Debug Mode

Der *Debug Mode* ist ein spezieller Prozessormodus in dem nicht normale Befehle ausführt werden, sondern den sogenannten *Debug Handler*. Der Ablauf im *Debug Mode* ist in Abbildung 4.2 dargestellt. Wird der Nucleus angehalten um Debuggt zu werden, wechselt er in den *Debug Mode*. Der *Debug Mode* wird Betreten, wenn eine der folgenden Anforderungen erfüllt ist:

- Das *debug_haltrequest*-Signal ist gesetzt.

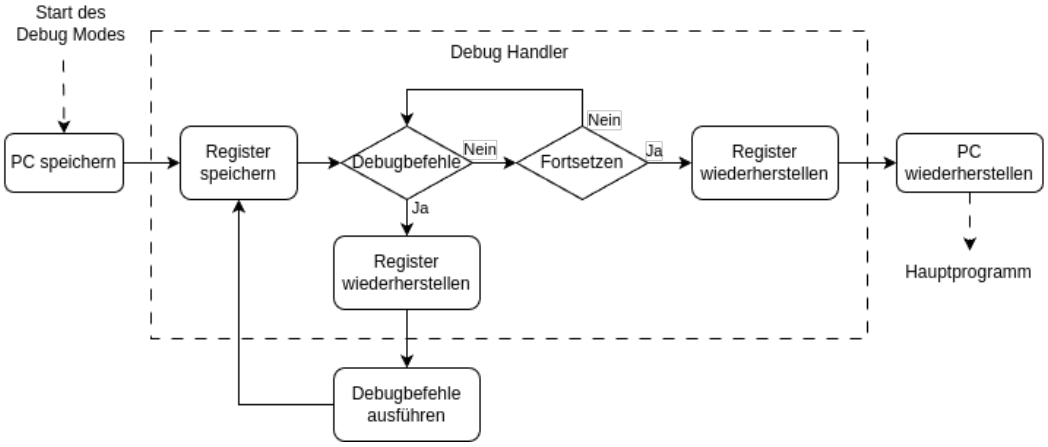

Abbildung 4.2: Ablauf im *Debug Mode*

- Der Breakpoint-Befehl wurde gelesen.
- Ein Maschinenbefehl wurde ausgeführt, und die Single Step-Funktion ist aktiv.

Der Nucleus führt im *Debug Mode* eine Routine aus – den *Debug Handler*. Dafür wird beim Betreten des *Debug Modes* der aktuelle Wert des Programmzählers im CSR *DPC (Debug Programm Counter)* abgespeichert. Anschließend wird der Programmzähler auf die Startadresse des *Debug Handlers* gesetzt und der *Debug Handler* wird ausgeführt.

Der *Debug Handler* is ein kleines Programm, das als Maschinenbefehle im Hauptspeichers liegt. In Abbildung 4.2 ist der Programmablauf dargestellt. Der letzte Befehl der abstrakten Befehle ist immer ein Breakpoint-Befehl. Damit startet der *Debug Handler* automatisch erneut, nachdem alle abstrakten Befehle ausgeführt wurden. Der *Debug Handler* besteht aus vier Abschnitten. Die Abschnitte werden, wenn nicht anders beschrieben, sequenziell Ausgeführt:

Setup

- Abspeichern von GPRs, die im *Debug Handler* verwendet werden.

Wait

- Warten, bis das *resume*-Bit oder das *run_commands*-Bit im *HARTCONTROL*-Register gesetzt ist.
- Ist das *resume*-Bit gesetzt, erfolgt ein Sprung in den Abschnitt *Exit*.
- Ist das *run_commands*-Bit gesetzt, erfolgt ein Sprung in den Abschnitt *Execute Commands*.

Execute Commands

- Wiederherstellen der zuvor gespeicherten GPRs.
- Der Programmzähler wird auf die Adresse gesetzt, die im *ABSTRACT0*-Register gespeichert ist. Dieses Register enthält den ersten auszuführenden abstrakten Befehl, siehe dazu Abschnitt 4.7.3.

Exit

- Wiederherstellen der zuvor gespeicherten GPRs.
- Der Programmzähler wird auf die Adresse gesetzt, die einem Befehl nach dem letzten ausgeführten Befehl entspricht – unmittelbar vor dem Start des *Debug Handlers*.

