Technische Hochschule Augsburg

Bachelorarbeit

Studienrichtung
Technische Informatik

Entwurf eines RISC-V-Prozessors mit quelloffenen Tools

Verfasser:
Lorenz Sommer
lorenzsommer1@gmail.com

im Fachgebiet Effiziente eingebettete System

Prüfer: Prof. Dr. Gundolf Kiefer
Zweitprüfer: Prof. Dr. Hubert Högl

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

Inhaltsverzeichnis

Abkürzungsverzeichnis

ALU	Arithmetic Logic Unit
ASCII	American Standard Code for Information Interchange
BIIS	Base Integer Instruction Set
CPU	Central Processing Unit
FPGA	Field Programable Gate Array
GCC	GNU Compiler Collection
ICSC	Intel Compiler for SystemC
IR	Instruction Register
LSB	Least Significant Bit
MEMU	Memory Unit
PC	Programm Counter
RISC	Reduced Instruction Set Computer
UART	Universal asynchronous receiver-transmitter
YOSYS	Yosys Open SYnthesis Suite

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Halbleiter durchdringen alle Facetten unserer modernen Gesellschaft und spielen eine zentrale Rolle in der Transformation dieser von einer analogen zu einer digitalen und nachhaltigen Gesellschaft [6]. Die COVID-19 Pandemie hat die systematischen Schwächen der europäischen Union in Hinsicht auf die Abhängigkeiten von Drittländern für Fertigung und Entwurf von Halbleitern aufgedeckt [6]. Jahrelange Industrietrends haben das Zentrum der Produktion nach Ostasien verlegt [10], während der Hauptmarktanteil der Halbleiterindustrie im Jahre 2023 mit 50% bei US-amerikanischen Unternehmen lag - verglichen mit nur 12% für ganz Europa [21].

Um dieser Faktenlage entgegenzuwirken hat die Europäische Union 2023 eine Verordnung verabschiedet, welche allgemein als der „Chips Act“ bekannt ist. Mit Ihr wurde ein "[...] ein Rahmen für die Stärkung der Resilienz der Union im Bereich der Halbleitertechnologien [...]"[6] etabliert. Durch Investitionen in die europäische Industrie und Forschung soll die Marktfähigkeit und Souveränität der Union im Bezug auf Halbleiter gestärkt werden. Ebenfalls wird die Ausbildung von Fachkräften zur Entwicklung von Halbleitern hervorgehoben und als wichtiger Aspekt für die Zukunft der europäischen Halbleiterindustrie identifiziert [6].

Prozessoren (CPUs) sind das Herzstück vieler elektronischen Geräte. Jedes Smartphone und jeder Computer haben mindestens einen Prozessor verbaut. Somit sind Prozessoren unverzichtbar für unsere Gesellschaft. Um die Abhängigkeit der europäischen Union von Prozessorherstellern aus Drittländern zu reduzieren, ist die Ausbildung von Fachkräften im Bereich der Rechnerarchitektur wichtig für ein digital souveränes Europa [6].

RISC-V (gesprochen „risc five “) ist eine quelloffene und frei verfügbare Befehlssatzarchitektur (ISA). Die RISC-V ISA hat sich seit ihrer ersten Ausgabe im Jahr 2014 und der Gründung des Verwaltungskörpers „RISC-V International“ mit heute 4500 Mitgliedern aus 70 Ländern in Industrie und Forschung etabliert [19, 8].

Die RISC-V ISA ist bewusst einfach aufgebaut, erweiterbar und modular [22, 8]. Aufgrund ihrer quelloffenen und lizenzgebührfreien Natur hat sich ein open-source-Ökosystem aus Entwicklungswerkzeugen und Anwendungssoftware um RISC-V gebildet [8]. Aus diesen Gründen bietet sich die RISC-V ISA für die Lehre im Bereich der Rechnerarchitektur und Hardwareentwicklung besonders an [8].

Die Forschungsgruppe Effiziente Eingebettete Systeme (EES) [7] bietet Studierenden an der Technischen Hochschule Augsburg die Möglichkeit tiefe Einblicke in die Fachgebiete Rechnerarchitektur, FPGA-Technologie und KI-Beschleunigung zu erlangen. Das von der EES-Forschungsgruppe gegründete Projekt „PicoNut“ nutzt die RISC-V ISA mit dem Ziel einen einfachen, minimalen und gleichzeitig erweiterbaren Prozessor auf FPGA-Basis zu entwickeln. Er soll von Studierenden der THA weiterentwickelt und für die Lehre an dieser eingesetzt werden. Sein Aufbau, seine Schnittstellen und seine Funktionsweise sollen gut dokumentiert sein, um Studierenden einen schnellen Einstieg in das Projekt zu erlauben und um das Vermitteln von Grundkonzepten der Hardwareentwicklung, insbesondere der Rechnerarchitektur, zu erleichtern.

1.2 Ziel der Arbeit

Ziel dieser Bachelorarbeit ist eine erste, minimale Implementierung des `RV32I` „Base Integer Instruction Set“ der RISC-V Befehlssatzarchitektur in Form des „minimalen Nucleus“. Der Begriff „Nucleus “ im Rahmen dieses Projektes bezieht sich auf die Teilkomponente des Prozessors, die Befehle ausführt. Dieser minimale Nucleus soll in SystemC beschrieben und so implementiert und dokumentiert sein, dass er für das PicoNut Projekt als leicht verständliche Grundlage für die Weiterentwicklung dient. Dies bedeutet, dass neben ausführlich dokumentiertem Quellcode auch eine technische Dokumentation bereitgestellt werden soll. Hiermit soll Studierenden der technischen Hochschule Augsburg sowie Außenstehenden, die mit und an dem PicoNut-Projekt arbeiten möchten, erleichtert werden einen Einstieg in dieses zu finden. Des weiteren wird ein Simulator zur Ausführung von C-Programmen bereitgestellt.

1.3 Aufbau der Arbeit

Zu Beginn dieser Arbeit werden in Kapitel 2 alle nötigen Grundlagen zum Verständnis der RISC-V ISA beschrieben. Weiterhin wird im Abschnitt 2.3 die Schnittstelle des PicoNut-Prozessors beschrieben. Das PicoNut-Projekt wird im Abschnitt 2.2

vorgestellt. Es gibt einen Überblick über dessen Motivation und Aufbau. Weiterhin wird das Gesamtsystem des PicoNut-Projekts vorgestellt. Zusätzlich wird auf die C++-Bibliothek SystemC eingegangen. Im Anschluss bietet das Kapitel 3 eine Übersicht über mit dieser Arbeit vergleichbaren Lösungen. Das Kapitel 4 detailliert die Anforderungen an den minimalen Nucleus. Das folgende Kapitel 5 bildet den Hauptteil dieser Arbeit und beschreibt den Aufbau und die Funktionsweise des minimalen Nucleus. Es wird beschrieben, wie der minimale Nucleus mit seinen Submodulen RISC-V-Befehle ausführen kann. Die Funktion der Submodule wird ebenfalls erläutert. Anschließend werden die Ergebnisse dieser Arbeit im Kapitel 6 dargelegt. Der PicoNut-Simulator wird beschrieben und es wird diskutiert, inwiefern dieser den minimalen Nucleus validieren kann. Weiterhin wird das synthetisierte Hardwaredesign anhand einer Statistik der genutzten Logikzellen analysiert. Zuletzt erfolge ein Vergleich mit dem VISCY-V-CPU [2].

2 Grundlagen

2.1 Die RISC-V Befehlssatzarchitektur

2.1.1 Aufbau der Befehlssatzarchitektur

Die RISC-V ISA ist in zwei Teile geteilt: Die „unprivilegierte“ [22] und die „privilegierte“ [23] ISA [20]. Der Inhalt der privilegierten Spezifikation ist für diese Arbeit nicht von Belang, da der minimale Nucleus keine Betriebsysteme unterstützen muss.

Die unprivilegierte ISA spezifiziert mehrere „Base Integer Instruction Sets “ (BIIS)[22]. Ein BIIS definiert einen Basissatz an Befehlen für dessen Wortbreite.

Name	Wortbreite	Beschreibung
`RV32I`	32-bit	Base Integer Instruction Set, 32 Register
`RV32E`	32-bit	Base Integer Instruction Set, **E**mbedded, 16 Register
`RV64I`	64-bit	Base Integer Instruction Set, 32 Register
`RV64E`	64-bit	Base Integer Instruction Set, **E**mbedded, 16 Register
`RV128I`	128-bit	Base Integer Instruction Set, 32 Register

Tabelle 2.1: Base Integer Instruction Sets der RISC-V ISA

Alle in der Tabelle 2.1 aufgelisteten BIIS implementieren die 40 Befehle des RV32I BIIS. Diese 40 Befehle umfassen grundlegende Rechenoperationen wie Addieren, Subtrahieren, Sprungbefehle und Speicherzugriffe. Eine vollständige Referenz aller RV32I-Befehle ist im Anhang A.1 zu finden.

Das Suffix `I` im Namen eines BIIS steht für „integer“ und bedeutet, dass dessen Befehle nur ganze Zahlen nutzen. Das Suffix `E` steht für „embedded“ und bedeutet, dass eine solche Implementierung nur 16 interne 32-bit Register besitzt. Somit kann die Chipfläche für Systeme mit besonderen Anforderungen reduziert werden [22].

Modularität erreicht die RISC-V ISA durch Erweiterungen - sogenannte „extensions“. Diese Erweiterungen bauen auf den BIIS auf und fügen diesen neue Befehle und Funktionalität hinzu [22]. Da es eine große Anzahl an Erweiterungen gibt werden

diese hier nicht alle aufgezählt. Jedoch lohnt es sich ein einige Beispiele zu nennen. Die `M`-extension beinhaltet dedizierte Befehle für das Multiplizieren von Zahlen. Die `F`-extension fügt Funktionalität und Befehle für Gleitkommaarithmetik hinzu. Die `V`-extension erweitert eine Architektur um Vektorbefehle und ermöglicht starke Parallelisierung auf Datenebene. [22]
Mit den richtigen Entwicklungstechniken lassen sich diese Erweiterungen dynamisch an- und abschalten und erlauben somit, dass eine CPU modular auf eine Anwendung angepasst werden kann. Das `RV32I` BIIS kann alle Extensions außer der `A`-Extension emulieren [22].

2.1.2 RISC-V Befehlsgruppen

Die 40 Befehle des `RV32I` BIIS sind in 5 Befehlsgruppen aufgeteilt [22].

- Die Befehlsgruppe „Integer Computational Instructions“ (`Ganzzahlige Rechenoperationen`) umfasst arithmetische und logische Befehle. Diese Befehle bilden aus zwei Operanden, typischerweise `A` und `B`, ein Ergebnis `Y`, welches in ein internes Register gespeichert wird. Es wird weiterhin in „register-register“- und „register-immediate“-Befehle unterschieden. Register-Register-Befehle nutzen als Operanden zwei Werte aus den internen Registern. Register-Immediate-Befehle bilden das Ergebnis aus dem Wert eines internen Registers und einem *Direktwert*. Direktwerte, auch „immediate values“, sind Konstanten die in einem Befehlswort enthalten sind. Neben den arithmetischen ALU-Befehlen zählen auch die Befehle `lui` und `auipc` zu den ganzzahligen Rechenoperationen.

- Kontrollflussbefehle oder „Control transfer instructions“ sind Befehle, die das „springen“ innerhalb eines Programms ermöglichen. Mit ihnen werden zum Beispiel Funktionsaufrufe und „if“-Bedingungen in Programmen implementiert, was in beiden Fällen dem Manipulieren der aktuellen Position in dem sequenziell im Speicher liegenden Programm entspricht. Zusätzlich wird zwischen „unconditional jumps“ (unbedingten Sprünge) und „conditional branches“ (bedingten Sprünge) unterschieden. Unbedingte Sprungbefehle springen sofort an eine neue Position im Programm. Bedingte Sprünge prüfen vor dem Sprung eine Bedingung und springen nur dann, wenn diese erfüllt ist. Springen sie nicht, wird der nächste Befehl im Programm geladen.

- Speichertransaktionen sind in der Befehlsgruppe „load and store instructions “ enthalten. „load“-Befehle initiieren eine Bustransaktion und laden einen Datenwert von einer Adresse im Speicher in ein internes Register. Ähnlich

speichern „store“-Befehle einen Datenwert aus einem internen Register in eine Adresse im Speicher.

- Der `fence`-Befehl ist der einzige Befehl der Gruppe „memory ordering instructions“. Er wird genutzt um Barrieren für die Speichersynchronisation zu errichten.
- Die Befehlsgruppe „environment call and breakpoints“-Befehlsgruppe enthält die Befehle `ecall` und `ebreak`. Der ecall-Befehl stellt eine Anfrage an die Ausführungsumgebung. Der ebreak-Befehl gibt die Kontrolle an einen Debugger ab.

