



HELMUT SCHMIDT
UNIVERSITÄT

Universität der Bundeswehr Hamburg

Henning Wolf

Systementwurf auf dem Nexys4 DDR mit verschiedenen SoftCore-Prozessoren

Studienarbeit

Fakultät für Elektrotechnik

Studiengang: Informatik-Ingenieurwesen

Matr.-Nr. 874320 / ET2014

Betreuer: Univ.-Prof. Dr. phil. nat. habil. Bernd Klauer

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst, keine anderen als die im Quellen- und Literaturverzeichnis genannten Quellen und Hilfsmittel, insbesondere keine dort nicht genannten Internet-Quellen benutzt, alle aus Quellen und Literatur wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe und dass die auf einem elektronischen Speichermedium abgegebene Fassung der Arbeit der gedruckten entspricht.

Hamburg,

.....

(Datum)

(Unterschrift)

Inhaltsverzeichnis

1	Einleitung	1
1.1	Ziel der Arbeit	1
1.2	Struktur der Arbeit	2
2	Grundlagen	3
2.1	Softcore-Prozessoren	3
2.1.1	Vergleich SoftCores und HardCores	3
2.1.2	Typen	4
2.2	Memory Management Unit	5
2.3	Peripherie	5
2.3.1	UART	5
2.3.2	Interrupt	6
2.3.3	Timer	7
2.4	Compiler	7
2.4.1	Compiler-Modi	8
2.5	Linux	10
2.5.1	Treiber	12
2.5.2	Device Tree	12
2.5.3	Buildroot	13
3	Soft Cores	16
3.1	MicroBlaze basierte Systeme	16
3.1.1	Erstellen der Hardwarekonfiguration	17
3.1.2	Ausführen des Linux Images auf dem Artix7	27
3.2	lowRISC basierte Systeme	28
3.3	LEON3 basierte Systeme	35
3.3.1	LEON3 Hardware	36
3.3.2	Linux für LEON3	42

Inhaltsverzeichnis

4 Zusammenfassung und Ausblick	45
Literatur	46

Abkürzungsverzeichnis

FPGA Field Programmable Gate Array

XMD Xilinx Microprocessor Debugger

mb Microblaze Target

mdm Microprocessor Debug Module Target

SDK Software Development Kit

USB Universal Serial Bus

ASIC Application-Specific Integrated Circuit

SPI Serial Peripheral Interface

UART Universal Asynchronous Receiver Transmitter

CPU Central Processing Unit

IRQ Interrupt-Request

ISA Instruction Set Architecture

MIG Memory Interface Generator

AHB Advanced High-performance Bus

MMCM Mixed-Mode Clock Manager

COREGEN CORE Generator

LRU Last Recently Used

TLB Translation Lookaside Buffer

DSU Debug Support Unit

FPU Floating-Point Unit

MMU Memory Management Unit

SMP Symmetric Multi-Processing

SMAC Signed Multiply-Accumulate

UMAC Unsigned Multiply-Accumulate

ARM Advanced RISC Machines

FIFO First In-First Out

POSIX Portable Operating System Interface

KDE K Desktop Environment

GNOME GNU Object Model Environment

TCP/IP Transmission Control Protocol/Internet Protocol

BIOS Basic Input/Output System

RTOS Real-Time Operating System

PWM Pulse Width Modulation

DMA Direct Memory Access

PLB Processor Local Bus

AXI Advanced eXtensible Interface Bus

SRMMU SPARC Reference Memory Management Unit

AMBA Advanced Microcontroller Bus Architecture

L4PRHS Linux for Partial Reconfigurable Heterogeneous System

PRHS Partial Reconfigurable Heterogeneous System

VHDL Very High Speed Integrated Circuit Hardware Description Language

SoC System-on-Chip

MAC Multiply-Accumulate

ESA European Space Agency

GUI Graphical User Interface

GTK Gimp Toolkit

UUCP Unix to Unix Copy

MPU Memory Protection Unit

FAT File Allocation Table

1 Einleitung

Die Technologieentwicklung, im speziellen die der Computer, kann auf eine rasante Entwicklung in den letzten Jahrzehnten zurückblicken. Jedoch ist diese Technik mittlerweile so hochentwickelt, dass es mit zunehmender Zeit schwieriger wird, die Bauteile noch schneller, kleiner und effizienter herzustellen, denn auch hier gibt es Grenzen. So liegt es an den Entwicklern neue Wege zu finden, um diese Beschränkungen zu umgehen.

So kam es zur Entwicklung der Field Programmable Gate Arrays ([FPGAs](#)), welche es nun ermöglichen sollten, komplexe Logikschaltungen rekonfigurierbar herzustellen, sodass diese jederzeit verändert und damit den Gegebenheiten und Anforderungen angepasst werden konnten. Durch die Optimierung während der Laufzeit können Kosten gesenkt und Ressourcen geschont werden. Mittlerweile sind moderne [FPGAs](#) mit genügend Ressourcen ausgestattet, um eine große Anzahl an Logikgattern zu berechnen und sind in der Lage komplexe Rechenoperationen durchzuführen.

1.1 Ziel der Arbeit

Das Ziel dieser Arbeit ist es, verschiedene Soft-Prozessoren zu betrachten, diese an das Digilent Nexys4 DDR [FPGA](#)-Board anzupassen und zu synthetisieren. Etwaige Fehler wurden behoben, welche auf Grund der OpenSource-Projekt durchaus vorkamen. Behandelt wurden in diesem Fall die OpenSource-SoftCores *lowRISC*, welcher auf der RISC-V Struktur basiert, sowie der *LEON3*-Prozessor, der auf der SPARC V8 Architektur aufgebaut ist. Des Weiteren wurde der, vom Hersteller des Boards angebotene, MicroBlaze konfiguriert und synthetisiert.

Neben der Hardware sollte auf jedem der Systeme das Betriebssystem Linux ausgeführt werden, um einen realistischen Vergleich herzustellen.

1.2 Struktur der Arbeit

Zur Beschreibung des Vorgehens, wird die Arbeit wie folgt gegliedert.

Zu Beginn der Arbeit werden im Kapitel 2 die theoretischen Grundlagen vermittelt. Als Ausgangspunkt wird im Kapitel 2.1 auf die Definition und den Unterschied zwischen SoftCore- beziehungsweise HardCore-Prozessoren eingegangen, gefolgt von einer näheren Erklärung der Memory Management Unit (MMU) in Kapitel 2.2. Die Kommunikation zwischen Peripheriegeräten und dem System findet über Schnittstellen statt. Die mit diesem Thema zusammenhängenden Unterkapitel befinden sich im Abschnitt(2.3). Dies beinhaltet die im Rahmen dieser Arbeit genutzte Schnittstelle Universal Asynchronous Receiver Transmitter (UART), welche zur Kommunikation mit dem Zielsystem benötigt wird(2.3.1). Ebenfalls sind die sogenannten Interrupts ein wichtiger Bestandteil dieser Systeme, näher beschrieben in Kapitel 2.3.2. Auf Seiten der Software wurde näher auf die Funktionsweise eines Compilers(2.4) eingegangen und in diesem Fall besonders auf die verschiedenen Modi des Compilers(2.4.1). Das Kapitel 2.5 erklärt die Funktionsweise eines Betriebssystems, wie hier anhand des Beispiels Linux. Hierbei gibt es drei Unterkapitel. Das Kapitel 2.5.1, welches die wesentlichen Eigenschaften und Funktionsweisen eines Treibers erläutert, gefolgt von dem Kapitel 2.5.2 in dem der Device Tree näher beschrieben wird, sowie dem Kapitel 2.5.3, welches das sogenannte Buildroot erklärt. Dieses beschreibt ein Tool, welches zu Erzeugung eines Betriebssystems genutzt werden kann.

Die praktische Implementierung in Kapitel 3 gliedert sich in drei Teile. Im ersten Teil geht es um die Implementierung des Xilinx MicroBlaze-Prozessor(Kapitel 3.1), sowie die Ausführung des erzeugten Linuxsystems(3.1.2). In Kapitel 3.2 wird Schritt für Schritt erklärt, wie das *lowRISC*-System zu konfigurieren ist. Des Weiteren wird in diesem Kapitel auf die Generierung des dazugehörigen Linux eingegangen. Ähnlich aufgebaut ist das Kapitel 3.3, in welchem erst die Hardware (3.3.1) des LEON3 und anschließend, in Kapitel 3.3.2, das passende Betriebssystem näher erläutert werden..

Im Kapitel 4 wird die Arbeit noch einmal zusammengefasst und ein kurzer Ausblick gegeben.

2 Grundlagen

2.1 Softcore-Prozessoren

Ein SoftCore (engl. „Software-Kern“) ist ein Prozessor, Mikrocontroller oder ein Signalprozessor, welcher als virtuelle Einheit in einem [FPGA](#) oder Application-Specific Integrated Circuit ([ASIC](#)) integriert wird. Dies bietet die Möglichkeit einem digitalen Schaltungsdesign jeden beliebigen Prozessor hinzuzufügen.

Umgesetzt wird dieser Vorgang im [FPGA](#) durch reine Anwenderlogik, welche dementsprechend konfiguriert werden muss.

Das typische Anwendungsgebiet eines SoftCores ist die Lösung von komplizierten Aufgaben, mit denen klassische „State Machines“ überfordert sind oder die Effektivität nicht mehr gegeben ist. Somit werden SoftCores oftmals nachträglich in bestehende Designs eingefügt, wenn deren Umfang sich extrem vergrößert hat.[8]

2.1.1 Vergleich SoftCores und HardCores

Vergleicht man den SoftCore- mit dem HardCore-Prozessor, lassen sich folgende Aussagen treffen:

Vorteile:

- Flexible Anwendung: Das [FPGA](#) kann bei Bedarf mit einem SoftCore versehen werden, jedoch wird nicht von vorneherein Platz für einen HardCore verschwendet, welcher dann letztendlich ungenutzt bleibt. Daraus ergeben sich deutliche Vorteile im Hinblick auf die Kosten.
- Konfigurierbarkeit: Beim SoftCore deutlich flexibler, da hier unter anderem die Größe der Datenpfade und die Anzahl der Zusatzmodule variiert werden kann.

Nachteile:

- SoftCores haben auf Grund ihrer Flexibilität einen deutlichen Geschwindigkeitsnachteil

2.1.2 Typen

Es gibt eine große Anzahl verschiedener SoftCores. Diese unterscheiden sich aber maßgeblich in der Größe der Datenpfade. Somit wird unterschieden zwischen 8-,16- und 32-Bit Systemen.[8]

8-Bit SoftCores

Name	Quellcode	Hersteller
LatticeMico8	Verilog und VHDL	Lattice
PicoBlaze	VHDL	Xilinx
Proteus	VHDL	LogicSolutions

Tabelle 2.1: 8-Bit SoftCores nach [8]

16-Bit SoftCores

Name	Quellcode	Hersteller
NEO430	VHDL	neo430@GitHub
OpenMSP430	Verilog	OpenCores
TG68	VHDL	OpenCores

Tabelle 2.2: 16-Bit SoftCores nach [8]

32-Bit SoftCores

Name	Quellcode	Hersteller
LEON	VHDL	Gaisler Research
MicroBlaze	Nein	Xilinx
OpenRISC	Verilog	OpenCores
NIOS II	Nein	Altera

Tabelle 2.3: 32-Bit SoftCores nach [8]

2.2 Memory Management Unit

Bei der **MMU** handelt es sich um einen Hilfsbaustein des Betriebssystems, welcher die Speicherverwaltung des Arbeitsspeichers beschleunigt. Umgesetzt wird diese Verwaltung mit Hilfe eines Verfahrens, welches virtuelle Adressen auf physikalische Adressen abbildet. Der gesamte Speicher wird in einzelne Speicherbereiche unterteilt und die **MMU** überwacht die Einhaltung dieser Bereiche und unterbricht gegebenenfalls Instruktionen, die auf einen verbotenen Bereich zugreifen wollen.[6]

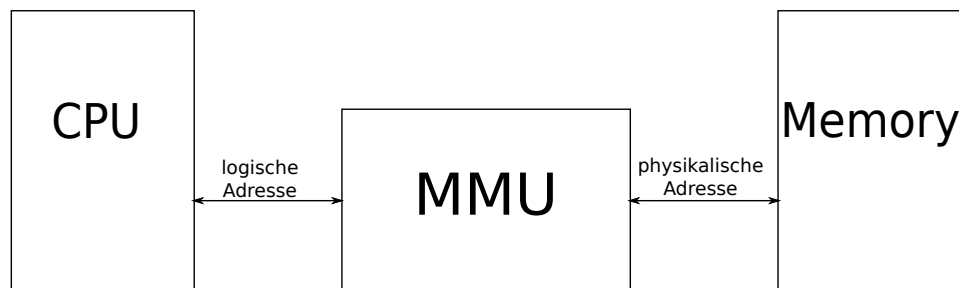


Abbildung 2.1: Blockschaltbild einer MMU

2.3 Peripherie

Ein Peripheriegerät kann einer Hardware hinzugefügt werden, um die Fähigkeiten zu erweitern. Diese Geräte sind optional und dienen zur Ein- und Ausgabe von Daten.