4.4 Remote Bitbang Module

Das *Remote Bitbang Module* dient als Schnittstelle zwischen *Debug Transport Module* und OpenOCD. Normalerweise steuert OpenOCD direkt einen JTAG-Adapter, welcher mit den JTAG-Signalen an das Debug Transport Modul angeschlossen wird. Da das in einer Simulation nicht möglich ist, werden die JTAG-Signale per Remote Bitbang-Protokoll an das *Remote Bitbang Module* gesendet. Dieses emuliert die JTAG-Hardwareschnittstelle nach Tabelle 2.1.

Dazu implementiert das Modul einen TCP/IP-Socket. Der Socket öffnet einen Port mit dem sich OpenOCD, nach entsprechender Konfigurierung verbindet. Die JTAG-Schnittstelle wird durch folgende Funktionen realisiert. Das Modul hat eine Callback-Funktion, die Aufgerufen wird, wenn sich der Pegel der JTAG Eingangssignale *tck*, *tms* oder *tdi* ändert. Der Pegel des *tdo*-Signals wird intern gespeichert und via Funktionsaufruf von außen kontinuierlich auf den aktuellen Pegel gesetzt. [4]

4.5 Debug Transport Module (DTM)

Das *Debug Transport Module* dient als Schnittstelle zwischen einem JTAG-Anschluss und dem *Debug Module*. Die Schnittstelle zwischen *Debug Transport Module* und dem *Debug Module* bildet das *Debug Module Interface*. [9]

Das *Debug Transport Module* orientiert sich nach der Definition eines TAP im JTAG-Standard der den Zugriff auf erweiterte spezielle definierte JTAG-Register ermöglicht

[8]. Diese sind für die Ansteuerung des *Debug Module Interfaces* und damit des *Debug Modules* zuständig. [9] Eine Übersicht über alle implementierten Register ist in Tabelle 4.1 dargestellt.

Register	JTAG Standard	Beschreibung
BYPASS	Ja	1-bit breites Register dient dazu ICs weiter hinten in der JTAG-Kette zu erreichen.
IDCODE	Ja	Read-only Register mit dem Wert `0xdeadbeef`.
DTMCS	Nein	DTM Control Status um das *Debug Transport Module* zurückzusetzen.
DMI	Nein	Lese- und Schreibzugriff auf den DMI-Bus um DM's anzusteuern.

Tabelle 4.1: Register des *Debug Transport Modules*

Das *BYPASS*-Register hat eine Breite von Eins. Es dient dazu, JTAG-Daten schnell durch zuleiten. Das ist hilfreich um ICs weiter hinten in der JTAG-Kette zu erreichen. [8]

Das *IDCODE*-Register enthält eine Eindeutige Chip-ID und hilft zur Identifikation des ICs. Im implementierten *Debug Transport Module* ist der Wert `0xdeadbeef`. [8]

Das *DTMCS*-Register (Debug Transport Module Control and Status) wird verwendet um den *Debug Module Interface*-Bus zurückzusetzen und um den Status des *Debug Transport Modules* auszulesen. Außerdem ist die Adressbreite des *Debug Module Interface*-Busses hier abgespeichert. [9]

Das *DMI*-Register wird genutzt um den *Debug Module Interface*-Bus anzusteuern. Der Aufbau des Registers ist in Abbildung 4.3 dargestellt.

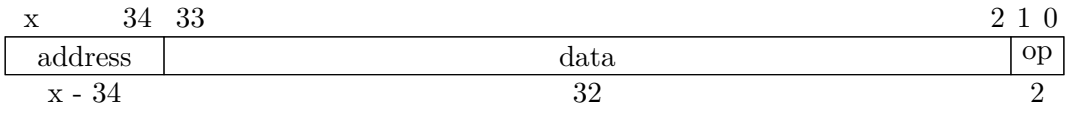

Abbildung 4.3: Aufbau des *DMI*-Registers

- *address:* Adresse des Registers auf dem *Debug Module Interface*-Bus
  Die Adressbreite ist Implementierungsabhängig.
- *data:* Daten die gelesen oder geschrieben werden
- *op:* Opcode gibt an ob gelesen oder geschrieben werden soll:
  - Lesezugriff: `0b01`

– Schreibzugriff: `0b10`

[9]

4.6 Debug Module Interface (DMI)

Das *Debug Module Interface* ist die Schnittstelle zwischen dem *Debug Transport Module* und dem *Debug Module*. Es ist ein einfaches Master-Slave Bussystem, in dem das *Debug Transport Module* der Master und das *Debug Module* der Slave ist. [9]

Da die Soft-Debug-Infrastruktur als reines Softwaremodell implementiert ist, wurde der *Debug Module Interface*-Bus mit normalen Public-Funktionen, die das *Debug Module* implementiert, realisiert. Diese werden vom *Debug Transport Module* Aufgerufen um Register, die an dem *Debug Module Interface*-Bus angesteuert sind, zu lesen und zu schreiben. [4] Das spart die aufwendige Auslegung eines eigenen Bussystems mit echten simulierten Hardwaresignalen. Das beschleunigt die Ausführungszeit der Simulation.