2.1.3 RISC-V Befehlskodierungen

Das `RV32I` BIIS definiert 6 Befehlskodierungen. Diese Kodierungen sind notwendig, da unterschiedliche Befehle unterschiedliche Informationen für ihre Ausführung benötigen. Zum Beispiel nutzen Register-Register-Befehle zwei interne Registerwerte als Operanden und müssen demnach zwei 5-bit Werte für die Wahl dieser in ihrem Befehlswort enthalten. Register-Immediate-Befehle nutzen nur *einen* Registerwert und einen Direktwert als Operanden. Aus diesem Umstand entsteht die Notwendigkeit für verschiedene Befehlsformate. Die Abbildung 2.1 zeigt wie die Kodierungen aufgebaut sind [22].

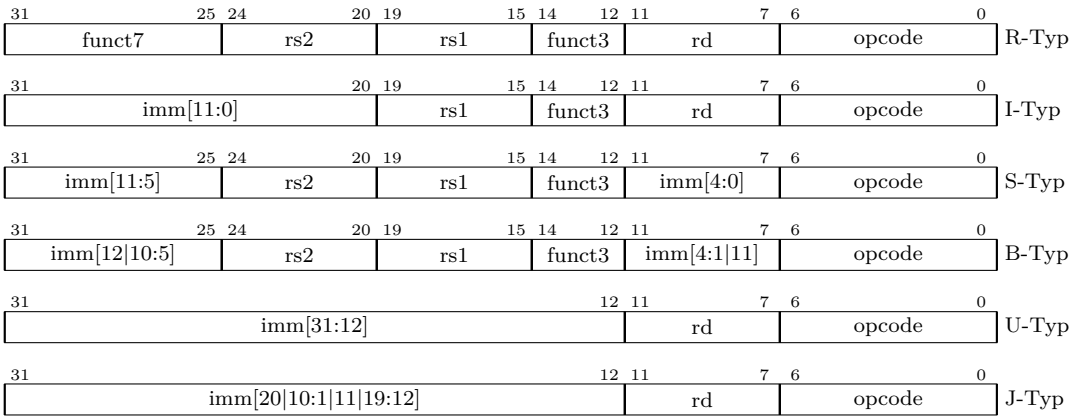

Abbildung 2.1: RISC-V Befehlskodierungen

Das `opcode`-Feld („operation code“, im Weiteren nur „der opcode“) dient zur Unterscheidung von Befehlen. Er umfasst 7 Bit. Die Befehle des RV32I BIIS nutzen 10 verschiedene opcodes. Weitere opcodes sind von Erweiterungen der RISC-V ISA belegt oder für zukünftige Erweiterungen reserviert.

Die Felder `rd` („destination register“), `rs1` („source register 1“) und `rs2` („source register 2“) indizieren jeweils eines der 32 internen Register. `rd` ist das Zielregister von Operationen, die einen Wert abspeichern. `rs1` und `rs2` sind Quellregister für abgespeicherte Werte, die von bestimmten Operationen zur Verarbeitung verwendet werden.

Zur weiteren Unterscheidung zwischen einzelnen Befehlen innerhalb einer Befehlsgruppe mit dem selben opcode dienen die Felder `funct3` und `funct7`. Mit `funct3` wird zwischen einzelnen Operationen gewählt, während `funct7` bei gleichen Werten von `funct3` zusätzlich zur Unterscheidung dient.

Direktwerte sind in den Feldern `imm[]` kodiert. Diese Direktwerte werden je nach Befehlsformat verschieden dekodiert.

2.2 Einführung in das Projekt PicoNut

2.2.1 Projektziele- und Aufbau

Das Projekt PicoNut der Technischen Hochschule Augsburg (THA) [24] im Rahmen der EES-Arbeitsgruppe [7] 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-Projekts sind die Unterstützung von Betriebssystemen wie Linux und FreeRTOS sowie die Integration von Hardware-Beschleunigern für KI-Anwendungen. In diesem Zusammenhang soll der Prozessor für die Lehre und Forschung an der THA eingesetzt werden.

Zum Zeitpunkt der Verfassung dieser Arbeit besteht das Projektteam des PicoNut aus den folgenden Mitgliedern:

- Lukas Bauer (`lukas.bauer@tha.de`): Verwaltung des Projekts und des Buildsystems sowie der Versionierungssoftware (git). Entwicklung des UART-Peripheriemoduls.
- Marco Milenkovic (`marco.milenkovic@tha.de`): Entwicklung der Simulations-MemU („sim-only memu“) und der „CSoft-Schnittstelle“.
- Claus Janicher (`claus.janicher1@tha.de`): Entwicklung der Hardware-MemU.

- Lorenz Sommer (`lorenz.sommer@tha.de`): Entwicklung des minimalen Nucleus.
- Johannes Hofmann (`johannes.hofmann@tha.de`): Entwicklung des GPIO-Peripheriemoduls.
- Gundolf Kiefer (`gundolf.kiefer@tha.de`): Gründer in Leitender Funktion.
- Michael Schäferling (`michael.schaeferling@tha.de`): Laborbetreuung und Serververwaltung.

2.2.2 Übersicht des Gesamtsystems

Die Abbildung 2.2 zeigt, wie sich das PicoNut-*Gesamtsystem* zusammensetzt.

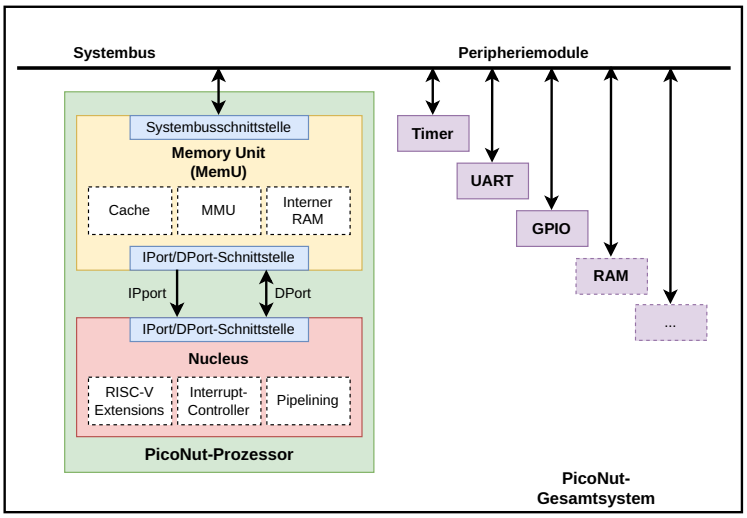

Abbildung 2.2: Übersicht des PicoNut-Gesamtsystems

Der PicoNut-Prozessor ist über einen Systembus mit Peripheriemodulen verbunden. Das für den Systembus genutzte Protokoll ist frei wählbar, jedoch sind bereits erste Module mit einer Wishbone-Schnittstelle ausgestattet. Neben den in der Abbildung 2.2 in violett gezeigten Peripheriemodulen sind weitere Module wie ein VGA-Controller oder Schnittstellen für weitere Kommunikationsprotokolle wie `I2C` oder `CAN` denkbar. In der Abbildung gestrichelte Kästen zeigen optionale Module und Optionen. Der Nucleus, die Memory Unit (MemU) und das Gesamtsystem sind durch das zu- und abschalten von Features und Modulen modular.

Der PicoNut-*Prozessor* setzt sich aus zwei Teilkomponenten zusammen.

- Nucleus: Die Zentrale Ausführungseinheit des Prozessors.

- MemU (Memory Unit): Übersetzungsglied zwischen der im Kapitel 2.3 beschriebenen IPort/DPort-Schnittstelle und dem Systembus. Ihre primäre Aufgabe ist das durchführen von Speicheranfragen.

Nucleus und MemU sind ebenfalls austauschbar. Es soll vermieden werden, dass ein einziger Nucleus kontinuierlich weiterentwickelt wird. Anstatt dessen sollen entweder komplett neue Nuclei oder zuschaltbare Optionen für den minimalen Nucleus entwickelt werden, um diverse Konfigurationen zu erlauben. Mögliche zuschaltbare Optionen für den Nucleus sind diverse RISC-V-Extensions (insbesondere CSR) oder ein Interrupt-Controller. Denkbar ist ebenfalls das Erweitern des Nucleus auf eine Pipelining-Architektur. Ähnlich verhält es sich mit der MemU. Denkbare Optionen sind die Unterstützung von Caching für schnellere Speicherzugriffe und Memory Management für die Unterstützung von Betriebssystemen wie Linux.

Der Arbeitsspeicher eines PicoNut-Gesamtsystems kann entweder als Peripheriemodul am Systembus angeschlossen werden oder in der MemU enthalten sein.

2.2.3 Tools und Software

Hervorzuheben ist, dass sich dieses Projekt primär an quelloffenen Tools für Hardwarebeschreibung, Synthese und Kompilierung bedient.

Die Hardwarebeschreibung des PicoNut nutzt die C++-Bibliothe *SystemC* [13].

Die SystemC Hardwarebeschreibung des minimalen Nucleus wird mit dem „Intel Compiler for SystemC “ (ICSC) [15] in äquivalenten SystemVerilog Code übersetzt, welcher zur Synthese genutzt wird.

Um das Design auf FPGA-Boards wie das Radiona ULX3S oder OrangeCrab zu übertragen, wird dieses mit dem offenen Syntheseframework „YOSYS “ (Yosys Open SYnthesis Suite) [26] synthetisiert.

2.2.4 Die „C-Soft“-Schnittstelle

Die „C-Soft“-Schnittstelle ermöglicht die Kommunikation von C++-Softwaremodellen und simulierter Hardware. Mit ihr können komplexe Hardwaremodule in Software modelliert, in eine Simulationsumgebung eingebunden werden und dort mit Hardwaremodellen kommunizieren. Durch diesen Ansatz stehen die Geschwindigkeit, Features und Ressourcen der Programmiersprache C++ zur Emulation von Hardwaresystemen zur Verfügung. Die Schnittstelle wurde von Marco Milenkovic im Rahmen des PicoNut-Projekts entwickelt.

2.3 Die IPort/DPort Schnittstelle

2.3.1 Zweck und Übersicht

Die IPort/DPort-Schnittstelle wurde in der Konzeptionsphase des PicoNut-Projekts von den Teammitgliedern (federführend Prof. Dr. Gundolf Kiefer) definiert und ausgearbeitet. Sie ist die Schnittstelle zwischen dem Nucleus und der MemU und besteht aus zwei Bussen - dem „Instruction Port“ (im Weiteren nur *IPort*) und dem „Data port“ (im Weiteren nur *DPort*). Der IPort dient zum Laden neuer Befehlswörter. Da Befehle sets nur aus dem Speicher gelesen werden, ist der IPort unidirektional und führt ausschließlich Lesetransaktion durch. Über den DPort werden durch `load`- und `store`-Befehle Daten über die MemU entweder aus dem Arbeitsspeicher geladen oder in diesen geschrieben. Daten können auch in die Register von Peripheriemodulen geschrieben oder von diesen gelesen werden.
Eine Trennung dieser beiden Kernaufgaben in zwei Busse vereinfacht die Architektur des Prozessors, da für eine einkanalige Busverbindung zusätzliche Entscheidungslogik vonnöten wäre, welche unter anderem auch das Steuerwerk verkomplizieren würde.

Diese Schnittstelle *muss* von allen zukünftigen Nuclei und MemUs des PicoNut-Projekts unterstützt werden, damit diese austauschbar sind.

2.3.2 Bussignale der IPort/DPort-Schnittstelle

Die Tabelle 2.2 zeigt die Signale der Busleitungen, deren Treiber und mögliche Signalbreiten.

Signalname	Bedeutung	Treiber	Signalbreite
`stb`	strobe	Nucleus	1
`we` (Nur DPort)	write enable	Nucleus	1
`bsel`	byteselect	Nucleus	1 bit pro Byte der Datenleitungen
`adr`	address	Nucleus	32
`wdata` (Nur DPort)	write data	Nucleus	Beliebiges vielfaches von 32
`ack`	acknowledge	MemU	1
`rdata`	read data	MemU	Beliebiges vielfaches von 32
`clk`	clock	Nucleus	1

Tabelle 2.2: Bussignale der IPort/DPort-Schnittstelle

Da die IPort-Schnittstelle ausschließlich für lesende Transaktionen verwendet wird, können die Signale `we` und `wdata` weggelassen werden. Da es nur zwei Busteilnehmer

gibt und der Nucleus alle Transaktionen beginnt, sind das Protokoll und dessen Signalnamen aus der Sicht des Nucleus formuliert. Die Bitbreiten der `wdata` und `rdata` Busleitungen müssen identisch sein. Der DPort- und IPort-Bus können jedoch unterschiedliche Bitbreiten an den Datenleitungen aufweisen. Beide Busse verwenden das gleiche Protokoll.