2.3.1 UART

Um mit Peripheriegeräten zu kommunizieren, kann **UART** als serielle Schnittstelle genutzt werden. Bei dieser Art der Kommunikation werden serielle Daten zwischen dem Board (*Master*) und dem Empfänger (*Slave*) ausgetauscht.

Dabei spielen die TxD und die RxD-Datenleitungen eine wichtige Rolle, da der Datenaustausch über diese Leitungen realisiert wird.

Im Gegensatz, zum Beispiel, zu einer Serial Peripheral Interface (**SPI**)-Schnittstelle, wird kein Clock-Signal übertragen um Daten zu validieren. Des Weiteren wird die Verbindung mit einer definierten Geschwindigkeit realisiert, der sogenannten *Baudrate*. [4]

Die Baudrate bezeichnet dabei die Anzahl der Bits, welche pro Sekunde übertragen werden. **UART** konvertiert die Bytes in serielle Bits, überträgt diese über eine einzelne

Leitung und liest die zugehörigen Start- und Stop-Bits aus.

Das sogenannte *Character*(Zeichen) besteht aus einer konfigurierbaren Anzahl an Datenbits (in den meisten Fällen 7 oder 8), aus einem *low-level* Start-Bit , einem optionalen Parity-Bit und einem oder mehreren logischen *high-level* Stop-Bits.

Das Start-Bit teilt dem Receiver mit, dass ein neues *Character* empfangen wird. Die nächsten Bits, je nachdem wie viele Daten-Bits vorher konfiguriert wurden , stellen dann den Inhalt des *Character* dar. Darauf folgt das optionale Parity-Bit, welches anzeigt, ob die Anzahl der mit '1' belegten Daten-Bits gerade oder ungerade ist. Am Ende der Folge stehen dann entweder ein oder zwei *high-level* Stop-Bits, welche dem Receiver eindeutig signalisieren, dass die Übertragung vollständig ist. Dadurch, dass das Start-Bit *high-level* (1) und das Stop-Bit *low-level* (0) ist, ist immer eine klare Abgrenzung zwischen dem derzeitigen und dem folgenden *Character* möglich.

Die Datenübertragung lässt sich nach 2.2 darstellen.

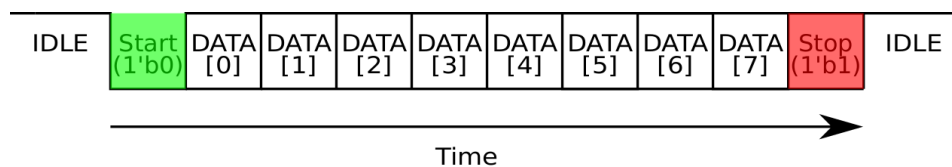


Abbildung 2.2: Datenübertragung per [UART\(8n1\)](#)

2.3.2 Interrupt

Neben der Central Processing Unit ([CPU](#)) und dem Datenspeicher haben die meisten Computersysteme Peripherie. Es handelt sich dabei um verbaute oder um an Schnittstellen angeschlossene Geräte. Damit die [CPU](#) die Nachricht erhält, dass Daten an solch einer Schnittstelle beziehungsweise Verbindung anliegen, muss es eine Möglichkeit geben den Prozessor zu unterbrechen. Hier gibt es die Art des sogenannten Polling, bei dem der Prozessor alle vorhandenen Eingabegeräte zyklisch abfragt. Ein effektiveres Verfahren ist die Unterbrechungsanforderung (Interrupt), welche eintritt, wenn Daten anliegen.

Sobald ein Gerät Daten zur Weiterverarbeitung besitzt, wird ein Interrupt-Request ([IRQ](#)) auf der dafür vorgesehenen Interrupt-Leitung gesendet. Daraufhin unterbricht der Prozessor seine Arbeit und verarbeitet die Daten des Gerätes.[13]

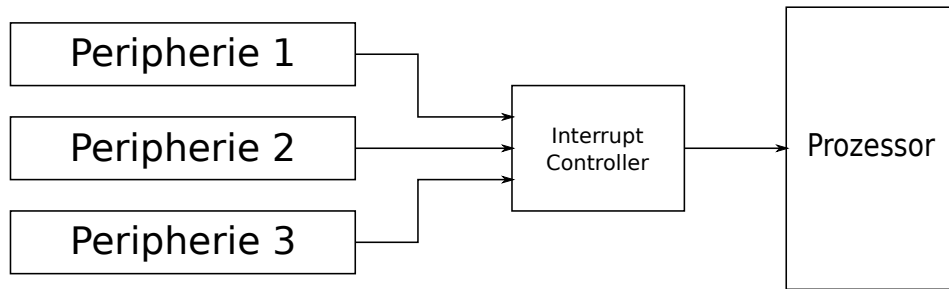


Abbildung 2.3: Interrupt-Controller

2.3.3 Timer

Ein weiterer essenzieller Teil zur Ausführung eines Betriebssystems, wie zum Beispiel Linux, ist der Timer. Dieser hilft dem Betriebssystem dabei, das so genannte Scheduling durchzuführen. Hierbei wird mit Hilfe von *Timer*-gesteuerten Interrupts dafür gesorgt, dass Nutzerprogramme, welche sich in der Ausführung befinden, unterbrochen werden und in den Betriebssystem-Modus geschaltet wird. Nun kann das Betriebssystem die anstehenden Prozesse neu planen.

2.4 Compiler

Der Compiler ist ein Werkzeug, welches einen Quelltext, der in einer höheren Programmiersprache (C++, C, Java) geschrieben wurde, in Maschinenbefehle umsetzt. Damit der Prozessor diese Instruktionen ausführen kann, werden die lesbaren Programmierbefehle übersetzt. Nach dem so genannten Kompilieren steht das Programm zur Ausführung bereit. Die Aufgaben des Compilers bestehen grundsätzlich aus drei Schritten:[\[12\]](#)

1. Lexikalische Analyse: Der Compiler zerlegt die Wörter und Zeichen des Quelltextes in verschiedene Klassen. Dabei werden überflüssige oder fehlende Zeichen als Fehler erkannt und vom Compiler, je nach Schwere des Verstoßes, als Warning oder Error ausgegeben.
2. Parsing: Es folgt die Prüfung des Codes auf syntaktische Korrektheit.
3. Semantische Analyse: Der Quelltext wird auf Sinnhaftigkeit geprüft. Zum Beispiel ob Befehle wirklich die korrekten Parameter haben.

Im Vergleich zum Interpreter gibt es folgende Vor- und Nachteile.

Vorteile:

- effiziente Übersetzung in ausführbaren Code
- Optimierung des generierten Codes

Nachteile:

- Kompilieren benötigt Zeit und Ressourcen
- Neu-Kompilierung nach Quelltextänderung
- Jede Programmiersprache benötigt einen eigenen Compiler

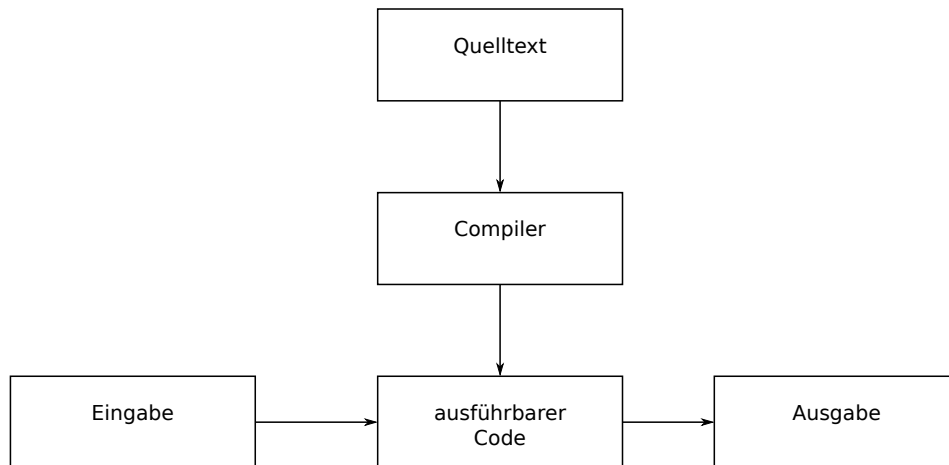


Abbildung 2.4: Funktionsweise des Compilers

2.4.1 Compiler-Modi

Der im Kapitel zuvor beschriebene Compiler kann, je nach Zielsystem, in verschiedenen Modi betrieben werden. Hier bei ist der ausschlaggebende Punkt, ob auf dem System ein Betriebssystem ausgeführt werden soll oder nicht.

- Non-OS mode: Programme ohne Zugriff auf Ein- bzw. Ausgabegeräte. Programme, welche in diesem Modus ausgeführt werden, haben keine periphere Unterstützung. Der Bare-Metal-Mode wird meistens genutzt, um Instruction Set Architecture (ISA)- und Cache-Regressionstests durchzuführen. Hierbei entscheidet der

Rückgabewert des Programmes über Erfolg beziehungsweise Misserfolg des Tests. 0 bedeutet Erfolg, Zahlen ungleich 0 geben den jeweiligen Fehlercode an. Bare-Metal-kompilierte Programme laufen auf einem [FPGA](#) und in einer Simulation im Hintergrund.

Um nun mit einem Bare-Metal-Programm eine Ausgabe zu erzeugen, müssen Bibliotheken verwendet werden, welche die Art und Weise der Ausgabe beinhalten.