4.7 Debug Module (DM)

Das *Debug Module* ist das Herzstück der Debug-Infrastruktur. Es setzt die Anweisungen, die es über den *Debug Module Interface*-Bus erhält so um, dass die implementierte Hardware korrekt darauf reagiert. Die *RISC-V External Debug Specification* [9] gibt dafür Register vorgesehen die an den *Debug Module Interface*-Bus angeschlossen sind. Die Schnittstelle mit dem *Debug Module Interface*-Bus wird in Abschnitt 4.7.1 beschrieben.

Der PicoNut hat zum Zeitpunkt der Erstellung dieser Arbeit als ausführenden Teil des Prozessorkerns den *Minimalen Nucleus*. Der Minimale Nucleus ist eine minimale Implementierung des PicoNut-Nucleus. Dieser wurde von Lorenz Sommer <Lorenz.Sommer@tha.de> an der Technischen Hochschule Augsburg im Zuge seiner Bachelorarbeit 2024 entwickelt. Der Minimale Nucleus ist ein Einkerner, daher ist für die Debug Infrastruktur nur ein *Debug Module* notwendig. [10]

Für Breakpoints wird der Befehl an dem ein Breakpoint gesetzt ist im Speicher mit dem Breakpoint-Befehl ausgetauscht. Liest der Nucleus als nächsten Befehl den Breakpoint-Befehl wird automatisch in den *Debug Mode* gewechselt. Im *Debug Module* ist deshalb keine extra Logik notwendig.

Für die Hardwareunterstützung der Single Step-Funktion, also das Ausführen *eines* Maschinenbefehls und danach sofortigen Betreten des *Debug Modes*, ist keine eigene Logik im *Debug Module* notwendig. Diese Funktionalität wird mit dem *step*-Bit im *DCSR (Debug Control and Status)* CSR-Register, sowie Logik im Nucleus erreicht. Ist das Bit gesetzt wird automatisch nach dem Ausführen eines Maschinenbefehls in den *Debug Mode* gewechselt. [9]

Es wurden nur die Funktionen implementiert um standardmäßiges Debugging zu ermöglichen. Das standardmäßige Debugging umfasst die Funktionen die in Abschnitt 4.1 beschrieben wurden. Damit lassen sich Programme weitreichend Debuggen. Bei den nicht implementierten Funktionen handelt sich um kleinere Optimierungen die es Beschleunigen Daten zwischen *Debug Host* und PicoNut auszutauschen. Mehr zu den nicht implementierten Funktionen im Fazit 6.

4.7.1 Anbindung an den DMI-Bus

Das *Debug Module* hat zudem eine Anbindung an den *Debug Module Interface*-Bus. Die Register, die von dort aus erreichbar sind, sind in der *RISC-V External Debug Specification* [9] Spezifikation beschreiben und danach implementiert. Diese sind in Tabelle 4.2 aufgeführt. Diese Implementierung ist ausreichend um die Anforderungen zu erfüllen.

Register	Beschreibung
DATA0 - DATA1	Register für Datenaustausch zwischen Debugger und PicoNut
DMCONTROL	Steuerung des DM's und der ausgewählten Harts.
DMSTATUS	Status des DM's und der ausgewählten Harts.
ABSTRACTCS	Control Register zur Steuerung der abstrakten Befehle
COMMAND	Quellregister für die abstrakten Befehle
PROGBUF0 - PROGBUF2	Enthält direkt vom Debugger kommende Maschinenbefehle

Tabelle 4.2: System Register im Debug Module

Die Register *DATA0 - DATA1* sind Speicherregister und dienen für den Datenaustausch zwischen *Debug Host* und PicoNut. Beispielsweise werden beim Lesen eines GPR's zuerst der Inhalt in das *DATA0*-Register geladen. Anschließend liest der *Debug Host* die Daten aus dem *DATA0*-Register [9]. Damit der Prozessor auf diese Register zugreifen kann sind sie zusätzlich an den Systembus mit Lese- und Schreibrechten angebunden. Diese werden in Abschnitt 4.7.3 beschrieben.