2.3.3 Das IPort/DPort-Protokoll

Um die Funktion des IPort/DPort-Protokolls zu verdeutlichen, zeigen die Abbildungen 2.3 und 2.4 beispielhafte Abläufe von Transaktionen der Schnittstelle. Die farbig markierten Felder signalisieren, dass Daten und Werte in diesen Intervallen stabil anliegen und valide Signalwerte haben müssen. Die blauen Linien verbinden den Anfang und das Ende einer Transaktion.

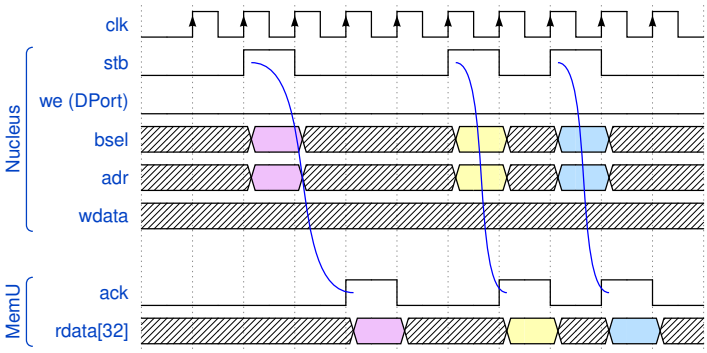

Abbildung 2.3: Lesezyklus IPort/DPort

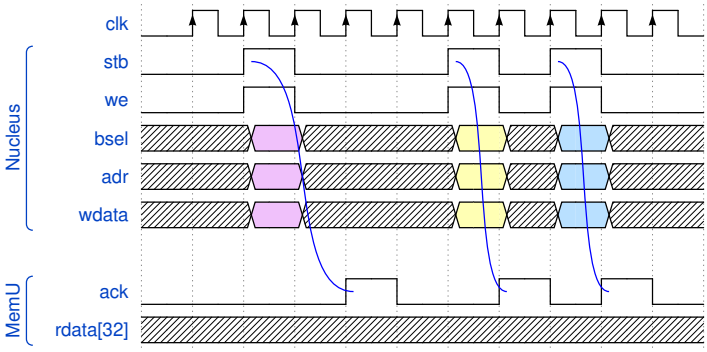

Abbildung 2.4: Schreibzyklus DPort

Eine Bustransaktion wird mit dem Setzen des `strobe`-Signals begonnen. Es darf nur für einen Taktzyklus gesetzt sein. Während das `strobe`-Signals gesetzt ist, müssen die Werte der Signale `adr`, `wdata` und `bsel` valide sein und stabil anliegen. Ist das `we`-Signal zeitgleich mit dem `strobe`-Signal gesetzt, beginnt eine Schreibtransaktion

(siehe 2.4). Es darf ebenfalls nur für einen Taktzyklus den Wert logisch `1` haben. Ist das `we`-Signal zu Beginn einer Transaktion nicht gesetzt, ist diese automatisch eine Lesetransaktion.

Das `bsel`-Signal dient der Auswahl valider Bytes aus dem zu lesenden oder schreibenden Datenwort. Eine genaue Erklärung des byteselect-Mechanismus ist im Kapitel 5.5.6 zu finden.

Die Adresse einer Bustransaktion ist im `adr`-Signal enthalten. Dies kann eine Speicheradresse oder die Adresse eines Registers in einem Peripheriemodul sein. Für den minimalen Nucleus ist dieses Signal 32 Bit breit.

Das zu schreibende Datenwort einer Schreibtransaktion ist im `wdata`-Signal enthalten. Es wird vom Nucleus durch die MemU in den Arbeitsspeicher oder in ein Register eines Peripheriemoduls geschrieben. Es wird nur bei Schreibtransaktion belegt.

Das `ack`-Signal quittiert Bustransaktionen und wird von der MemU gesetzt. Die MemU darf dieses Signal erst dann setzen, wenn die vom Nucleus angefragte Lese- oder Schreibtransaktion vollständig ausgeführt ist. Für Schreibtransaktionen bedeutet dies, dass das Schreiben des Datenwortes in den Arbeitsspeicher oder ein Peripheriemodul durch dessen Schnittstelle bestätigt wurde. Bei Lesetransaktionen bedeutet dies, dass der vom Nucleus angefragte Datenwert aus dem Hauptspeicher stabil am *rdata*-Signal anliegt. Es darf den logischen Wert `1` nur für die Dauer eines Taktzyklus halten.

Das eingehende Datenwort einer Lesetransaktion ist im `rdata`-Signal enthalten. Es wird über die MemU durch eine vom Nucleus eingeleitete Lesetransaktion aus dem Speicher oder einem Peripheriemodul gelesen und in einem internen Register gespeichert. Es muss valide Daten enthalten und stabil anliegen, während das `ack`-Signal gesetzt ist.

Das „clk“-Signal gibt den synchronen Taktzyklus der Busteilnehmer vor.

2.3.4 „Overlap“-Modus der Schnittstelle

Das Protokoll, so wie es bisher beschrieben wurde, stellt die „single mode“-Variante (im Weiteren nur „Einzelmodus “) des Protokolls dar. Dies beduetet, dass zu jedem Zeitpunkt nur eine einzige Bustransaktion im Gange sein darf. Eine neue Transaktion darf (durch das setzen des `stb`-Signals) nur dann begonnen werden, wenn das `ack`-Signal vorher einen Taktzyklus lang den logischen Wert `1` hatte und damit die vorherige Transaktion quittiert ist. Zwar vereinfacht dies die Implementierung des

Protokolls, nutzt die Busleitungen jedoch nicht optimal aus. Selbst wenn das `ack`-Signal direkt im folgenden Taktzyklus nach dem setzen des `strobe`-Signals eintrifft, vergeht mindestens ein Taktzyklus in dem keine Daten übertragen werden (siehe Abbildung 2.4). Der „overlap-mode“ (im Weiteren nur “Überlappungsmodus“) erlaubt, dass maximal eine Zusätzliche Transaktion vor der Quittierung der zuletzt begonnenen Transaktion initiiert werden darf. Das `stb`-Signal darf noch vor dem eintreffen der nächsten Quittierung erneut gesetzt werden. So können zu jeder zeit maximal zwei Transaktionen gleichzeitig erfolgen.

Die Abbildungen 2.6 und 2.5 zeigen Transaktionen im Überlappungsmodus beispielhaft in Timingdiagrammen. Die Antwortzeiten der MemU sind beliebig gewählt.

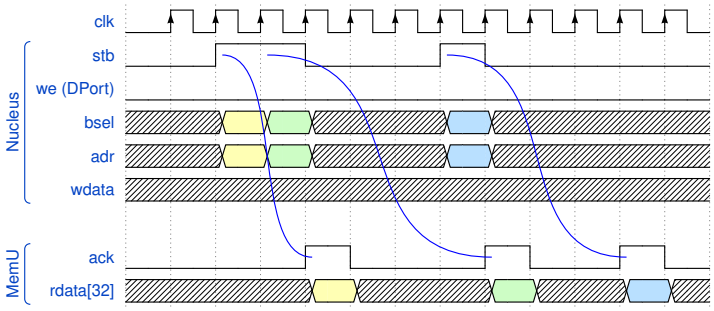

Abbildung 2.5: Lesezyklus DPort/IPort mit Überlappung

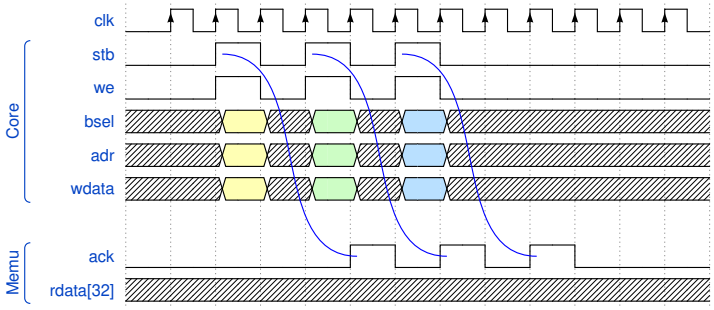

Abbildung 2.6: Schreibzyklus DPort mit Überlappung

Wie in der Abbildung 2.5 zu erkennen ist, verbessert sich der Datendurchsatz der Busleitung bei langsamen und variierenden Antwortzeiten der MemU auch im Überlappungsmodus nicht ausschlaggebend. Optimaler Durchsatz ist nur erreichbar, wenn die MemU innerhalb eines einzigen Taktzyklus nach dem setzen des `stb`-Signals das `ack`-Signal setzt. Ist dies nicht der Fall, sind alle Transaktionen nur einen Taktzyklus kürzer, da nicht auf Eintreffen des `ack`-Signals gewartet werden muss. Antwortet die MemU jedoch sofort, verdoppelt sich der Busdurchsatz.

Eine optimale Auslastung der Busleitungen mit doppeltem Durchsatz ist in der Abbildung 2.7 dargestellt.

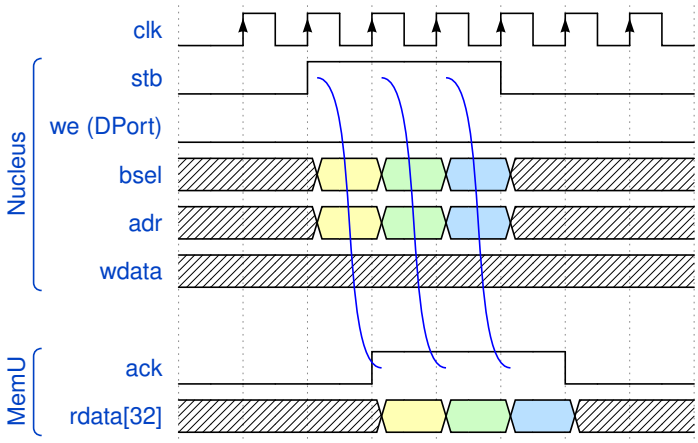

Abbildung 2.7: Optimale Ausnutzung im Überlappungsmodus

Unterstützt die MemU den Überlappungsmodus, kann ein Nucleus das Busprotokoll je nach Bedarf in beiden Betriebsmodi gleichzeitug nutzen. Der Einzelmodus ist kompatibel zum Überlappungsmodus. Der minimale Nucleus unterstützt den Überlappungsmodus *nicht*.

2.4 Hardwareentwicklung mit SystemC

SystemC ist eine auf die Programmiersprache C++ aufbauende Bibliothek, welche diese um HDL-Konzepte wie die Modellierung von Zeiten, Parallelität und die Beschreibung von Strukturen erweitert [16, 5]. SystemC fügt C++ *keine* neue Syntax hinzu. Es werden lediglich Klassen zum Definieren von Modulen, Prozessen, Ports und Signalen hinzugefügt[5]. Das alleinstellende Merkmal von SystemC, ist, dass mithilfe der Bibliothek die Verifikation, die Hardwarebeschreibung, Softwareentwicklung und der Systementwurf mit nur einer Sprache, nämlich C++, erfolgen kann [5].

Eine SystemC-Hardwarebeschreibung kann mit einem beliebigen C++-Compiler kompiliert werden und ergibt ein Softwaremodell dieser. Dieses Softwaremodell hat das gleiche Verhalten wie die beschriebene Hardware und kann in einer Simulationsumgebung zu Verifikation derer genutzt werden [5]. Weiterhin können alle Typischen C++-Entwicklungstools für die Entwicklung mit SystemC genutzt werden. Insbesondere ermöglicht dies die Nutzung von Softwaredebuggern [5].

SystemC-Quellcode ist jedoch nicht immer synthetisierbar. C++ ist flexibel und erlaubt das erstellen von validem SystemC-Quellcode, der jedoch nie in Hardware realisierbar wäre [5]. Deshalb wurde das *SystemC Synthesizable Subset* [1] definiert. Es beschreibt ein synthetisierbares Subset von SystemC.

Für die Lehre im Bereich Hardwareentwicklung bietet die Entwicklung mit SystemC somit eine Reihe von Vorteilen. Hardware-Prototypen können schnell erstellt werden, da C++-Code beliebig komplex sein kann und in Hardware komplexe Algorithmen mit einfachem C++-Code implementiert werden können. Des weiteren ist das einbinden beliebig komplexer C++-Funktionen- und Klassen möglich, was zum Beispiel für die „C-Soft-Schnittstelle“ (siehe 2.2.4) genutzt wurde. So können synthetisierbare SystemC-Module mit reinen Softwaremodulen zur Simulation gemischt werden. Dadurch können komplexe Simulationsumgebungen einfach aufgebaut werden.