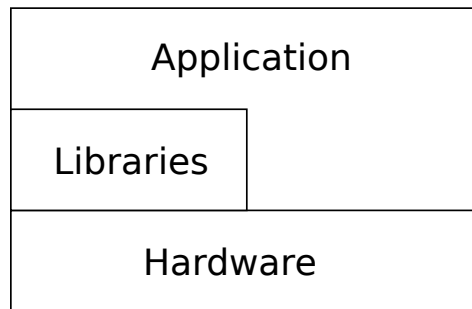


Abbildung 2.5: Grafische Darstellung der Non-OS-Architektur

- Linux mode: Benutzerprogramme mit Linux-Unterstützung. Mit Hilfe dieser Programme lassen sich Verhaltenssimulationen auf dem [FPGA](#)-Board durchführen. Diese Programme erhalten Multi-Thread und Peripherie-Unterstützung vom Linux-Kernel[15]

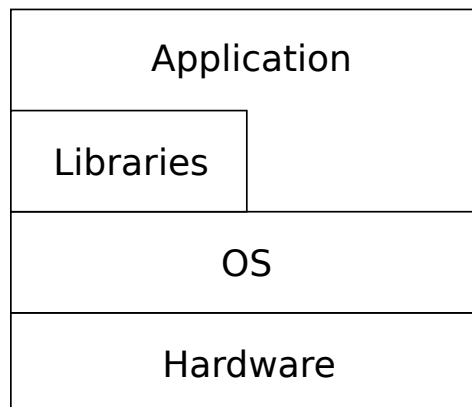


Abbildung 2.6: Grafische Darstellung der Linux-Architektur

2.5 Linux

Das verwendete System Linux wurde als freies Betriebssystem von Linus Benedict Torvald entwickelt. Es basiert auf dem Unix-Modell, jedoch schrieb Torvald den sogenannten Kernel neu. Seit dem gilt Linux als Open-Source-Projekt und Entwickler versuchen stetig das System zu verbessern.[\[17\]](#)

Die wohl wichtigste Eigenschaft die Linux bzw. Unix für den Einsatz beliebt macht, ist die Portabilität, da es weitestgehend rechnerunabhängig läuft. Weitere Eigenschaften die das Betriebssystem zu einem der weitverbreitetsten Systeme überhaupt gemacht haben, sind:

[\[17\]](#)

- **Multi-Tasking** Das parallele Nutzen verschiedener Programme erlaubt jedem Benutzer gleichzeitige Aktionen durchzuführen, ohne auf den Abschluss der letzten Tätigkeit zu warten.
- **Time-Sharing** Das Priorisieren von einzelnen Prozessen erlaubt es, dass mehrere Prozesse gleichzeitig ablaufen, indem abwechselnd Platz im Hauptspeicher beziehungsweise in der [CPU](#) zugewiesen wird.
- **Sprachenvielfalt** Neben der Sprache C stellt Linux/Unix viele Programmiersprachen wie C++, Java, Python und viele mehr zur Verfügung. Dadurch, dass es sich um ein Open-Source-Projekt handelt, wird diese Bibliothek, je nach System, ständig erweitert.
- **Grafische Benutzeroberfläche** Linuxrechner, im speziellen die Anwendungen, nutzen in den meisten Fällen GNU Object Model Environment ([GNOME](#)) oder K Desktop Environment ([KDE](#)) als grafische Oberfläche. Durch die weitverbreitete Nutzung dieser Oberflächen wurden diese kontinuierlich weiterentwickelt und bieten so eine hohe Vielfalt an Anpassungsmöglichkeiten.
- **Netzwerk** Für die Kommunikation zwischen Server und Client hat sich das Betriebssystem bewährt, da es über ein umfangreiches Paket an Software verfügt. Neben den Standardprotokollen Transmission Control Protocol/Internet Protocol ([TCP/IP](#)) (IPv6 wird ebenfalls unterstützt), werden weitere Formate verwendet, wie zum Beispiel Unix to Unix Copy ([UUCP](#)), welches eine simple Form der Kommunikation zwischen Linux/Unix-Rechnern darstellt.

2 Grundlagen

Grundsätzliche bestehen Betriebssysteme wie Linux/Unix aus drei Hauptkomponenten. Der **Kern** (engl. *Kernel*) bildet die grundlegenden Funktionen, wie zum Beispiel die Organisation und Verwaltung von Speicher, der Prozesse, sowie Ein- und Ausgänge und sämtliche Kommunikationsaufgaben.

Das **Dateisystem** (engl. *File System*) ermöglicht das Speichern von Dateien und errichtet einen Dateibaum und ist somit für die Datenorganisation zuständig.

Die dritte Komponente stellt der **Befehlsübersetzer** dar (engl. *shell*), welcher anhand einer Befehlssprache die Kommunikation ermöglicht und dem Benutzer erlaubt mit sämtlichen Peripheriegeräten zu interagieren, ohne dass die im Hintergrund laufenden Prozesse berücksichtigt werden müssen. [9]

Grafisch lässt sich der Aufbau wie folgt darstellen:

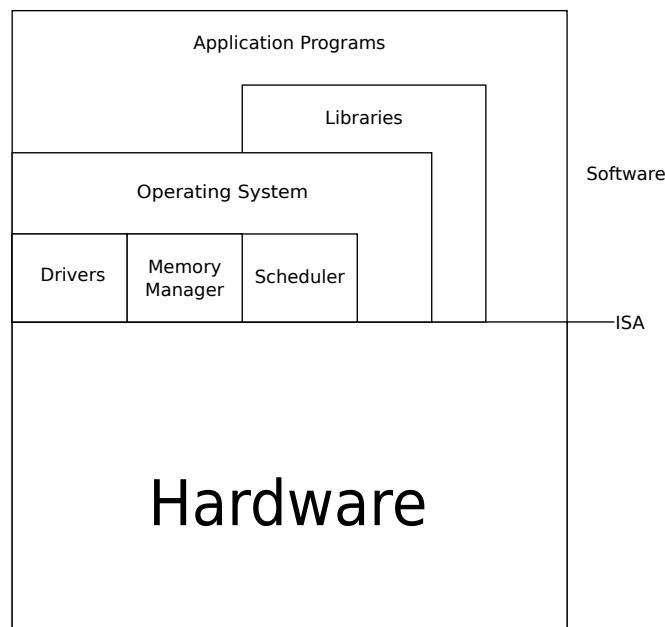


Abbildung 2.7: Computer System Architektur nach [14]

Um nun mit den verschiedenen Geräten über eine Vielzahl von Schnittstellen kommunizieren zu können, benötigt jedes Gerät einen eigenen Treiber. Dieser Treiber ist im Betriebssystem als „*Wissen*“ hinterlegt und beinhaltet Information über das Gerät und dessen Zugriffsmöglichkeiten und wird im folgenden Unterkapitel näher erklärt. [10]

Der Anwender hat nun die Möglichkeit über die Treiber auf die Schnittstellen zuzugreifen, in dem diese vom Treiber gesteuert werden.

2.5.1 Treiber

Der Treiber gilt als „*Lexikon*“ der jeweiligen Geräte für das Betriebssystem. Diese Aufgabe ist enorm wichtig für ein funktionierendes System, sodass der Stellenwert eines Treibers sehr hoch ist. Einfach erklärt, besteht ein Treiber aus einer Reihe von Funktionen, die den Zugriff auf das Gerät steuern und so eine Kommunikation ermöglichen. So muss für jedes Gerät ein eigener Treiber implementiert werden, der die offene Schnittstelle im Betriebssystemkern füllt.

Dabei sind zwei Begrifflichkeiten wichtig, in die der Speicherbereich aufgeteilt wird:

- **Kernelspace** : Dieser Bereich wird ausschließlich vom Kernel benutzt
- **Userspace** : Dieser Speicher wird von Applikationen genutzt und kann nicht, oder in Ausnahmefällen, nur eingeschränkt vom Kernel genutzt werden

Die Treiber werden grundsätzlich mit in den Kernel kompiliert oder sind als Modul dynamisch angelegt, um zum Kernel, während der Laufzeit, hinzugefügt zu werden. [16]

Bei der Implementierung von Gerätetreibern wird zwischen drei Gerätetypen unterschieden ([16]):

1. **Character-Devices** Auf diese Geräte kann wie auf einen Stream von Bytes zugegriffen werden, indem Portable Operating System Interface (**POSIX**)-Funktionen genutzt werden, wie beispielsweise *open()*, *write()*, *read()* oder *close()*. Für die Nutzung wird für jedes Gerät eine *Gerätedatei* (vgl. *device nodes*) angelegt.
2. **Block Devices** Im Gegensatz zu den *Character-Devices*, wird auf die *Block-Devices* immer per Random Access zugegriffen. Die Daten werden blockweise gelesen. Diese *Block-Devices* können ein Dateisystem aufnehmen und gemountet werden.
3. **Netzwerkschnittstellen** Netzwerkgeräte erhalten keine Gerätedatei, sondern sind in einer globalen Liste hinterlegt. Der Datenaustausch findet immer asynchron statt, sodass die Netzwerkkarte jederzeit bereit sein muss Daten zu empfangen.

2.5.2 Device Tree

Der *Device-Tree* beinhaltet Informationen, welche dem Kernel bei dem Bootvorgang helfen. Da das System angeschlossene Geräte nicht automatisch erkennen und die dazu

passenden Treiber laden kann, geschieht das in dieser *.dts*-Datei. Durch das Kompilieren wird daraus der sogenannte *Device Tree Blob*. Dieser wird zusammen mit dem Kernel vom Bootloader geladen und im Hauptspeicher abgelegt. Im Anschluss daran wird daraus eine Baumstruktur. In dieser sind die Geräte als Knoten angelegt und es werden Informationen zu den dazugehörigen Treiber hinterlegt, welche dem Betriebssystem zur Verfügung gestellt werden. Die Verbindung zwischen Device-Tree und dem Treiber wird über eine eindeutige Identifikation per *Compatible* festgelegt.

2.5.3 Buildroot

Buildroot ist ein Tool, welches das Erstellen eines kompletten Linux-Systems per *Cross-Compiling* für eingebettete Systeme vereinfacht und automatisiert. Neben dem *cross-compiled* Toolchain können ebenfalls das *Root-Dateisystem*, ein *Linux-Kernel-Image* und ein *Bootloader* für das Zielsystem erstellt werden. In einem Konfigurationsprogramm lassen sich vor der Erstellung sämtliche Optionen und Erweiterungen ab- beziehungsweise anwählen.

Das Tool wird größtenteils dafür genutzt, um die Systeme für andere Prozessoren, außer die klassischen x86-Prozessoren, zu erstellen, wie zum Beispiel Advanced RISC Machines ([ARM](#))-Prozessoren.[3]

Toolchain

Bei der *Toolchain* handelt es sich um eine Reihe von Tools zum Erstellen und Debuggen von Code für eine Zielarchitektur. Des Weiteren beinhaltet sie einen Compiler, einen Linker, sowie verschiedene Bibliotheken.

Die Toolchain ist essenziell, um die drei weiteren Elemente eines eingebetteten Systems zu generieren: den Bootloader, den Kernel und das Root-Dateisystem.

Hierbei wird im wesentlichen zwischen zwei *Toolchain*-Arten differenziert:

1. Die interne Toolchain: Die *Toolchain* wird aus einer Quelle heraus vom *Buildroot* generiert
2. die externe Toolchain: Es wird eine bereits vorhandene *Toolchain* verwendet

Das Generieren einer eigenen *Toolchain* benötigt entsprechend mehr Zeit und verlängert den ganzen Kompiliervorgang, bietet jedoch deutlich mehr Konfigurationsmöglichkeiten im Vergleich zu bereits vorhandenen *Toolchains*.

Kernel

Der *Kernel*, welcher bereits in Kapitel 2.5 beschrieben wurde, kann ebenfalls durch *Buildroot* konfiguriert und erzeugt werden.

Dieser steuert alle Prozessor- und Speicherzugriffe, unterhält die wichtigsten Treiber und greift direkt auf die Hardware zu. Dadurch, dass er die unterste Schicht des Systems beschreibt, ist er die Basis in der Kommunikation zwischen Hard- und Software.

Zu seinen Aufgaben gehören neben der parallelen Verarbeitung verschiedener Aufgaben, auch die Einhaltung zeitkritischer Grenzen, sowie die Offenheit für unterschiedlichste Anwendungen und Erweiterungen. Im Falle des Linux-Systems gilt der Kernel als Vermittler im System, so ist die grafische Oberfläche komplett unabhängig vom Linux-Kernel.[5]

Root-Dateisystem

Der Kernel enthält keine Programme oder Dämonen, die das System an sich nutzbar machen. Hierfür ist das *Root-Dateisystem* zuständig.