Das *DMCONTROL*-Register steuert das *Debug Module*, sowie den angebunden Nucleus. Die verwendeten Funktionen des Registers umfassen das *haltreq*-Bit und das *resumereq*-Bit. Diese Flags werden dazu verwendet den Nucleus asynchrone Anzuhalten, siehe dazu Abschnitt 4.7.4, bzw. fortsetzten zu lassen.

Das *DMSTATUS*-Register ist als Read-Only Register implementiert und hält Statusinformationen über das *Debug Module*, sowie den angebunden Nucleus. Die verwendeten Funktionen des Registers umfassen zwei *ack*-bits. Jeweils für das *haltreq*-Bit und das *resumereq*-Bit des *DMCONTROL*-Registers. Außerdem wird der aktuelle Status des Nucleus in den Bits *allrunning* und *allhalted* explizit festgehalten.

Das *ABSTRACTCS*-Register, ausgeschrieben Abstract Control Status, enthält Informationen über die Generierung der *abstrakten Befehle*. Die Abstrakten Befehle werden in Abschnitt 4.7.2 beschrieben.

Das *COMMAND*-Register hält die abstrakten Informationen für die zu generierenden Abstrakten Befehlen. Das *COMMAND*-Register wird in Abschnitt 4.7.2 näher beschrieben.

Die Register *PROGBUF0 - PROGBUF2* sind Speicherregister und dienen dazu direkte RISC-V Maschinenbefehle zu halten, die vom Nucleus ausgeführt werden können. Damit der Nucleus auf diese Register zugreifen kann sind sie zusätzlich an den Systembus mit Leserechten angebunden. Die Anbindung an den Systembus wird in Abschnitt 4.7.3 beschrieben.

4.7.2 Abstrakte Befehle

Damit der Nucleus Befehle vom *Debug Host* während der Laufzeit ausführen kann sind in der *RISC-V External Debug Specification* [9] zwei Wege beschrieben. Mit den Registern *PROGBUF0 - PROGBUF2* lassen sich direkt vom Benutzer gestellte Maschinenbefehle auf dem Nucleus ausführen. Der Nachteil dieser Methode ist, dass es mehrere Maschinenbefehle braucht um Beispielsweise eine Speicheradresse zu lesen und den Inhalt in das *DATA0*-Register zu schreiben. Somit ist ein erhöhter Datenaustausch zwischen *Debug Host* und PicoNut die Folge.

Eine andere Methode ist der abstrakte Befehl. Dabei wird der Befehl wird in abstrakter Form an das *Debug Module* übermittelt und erst im *Debug Modul* zu mehreren echten Maschinenbefehlen umgewandelt. Der abstrakte Befehl wird in das *COMMAND*-Register geschrieben. Damit müssen deutlich weniger Daten zwischen *Debug Host* ausgetauscht werden, da nur ein Register beschrieben werden muss.

Nachdem das *COMMAND*-Register beschrieben wurde werden die entsprechenden Maschinenbefehle erzeugt.Es werden drei Arten von abstrakten Befehlen unterstützt:

- GPRs lesen und schreiben.
- CSRs lesen und schreiben.
- Speicher lesen und schreiben.

Die erzeugten Maschinenbefehle werden in den Systembusregistern *ABSTRACT0 - ABSTRACT7* abgespeichert. [9]

4.7.3 Anbindung an den Systembus

Das *Debug Module* ist an den Systembus angebunden. Das ermöglicht es dem Nucleus auf bestimmte Register im *Debug Module* über den Systembus zu zugreifen. Die Register werden für den Datenaustausch zwischen *Debug Host* und PicoNut, sowie zur Steuerung des Verhaltens des Nucleus genutzt. In Tabelle 4.3 ist eine Übersicht der Register dargestellt.

Register	Beschreibung
DATA0 - DATA1	Register für Datenaustausch zwischen Debugger und PicoNut.
PROGBUF0 - PROGBUF2	Enthält direkt vom Debugger kommende Maschinenbefehle.
ABSTRACT0 - ABSTRACT7	Enthält vom Debug Modul generierte Maschinenbefehle, siehe Abschnitt 4.7.2.
HARTCONTROL	Control Register zur Steuerung des Harts.
HARTSTATUS	Status Register für Status des Harts.