3 Stand der Technik

3.1 Der VISCY-V-Prozessor

Der VISCY-V-CPU ist ein im Rahmen der Arbeit „Entwicklung einer minimalen RISC-V-CPU zu Lehrzwecken“ von Patrick Arlt entwickelter RISC-V-Prozessor [2]. Er ist in VHDL beschrieben. Der VISCY-V-CPU implementiert nur einen Teil des `RV32I` RISC-V BIIS. Grund hierfür war, dass dieser Prozessor von Studierenden der Technischen Hochschule Augsburg im Rahmen des Praktikums „Hardware-Systeme“ nachgebaut werden soll [2] und deshalb in seiner Komplexität reduziert werden musste. Um den Befehlssatz zu reduzieren, wurde der Fakt ausgenutzt, dass sich manche `RV32I`-Befehle durch die Kombination aus zwei anderen Befehlen ersetzen lassen. Zum Beispiel: Der Befehl `auipc` kann durch eine Kombination der Befehle `lui` und `addi` emuliert werden [2]. Weiterhin beschränkt sich der VISCY-V-CPU auf 32-bit Speicherzugriffe (`lw` und `sw`) und seine `ALU` kann Daten nur um eine Stelle schieben [2].

All dies führt dazu, dass der VISCY-V-CPU mit GCC kompilierte C-Programme *nicht* ausführen kann. Programme müssen somit in RISC-V-Assembly geschrieben werden und es muss darauf geachtet werden, dass bestimmte Befehle nicht verwendet werden dürfen und durch andere ersetzt werden müssen.

Im Vergleich dazu steht der PicoNut-Prozessor, welcher ebenfalls zu Lehrzwecken eingesetzt werden soll. Jedoch soll er von Studierenden nicht nachgebaut, sondern im Rahmen des PicoNut-Projekts von Studierenden *weiterentwickelt* werden. Aus diesem Grund soll dessen erster, minimaler und leicht verständlicher *Nucleus* als Grundlage und Referenz für das gesamte Projekt dienen. In diesem Zusammenhang, soll er auch das ausführen von mit GCC kompilierten C-Programmen unterstützen. Hierzu müssen alle Befehle des `RV32I` BIIS (bis auf drei Ausnahmen, siehe 5.2) unterstützt werden.

3.2 Vergleichbare RISC-V-Prozessoren

Der „RISC-V Exchange“ [18] auf der offiziellen RISC-V-Webseite zeigt die große Bandbreite der verfügbaren RISC-V-Prozessoren. Seit seiner Veröffentlichung, ist das Interesse an RISC-V stark gewachsen [8]. Dies ist nicht nur an der Anzahl der verschiedenen Prozessoren zu erkennen, sondern auch an den steigenden Zahlen von Publikationen im Bereich RISC-V [11]. Die Seite „IEEE Xplore“[12] gibt bei der Suche mit dem Schlagwort „risc-v“ im Zeitraum 2023 bis 2024 *799* Ergebnisse zurück (Stand 21.10.2024).

Die RISC-V-Prozessoren *PicoRV32* [25], *RISCO 5* [3] oder *NEORV32* [17] sind quelloffen und frei verfügbar. Alle diese Prozessoren unterstützen das `RV32I` BIIS und verfolgen Falle des *PicoRV32* und des *NEORV32* einen ähnlichen Ansatz zur Konfigurierbarkeit. Diese Prozessoren geben einen Ausblick darauf, was im PicoNut-Projekt in Zukunft erreicht werden kann.

Der *ParaNut*-Prozessor der Technischen Hochschule Augsburg ist ein quelloffener, konfigurierbarer RISC-V-Prozessor mit Fokus auf Parallelisierung und Skalierung [4]. Der ParaNut-Prozessor ist in SystemC modelliert und hat für diese Arbeit als Nachschlagewerk für SystemC-Modellierung gedient.

4 Anforderungsanalyse des minimalen Nucleus

In diesem Kapitel werden die Anforderungen an den im Rahmen dieser Arbeit entwickelten minimalen Nucleus dargestellt.

4.1 Anforderungen an den minimalen Nucleus

Der minimale Nucleus soll als erste, minimale Referenzimplementierung des RISC-V `RV32I` BIIS für das PicoNut-Projekt dienen. Er soll ein in der in der Programmiersprache C geschriebenes und mit `GCC` kompiliertes, „Hello World!“-Programm ausführen können. Der minimale Nucleus soll zusammen mit einer MemU einen ersten, funktionierenden PicoNut-Prozessor bilden, auf welchen im weiteren Projektverlauf möglichst einfach aufgebaut werden kann.

Der minimale Nucleus ist *minimal*, wenn er ...

- ... die zur Ausführung von mit GCC kompilierten C-Programmen *nötigen* Befehle des `RV32I` BIIS ausführen kann.
- ... nur *nötige* Logik enthält.

4.1.1 Anfordrungen an die Hardwarebeschreibung

Die Hardware des minimalen Nucleus soll in SystemC beschrieben sein. Der SystemC-Quellcode der Submodule und des gesamten minimalen Nucleus muss fehlerfrei mit ICSC in äquivalenten SystemVerilog-Code konvertierbar sein. Die Hardwarebeschreibung muss synthetisierbar sein.

Die Hardwarebeschreibung des minimalen Nucleus soll komplexe Algorithmen vermeiden, um diese leicht wartbar und verständlich zu halten. Die Submodule des minimalen Nucleus sollen in eigene Quellcode- und Header-Dateien aufgeteilt sein. Dies erleichtert die Übersicht über diese und erlaubt einen modularen Aufbau. Des weiteren werden lange und unübersichtliche Quellcodedateien vermieden. Die Quellcodedateien sollen ausführlich dokumentiert sein und einer einheitlichen Code- und Namenskonvention folgen.

4.1.2 Anforderungn an die Schnittstellen des minimalen Nucleus

Der minimale Nucleus soll die im Rahmen des PicoNut-Projekts entworfene IPort/DPort-Schnittstelle (siehe 2.3) und deren Protokoll unterstützen. Die Busleitungen dieser Schnittstelle sind somit die Ein- und Ausgänge des minimalen Nucleus. Er soll über keine weiteren Ein- oder Ausgänge verfügen.

5 Der minimale PiconNut Nucleus

5.1 Übersicht

Dieses Kapitel beschreibt den Aufbau des minimalen PicoNut-Nucleus, seine Untermodule, deren Aufbau sowie deren Zusammenspiel. Relevante Rahmenbedingungen und Einschränkungen werden erläutert.

Die RISC-V Spezifikation [22] definiert wie Befehle strukturiert sind und wie diese von einem Prozessor auszuführen sind. Sie definiert jedoch nicht die notwendigen Submodule oder die Architektur einer RISC-V CPU [22]. Die konkreten Schritte zu einer RV32I RISC-V Implementierung sind somit dem Entwickler überlassen.

Der Aufbau des minimalen Nucleus ist an den des VISCY-V-Prozessors [2] angelehnt.

5.2 Rahmenbedingungen und Einschränkungen

Die Implementierung des minimalen Nucleus richtet sich nach den im Kapitel 4 geschilderten Anforderungen. Somit ergeben sich die folgenden Einschränkungen.

Der minimale Nucleus ...

- ... unterstützt keine CSR-Befehle.
- ... enthält keine Ausführungspipeline.
- ... kann keine Exceptions auslösen.
- ... unterstützt keine Skalierung zu einer multi-core-Architektur und ist nur in single-core-Architekturen zu verwenden.
- ... enthält keine Logik zur branch prediction.
- ... enthält keinen interrupt-Controller.
- ... unterstützt alle `RV32I`-Befehle bis auf die Befehle `ecall`, `ebreak` und `fence`.

Die Befehle `ecall` und `ebreak` sind für das ausführen von GCC-kompilierten C-Programmen ohne Betriebssystem und Debugger nicht notwendig und somit nicht unterstützt. Der `fence`-Befehl ist ebenfalls nicht unterstützt, da nur der minimale Nucleus auf den Arbeitsspeicher zugreift und somit keine Speicherbarrieren vonnöten sind.

Die RISC-V-Spezifikation [22] setzt byte-adressierbaren Speicher voraus. Sie gibt weiterhin vor, dass alle 32-bit Wörter im Hauptspeicher „four-byte aligned“ sein müssen. Dies bedeutet, dass beginnend ab der Speicheradresse `0x00000000` alle vier Byte ein neues Befehls- oder Datenwort beginnt. Dies führt auch dazu, dass die untersten zwei Bit einer vom Nucleus ausgehenden Adresse immer den Wert `0` haben.

5.3 Arbeitsweise

Um einen Befehl auszuführen, durchläuft der minimale Nucleus drei Schritte.

- Befehl holen: Ein neuer Befehl wird über die IPort-Schnittstelle durch eine Speichertransaktion in das Befehlsregister `(IR)` geladen. Die Adresse des zu ladenden Befehls ist der Inhalt des Programmzählers `(PC)`. Dieser Schritt umfasst drei Taktzyklen.
- Befehl dekodieren: Das Befehlswort wird dekodiert und das Steuerwerk (`Controller`) setzt die zugehörigen Statussignale. Dieser Schritt wird in einem Taktzyklus ausgeführt.
- Befehl ausführen: Ausführen des Befehls. Dieser Schritt wird für alle Befehle, außer `load`- oder `load`-Befehlen, innerhalb eines Taktzyklus ausgeführt. Die mit `load`-Befehlen einhergehende Speichertransaktion nimmt vier Taktzyklen in Anspruch. Die Speichertransaktion von `store`-Befehlen benötigt drei Taktzyklen.

Nach einem erfolgreich ausgeführten Befehl, beginnt der Ablauf erneut mit dem „Befehl holen“ Schritt. Man kennt diese Abfolge auch unter dem Begriff „fetch-decode-execute“.

5.4 Gesamtsystem des minimalen Nucleus

In diesem Abschnitt wird das Gesamtsystem des minimalen Nucleus vorgestellt. Die Abbilding 5.1 zeigt den minimalen Nucleus als Blockdiagramm.

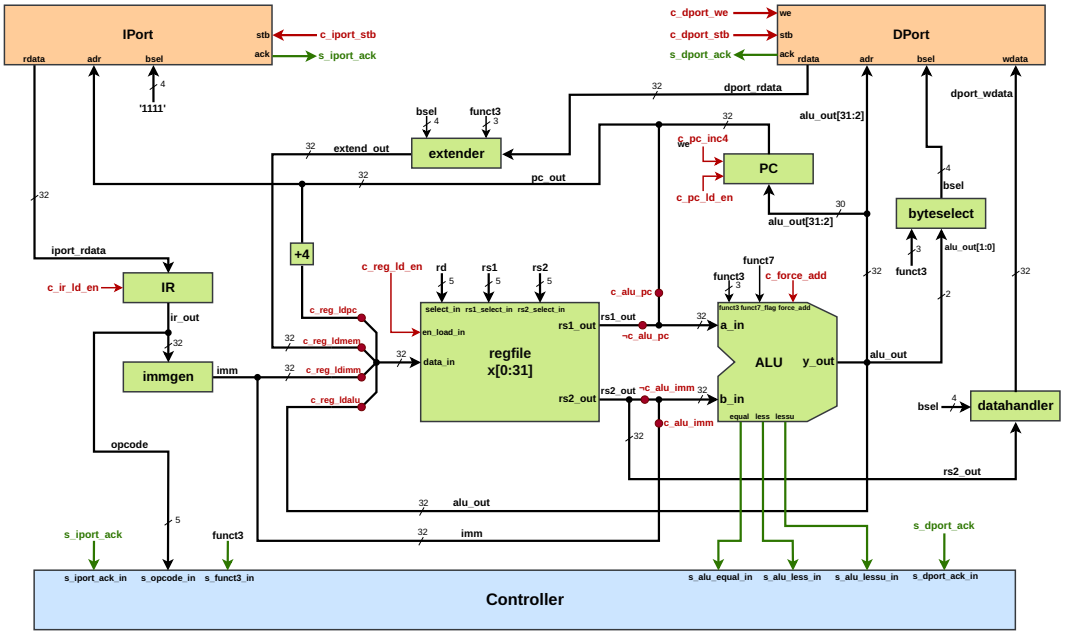

Abbildung 5.1: Gesamtsystem des minimalen Nucleus

Signale in roter Schrift sind Steuersignale. Sie sind Ausgänge des Controller-Submoduls. Rote Punkte in Signalleitungen sind „Ventile“ und steuern den Signalfluss. Sie wählen zwischen möglichen Eingangsquellen an den Eingängen bestimmter Submodule. Signale in grüner Schrift sind Statussignale. Sie sind die Eingänge des Controller-Submoduls.