Wichtige Inhalte sind:

- init-Dämon: Startet Dämonen, Systemeinstellungen sowie Login-Programme
- System-Konfiguration
- Device-Nodes
- Bibliotheken und Kernel-Module

Für die Zusammenstellung dieses Dateisystems kann *Buildroot* verwendet werden, wodurch *BusyBox* sowie weitere Programmpakete integriert werden.[7]

Bootloader

Der Bootloader, auch Boot Manager genannt, ist ein Programm, dessen Aufgabe es ist, das Betriebssystem in den Arbeitsspeicher zu laden. In diesem Fall ist er dafür verantwortlich, den Kernel mit den gewünschten Parametern zu laden und den Speicher zu initialisieren, bevor der eigentliche Boot-Prozess startet.[11]

BusyBox

BusyBox kombiniert winzige Versionen vieler gängiger UNIX-Dienstprogramme in einer einzigen kleinen ausführbaren Datei. Es bietet Ersatz für die meisten Dienstprogramme, die normalerweise in GNU fileutils, Shellutils zu finden sind. Die Dienstprogramme in BusyBox haben im Allgemeinen weniger Optionen als ihre voll ausgestatteten GNU-Gegenstücke, welche jedoch die erwartete Funktionalität bieten und vom Verhalten sehr ähnlich sind. Ebenfalls bietet die BusyBox eine ziemlich vollständige Umgebung für eingebettete Systeme.

BusyBox wurde größenoptimiert und für begrenzte Ressourcen geschrieben. Es ist auch extrem modular, sodass Befehle (oder Funktionen) zur Kompilierzeit ein- und ausgeschlossen werden können. Dies erleichtert die schnelle Anpassung der eingebetteten Systeme.[\[2\]](#)

3 Soft Cores

3.1 MicroBlaze basierte Systeme

Microblaze

Der MicroBlaze SoftCore von Xilinx ist eine 32-Bit RISC Harvard Architektur mit einem umfangreichen Befehlssatz und ist optimiert für eingebettete Anwendungen. Neben dieser Flexibilität besitzt er drei unterschiedliche Voreinstellungen:

1. Microcontroller: Geeignet für Baremetall-Code
 - 32-Bit Prozessorkern (in der Regel ohne [MMU](#))
 - Externer Speichercontroller
 - [SPI](#) Controller
 - I2C-Controller
 - [UART](#)
 - Interrupt-Controller
 - Timer
2. Echtzeitprozessor: Deterministische Echtzeitverarbeitung auf einem Real-Time Operating System ([RTOS](#))
 - Alle Mikrocontroller Blöcke
 - Instruktioncache
 - Memory Protection Unit ([MPU](#))
 - Datencache
 - DDR-Controller
3. Anwendungsprozessor: Embedded Linux-fähig
 - Alle Echtzeit-Prozessor-Blöcke

- 32-Bit-Prozessorkern
- MMU
- Ethernet-Controller

Ausgehend von diesen Presets lässt sich das Design mit Hilfe von spezifischen Prozessoroptionen anpassen. Diese Vielzahl an Anpassungen sind per Drag & Drop sehr einfach hinzuzufügen. Dazu zählen unter anderem UART-, Pulse Width Modulation (PWM)- und Direct Memory Access (DMA)-Bausteine, sowie serielle Schnittstellen. Dadurch wird sichergestellt, dass die spezifischen Anforderungen der Anwendung erfüllt werden.

Zusammengefasst besitzt der Microblaze folgende Schlüsselfähigkeiten:

- 32 mal 32-Bit-*General-Purpose*-Register
- 32-Bit-Befehlswort mit drei Operanden und zwei Adressierungsmodi
- 32-Bit-Adressbus, erweiterbar auf 64-Bit
- Optionale Floating-Point Unit (FPU)
- Advanced eXtensible Interface Bus (AXI)4-Interface
- Verschiedene Zustandsmodi (Sleep, Hibernate, Suspend Mode/Instructions)
- Unterstützt entweder Processor Local Bus (PLB)- oder AXI-Schnittstellen
- Big-Endian / Little-endian Unterstützung
- Optionale MMU, sowie separate, konfigurierbare Daten- und Instruktionscaches

3.1.1 Erstellen der Hardwarekonfiguration

Erstellen eines Projekts

Starten Sie Vivado und wählen Sie *New Project* aus. Hierbei wählt man *RTL Project* als Projekttyp, wobei der Haken bei *"Do not specify sources at this time"* gesetzt sein sollte.

Nun gilt es unter dem Tab *Boards* das korrekte Board auszuwählen, in diesem Fall das Nexys4 DDR. Anschließend klicken Sie auf *Finish* und das Projekt wird erstellt.

Erstellen des Block Designs

Im Flow Navigator auf der linken Seite wählt man nun *Create Block Design* aus und gibt diesem Design einen beliebigen Namen. Es öffnet sich das Block Design, bei dem der Tab *Board* zu wählen ist. Hier werden nun alle passenden Bausteine für das Nexys4 DDR-Board angezeigt.

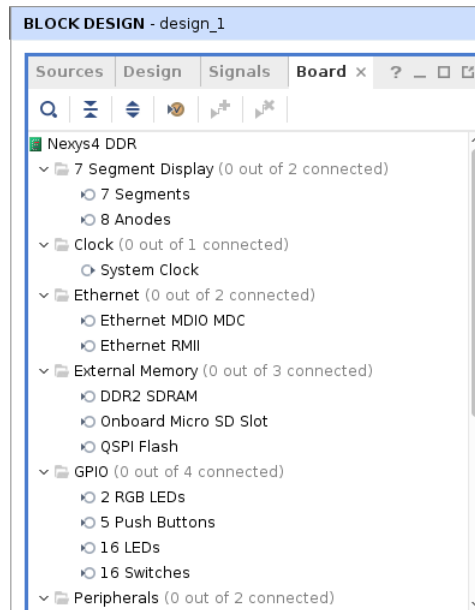


Abbildung 3.1: IP Integrator Block Design Diagram

Hinzufügen der Bausteine

Durch doppeltes Klicken auf die Bausteine werden zunächst die *System Clock*, *USB UART* und *DDR2 SDRAM* hinzugefügt. Die folgenden Fenster werden jeweils mit OK bestätigt. Im Design Diagram kann nun über das „+“-Symbol der Microblaze und der *AXI Timer* eingefügt werden.

Konfiguration des Clocking Wizard

Der *Clocking Wizard* muss wie im folgenden Bild konfiguriert werden.

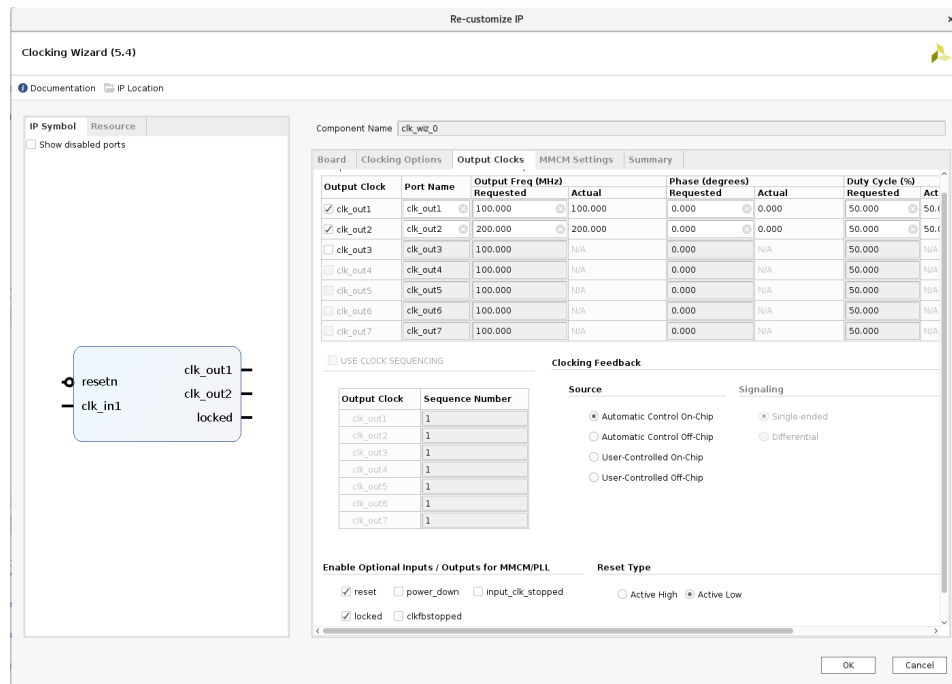


Abbildung 3.2: Einstellungen des Clocking Wizard

Dabei wird eine zweite Output Clock auf 200MHz gesetzt, sowie der Reset Type auf „Active Low“.

Anpassung der bestehenden Verbindungen

Die bestehende Verbindung zwischen der SysClock und dem MIG muss gelöscht werden und eine neue Verknüpfung zwischen dem Clocking Wizard und dem MIG erstellt werden.

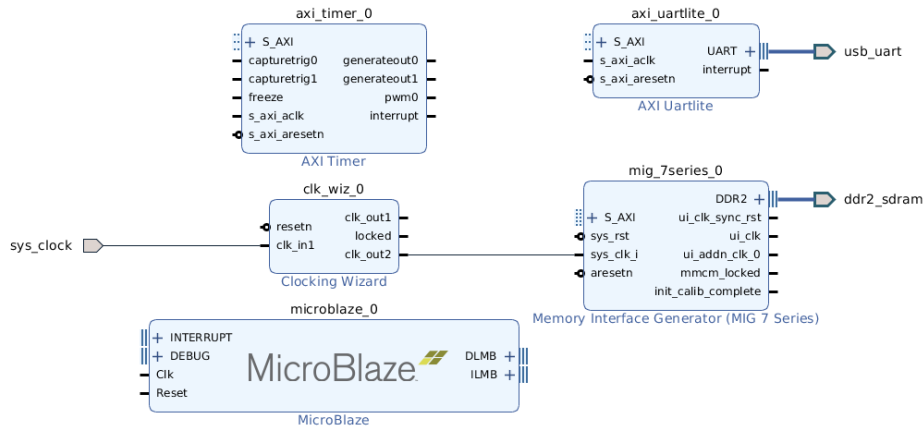


Abbildung 3.3: Neue Verbindung zwischen ClkWiz und MIG7

Block Automation

Ab hier greift die *Block Automation* ein, bei der die im nachfolgenden Bild angezeigten Einstellungen zu übernehmen sind. Besonders wichtig ist hier die Auswahl des Interrupt Controllers.

3 Soft Cores

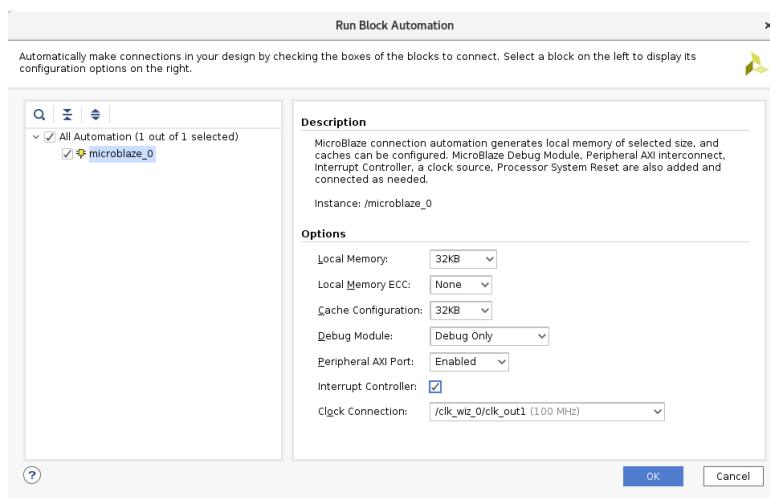


Abbildung 3.4: Einstellungen der Block Automation

Im Anschluss wird die *Connection Automation* gestartet und alle Bausteine ausgewählt. Durch die Schaltfläche *Regenerate Layout* kann das Diagramm neu angeordnet werden.