Tabelle 4.3: System Register im Debug Module

Die Regiser *DATA0 - DATA1* und *PROGBUF0 - PROGBUF2* werden in Abschnitt 4.7.1 beschrieben.

Die Register *ABSTRACT0 - ABSTRACT7* werden im Abschnitt 4.7.2 beschrieben.

Das *HARTCONTROL*-Register wird dazu verwendet, den Ablauf des *Debug Handler*-Programmes zu steuern. Das *Debug Handler*-Programm fragt in der Warteschleife kontinuierlich das *HARTCONTROL*-Register ab. Der Aufbau des Registers ist in

Abbildung 4.4 zu sehen. Das *resumereq*-Bit gibt an, ob das Hauptprogramm fortgesetzt werden soll. Das *run_commandsreq*-Bit gibt an, ob die abstrakten Befehle ausführt werden sollen. Der Ablauf des *Debug Handler*-Programmes wird in Abschnitt 4.3 beschrieben.

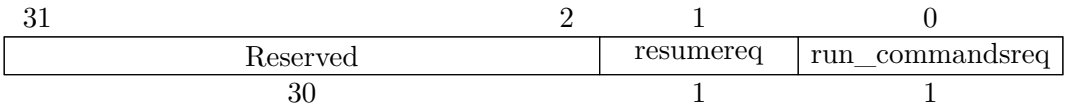

Abbildung 4.4: Aufbau des HARTCONTROL-Registers

Das *HARTSTATUS*-Register ist das Gegenteil des *HARTCONTROL*-Registers. Der Aufbau ist in Abbildung 4.5 dargestellt. Der Nucleus schreibt in dieses Register den aktuellen Status Nucleus im *Debug Mode*. Die Statusbits sind: *halted*, *running* und *commands_running*.

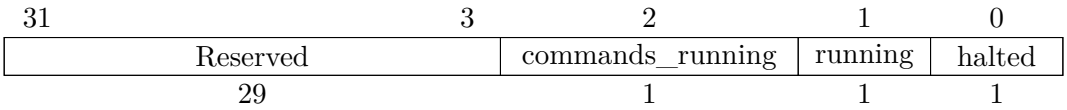

Abbildung 4.5: Aufbau des HARTSTATUS-Registers

4.7.4 Asynchrones Halten

Um einen Nucleus auch asynchron anhalten zu können gibt das Hardwaresignal *debug_haltrequest*, dass in den Nucleus führt. Die Quelle des Signals ist das *haltreq*-Bit aus dem *DMCONTROL*-Register. Ist das Signal gesetzt wechselt der Nucleus nach Beendigung des aktuell ausgeführten Befehls in den *Debug Mode*. Das verhalten des Nucleus im *Debug Mode* wird in Abschnitt 4.3 beschrieben.

5 Simulator mit Soft-Debug-Infrastruktur

Das Referenzdesign mit Soft-Debug-Infrastruktur für den PicoNut stellt eine Simulationsumgebung bereit, die einen minimalen Nucleus, eine Soft-Memory Unit und einen Soft-Debug-Infrastruktur integriert. Dieses System ermöglicht das Debuggen von reinen Konsolenanwendungen, die auf dem simulierten PicoNut ausgeführt werden. Beim Ausführen des Referenzdesigns wird nur der PicoNut gestartet. Um nun eine Debugsitzung zu starten muss OpenOCD und GDB separat gestartet werden. Eine laufende Debugsitzung ist in Appendix A.1 zu sehen.

Ein kleines Testprogramm wurde verwendet, um die Debug-Funktionalitäten des Systems zu testen. Das in C geschriebene Testprogramm gibt die Wörter „PicoNut/-RISC-V“ als ASCII-Art auf der Konsole aus. Der Programmcode ist in Appendix A.2 zu sehen. Mithilfe des GDB Debuggers, sowie der Soft-Debug-Infrastruktur konnte der Programmablauf in der Simulation überwacht, Breakpoints gesetzt und die Registerinhalte und Variablen während der Ausführung ausgelesen, sowie überschreiben werden. Dieses einfache, aber effektive Testprogramm diente dazu, die grundlegenden Debugging-Funktionen des Referenzdesigns manuel zu validieren.