5.4.1 Ein- und Ausgangsports des minimalen Nucleus

Die Ein- und Ausgangsports des minimalen Nucleus sind die Signale der IPort- und DPort-Schnittstellen. Die Signale `rdata` und `ack` beider Ports sind Eingänge. Die Signale `bsel`, `adr`, `stb`, `wdata` und `we` sind Ausgänge. Das `bsel`-Signal des IPort hat konstant auf den Wert binär `1111`, da Befehle stets ganze 32-bit Datenwörter sind und auch als solche geladen werden müssen. Die Signale `we` und `wdata` des IPort sind nicht verbunden (siehe 2.3.2). Das `adr`-Signal des DPorts kann auf seine obersten 30 Bits reduziert werden (siehe 5.2).

Um die Busleitungen zu synchronisieren, sind alle Ausgänge des minimalen Nucleus mit taktsynchronen Registern versehen und sind somit um jeweils einen Taktzyklus verzögert. Neben der Synchronisation der Busleitungen dienen diese Register als Schutz gegen das Entstehen langer Logikpfade über die Grenzen des minimalen Nucleus hinaus.

5.4.2 Signale des Gesamtsystems

Auf der Ebene des Gesamtsystems wurden einige Signale deklariert, die die Verbindungen zwischen den Submodulen herstellen. Alle diese Signale sind entweder Ausgänge von Submodulen oder kommen von der IPort- oder DPort-Schnittstelle.

Signalname	Bitbreite	Beschreibung	Herkunft
`rd`	5	Zielregister	`IR[11:7]`
`rs1`	5	Quellregister `rs1`	`IR[19:15]`
`rd`	5	Quellregister `rs2`	`IR[24:20]`
`funct3`	3	`funct3`-Block	`IR[14:12]`
`funct7`	1	Reduzierter `funct7`-Block	`IR[30]`
`bsel`	4	Bysteselect-Signal	Ausgang `bsel_out` des Byteselectors
`imm_out`	32	32-bit Direktwert	Ausgang `imm_out` des Immediate-Generators
`rs1_out`	32	Regfile Ausgang	Ausgang `rs1_out` des Regfiles
`rs2_out`	32	Regfile Ausgang	Ausgang `rs2_out` des Regfiles
`alu_out`	32	Ausgang der ALU	Ausgang `y_out` der ALU
`extend_out`	32	Ausgang der ALU	Ausgang `extend_out` des Extenders.
`PC`	32	Programmzähler	Ausgang `pc_out` des PC
`dport_rdata`	32	`rdata`-Signal des DPort	Von DPort
`iport_rdata`	32	`rdata`-Signal des DPort	Von IPort
`iport_wdata`	32	`wdata`-Signal des DPort	Ausgang `data_out` des Datahandlers

Tabelle 5.1: Signale des minimalen Nucleus

Eine Besonderheit ist das `opcode`-Signal. Wie der Tabelle 2.1.3 zu entnehmen ist, ist der Opcode in den sieben unteren Bits eines Befehlswortes kodiert. Jedoch genügt es für diese Implementierung die *fünf Bits IR[6:2]* als `opcode`-Signal zu verwenden, da alle `RV32I` Opcodes den Wert binär `11` in den untersten zwei Bits haben (siehe A.1).

Das `funct7`-Signal kann ebenfalls reduziert werden. Betrachtet man die Tabelle A.1 wird ersichtlich, dass das `func7`-Signal nur zwei Werte annehmen kann: `0x0` und

0x20. Da das Signal den obersten 7 Bits eines Befehlswortes entspricht und hexadezimal 0x20 binär 0100000 entspricht, kann man das funct7-Signal auf IR[30] reduzieren. Diese Aussage gilt für Befehle der Typen R und I.

5.5 Submodule des minimalen Nucleus

Der minimale Nucleus besteht aus 9 Submodulen. Deren Funktion, Aufbau und Besonderheiten werden im Folgenden beschrieben.

5.5.1 ALU (Arithmetic logic unit)

Das ALU Submodul (im Weiteren nur „die ALU“) führt logische und arithmetische Befehle aus. Aus den Operanden „A“ und „B“ wird anhand des momentanen Befehlswortes das Ergebnis „Y “ gebildet. Die Tabellen 5.2 und 5.3 zeigen die Ein- und Ausgangsports der ALU.

Portname	Bitbreite	Beschreibung
a_in	32	Operand A.
b_in	32	Operand B.
funct3_in	3	Kontrollsignal zur Wahl der Operation. Entspricht IR[14:12]
funct7_flag_in	1	Kontrollsignal zur Unterscheidung der ADD/-SUB und SRA/SRL Befehle. Entspricht IR[30].
force_add_in	1	Zwingt die ALU zur Addition der Operanden.

Tabelle 5.2: Eingangsports des ALU Moduls

Portname	Bitbreite	Beschreibung
y_out	32	Ergebnis Y.
equal_out	1	Statussignal, gesetzt wenn A == B sonst 0.
less_out	1	Statussignal, gesetzt wenn A < B sonst 0.
lessu_out	1	Statussignal, gesetzt wenn (unsigned) A < (unsigned) B sonst 0.

Tabelle 5.3: Ausgangsports des ALU Moduls

Anhand Eingangssignalwerte führt die ALU die in der Tabelle 5.4 aufgelisteten Operationen aus. Das Operationsergebnis Y wird am Ausgangsport y_out ausgegeben.

Operation	Kurzform	funct3_in	funct7_flag_in	Beschreibung
ADD	$y = a + b$	0x0	0	Addition
SUB	$y = a - b$	0x0	1	Subtraktion
AND	$y = a\&b$	0x7	0	Bitweises UND
OR	$y = a \mid b$	0x6	0	Bitweises ODER
XOR	$y = a \oplus b$	0x4	0	Bitweises XOR
SLL	$y = a \ll b[4:0]$	0x1	0	Logisches Linksschieben
SRL	$y = a \gg b[4:0]$	0x5	0	Logisches Rechtsschieben
SRA	$y = a \gg b[4:0]$	0x5	1	Arithmetisches Rechtsschieben (Um Vorzeichen erweitert)
SLT	$y = a < b?1:0$	0x2	0	Kleiner als
SLTU	$y = a < b?1:0$	0x3	0	Kleiner als (Ohne Vorzeichen, um Null erweitert)

Tabelle 5.4: ALU Operationen

Eine Besonderheit ist die Begrenzung des Operanden `B` auf `b_in[4:0]` bei den Schiebebefehlen `SLL`, `SRL` und `SRA`. Diese Begrenzung ist durch die RISC-V-Spezifikation [22] vorgegeben. Das Schieben von 32-bit Wörtern um mehr als 32 Stellen in eine Richtung ist nicht sinnig.

Das Modul enthält keine Register und besteht aus rein kombinatorischer Logik.

5.5.2 Regfile (Register file)

Das „Register file“-Submodul (im Weiteren nur „`das Regfile`“) enthält 32 32-bit Register die als Zwischenspeicher für Rechenergebnisse und Adressen dienen. Die Tabellen 5.5 und 5.6 zeigen die Ein- und Ausgangsports des Moduls.

Portname	Bitbreite	Beschreibung
`data_in_in`	32	Dateneingang.
`select_in`	5	Wählt das Register aus, in das Daten gespeichert werden.
`rs1_select_in`	5	Wählt das Register aus, dessen Wert an den Ausgangsport `rs1_out` ausgegeben wird.
`rs2_select_in`	5	Wählt das Register aus, dessen Wert an den Ausgangsport `rs2_out` ausgegeben wird.
`en_load_in`	1	Steuersignal zum Aktivieren oder Deaktivieren der Speicherung von Eingangsdaten.
`clk`	1	Taktsignal
`reset`	1	Low-aktiver Reset

Tabelle 5.5: Eingangsports des Regfile Moduls

Portname	Bitbreite	Beschreibung
`rs1_out`	32	Wert des Registers, das durch `rs1_select_in` ausgewählt wurde.
`rs2_out`	32	Wert des Registers, das durch `rs2_select_in` ausgewählt wurde.

Tabelle 5.6: Ausgangsports des Regfile Moduls

Das Regfile besteht aus kombinatorischer und taktsynchroner Logik. Dies bedeutet, dass sich die Werte der internen Register erst zur nächsten steigenden Taktflanke ändern. Die Ausgangsports `rs1_out` und `rs2_out` haben konstant den Wert der durch die Eingangsignale `rs1_select_in` und `rs2_select_in` gewählten Register. Alle Register haben den Resetwert `0x00000000`.

5.5.3 PC (Program counter)

Das „Program counter “-Submodul (im Weiteren nur „`der PC`“) ist ein 30-bit Register. Zweck dieses Moduls ist es, die Speicheradresse des aktuellen Befehls zu speichern. Die Tabellen 5.7 und 5.8 zeigen die Ein- und Ausgangsports des Moduls.

Portname	Bitbreite	Beschreibung
`pc_in`	32	Dateneingang.
`inc_in`	1	Steuersignal. Wenn gesetzt, wird der PC zur nächsten steigenden Taktflanke um `0x4` erhöht.
`en_load_in`	1	Steuersignal. Wenn gesetzt, übernimmt das interne Register zur nächsten steigenden Taktflanke den Wert, der am Port `pc_in` anliegt.
`clk`	1	Taktsignal
`reset`	1	Low-aktiver Reset

Tabelle 5.7: Eingangsports des PC Moduls

Portname	Bitbreite	Beschreibung
`pc_out`	32	Konstanter Ausgang des internen Registers.

Tabelle 5.8: Ausgangsports des PC Moduls

Es ist zu beachten, dass das interne Register des PC *30 Bit* anstatt 32 Bit breit ist. Der Grund hierfür ist, dass nach der RISC-V-Spezifikation der Programmspeicher „4-byte aligned“ sein muss (siehe 5.2). Dies bedeutet hier konkret, dass die untersten 2 Bit nicht von Relevanz sind und somit weggelassen werden können. Der Ausgang `pc_out` des `PC` ist jedoch *32 Bit* breit. Die unteren zwei Bit werden konstant auf 0 gesetzt. Dies wurde entschieden, um Verwirrung bei den Bitbreiten an den Eingängen anderer Module zu vermeiden, welche `pc_out` als Eingang haben. Somit muss das erweitern von 30 auf 32 Bit auch nur an einer Stelle geschehen, anstatt in jedem weiteren Submodul. Der Eingangsport des `PC` ist um Konsistenz zu wahren ebenfalls 32 Bit breit.

Der PC enthält kombinatorische und taktsynchrone Logik. Der Wert des internen Registers nimmt erst zur nächsten steigenden Taktflanke den Wert des Eingangssignals `pc_in` an, wenn `en_load_in` gesetzt ist. Das Signal `pc_out` hat konstant den Wert des internen Registers. Der Resetwert (und damit die Startadresse) ist über das PicoNut-Build-System konfigurierbar.

5.5.4 IR (Instruction register)

Das „Instruction Register “-Submodul (im Weiteren nur „`das IR`“) ist ein 32-bit Register. Es dient dazu, das aktuelle Befehlswort zu speichern und für die anderen Submodule des Nucleus in Form von weiteren Einzelsignalen bereitzustellen (siehe 5.4.2). Die Tabellen 5.9 und 5.10 zeigen die Ein- und Ausgangsports des Moduls.

Portname	Bitbreite	Beschreibung
`ir_in`	32	Eingangsdaten. Das interne Register übernimmt diesen Wert zur nächsten steigenden Taktflanke.
`en_load_in`	1	Steuersignal. Wenn gesetzt, übernimmt das interne Register zur nächsten steigenden Taktflanke den Wert, der am Port ‘ir_in‘ anliegt.
`clk`	1	Taktsignal
`reset`	1	Low-aktiver Reset

Tabelle 5.9: Eingangsports des IR Moduls

Portname	Bitbreite	Beschreibung
`ir_out`	32	Konstanter Ausgang des internen Registers.

Tabelle 5.10: Ausgangsports des IR Moduls

Der Signalwert am Port `ir_in` wird zur nächsten steigenden Taktflanke in das interne Register übernommen. Der Ausgangsport `ir_out` ist konstant der Wert des internen Register. Der Resetwert ist `0x00000000`.

5.5.5 Immgen (Immediate generator)

Das „Immediate generator “-Submodul (im Weiteren nur „`der Immediate-Generator`“) ist ein kombinatorisches Modul, welches die in Befehlen enthaltenen Direktwerte dekodiert und als 32-bit-Wert konstant an seinem Ausgangsport ausgibt. Die Tabellen 5.11 und 5.12 zeigen die Ein- und Ausgangsports des Moduls.