Manuelle Verbindungen

Der angelegte Interrupt Controller ist mit einem *Concat* verbunden, auf dessen Eingänge manuell die Interruptleitungen des *AXI Uartlite* und des *AXI Timers* gelegt werden. Die Verbindungen sind im folgenden Bild orange gekennzeichnet.

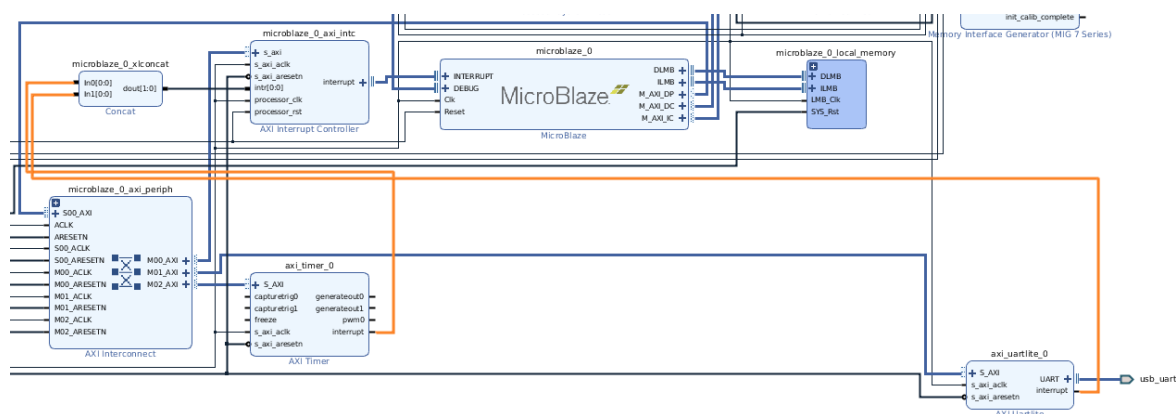


Abbildung 3.5: Manuell erzeugte Verbindungen der Interruptleitungen

Microblaze Konfiguration

Der sogenannte MicroBlaze Configuration Wizard ermöglicht es, bereits voreingestellte Konfigurationen zu wählen.

- Minimum Area: Minimaler MicroBlaze Core, kein Cache und kein Debug
- Maximum Performance: Maximale Leistung, großer Cache und Debug
- Maximum Frequency: Maximale Frequenz, kein Cache und kein Debug mit wenigen Ausführungseinheiten
- Linux with MMU: Hohe Leistung, Linux-optimiert, MMU aktiviert, großer Cache und Debug
- Low-end Linux with MMU: Für low-end-Systeme, MMU aktiviert, kleiner Cache und Debug
- Typical: Kompromiss zwischen Leistung, Frequenz und Größe
- Frequency Optimized: Bestmögliche Frequenz, MMU aktiviert

Der Microblaze ist wie folgt durch einen Doppelklick zu konfigurieren.

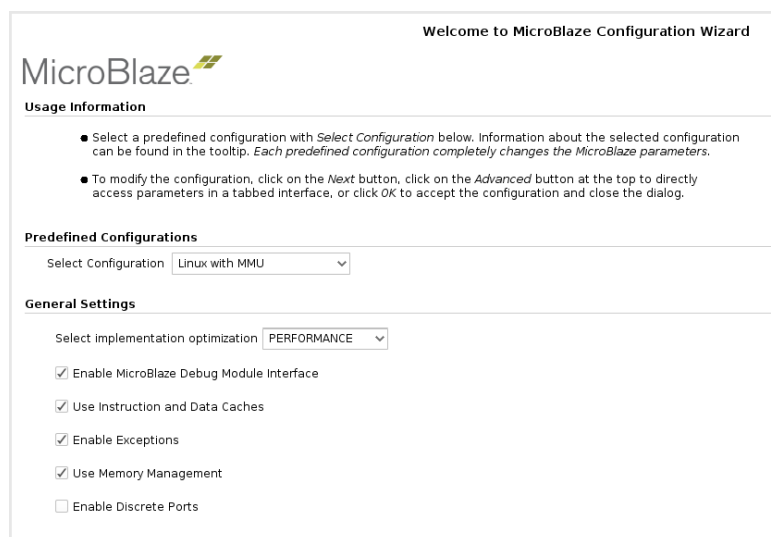


Abbildung 3.6: Konfiguration des MicroBlaze

Die neuen Einstellungen werden durch das Klicken des OK-Buttons übernommen.

Anpassung des AXI Uartlite-Bausteins

Der AXI Uartlite-Baustein muss ebenfalls angepasst werden. Hierbei gilt es im Tab „IP Configuration“ die Baudrate auf 115200 anzupassen. Diese Änderung ist mit OK zu bestätigen.

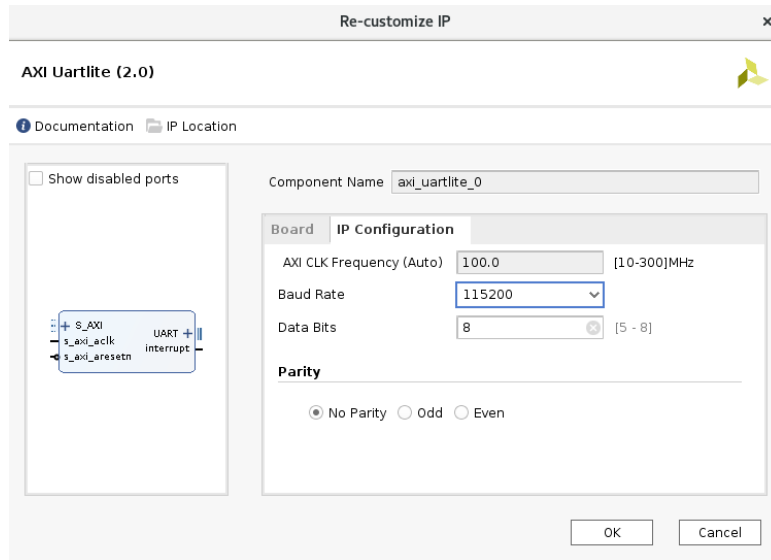


Abbildung 3.7: Änderung der Baudrate des AXI Uartlite

Erstellen des HDL Wrappers

Durch das Klicken der rechten Maustaste auf das Design im Source-Tab wird nun der HDL Wrapper erstellt.

Generierung des Bitfiles

Im nächsten Schritt wird das Bitfile generiert. Dafür ist im Flow Manager der Button „Generate Bitstream“ vorgesehen. Die „Number of jobs“ erhöht dabei die Geschwindigkeit je nach Prozessor und verfügbaren Kernen.

Im Anschluss wird über File -> Export -> Export Hardware das generierte Bitstream für die SDK, welche als nächstes gestartet wird, nutzbar gemacht.

Device Tree

Um den dazugehörigen Device Tree zu generieren, wird das Device-Tree Repository benötigt, welches aus dem Xilinx Github heruntergeladen werden kann: <https://github.com/Xilinx/device-tree-xlnx.git>

Nach dem Entpacken wird der Ordner nun im Software Development Kit (SDK) als Repository eingebunden. Dies geschieht über die Schaltfläche „Xilinx Tools“ beziehungsweise Repositories. Hier kann nun der Pfad des Ordners nur für ein bestimmtes Projekt oder für alle Projekte hinterlegt werden.

Erzeugen des Board Support Packages

Das Erstellen des Device-Tree geschieht über „File“ und anschließend „New“, sowie den Button „Board Support Package“. Dabei sollte nochmal die voreingestellten Parameter überprüft werden und als Board Support Package OS *devicetree* ausgewählt werden.

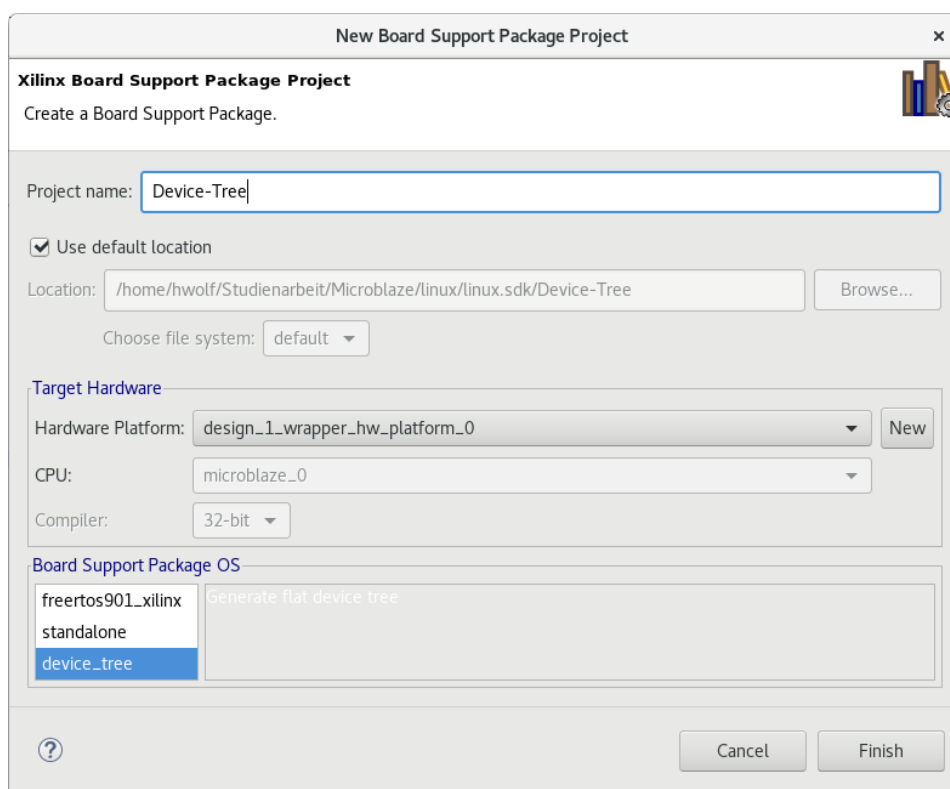


Abbildung 3.8: Erstellen eines Device-Tree als Board Support Package

Das folgenden Fenster kann ohne Weiteres mit OK bestätigt werden.

Zusammenführung der *Device Tree Source*-Dateien

Die entstandenen *.dtsi* beziehungsweise *.dts*-Dateien müssen zusammengeführt werden und in einer *.dts*-Datei gespeichert werden. Hierzu sind beide Dateien im Texteditor zu öffnen, um die *.dts* in die *.dtsi*-Datei zu kopieren. Die resultierende Datei wird, wie bereits erwähnt, als alleinstehende *.dts*-Datei gespeichert, um später in das buildroot-Tool eingefügt zu werden.

```

1 /dts-v1/;
2 / {
3     #address-cells = <1>;
4     #size-cells = <1>;
5     compatible = "xlnx,microblaze";
6     model = "Xilinx MicroBlaze";
7     chosen {
8         bootargs = "console=ttyUL0,115200";
9         linux,stdout-path = &axi_uartlite_0;
10        stdout-path = &axi_uartlite_0;
11    };
12    aliases {
13        serial0 = &axi_uartlite_0;
14    };
15    memory {
16        device_type = "memory";
17        reg = <0x80000000 0x8000000>;
18    };
19    cpus {
20        #address-cells = <1>;
21        #cpus = <1>;
22        #size-cells = <0>;
23        microblaze_0: cpu@0 {

```

Listing 3.1: Artix7.dts-Datei

Buildroot

Das bereits genannte Buildroot kann unter <https://buildroot.org/download.html> heruntergeladen werden. Das in dieser Arbeit verwendete Buildtool besitzt die Version 2018.02.

Neben dem Buildroot werden Linux Kernel *defconfig*-Dateien benötigt, welche für das Nexys4 angepasst wurden und auf folgenden Github zur Verfügung stehen :

<https://github.com/HenW/microblaze/tree/master>

Die Dateien werden in das Verzeichnis */board/nexys/microblaze* kopiert, welche vorher neu angelegt werden müssen. Es ist wichtig zu beachten, dass die Basisadresse des Kernels als 0x80000000 gesetzt ist und gegebenenfalls angepasst werden muss.