6 Fazit

6.1 Zusammenfassung

In dieser Arbeit wurde eine Debug-Infrastruktur für den PicoNut, für eine Simulationsumgebung, in der Programmiersprache C++ entwickelt. Damit ist es möglich Programme die auf dem PicoNut ausgeführt werden mithilfe von GDB zu debuggen. Es ist möglich Programme Anhalten und Fortsetzen zu lassen, Breakpoints zu setzen und mit Single Step-Funktion das Programm auszuführen. Außerdem ist es möglich GPRs, CSRs und Speicher auszulesen und zu manipulieren. Das ist essenziell für Entwicklung komplexerer Software auf dem PicoNut, wie zum Beispiel dem 3D-Spieleklassiker Doom oder Betriebssystem wie FreeRTOS und Linux.

6.2 Ausblick

Die implementierte Debug-Infrastruktur ist eine eher minimale Implementierung. Die *RISC-V External Debug Specification* [9] sieht noch Erweiterungen vor wie die Unterstützung für Mehrkernprozessoren, ein *Trigger Module*, und Optimierungen für schnellere Speicherzugriffe. Eine Reset-Infrastruktur, um vom *Debug Host* das System zurückzusetzen zu können ist zudem von der Spezifikation vorgesehen, sowie nützlich für noch effizienteres Debugging.

Es wurde die Debug-Infrastruktur geschaffen, die bisher nur in einer Simulationsumgebung funktioniert. Der PicoNut ist jedoch ein Prozessor, der auch auf echter FPGA-Hardware funktionieren soll. Daher besteht der nächste und wichtigste Schritt darin, die Debug-Infrastruktur mithilfe von synthetisierbarem SystemC-Code zu implementieren. Das ist insbesondere wichtig, um das Verhalten der zu entwickelten Hardware besser analysieren und verstehen zu können. Damit können auch Fehler in der echten Hardware auf dem FPGA effizienter gefunden werden.

Literaturverzeichnis

[1] Bauer, Lukas: "Weiterentwicklung der Debug-Infrastruktur des ParaNut-Prozessors". Bachelorarb. TH Augsburg, 2023 (siehe S. 6).

[2] *Bitbang-Protokoll*. Wikipedia. 2025. URL: https://en.wikipedia.org/wiki/Bit_banging (besucht am 10. 04. 2025) (siehe S. 5, 6).

[3] Borneo, Antonio: *Remote Bitbang OpenOCD Developer's Guide*. 2024. URL: https://github.com/openocd-org/openocd/blob/master/doc/manual/jtag/drivers/remote_bitbang.txt (siehe S. 6).

[4] Forschungsgruppe EES: *PicoNut Manual*. TH Augsburg. 2025. URL: https://ees.tha.de/piconut/manual/ (siehe S. 3, 4, 10, 12).

[5] Forschungsgruppe EES: *Webseite Effiziente Eingebettete Systeme*. TH Augsburg. 2025. URL: https://ees.tha.de/ (besucht am 10. 04. 2025) (siehe S. 3).

[6] Forschungsgruppe EES: *Webseite PicoNut-Projekt*. TH Augsburg. 2025. URL: https://ees.tha.de/piconut/index.html (besucht am 10. 04. 2025) (siehe S. 3).

[7] GDB developers: *GDB: The GNU Project Debugger*. 2025. URL: www.sourceware.org/gdb/ (besucht am 10. 04. 2025) (siehe S. 5).

[8] IEEE: *IEEE Standard Test Access Port and Boundary-Scan Architecture*. 1990. DOI: 10.1109/IEEESTD.1990.114395. URL: https://ieeexplore.ieee.org/document/211226 (siehe S. 11).

[9] Newsome, Tim; Wachs, Megan: *RISC-V External Debug Support*. 0.13.2. 2019. URL: https://riscv.org/wp-content/uploads/2024/12/riscv-debug-release.pdf (siehe S. 1, 2, 4, 6, 7, 10–15, 18).

[10] Sommer, Lorenz: "Entwurf eines RISC-V-Prozessors mit quelloffenen Tools". Bachelorarb. TH Augsburg, 2024 (siehe S. 12).

[11] The OpenOCD Project: *Open On-Chip Debugger: OpenOCD User's Guide*. 0.12.0. The OpenOCD Project. 2022. URL: https://openocd.org/doc-release/pdf/openocd.pdf (siehe S. 5).