Portname	Bitbreite	Beschreibung
`data_in`	32	Befehlswort, aus dem der Direktwert dekodiert werden soll.

Tabelle 5.11: Eingangsports des Immediate-Generator Moduls

Portname	Bitbreite	Beschreibung
`imm_out`	32	Direktwert, der aus einem Befehlswort dekodiert wurde.

Tabelle 5.12: Ausgangsports des Immediate-Generator Moduls

Da die sich Kodierung der Direktwerte zwischen den RISC-V-Befehlsformaten unterscheidet, liest das Modul den im Eingangssignal `data_in` enthaltenen `opcode` und

erkennt so, welche Dekodierungsmethode angewendet werden muss. Befehle des `R-Typs` enthalten keine Direktwerte. Die Abbildung 5.2 zeigt wie Direktwerte der verschiedenen Befehlskodierungen aus dem Befehlswort `inst[31:0]` gebildet werden. Aus Platzgründen haben einzelne 1-bit Felder keine Beschriftung mit `inst[x]`. Diese Felder mit der Beschriftung `[x]` haben ebenfalls ebenfalls die Quelle `inst[31:0]`.

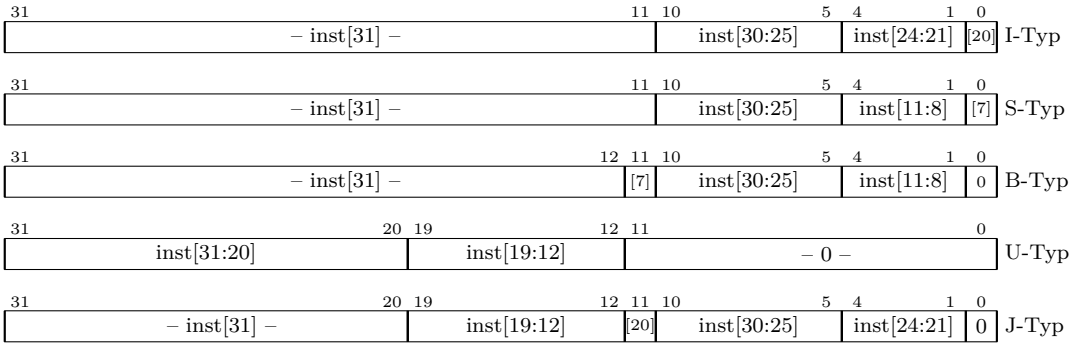

Abbildung 5.2: RISC-V Direktwert-Dekodierung nach Kodierungstyp [22]

Wie in der Abbildung 5.2 zu erkennen ist, werden Direktwerte durch Aufspalten und Restrukturieren des eingehenden Befehlswortes gebildet.

Der Immediate-Generator besteht aus rein kombinatorischer Logik.

5.5.6 Byteselector

Das „Byteselector“-Modul (im Weiteren nur „der Byteselector“) Modul generiert das Signal `bsel`. Das `bsel`-Signal ist notwendig, um das Ausführen von Byte-und Halbwort-Speichertransaktionen zu ermöglichen. Die Tabellen 5.13 und 5.14 zeigen die Ein- und Ausgangsports des Moduls.

Portname	Bitbreite	Beschreibung
`adr_in`	2	Die niedrigsten zwei Bits des `alu_out` Signals, die die Ausrichtung einer berechneten Adresse darstellen.
`funct3_in`	3	`funct3` Block des aktuellen Befehls.

Tabelle 5.13: Eingangsports des Byteselector Moduls

Portname	Bitbreite	Beschreibung
`bsel_out`	4	Ausgehendes Byteselect Signal.
`invalid_out`	1	Gesetzt wenn Addresse invalide ist.

Tabelle 5.14: Ausgangsports des Byteselector Moduls

Da Speichertransaktionen stets eine „4-byte algined“ (4-byte ausgerichtete) 32-bit Adresse verwenden, muss ein Zusätzlicher Mechanismus implementiert werden um auch einzelne Bytes und Halbwörter in den Speicher zu schreiben oder aus diesem zu lesen. „4-byte aligned“ bedeutet, dass die untersten zwei Bits der Adresse stets 0 sind. Das `bsel`-Signal ist 4 Bit breit und kodiert, welche der vier Bytes eines 32-Bit Wortes bei einer Speichertransaktion als valide zu betrachtet werden. Die 4 Bit des `bsel`-Signals entsprechen je einem Byte des Datenwortes. Hat ein Bit den Wert logisch `1` ist das zugehörige Byte valide, sonst nicht.

Das `bsel`-Signal wird anhand der Signale `funct3` und `adr_in` generiert. Das `funct3`-Signal erlaubt die Unterscheidung zwischen Byte-, Halbwort- und Wort-Transaktionen und legt die Anzahl der gesetzten Bits fest. Das `adr_in`-Signal entspricht den untersten zwei Bits der Zieladresse. Zwar sind diese Bits für den tatsächlichen Speicherzugriff nicht relevant, jedoch sie sind relevant, um einzelne Bytes und Halbwörter auch in die Oberen Bytes einer Zieladresse zu schreiben. Die RISC-V-Spezifikation gibt vor, dass der Hauptspeicher byte-adressierbar sein muss. Daraus folgt, dass alle möglichen Adressen natürlich „byte-aligned“ sind. Weiterhin sind alle Adressen mit den Werten `0x01` und `0x10` in den untersten zwei Bits „halfword aligned“. Dies bedeutet, dass alle 16 Bit (beginnend an der Speicheradresse `0x00000000`) ein Halbwort liegt.

Die Tabelle 5.15 zeigt die möglichen Kombinationen der Ein- und Ausgangssignale des Bysteselector-Moduls.

adr_in	funct3_in	bsel_out	invalid_out
00	lb/lbu/sb (0x0 oder 0x4)	0001	0
01	lb/lbu/sb (0x0 oder 0x4)	0010	0
10	lb/lbu/sb (0x0 oder 0x4)	0100	0
11	lb/lbu/sb (0x0 oder 0x4)	1000	0
00	lh/lhu/sh (0x1 oder 0x5)	0011	0
10	lh/lhu/sh (0x1 oder 0x5)	1100	0
01	lh/lhu/sh (0x1 oder 0x5)	0000	1
11	lh/lhu/sh (0x1 oder 0x5)	0000	1
00	lw/sw (0x2)	1111	0
01	lw/sw (0x2)	0000	1
10	lw/sw (0x2)	0000	1
11	lw/sw (0x2)	0000	1

Tabelle 5.15: Byteselector Generierungstabelle

Der Byteselector besteht aus rein kombinatorischer Logik.

5.5.7 Extender

Das „extender“-Modul (im Weiteren nur „der Extender) erweitert aus dem Speicher geladene Werte auf 32-bit, wenn nötig. Die Tabellen 5.16 und 5.17 zeigen die Ein- und Ausgangsports des Extenders.

Portname	Bitbreite	Beschreibung
`data_in`	32	Eingehendes Datenwort.
`funct3_in`	3	`funct3`-Block des aktuellen Befehls.
`bsel_in`	4	Byteselect-Signal.

Tabelle 5.16: Eingangsports des Extender Moduls

Portname	Bitbreite	Beschreibung
`extend_out`	32	Verarbeitetes Datenwort.

Tabelle 5.17: Ausgangsports des Extender Moduls

Wie bereits im vorherigen Abschnitt 5.5.6 beschrieben, unterstützt der minimale Nucleus das laden von Bytes und Halbwörtern aus dem Speicher.

Speicherladetransaktionen liefern immer einen 32-Bit Wert. Deshalb ist es notwendig, den validen Datenanteil eines geladenen 32-Bit Wortes auf ein insgesamt valides 32-Bit Wort zu erweitern. Valide bedeutet in diesem Zusammenhang, dass der gewünschte Dateninhalt mit dem LSB des Datenwortes beginnt. Bei eingehenden Datenwörtern aus dem Speicher, ist dies Aufgrund von Byte- und Halbwort-Ladetransaktionen nicht immer der Fall.

Der Extender nutzt das vom Byteselector 5.5.6 generierte `bsel`-Signal, welches vorgibt, welches Byte des eingehenden Datenwortes als valide zu betrachten sind. Mit dieser Information werden die validen Bytes des Datenwortes in die unteren Bytes des ausgehenden Datenwortes verschoben. Hat `bsel` den Wert `1111` wird das gesamte Datenwort ohne Änderungen übernommen, da alle Bytes valide sind.

Die Tabelle 5.18 zeigt die Funktion des Extenders exemplarisch.

Eingehendes Datenwort	bsel	Ausgabe
0x12345678	0001	0x00000078
0x12345678	0010	0x00000056
0x12345678	0100	0x00000034
0x12345678	1000	0x00000012
0x12345678	0011	0x00005678
0x12345678	1100	0x00001234

Tabelle 5.18: Beispielhafte Extender-Ausgabetabelle

Zusätzlich verwendet der Extender das `funct3`-Signal, um zu entscheiden ob eine Vorzeichenerweiterung durchgeführt werden muss. Für die Vorzeichenerweiterung wird bei Byte-Zugriffen das achte Bit des bereits im vorherigen Schritt angepassten Datenworts direkt auf die oberen 24 Bit abgebildet. Analog geschieht es bei Halbwortzugriffen, nur wird das sechzehnte Bit abgebildet.

Der Extender besteht aus rein kombinatorischer Logik. Das eingehende Datenwort wird in seine vier Bytes aufgetrennt und valide Bytes werden entsprechend dem `bsel`-Signal in die unteren Bytes des ausgehenden Datenwortes eingefügt. Anhand des `funct3`-Signals wird das resultierende Datenwort eventuell noch um ein Vorzeichen erweitert.

5.5.8 Datahandler

Das „datahanlder“-Modul (im Weiteren nur „der Datahandler“) erfüllt eine Ähnliche Funktion wie das im Abschnitt 5.5.7 beschriebene Extender-Modul. Die RISC-V-Spezifikation gibt vor, dass der valide Dateninhalt eines internen Registers immer die unteren Bits dessen besetzt. Um das speichern eines Datenwertes in ein bestimmtes Byte im Hauptspeicher zu ermöglichen, muss das ausgehende 32-bit Datenwort entsprechend manipuliert werden. Die Funktion der Module Datahandler und Extender ist somit fast exakt invers. Der Extender bewegt valide Dateninhalte anhand des `bsel`-Singals in die unteren Bytes eines Datenwortes, der Datahandler bewegt Dateninhalte aus den unteren Bytes anhand des `bsel`-Signals in die Position der validen Bytes.

Die Tabellen 5.19 und 5.20 zeigen Ein- und Ausgangsports des Datahanlders.

Portname	Bitbreite	Beschreibung
`data_in`	32	Eingehendes Datenwort.
`bsel_in`	4	Byteselect-Signal.

Tabelle 5.19: Eingangsports des Datahandler Moduls

Portname	Bitbreite	Beschreibung
`data_out`	32	Verarbeitetes Datenwort.

Tabelle 5.20: Ausgangsports des Datahandler Moduls

Die Tabelle 5.21 zeigt die Funktion des Datahandler-Moduls exemplarisch.

Eingehends Datenwort	bsel	Ausgabe
0x00000012	1000	0x12000000
0x00000012	0100	0x00120000
0x00000012	0010	0x00001200
0x00000012	0001	0x00000012
0x00001234	1100	0x12340000
0x00001234	0011	0x00001234
0x12345678	1111	0x12345678

Tabelle 5.21: Exemplarische Datahandler-Ausgabetabelle

Der Datahandler bestet aus rein kombinatorischer Logik. Die durch das `bsel`-Signal markierten validen Bytes des eingehenden Datenwortes werden neu angeordnet und bilden das ausgehende Datenwort.

5.5.9 Controller

Die bisher beschriebenen Submodule des minimalen Nucleus bilden dessen Operationswerk. Das „controller“-Modul (im weiteren nur „der Controller“) ist dessen Steuerwerk. Der Controller ist ein Moore-Automat mit zweiundzwanzig Zuständen. Er hat sieben Eingangssignale, sogenannte *Statussignale* („status signals“).
Vier dieser Statussignale sind Ausgänge anderer Teilodule des Nucleus und zwei der Statussignale sind Busleitungen der IPort/DPort-Schnittstelle und damit Eingänge des minimalen Nucleus. Anhand der Statussignale und dem aktuellen Zustand wird der nächste Zustand bestimmt. Je nach Zustand wird eine Menge an sogenannten *Steuersignalen* („control signals“) auf logisch `1` oder `0` gesetzt. Diese Signale, fünfzehn an der Zahl, sind die Ausgänge des Controllers. Zwölf diesr Signale sind mit

dem Schaltnetz verbunden. Die übrigen drei Steuersignale sind Ausgänge des minimalen Nucleus in Form von IPort/DPort-Bussignalen.