Die im vorherigen Schritt erstellte *.dts*-Datei muss ebenfalls in das erstellte Verzeichnis eingefügt werden.

Des weiteren muss die Buildroot *defconfig*-Datei „nexys_microblaze_defconfig“ im *configs*-Ordner abgelegt werden.

Einrichten der Konfigurationsdatei

Nachdem das System, wie in der vorherigen Schritt beschrieben, vorbereitet wurde, muss nun folgender Befehl mit Hilfe eines Terminals im Buildroot-Verzeichnis ausgeführt werden:

```
1 make nexys_microblaze_defconfig
```

Listing 3.2: Generierung der *defconf*-Datei

Dieser Befehl erzeugt eine sogenannte *.config* Datei für das zugehörige buildroot.

Erzeugen des Kernels

Um den Kernel und das passende Filesystem zu erstellen, wird der Befehl *make* benötigt.

```
1 make
```

Listing 3.3: Erstellung des Kernels

Dieser Vorgang benötigt je nach Rechenleistung einige Minuten. Nach erfolgreichem Abschluss des Vorgangs wird eine „simpleImage.artix7“-Datei erstellt, welche zum Booten des Linux auf dem Nexys4 DDR Artix7 **FPGA** Modul benötigt wird.

3.1.2 Ausführen des Linux Images auf dem Artix7

Das Linux Image muss jetzt in einen Ordner kopiert werden, welcher im Anschluss über die [SDK](#) Shell leicht zu erreichen ist. Das [FPGA](#)-Board wird nun mit dem Rechner über Universal Serial Bus ([USB](#)) verbunden und eingeschaltet. Über das Xilinx [SDK](#) kann nun eine Verbindung über den korrekten Port mit der entsprechenden Baudrate (115200) hergestellt werden.

Starten der Xilinx Microprocessor Debugger ([XMD](#))-Engine

Um die Verbindung zu prüfen, kann, nachdem das [FPGA](#)-Board konfiguriert wurde, ein Hello World Programm ausgeführt werden. Andernfalls kann direkt im SDK über Xilinx Tools, beziehungsweise „Launch Shell“ ein Terminal geöffnet werden.

Mit dem Befehl:

```
1 XMD
```

Listing 3.4: Öffnen des [XMD](#)

wird eine [XMD](#) Engine gestartet.

Anschließend wird über

```
1 connect mb mdm
```

Listing 3.5: Herstellen der Verbindung zum Microblaze Target

eine Verbindung zum Microblaze Target ([mb](#)) als Microprocessor Debug Module Target ([mdm](#)) hergestellt.

Nun gilt es das Kernel Image auf das [FPGA](#)-Modul zu laden. Hierzu muss innerhalb des Shells in den korrekten Ordner navigiert werden. Über die bestehende Verbindung kann nun mit Hilfe von

```
1 dow simpleImage.artix7
```

Listing 3.6: Download des Images

das Image heruntergeladen werden.

3.2 lowRISC basierte Systeme

Installation der benötigten Tools

Um die RISC-V Tools erstellen zu können, werden verschiedene Pakete benötigt, welche vorher zu installieren sind.

- autoconf: Produziert Shell Skripte zur automatischen Konfiguration von Software
- automake: Tool zur Generierung von Makefile-Dateien
- autotools-dev: Erneuert die Infrastruktur für config-Dateien
- libmpfr-dev: Mehrfach genaue Arithmetikbibliothek – Entwicklungswerkzeuge
- libmpfr-dev: siehe libmpfr-dev
- libgmp-dev: siehe libmpfr-dev
- gawk: GNU-Projekt Implementierung der AWK Programmiersprache
- build-essentials: Benötigte Pakete zur Kompilierung eines Pakets
- bison: YACC-kompatibler Parser-Generator
- flex: schneller lexikalischer-Analysator
- texinfo: hypertextfähiges Dokumentationssystem
- gperf: Hash-Funktionsgenerator
- libncurses5-dev: Ncurses-Bibliotheken für Entwickler
- libusb-1.0.-0: Programmbibliothek zum Schreiben und Lesen von USB-Geräten
- libboost-dev: Boost C++ Bibliotheken
- Git: Software zur Versionsverwaltung von Dateien

```

1 Sudo apt-get install autoconf automake autotools-dev curl \
2 Libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison \
3 Flex texinfo gperf libncurses5-dev libusb-1.0-0 libboost-dev \
4 Git

```

Listing 3.7: Installation der benötigten RISC-V-Tools

Download des lowRISC-Git-Repository

Hierbei ist es empfehlenswert, dass gesamte Repository zu klonen, statt einzelne Untermodule.

```

1 cd ~/lowRISC/DIR
2 #download des untether-v0.2-Branch
3 git clone -b untether-v0.2 --recursive https://github.com/lowrisc/lowrisc-
  chip.git
4 cd lowrisc-chip

```

Listing 3.8: Download des Repositories

Setzen der RISC-V-Variablen

Um die passende Umgebung zu schaffen, müssen die Variablen gesetzt werden. Hierfür ist zu erst der *TOP*-Pfad anzugeben:

```

1 export TOP=[Gesamter Pfad zu dem lowRISC-Verzeichnis]

```

Listing 3.9: Setzen der TOP-Variable

Bevor nun das Setupskript *lowrisc-chip/set_riscv_env.sh* ausgeführt wird, um die restlichen Variablen anzupassen, muss in diesem Skript das passende Board angegeben werden. Im Falle des verwendeten Boards Nexys4DDR wie folgt:

```

1 [...]
2 #choose the FPGA board (KC705 in default)
3 export FPGA_BOARD=nexys4

```

Listing 3.10: Anpassung des Zielsystems

Ausgeführt wird die angepasste Datei im Terminal mit:

```

1 source set_riscv_env.sh

```

Listing 3.11: Umgebungsvariablen

Es gilt zu beachten, dass das Einrichten dieser Umgebung jeweils nur für das genutzte Terminal gilt. Wird ein neues beziehungsweise ein weiteres Terminal geöffnet, müssen

vorherigen Schritte erneut ausgeführt werden.

Bauen des RISC-V Cross-Kompilierung-Tools

In dem Repository wurde ein Skript mitgeliefert, welches die meisten Tools für die Cross-Kompilierung und Spike vorbereitet. Dazu gilt es in den *\$TOP/riscv-tools*-Ordner zu navigieren und das Skript auszuführen.

```
1 ./build.sh
```

Listing 3.12: Ausführen des *build*-Skripts

Nach der erfolgreichen Kompilierung sollten der Cross-Compiler, sowie Spike zur Verfügung stehen. Geprüft werden kann dies mit:

```
1 which riscv64-unknown-elf-gcc
```

Listing 3.13: Überprüfung der Erreichbarkeit des Compilers

Erstellen des Linux-Cross-Compilers

Neben dem RISC-V-Cross-Compiler, welcher für das System an sich benutzt wird, wird ebenfalls ein Cross-Compiler für Programme, welche unter Linux ausgeführt werden sollen, benötigt.

Hierfür werden lediglich die Variablen innerhalb der RISC-V-Umgebung angepasst.

```
1 cd $TOP/riscv-tools/riscv-gnu-toolchain
2 #wenn der Ordner "build" bereits existiert, kann der nächste Schritt
   ignoriert werden
3 mkdir build
4 cd build
5 ../configure --prefix=$RISC_V
6 make -j(Anzahl Prozessorkerne) linux
```

Listing 3.14: Anpassung der RISC-V-Umgebung

Auch hier kann die Verfügbarkeit überprüft werden.

```
1 which riscv64-unknown-linux-gnu-gcc
```

Listing 3.15: Überprüfung der Erreichbarkeit des Linux-Compilers

Kompilieren des Linux Kernel

Für den Linux Kernel, welcher mit Hilfe von Spike simuliert oder auf dem FPGA-Board gebootet werden kann, muss ebenfalls die Umgebung angepasst werden.

Es werden verschiedene Dateien benötigt, die unter den folgenden Verlinkungen heruntergeladen werden können.

```
1 # Setzen der RISC-V Umgebungsvariablen
2 cd $TOP/riscv-tools
3 curl https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.14.41.tar.xz \
4 | tar -xJ
5 cd linux-3.14.41
6 git init
7 git remote add origin https://github.com/lowrisc/riscv-linux.git
8 git fetch
9 git checkout -f -t origin/untether-v0.2
10 make ARCH=riscv defconfig
11 make ARCH=riscv -j vmlinux
```

Listing 3.16: Download und Anpassung des Kernels

Nach der Kompilierung sollte das Linux Kernel Image erreichbar sein.

```
1 ls -l vmlinux
```

Listing 3.17: Erreichbarkeit des Kernel-Images

BusyBox

BusyBox wird im Root-Image genutzt um die grundlegende Shell-Umgebung bereitzustellen. Um ein eigenes Image zu erstellen, muss die BusyBox-Binärdatei vorher generiert werden.

```

1 # Setzen der RISC-V Umgebungsvariablen
2 cd $TOP/riscv-tools
3 curl -L http://busybox-net/downloads/busybox-1.21.1.tar.bz2 | tar -xJ
4 cd busybox-1.21.1
5 cp $TOP/riscv-tools/busybox_config .config
6 make -j (Anzahl Prozessorkerne)

```

Listing 3.18: Download der BusyBox

Nachdem die Kompilierung erfolgreich abgeschlossen, wurde die Binärdatei im selben Verzeichnis generiert.

```

1 ls -l busybox

```

Listing 3.19: BusyBox

Mit Hilfe des *make_root*-Skripts kann das Root-Image (root.bin) erstellt werden:

```

1 $TOP/riscv-tools/make_root.sh

```

Listing 3.20: Generieren des Root-Image

FPGA-Demo

Um nun das FPGA-Projekt zu kompilieren wird die Bitstream-Datei mit Hilfe eines Skripts erstellt.

```

1 cd $TOP/fpga/board/nexys4
2 make bitstream

```

Listing 3.21: Erzeugen des Bitfiles

Die Bitstream-Datei ist anschließend unter dem Pfad *lowrisc-chip-imp/lowrisc-chip-imp.runs/impl_1/chip_top.bit* zu finden.

Kompilieren und Ausführen von Bare-Metal-Programmen

In dem *lowrisc*-Verzeichnis sind bereits vier Bare-Metal Beispiele hinterlegt. Mit Hilfe einfacher Skripte können diese kompiliert werden und die neue Bitstream-Datei generiert werden. Für das klassische Hello-World Programm ist folgender Aufruf notwendig:

```
1 cd $TOP/fpga/board/nexys4
2 make hello
```

Listing 3.22: Erzeugen und kompilieren eines Beispielprogrammes

Anschließend befindet sich im Verzeichnis *lowrisc-chip-imp/lowrisc-chip-imp.runs/impl_1* eine *chip_top.new.bit*-Datei, welche auch zum Konfigurieren des FPGA genutzt werden sollte. Mit Hilfe von Vivado kann der FPGA mit der Bitstream-Datei konfiguriert werden.

Der *Minicom* wird genutzt, um die [UART](#)-Schnittstelle zu testen.

```
1 minicom -D /dev/(passende USB-Schnittstelle)
```

Listing 3.23: Aufruf des *Minicom*

Es sollte als Ausgabe Hello World erscheinen.