[12] *Webseite SystemC*. Accellera Systems Initiative Inc. 2025. URL: https://systemc.org/ (besucht am 10. 04. 2025) (siehe S. 3).

[13] *Webseite TH Augsburg*. TH Augsburg. 2025. URL: https://www.tha.de/ (besucht am 11. 04. 2025) (siehe S. 3).

Ich, Johannes Hofmann, versichere hiermit, dass ich die vorliegende Arbeit mit dem Thema

> *Implementierung einer Debug-Infrastruktur für einen RISC-V-Prozessor in Simulationsumgebung*

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Augsburg, den 8. Juli 2025

Johannes Hofmann

A Anhang

A.1 Debug Session

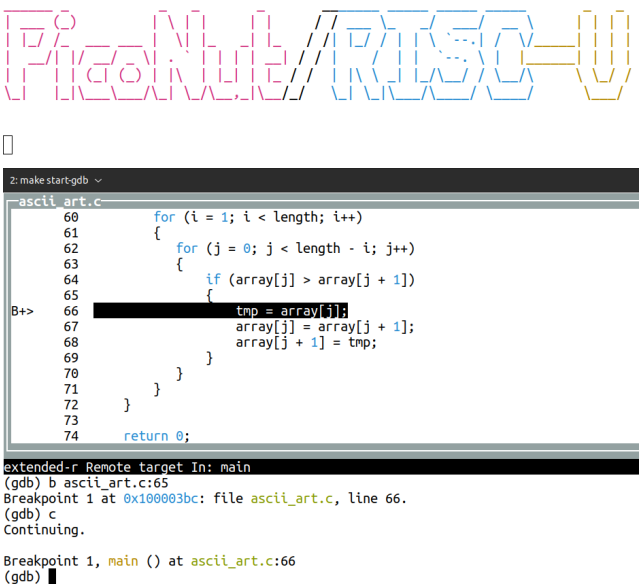

Abbildung A.1: Debug Session mit GDB im Terminal

A.2 Testprogramm

Listing A.1: Code des Testprogrammes

```
1  #include <stdio.h>
2
3  #define KNRM  "\x1B[0m"
4  #define KRED  "\x1B[31m"
5  #define KGRN  "\x1B[32m"
6  #define KYEL  "\x1B[33m"
7  #define KBLU  "\x1B[34m"
8  #define KMAG  "\x1B[35m"
9  #define KCYN  "\x1B[36m"
10 #define KWHT  "\x1B[37m"
11
```

```
12  int main() {
13
14      while(1) {
15
16          printf("%s______ _            _   _       _       %s__%
                s_____ _____ _____ _____       %s_   _ \n", KMAG, KNRM
                , KBLU, KYEL);
17          printf("%s| ___ (_)          | \\ | |     | |     %s/ /%s
                ___ \\_   _/  ___/  __ \\     %s| | | |\n", KMAG, KNRM
                , KBLU, KYEL);
18          printf("%s| |_/ /_  ___ ___ |  \\| |_   _| |_   %s/ /%s| |
                _/ / | | \\ `--.| /  \\/%s_____| | | |\n", KMAG, KNRM,
                 KBLU, KYEL);
19          printf("%s|  __/| |/ __/ _ \\| . ` | | | | __| %s/ /%s |
                   /  | |  `--. \\ |  %s|______| | | |\n", KMAG, KNRM,
                 KBLU, KYEL);
20          printf("%s| |   | | (_| (_) | |\\  | |_| | |_ %s/ /%s  |
                |\\ \\ _| |_/\\__/ / \\__/\\     %s\\ \\_/ /\n", KMAG,
                 KNRM, KBLU, KYEL);
21          printf("%s\\_|   |_|\\___\\___/\\_| \\_/\\__,_|\\__%s/_/%s
                   \\_| \\_|\\___/\\____/ \\____/      %s\\___/ \n\n\n
                ", KMAG, KNRM, KBLU, KYEL);
22
23          int array[10] = {3, 7, 2, 9, 1, 5, 6, 4, 8, 0};
24          const int length = sizeof(array) / sizeof(int);
25          int i, j, tmp;
26
27          for (i = 1; i < length; i++)
28          {
29             for (j = 0; j < length - i; j++)
30             {
31                 if (array[j] > array[j + 1])
32                 {
33                     tmp = array[j];
34                     array[j] = array[j + 1];
35                     array[j + 1] = tmp;
36                 }
37             }
38          }
39      }
40
41      return 0;
42  }
```