Die Tabellen 5.22 und 5.23 zeigen die Ein- und Ausgangsports des Controllers. Die Tablle der Ausgangsports verzichtet auf die Spalte „Bitbreite“, da alle Steuersignale nur ein Bit breit sind.

Signalname	Bitbreite	Beschreibung
`s_opcode_in`	5	**opcode** des aktuellen Befehls.
`s_funct3_in`	3	**funct3**-Block des aktuellen Befehls.
`s_alu_less_in`	1	Statussignal der **ALU**.
`s_alu_lessu_in`	1	Statussignal der **ALU**.
`s_alu_equal_in`	1	Statussignal der **ALU**.
`s_dport_ack_in`	1	**ack**-Signal der IPort-Schnittstelle.
`s_iport_ack_in`	1	**ack**-Signal von DPort-Schnittstelle.
`clk`	1	Taktsignal
`reset`	1	Low-aktiver Reset

Tabelle 5.22: Eingangsports des Controller Moduls (Statussignale)

Signalname	Beschreibung
`c_iport_stb`	IPort **stb**-Signal.
`c_dport_stb`	DPort **stb**-Signal.
`c_dport_we`	IPort **we**-Signal.
`c_reg_ld_en`	Aktiviert laden des **Regfiles** mit Wert am Eingang.
`c_reg_ldpc`	Setzt den Eingang des Regfiles auf **pc_out**.
`c_reg_ldmem`	Setzt den Eingang des Regfiles auf **dport_rdata_in**.
`c_reg_ldimm`	Setzt den Eingang des Regfiles auf **imm_out**.
`c_reg_ldalu`	Setzt den Eingang des Regfiles auf **alu_out**.
`c_alu_pc`	Setzt den Eingang **A** der **ALU** auf **pc_out**.
`c_alu_imm`	Setzt den Eingang **B** der **ALU** auf **imm_out**.
`c_force_add`	Zwingt die **ALU** zur Additionsoperation.
`c_funct7_flag`	Steuersignal für die ALU.
`c_pc_inc4`	Aktiviert inkrementieren des PC um `0x4`.
`c_pc_ld_en`	Aktiviert laden des **PC** mit Wert am Eingang.
`c_ir_ld`	Aktiviert laden des **PC** mit Wert am Eingang.

Tabelle 5.23: Ausgangsports des Controller Moduls (Steuersignale)

Die Abbildung 5.3 ist ein Zustandübergansdiagramm des Controllers. Die Übergangsbedingungen sind jeweils an den Flaken abgebildet. Es wurde auf das Erstellen

großer Tabellen zur Darstellung der Übergangsbedingungen und den Folgezuständen verzichtet, um diesen Abschnitt übersichtlicher zu gestalten. Innerhalb der einzelnen Zustände zeigt die Abbildung, welche Steuersignale auf logisch `1` gesetzt werden, wenn sich der Automat in diesem Zustand befindet. Alle anderen Steuersignale haben dann den Wert logisch `0`. Die in grün geschriebenen Bedingungen an den Flanken des Diagramms sind logische Ausdrücke und wenn sie zu `true` (oder `1`) evaluieren, ist der Zustand entlang der Flanke der Folgezustand.

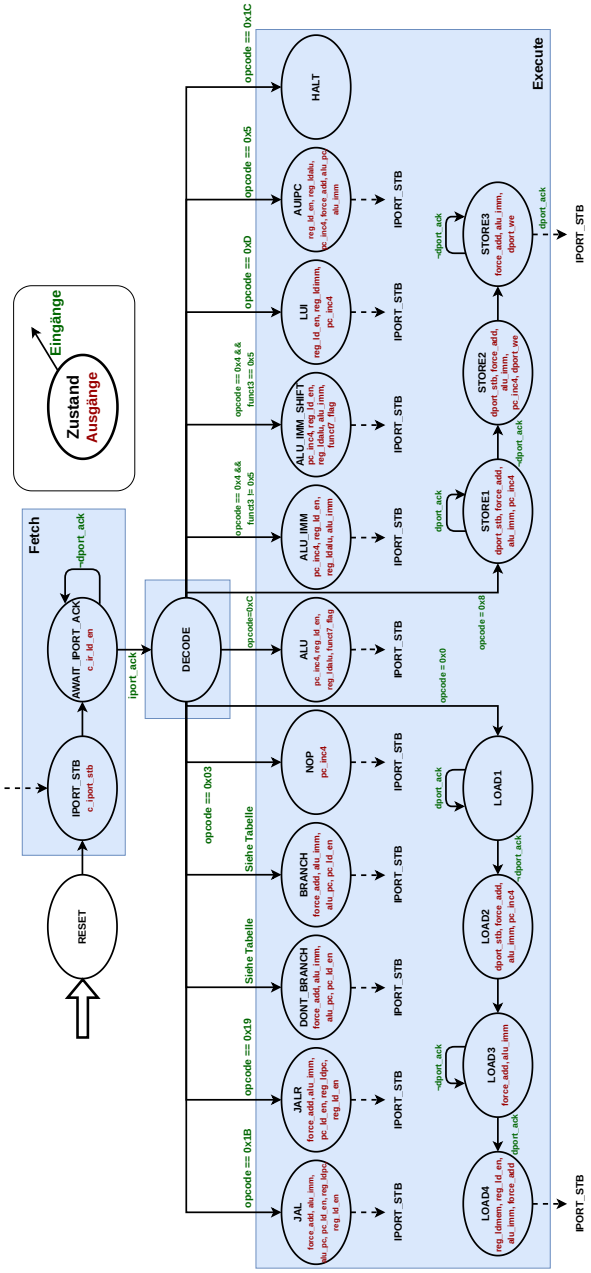

Abbildung 5.3: Controller Zustandsdiagramm

Da die Flanken zu den Zuständen `BRANCH` und `DONT_BRANCH` mehrere Bedingungen enthalten, sind diese in der Folgenden Tabelle 5.24 aufgelistet.

Branch-Typ	Bedingung
`BEQ` („Branch equal")	`funct3 == 0x0 && s_alu_equal == 1`
`BNE` („Branch not equal“)	`funct3 == 0x1 && s_alu_equal == 0`
`BLT` („Branch less than“)	`funct3 == 0x4 && s_alu_less == 1`
`BGE` („Branch greater or equal“)	`funct3 == 0x5 && s_alu_less == 0`
`BLTU` („Branch less than (unsigned)“)	`funct3 == 0x6 && s_alu_lessu == 1`
`BGEU` („Branch greater or equal (unsigned)“)	`funct3 == 0x7 && s_alu_lessu == 0`

Tabelle 5.24: Branch-Bedingungen

Beginnend mit dem `RESET`-Zustand prüft der Controller zu jeder steigenden Taktflanke die Statussignale, betritt Folgezustände und setzt zugehörige Steuersignale. Die Steuersignale steuern den Signalfluss im Schaltnetz, einzelne Submodule und die IPort/DPort-Schnittstelle. Dadurch wird die im Abschnitt 5.3 beschriebene Arbeitsweise realisiert.

Der Controller besteht aus kombinatorisher und taktsynchroner Logik. Ein internes 5-bit breites Register speichert den aktuellen Zustand. Zustandsübergänge finden immer zu steigenden Flanken des Taktsignals statt. Der Resetwert des Zustandsregisters ist `0`.

5.5.10 Zusätzliche Logik außerhalb der Submodule

Neben den neun Submodulen enhält der minimale Nucleus noch zusätzliche Logik außerhalb der Submodule.

Fas Regfile hat vier mögliche Eingangsquellen: `imm_out`, `pc_out`, `ext_out` und `alu_out`. Jeder dieser Quellen ist ein Steuersignal zugeordnet (siehe 5.23). Damit aus diesen vier Möglichen Eingangsquellen anhand der vier Steuersignale gewählt werden kann, ist eine Entscheidungslogik implementiert, die einem 4:1 32-bit Multiplexer mit einem 4-Bit `select`-Eingang entpsricht.

Ähnlich verhält es sich mit den Eingängen der ALU. Ist das Steuersignal `c_alu_pc` gesetzt, ist der ALU-Operand `A` (`a_in`) gleich `pc_out`, sonst `rs1_out`. Ist `c_alu_imm` gesetzt, ist der ALU-Operand `B`(`b_in`) gleich `imm_out`, sonst `rs2_out`.

Um redundante Hardware zu vermeiden, wurde die ALU so konstruiert, dass sie Register-Register-Befehle und Register-Immediate-Befehle zugleich ausführen kann.

Die beiden Befehlstypen unterscheiden sich jedoch durch das Fehlen eines `subi`-Befehls („subtract immediate“). Register-Register-Befehle nutzen das `funct7`-Signal um zwischen Subtraktion und Addition sowie zwischen arithmetischem und logischem Schieben zu unterscheiden. Somit darf das `funct7`-Signal nicht direkt an die ALU weitergeleitet werden, da sonst ein fehlerhafter `subi`-Befehle entstehen würde. Jedoch ist das Signal notwendig um zwischen den Befehlen `slri` und `srai` zu unterscheiden. Dieses Problem wurde durch einen zusätzlichen Schaltwerkszustand `ALU_SHIFT_IMM` für Schiebebefehle mit Direktwert (siehe 5.3) und das zusätzliche Steuersignal `c_funct7_flag` gelöst. Wenn Steuersignal `funct7_flag` gesetzt ist, wird das globale `funct7`-Signal auf den Wert `IR[30]` gesetzt. Ist es nicht gesetzt, ist der Wert dessen `0`. Der Zustand `ALU_SHIFT_IMM` setzt dieses Signal. Der Zustand `ALU_IMM` für alle übrigen Direktwertbefehle setzt es nicht.

Die Abbildung 5.1 zeigt ein Symbol mit der Beschriftung „+4“ im Pfad des `pc_out`-Signals hin zum Eingang des Regfiles. Hierbei handelt es sich um einen Addierer der das `pc_out`-Signal um `0x4` erhöht. Zwar hat der PC bereits einen internen Addierer um seinen Registerwert zu inkrementieren, jedoch geschieht dies erst zur nächsten steigenden Taktflanke. Der zusätzliche Addierer ist notwendig um die unbedingten Sprungbefehle `jal` und `jalr` in einem einzigen Taktzyklus auszuführen, da diese den aktuellen Wert des PC um `0x4` inkrementiert im Regfile abspeichern und gleichzeitig einen neuen Wert in den PC laden müssen.

5.6 Das Ausführen von Befehlen

Die vorherigen Abschnitte dieses Kapitels dienten dazu, den Aufbau des minimalen Nucleus aus einen Submodulen, wie er zu einem Gesamtsystem zusammengeführt wurde und welche Besonderheiten dabei entstanden sind. In diesem Abschnitt wird dargestellt, wie dieses Gesamtsystem alle `RV32I`-Befehle ausführen kann (mit Ausnahme der Befehle `ecall`, `ebreak` und `fence`).

5.6.1 Befehl laden („fetch“)

Das Laden eines Befehls beginnt mit dem Zustand `IPORT_STB`. Der Controller setzt das `stb`-Signal des IPort, geht in den Zustand `AWAIT_IPORT_ACK` über und setzt `c_ir_en_load` um das Laden des IR zu aktivieren. Die angefragte Adresse entspricht dem Wert des PC. Der Controller wartet in diesem Zustand, bis das IPort `ack`-Signal durch die MemU gesetzt wurde. Sobald dieses eintrifft, liegt ein valides 32-Bit Befehlswort am Eingang des IR an welches zur nächsten steigenden Taktflanke

in dieses übernommen und dort gehalten wird. Der Controller geht in den Zustand `DECODE` über.

5.6.2 Befehl dekodieren („decode“)

Im Zustand `DECODE`, evaluiert der Controller die Signale `opcode`, `func3`, `s_alu_equal`, `s_alu_less` und `s_alu_lessu`. Anhand dieser wird der zugehörige Folgezustand betreten, wie sie in der Abbildung 5.3 an den Flanken und in der Tabelle 5.24 beschrieben sind. Alle in der Tabelle 5.5.10 beschriebenen Signale sind jetzt für diesen Befehlszyklus valide.

5.6.3 Befehl ausführen („execute“)

Um Redundanz zu vermeiden, ist anzunehmen, dass das Steuersignal `c_pc_inc4` in jedem Zustand gesetzt ist, außer es wird explizit als nicht gesetzt angegeben. Es aktiviert das Inkrementiern des PC um `0x4` zur nächsten steigenden Flanke und setzt diesen damit auf die Adresse des nächsten Befehls im Speicher.