Vorbereiten des eigenen Images

Um RISC-V Linux zu starten, werden drei Dateien auf der im File Allocation Table ([FAT](#))-formatierten SD-Karte benötigt:

1. boot: Der Bootloader(Berkeley bootloader) um den Linux Kernel zu laden
2. vmlinux: Der RISC-V Linux Kernel
3. root.bin: Das ramdisk-Image

Der Linux Kernel und das Image können über das Terminal auf die SD-Karte kopiert werden.

```
1 cd $TOP/riscv-tools
2 cp linux-3.14.41/vmlinux /(Pfad der SD-Karte)/vmlinux
3 cp busybox-1.21.1/root.bin /(Pfad der SD-Karte)/root.bin
```


Der fehlende Bootloader kann wie folgt generiert werden:

```
1 cd $TOP/fpga/board/nexys4
2 make bbl
3 cp bbl/bbl /(Pfad der SD-Karte)/boot
```

Nachdem das Board, sowie die SD-Karte programmiert wurden, kann die SD-Karte in das Board gesteckt werden und mit Hilfe des *CPU Reset* ein Neustart der CPU erreicht werden. Nun sollte auf dem *Minicom* das lowRISC boot program zu sehen sein und Linux starten.

3.3 LEON3 basierte Systeme

Der LEON3 ist das Very High Speed Integrated Circuit Hardware Description Language ([VHDL](#))-Modell eines 32-Bit-Prozessors, welcher auf der SPARC V8-Architektur basiert. Das Modell ist in großem Umfang konfigurierbar und besonders für System-on-Chip ([SoC](#))-Designs geeignet. Ursprünglich wurde dieses System von der European Space Agency ([ESA](#)) entworfen, wird heutzutage jedoch von *Aeroflex Gaisler* weiterentwickelt und vertrieben.

Der LEON3-Prozessor verfügt über folgende Eigenschaften:[\[1\]](#)

- SPARC V8 Befehlssatz mit V8e Erweiterungen
- Fortgeschrittene 7-stufige Pipeline
- Hardware multiplizieren, teilen und Multiply-Accumulate ([MAC](#))-Einheiten
- Hochperformante IEEE-754-[FPU](#) mit vollem Pipelinespektrum
- Getrennter Befehls- und Datencache (Harvard-Architektur) mit Snooping
- Konfigurierbare Caches: 1 - 4 Wege, 1 - 256 kbytes / Weg. Random, LRR oder Last Recently Used ([LRU](#)) Ersatz
- Lokaler Befehls- und Daten-Scratchpad-RAM, 1 - 512 KByte
- SPARC Reference Memory Management Unit ([SRMMU](#)) mit konfigurierbarem Translation Lookaside Buffer ([TLB](#))
- Advanced Microcontroller Bus Architecture ([AMBA](#))-2.0 Advanced High-performance Bus ([AHB](#))-Busschnittstelle
- Erweiterte On-Chip-Debug-Unterstützung mit Befehls- und Daten-Trace-Puffer
- Symmetrische Multiprozessorunterstützung (Symmetric Multi-Processing ([SMP](#)))
- Power-Down-Modus und Clock-Gating
- Robustes und vollsynchrones Single-Edge-Clock-Design
- Bis zu 125 MHz in FPGA und 400 MHz in 0,13 um [ASIC](#)-Technologien
- Fehlertolerante und SEU-geschützte Version für Raumfahrtanwendungen

- umfangreich konfigurierbar
- Große Auswahl an Software-Tools: Compiler, Kernel, Simulatoren und Debug-Monitore
- Hohe Leistung: 1,4 DMIPS / MHz, 1,8 CoreMark / MHz (gcc -4.1.2)

3.3.1 LEON3 Hardware

Um diesen SoftCore zu erstellen, wird der LEON3/GRLIB Source Code benötigt welcher auf der Seite <https://www.gaisler.com/index.php/downloads> heruntergeladen werden kann. Diese Datei kann in einen beliebigen Ordner entpackt werden.

Für das hier verwendete Board Nexys4 DDR muss nun in einem Terminal in den Ordner:

```
1 ~/glib-gpl-2017.3-b4208/designs/leon3-digilent-nexys4ddr
```

Listing 3.24: Pfad zu dem passenden Design

navigiert werden.

Mit Hilfe des folgenden Aufruf erscheint eine grafische Oberfläche, welche alle Parameter des Designs enthält.

```
1 make xconfig
```

Listing 3.25: Aufruf des Konfigurationsmenüs

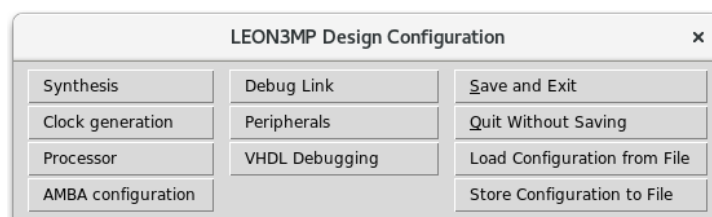


Abbildung 3.9: Grafische Oberfläche zur Konfiguration der Hardware

Hier lassen sich nun sämtliche Einstellungen wie gewünscht verändern und anpassen.

Zuerst muss der Prozessor angepasst werden. Hier ist es wichtig, dass bei dem Drop-Down Menü „Force values from example configuration“ die Option

„General-purpose-cfg“ ausgewählt wird. Das Fenster kann geschlossen und die Oberfläche nach dem Speichern beendet werden, um das Menü anschließend erneut aufzurufen. Der gleiche Ablauf ist zu wiederholen, lediglich ist nun die Option „Custom-configuration“ auszuwählen, um weitere Einstellungen vorzunehmen.

Durch die vorherigen Schritte wurde eine „Allzweck-Konfiguration“ vorgenommen, welche im Detail in der Tabelle 3.1 beschrieben ist.

Einheit	Wert	Beschreibung
dsu	1	Unterstützung der LEON3-DSU
fpu	-	Beinhaltet die FPU. Diese ist für die Großzahl an Systemen dringend empfohlen. Hierbei kann zwischen der GRFPU und der GRFPU-Lite ausgewählt werden. Diese unterscheiden sich in Geschwindigkeit und benötigter Flächenanforderung.
V8	2	Unterstützung für SPARC V8 MUL/DIV Instruktionen mit einer 5-Zyklen-Multiplikation. Bei geeigneter Zieltechnologie, benötigt eine Single-Zyklen-Multiplikation weniger Platz und bietet mehr Leistung
mac	0	Beinhaltet keine Unterstützung für SPARC V8e SMAC/UMAC
nwp	2	Fügt zwei Hardware-Watchpoints hinzu
icen/dcen	1	Prozessor-Cache
isets/dsets	2	Zwei-Wege Instruktions- und Datencache
irepl/drepl	2	Zufällige Ersetzung für Befehl- und Datencache oder möglicher LRU Algorithmus
isetsize/dsetsize	-	Cachegröße, je nach Umgebung
dsnoop	6	Aktivieren des Snooping mit zusätzlichen physischen Tags
mmuen	1	Aktivieren der MMU
itlbnun/dtlbnun	8	Jeweils acht Einträge für Look-a-Side-Puffer für Befehls- und Datentranslation der MMU
tlb_type	2	Nutzung von zwei separaten Look-a-Side Puffern mit schnellem Schreiben für Daten und Befehle
tlb_rep	0	Nutzen von LRU für die TLB
lddel	1	Ein-Zyklen Ladeverzögerung
tbuf	4	Nutzen von 4 KiB-Trace-Befehlpuffer
pwd	2	Timing-effiziente Power-Down Implementierung
smp	0	Deaktivierung der SMP-Unterstützung
bp	1	Aktivierung Verzweigungsprognose

Tabelle 3.1: General-Purpose-Configuration

Anpassung der UART-FIFO-Größe

Um mit dem **FPGA**-Board kommunizieren zu können, müssen die *Sender*- beziehungsweise *Empfänger*-First In-First Out (**FIFO**) angepasst werden. Diese können Werte von 1 bis 32 annehmen. Der Wert 1 sollte für einen platzsparendes System verwendet werden, in diesem Fall ist jedoch die 32 für eine maximale Übertragung empfehlenswert. Angepasst werden kann der Wert im *xconfig*-Menü unter dem Untermenü *Peripherals* beziehungsweise „*UARTS, timers and irq control*“.

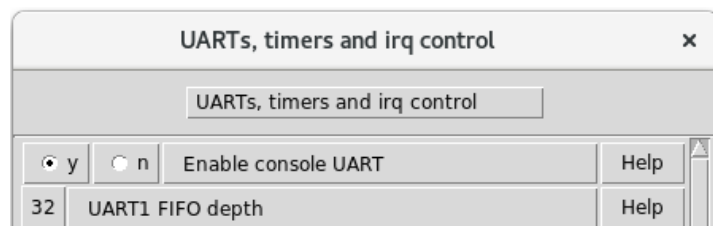


Abbildung 3.10: Anpassung der **FIFO**-Größe

Die getätigten Einstellungen ermöglichen es nun, die Synthese anzustoßen.

```
1 make vivado
```

Listing 3.26: Aufruf des Vivado-Programm

Das erstellte Bitfile wird anschließend mittels des Hardwaremanagers von Vivado auf das Board geladen.

Eigenschaften der Hardware

- Die Standardkonfiguration legt die Systemfrequenz auf 70MHz fest.
- Der verwendete DDR2SPA-Controller kann mit bis zu 150MHz getaktet werden. Die Standardkonfiguration legt diese Frequenz bei 140MHz fest. Um diesen DDR2SPA-Controller zu verwenden, muss dieser im *xconfig*-Menü aktiviert werden und die Parameter mit folgenden Werten versehen werden:

Enable power-on initialization:	Yes
DDR clock is aligned to system clock:	No
Memory Frequency:	140MHz
tRFC:	130ns
Column address bits:	10
Chip select bank size:	128 Mbyte
Data width:	16

- Wenn der Memory Interface Generator ([MIG](#))-Controller aktiviert ist, läuft dieser mit einer Frequenz von 70 MHz und der DDR2-Speicher mit 280MHz.
- Die [AHB](#)-Clock wird vom Mixed-Mode Clock Manager ([MMCM](#))-Modul in dem [MIG](#)-Controller generiert und kann mit Hilfe des Xilinx CORE Generator ([COREGEN](#)) gesteuert werden. Wenn der [MIG](#)-Controller nicht zur Verfügung steht, wird die [AHB](#)-Clock von dem CLKGEN erstellt und via *xconfig* kontrolliert.
- LED-Statusanzeige
 - LED 0/1: zeigen [USB/UART](#) RX/TX Aktivitäten an.
 - LED 2: [DSU](#) Debug-Modus
 - LED 3: zeigt den Prozessor im Fehlermodus an
 - LED 4: stellt das Ende des Kalibrierungsvorganges für den Speichercontroller dar

Die [DSU](#)-Schnittstelle ist aktiviert und über den [USB/UART](#)-Port zugänglich. Für die Ausgabe wird empfohlen das Debugging-Tool von Gaisler zu benutzen, welches den Namen *GRMON* trägt und auf der Herstellerseite kostenfrei heruntergeladen werden kann. Dieses Tool bietet verschiedene Aufrufparameter, welche in dem Manual näher beschrieben sind.

GRMON

In diesem Fall wird der *GRMON* mit folgendem Aufruf gestartet:

```
1 ./grmon -baud 460800 -uart /dev/ttyUSB1 -u -nb
```

Listing 3.27: Aufruf des GRMON

Erklärung der einzelnen Parameter:

- baud *baudrate*: Nutzen der angegebenen Baudrate. Die Standardeinstellung liegt bei 115200 baud. Höhere Baudraten benötigen eine entsprechende Unterstützung des Zielsystems.
- uart *device*: Standardmäßig nutzt der GRMON den ersten [UART](#)-Port des Hostrechners. Diese Einstellung kann mit Hilfe der speziellen Angabe überschrieben werden und so sichergestellt werden, dass mit dem gewünschten Gerät kommuniziert wird.
- u [*device*]: Dieser Befehl versetzt den UART1-Port in den FIFO-Debug Modus, wenn die Hardware es unterstützt. Ist dies nicht gegeben, wird der Loop-back-Modus aktiviert. Dieser Modus ermöglicht es via GRMON über den [UART](#) zu kommunizieren.
- nb: Das System soll bei sogenannten „Error Traps“ nicht anhalten. Dieser Punkt ist besonders wichtig für die Ausführung von Linux.

Nach dem Start können durch den Befehl

```
1 info sys
```

Listing 3.28: Abruf von Systeminformationen

weitere Informationen zum System ausgegeben werden. Hierbei fällt auf, dass kein SDRAM gefunden wurde.

```
ddr2spa0 Cobham Gaisler Single-port DDR2 controller
AHB: 40000000 - 48000000
AHB: FFF00100 - FFF00200
No SDRAM found
```

Abbildung 3.11: Fehlender SDRAM

Um diesen Fehler zu beheben muss eine Kalibrierung durchgeführt werden.

```
1 ddr2delay scan
```

Listing 3.29: Starten der Routine

Dieses Kommando führt eine Kalibrierungsroutine aus, die alle *tap*- und *read*-Verzögerungen durchsucht, um so eine funktionierende Einstellungen herzustellen.

Der *GRMON* ist anschließend mit dem Befehl *quit* zu beenden und erneut zu starten. Wiederholt man nun die Ausgabe der Systeminformationen, sollte SDRAM vorhanden sein.

```
ddr2spa0 Cobham Gaisler Single-port DDR2 controller
AHB: 40000000 - 48000000
AHB: FFF00100 - FFF00200
16-bit DDR2 : 1 * 128 MB @ 0x40000000, 8 internal banks
140 MHz, col 10, ref 7.8 us, trfc 135 ns
```

Abbildung 3.12: Konfigurierter SDRAM

3.3.2 Linux für LEON3

Das passende Linux Image für das LEON3-System wird am einfachsten mit LINUXBUILD erzeugt. Hierfür werden zwei essentielle Programme benötigt:

1. Linux Toolchain: Eine „Werkzeugkette“, welche für die Programmierung von Anwendungen und Betriebssystemen eingesetzt wird
2. LINUXBUILD: Zu Konfiguration des Kernels

Linux Toolchain

Bevor LINUXBUILD konfiguriert und genutzt werden kann, muss die SPARC/LEON Linux Toolchain installiert werden. Diese kann direkt von der Seite <http://gaisler.com/anonftp/linux/linux-2.6/toolchains/> heruntergeladen werden. Hierbei sollte die Kompatibilität der Versionen von Toolchain und Linuxkernel beachtet werden.

Die Toolchain muss in dem Ordner `/opt` installiert werden, da sonst die Umgebungsvariablen zu Fehlern führen können. Als Beispiel ist das Zielverzeichnis für die `sparc-gaisler-linux4.9` Toolchain der Ordner `/opt/sparc-gaisler-linux4.9`. Dieses Verzeichnis muss jetzt als Pfad angegeben werden.

```

1 $ export PATH=/opt/sparc-gaisler-linux4.9/bin:$PATH
2 $ which sparc-linux-gcc
3 /opt/sparc-gaisler-linux4.9/bin/sparc-linux-gcc

```

Listing 3.30: Anpassen der Umgebungsvariablen

LINUXBUILD

Das bereits angesprochen LINUXBUILD-Paket kann unter <https://www.gaisler.com/anonftp/linux/linux-2.6/linuxbuild/> heruntergeladen werden und an einem beliebigen Ort entpackt werden.

Nachdem mit Hilfe des Terminals in den entsprechenden Ordner navigiert wurde, kann durch verschiedene Aufrufe das Konfigurationsmenü aufgerufen werden.

- `make config`: Qt basierte GUI (benötigt Qt-3/4 Bibliotheken)
- `make gconfig`: Graphical User Interface (GUI) aufbauend auf Gimp Toolkit (GTK)
- `make menuconfig`: Ncurses basiertes Terminal Interface

Einrichten der Konfiguration

Der Menüpunkt „Install Linux“ führt letztendlich zur Installation des Images. Das doppelte Klicken auf den Unterpunkt „Execute Linux installation of latest stable leon-linux“ wählt hierbei immer die letzte stabile Version aus, welche extra für das LEON3-System konfiguriert ist.

Das sich im Anschluss öffnende Pop-Up-Fenster kann mit „Yes“ bestätigt werden und die Installation wird gestartet. Es gilt zu beachten, dass die Installation nicht mit einem Fehler beendet wird und reibungslos durchläuft. Das Fenster kann mit der *Return*-Taste geschlossen werden.

Der Pfad, zu der im vorherigen Schritt installierten Toolchain, muss jetzt im Linuxbuild angegeben werden. Dies ist in dem Menüpunkt *Step 2:package configuration* und dem Unterpunkt *Toolchain configuration* möglich.

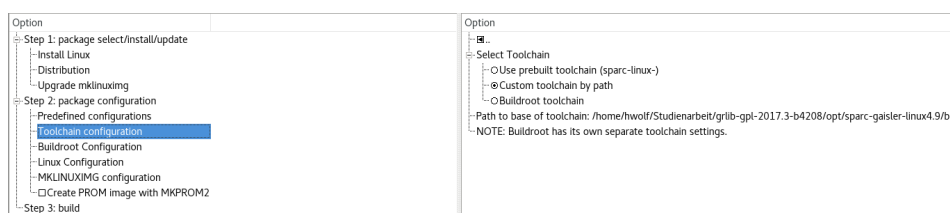


Abbildung 3.13: Angabe des Pfades zu der Toolchain

Der letzte Schritt ist das Erzeugen des Images. Unter dem Punkt „Step3: build“ kann die eigentliche Erzeugung über „Execute make build“ angestoßen werden. Auch hier kann das erscheinende Fenster mit „Yes“ bestätigt werden.

Wurde das Image erfolgreich erstellt, kann das Fenster sowie *xconfig* geschlossen werden und folgende Dateien befinden sich im Verzeichnis *~/output/images*.

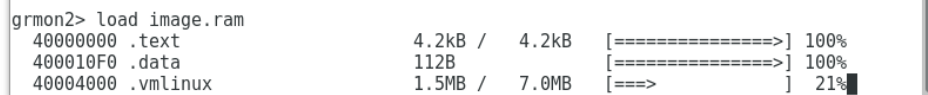
- image.ram: RAM Image (Zum Laden und Ausführen oder zum Nutzen des Bootloaders)
- image: Virtuelles Adressen Image mit Symbolen (Debugging)

Laden des Image

Das erstellte Image kann nun mit Hilfe des *GRMON* auf das Zielsystem geladen.

```
1 \load [Zielverzeichnis zum image.ram]
```

Listing 3.31: Laden des Image



```
grmon2> load image.ram
40000000 .text          4.2kB /  4.2kB  [=====>] 100%
400010F0 .data          112B   /   112B  [=====>] 100%
40004000 .vmlinux       1.5MB /  7.0MB  [====>] 21%
```

Abbildung 3.14: Laden des Image auf das Zielsystem

Nachdem der Ladevorgang erfolgreich abgeschlossen wurde, kann das Image mit Hilfe des folgenden Befehls ausgeführt werden.

```
1 run
```

Listing 3.32: Ausführen des Image

Das System bootet nun und weitere Informationen des Systems werden auf dem *grmon* dargestellt. Nach Abschluss des Bootvorgangs wird der Login-Bildschirm angezeigt, bei dem sich der Nutzer mit *root* anmelden kann.



```
Welcome to Buildroot
buildroot login: root
# cd ..
# ls
bin      init     linuxrc  opt      run      tmp
dev      lib      media    proc     sbin     usr
etc      lib32    mnt      root     sys      var
#
```

Abbildung 3.15: Linux für Leon3 in der Ausführung

4 Zusammenfassung und Ausblick

In dieser Arbeit wurden grundsätzliche Themen, welche mit der SoftCore-Entwicklung zusammenhängen in Form der Grundlagen erläutert und zum Teil, zum besseren Verständnis, grafisch dargestellt.

Des Weiteren wurden die drei genannten Systeme näher erläutert und es wurde Schritt für Schritt erklärt, wie diese Systeme angepasst werden müssen, um eine Ausführung auf dem Digilent Nexys4 DDR [FPGA](#)-Board zu ermöglichen. Ebenfalls wurde beschrieben wie ein, zu den einzelnen Systemen passendes, Betriebssystem konfiguriert werden muss. Die Eigenschaften der drei SoftCores wurden jeweils kurz erläutert um eine gewissen Vergleichbarkeit herzustellen. Diese Vergleichbarkeit ist es, die eine auf dieser Arbeit basierende Betrachtung interessant macht. Neben den drei Systemen gibt es an der Professur für Technische Informatik an der Helmut-Schmidt-Universität / Universität Hamburg der Bundeswehr ein weiteres System, das sogenannte Partial Reconfigurable Heterogeneous System ([PRHS](#)). Für dieses Framework steht ebenfalls ein Betriebssystem zur Verfügung, das Linux for Partial Reconfigurable Heterogeneous System ([L4PRHS](#)). So würde es sich durchaus anbieten, auf Grund dieser Arbeit und dem hier an der Universität vorhandenen Systems, einen Vergleich dieser Modell auf verschiedenen Ebenen durchzuführen. Mit Hilfe einer Benchmark könnte eine Analyse der Systeme, im Bezug auf Ergebnisse mit vorher festgelegtem Bezugswert, erstellt werden und in einer weiteren Arbeit festgehalten werden.

Literaturverzeichnis

- [1] AB, C. G.: *LEON3 Processor*, März 2018. <https://www.gaisler.com/index.php/products/processors/leon3>.
- [2] Andersen, E.: *BusyBox: The Swiss Army Knife of Embedded Linux*, Mai 2018. <https://busybox.net/about.html>.
- [3] Buildroot: *Buildroot - Making Embedded Linux Easy*. <https://buildroot.org>.
- [4] Cömert, E.: *PModCLS - UART Interface with VHDL*, 2011.
- [5] Geißler, O. und Ostler, U.: *Was ist ein Kernel?*, Jan. 2018. <https://www.datacenter-insider.de/was-ist-ein-kernel-a-679875/>.
- [6] ITWissen.info: *Speichermanager*, Okt. 2010. <https://www.itwissen.info/Speichermanager-memory-management-unit-MMU.html>.
- [7] Martina Hafner, A. K. und: *Ein Embedded-Echtzeit-Linux-System aufsetzen*, Mai 2012. <https://www.elektronikpraxis.vogel.de/ein-embedded-echtzeit-linux-a-360867/index3.html>.
- [8] Mikrocontroller.net: *FPGA Soft Core*. https://www.mikrocontroller.net/articles/FPGA_Soft_Core.
- [9] Noisefloor: *Ubuntuusers Wiki*, Mai 2016. <https://wiki.ubuntuusers.de/Dateisystem>.
- [10] Quade, J. und Kunst, E. K.: *Linux-Treiber entwickeln*. dpunkt.Verlag, 2015, ISBN 9783864902888.
- [11] Rouse, M.: *Bootloader*, Mai 2018. <https://www.searchdatacenter.de/definition/Bootloader>.
- [12] Schnabel, P.: *Compiler und Interpreter*. <https://www.elektronik-kompodium.de/sites/com/1705231.htm>.

- [13] Schnabel, P.: *Interrupt(IRQ)*. <https://www.elektronik-kompodium.de/sites/com/0610151.htm>.
- [14] Smith, J.E. und Nair, R.: *Virtual Machines*. Morgan Kaufmann, 2005, ISBN 9781558609105.
- [15] Song, W.: *Untethered lowRISC tutorial*, Dez. 2015. <http://www.lowrisc.org/docs/untether-v0.2/>.
- [16] Wietzke, J.: *Embedded Technologies*. Springer-Verlag, 2012, ISBN 9783642239960.
- [17] Wolfinger, C.: *Keine Angst vor Linux/Unix: Ein Lehrbuch für Linux- und Unix-Anwender*. Springer-Verlag, Berlin Heidelberg, 2013, ISBN 9783642320791.