5.6.3.1 Register-Register-Befehle

Für das Ausführen der Register-Register-Befehle `add`, `sub`, `xor`, `or`, `and`, `sll`, `srl`, `sra`, `slt` und `sltu` betritt der Controller den Zustand `ALU`. Die Steuersignale `c_reg_ld_en` und `c_reg_ldalu` werden gesetzt. Sie aktivieren das Laden des Regfiles und setzen dessen Eingang auf `alu_out`. Somit wird das Ergebnis der ALU-Operation zur nächsten steigenden Taktflanke im Regfile gespeichert. `c_funct7_flag` wird gesetzt um die Unterscheidung zwischen `add` und `sub` sowie `sra` und `srl` zu erlauben.

5.6.3.2 Register-Immediate-Befehle

Für das Ausführen der Register-Immediate-Befehle `addi`, `xori`, `ori`, `andi`, `slli`, `slti` und `sltiu` betritt der Controller den Zustand `ALU_IMM`. Das Steuersignal `c_alu_imm` wird gesetzt. Dadurch wird der ALU Operand `B` zu `imm_out`. `c_reg_ld_en` und `c_reg_ldalu` werden zum Abspeichern des Ergebnisses gesetzt.

Die Befehle `srli` und `srai` werden im Zustand `ALU_IMM_SHIFT` ausgeführt. Die Steuersignale werden analog zum `ALU_IMM`-Zustand gesetzt, jedoch wird zusätzlich `c_funct7_flag` gesetzt (siehe 5.5.10).

Der Befehl `lui` wird im Zustand `LUI` ausgeführt. Er dient dazu, 32-Bit Konstanten zu bilden [22] Der Direktwert des Befehls wird in das Register `rd` geladen. Hierzu werden die Steuersignale `c_reg_ld_en` und `c_reg_imm` gesetzt.

Der Befehl `AUIPC` dient dazu PC-relative Adressen zu generieren [22]. Durch Das Setzen von `c_alu_pc` und `c_alu_imm`, sowie `c_force_add` werden der Direktwert des Befehls und der PC addiert. Mit `c_reg_ld_en` und `c_reg_ldalu` wird dieses Ergebnis in das Regfile gespeichert.

5.6.3.3 Kontrollflussbefehle

Die Kontrollflussbefehle vom Branch-Typ `beq`, `bne`, `blt`, `bge`, `bltu` und `bgeu` werden, wenn ihre Bedingungen (siehe 5.24) erfüllt sind, im Zustand `BRANCH` ausgeführt. `c_pc_inc4` ist *nicht* gesetzt. Das im Direktwert kodierte „branch offset“ wird auf den PC addiert. Hierzu werden `c_force_add`, `c_alu_pc` und `c_alu_imm` gesetzt. Die resultierende Sprungadresse wird durch das Setzen von `pc_ld_en` zur nächsten steigenden Taktflanke im PC gespeichert. Jetzt hat der PC den Wert der Sprungadresse.

Sollte die Bedinung um den `BRANCH`-Zustand zu betreten nicht erfüllt sein, wird der Zustand `DONT_BRANCH` betreten, in dem lediglich der PC um `0x4` inkrementiert wird. `c_pc_inc4` ist wiederrum gesetzt.

Der unbedingte Sprungbefehl `jal` wird im Zustand `JAL` ausgeführt. Das Addressoffset `imm_out` wird mit `c_force_add` auf den PC addiert und bildet somit die Zieladresse. Die Rücksprungadresse (PC+4) wird durch setzen von `c_reg_ldpc` und `c_reg_ld_en` in das Zielregister `rd` gespeichert. `c_pc_inc4` ist *nicht* gesetzt.

Der unbedingte, indirekte Sprungbefehl `jalr` gleicht in der Ausführung dem Befehl `jal`, mit dem einzigen Unterschied, dass der PC mit der Summe aus einem Registerwert und einem Direktwert geladen wird. Er wird im Zustand `JALR` ausgeführt. `c_pc_inc4` ist `nicht` gesetzt.

5.6.3.4 Speichertransaktionen

Speichertransaktionen können nicht in einem einzigen Taktzyklus ausgeführt werden, da mit ihnen eine Bustransaktion einhergeht.

Die Ausführung der `load`-Befehle `lb`, `lh`, `lbu`, `lhu` und `lw` beginnt im Zustand `LOAD1`. In diesem Zustand wartet der Controller, bis das Bussignal `stb` des DPorts den Wert logisch `0` hat. Dieser Schritt ist notwendig um eventuelle Buskollisionen zu vermeiden. Ist dies so, geht er in den Zustand `LOAD2` über. Die Quelladresse der

Ladetransaktion wird durch Addieren des im Direktwert enthaltenen Offsets mit dem Register `rs1` erhalten. Hierzu werden `c_force_add` und `c_alu_imm` gesetzt. In diesem Zustand wird der PC durch setzen von `c_pc_inc4` inkrementiert. Der Controller betritt ohne Bedingung den Zustand `LOAD3`, in welchem er auf das setzen `ack`-Signals durch die MemU wartet und die Adresse durch setzen der Steuersignale `c_force_add` und `c_alu_imm` hält. Wenn das `ack`-Signal eintrifft, geht der Controller in den Zustand `LOAD4` über. Erst jetzt kann das valide anliegende Datenwort durch setzen von `c_reg_ldmem` und `c_reg_ld_en` in das Zielregister `rd` geladen werden. Dieser Zusätzliche Schritt ist notwendig um fehlerhaftes Verhalten zu vermeiden, wenn das Zielregister `rd` und das Quellregister `rs1` gleich sind. Danach geht der Controller in den Zustand `IPORT_STB` über.

Die `store`-Befehle `sb`, `sh` und `sw` beginnen ihre Ausführung im Zustand `STORE1`. In diesem Zustand wird ähnlich wie bei den `load`-Befehlen geprüft, ob `dport_ack` den Wert `0` hat. Ist dies der Fall, geht der Controller in den Zustand `STORE2` über. Die Zieladresse der Transaktion wird durch setzen von `c_force_add` und `alu_imm` berechnet. Zusätzlich wird `c_dport_we` gesetzt, um der MemU zu signalisieren, dass es sich um eine Schreibransaktion handelt. In diesem Zustand wird der PC durch setzen von `c_pc_inc4` inkrementiert. Der Controller geht ohne Bedingung in den Zustand `STORE3` über. Die Zieladresse wird weiterhin gehalten und der Controller wartet auf setzen `ack`-Signals durch die MemU. Wenn dieses eintrifft, ist die Transaktion abgeschlossen und der Controller geht in den Zustand `IPORT_STB` über.

5.6.4 Sonstige Befehle

Die im Abschnitt 5.2 erwähnten, nicht unterstützten Befehle `ecall` und `ebreak` versetzen den Controller in den Zustand `HALT`. Dieser kann nicht verlassen werden, setzt keine Steuersignale und stellt somit das Ende eines Programmes dar. Der Befehl `fence` führt zum Zustand `NOP` („no operation“). In diesem Zustand wird lediglich der PC um `0x4` inkrementiert.

6 Ergebnisse

In diesem Kapitel werden die Ergebnisse dieser Arbeit dargestellt. Der Simulator des PicoNut wird vorgestellt und die Simulation eines „Hello World!“-Programms wird analysiert. Es wird dargestellt, inwiefern die korrekte Funktion des Simulators den minimalen Nucleus validiert. Weiterhin wird das Syntheseergebnis des minimalen Nucleus vorgestellt und analysiert.

6.1 Der PicoNut-Simulator

6.1.1 Aufbau des Simulators

Die SystemC Hardwarebeschreibung des minimalen Nucleus wurde mit dem GCC C++-Compiler „g++“ zu einem Softwaremodell des Hardwaresystems kompiliert. Zusammen mit der „soft_memu“ entsteht ein Simulator des PicoNut-Prozessors, welcher im Stande ist, mit GCC kompilierte C-Programme auszuführen. Die `soft_memu` ist ein Softwaremodell einer MemU mit internem Speicher. Um den Inhalt dieses Arrays an die Hardwaresimulation weiterzugeben nutzt die `soft_memu` die *C-Soft-Schnittstelle*. Die `soft_memu` wurde von Marco Milenkovic im Rahmen des PicoNut-Projekts entwickelt.

Zur ASCII-Zeichenausgabe enthält der Simulator zusätzlich ein „soft_uart“-Modul. Es ist ein in Software implementiertes UART-Modul, welches ausgehende Zeichen direkt an die C++-Standardausgabe weitergibt. Dieses Modul nutzt die **C-Soft-Schnittstelle** zur Kommunikation mit der MemU. Das `soft_uart`-Modul wurde von Lukas Bauer im Rahmen des PicoNut-Projekts entwickelt.

Der Simulator des PicoNut ist zum aktuellen Zeitpunkt aus den folgenden Komponenten aufgebaut:

- Dem PicoNut-Prozessor bestehend aus dem minimalen Nucleus und der `soft_memu`.
- Dem `soft_uart`-Modul zur Ausgabe von ASCII-Zeichen an die Konsole.

Dieser Simulator kann ein GCC-kompiliertes C-Programm einlesen und es in den Speicher der `soft_memu` legen. Von dort werden über die IPort/DPort-Schnittstelle Befehlswörter in den Nucleus geladen und Speichertransaktionen abgewickelt.

Die Simulationsumgebung terminiert, wenn der Controller des minimalen Nucleus den Zustand `HALT` erreicht. Das *clock*-Signal hat eine Periode von zehn Nanosekunden.

6.1.2 Ausführen eines „Hello World!“-Programms und Analyse des Simulationsergebnisses

Um die Funktion des minimalen Nucleus zu bestätigen, wurde der im vorherigen Abschnitt 6.1.1 beschriebene Simulator genutzt um ein „Hello World!“-Programm auszuführen (Quellcode siehe A.2). Die Ausgabe des Text-Strings erfolgte auf der Konsole A.1.

Der Simulator kann mit dem Kommando `make run-tb` im Ordner `/piconut/systems/refdesign/hw` aufgerufen werden.

Die Tabelle 6.1 zeigt eine Statistik ausgewählter Datenpunkte, die durch die Analyse des vom Simulator generierten Tracefiles erstellt wurde. Das Tracefile enthält den zeitlichen Verlauf aller Signale innerhalb der Simulation und wurde mit dem Programm `GTKWave` eingesehen.

Datenpunkt	Anzahl	Erhoben durch
Anzahl der Taktzyklen	174260	Simulationszeit in Nanosekunden geteilt durch 10
Anzahl der IPort `stb`-Pulse	21927	Steigende Flanken des Signals
Anzahl der IPort `ack`-Pulse	21927	Steigende Flanken des Signals
Anzahl der PC Änderungen	21927	Wertewechsel des Signals
Anzahl der DPort `stb`-Pulse	9263	Steigende Flanken des Signals
Anzahl der DPort `ack`-Pulse	9263	Steigende Flanken des Signals
Anzahl ausgeführter `JAL`-Befehle	817	Zustand des Controllers
Anzahl ausgeführter `JALR`-Befehle	571	Zustand des Controllers
Anzahl genommener Branches	1429	Zustand des Controllers
Anzahl nicht genommener Branches	1417	Zustand des Controllers
Anzahl der **store**-Befehle	3671	Zustand des Controllers
Anzahl der **load**-Befehle	5646	Zustand des Controllers
Anzahl der ALU-Befehle	1707	Zustand des Controllers

Tabelle 6.1: Statistik ausgewählter Datenpunkte des „Hello World!“-Trace-Files

Die Anzahl der Änderungen des `PC` sowie die Anzahl der Pulse der IPort-`stb`- und IPort-`ack`-Signale gibt Aufschluss darüber, wie viele Befehle im Laufe der Simulation ausgeführt wurden. Diese müssen an der Anzahl gleich sein. Teilt man diesen Wert (21927) durch die Anzahl der abgelaufenen Taktzyklen (174260) erhält man eine durchschnittliche Ausführungsdauer pro Befehl von *7,94* Taktzyklen. Der minimale Nucleus verzögert seine Eingänge und Ausgänge mit taktsynchronen Registern (siehe Kapitel 5.4.1). Dadurch verlangsamen sich auch die Ausführungszeiten des minimalen Nucleus. 25,7% der ausgeführten Befehle sind `load`-Befehle, welche von Beginn der `fetch`-Phase bis Ende der `execute`-Phase in elf Taktzyklen ausgeführt werden. Sieben dieser Taktzyklen sind dem Ablauf des Controllers zuzuordnen. Die übrigen vier Taktzyklen sind den jeweils um einen Taktzyklus verzögerten `ack`- und `stb`-Signalen an den IPort- und DPort-Schnittstellen zuzuordnen. *store*-Befehle werden in zehn Taktzyklen ausgeführt. *store*-Befehle durchlaufen einen Controller-Zustand weniger als *load*-Befehle, wodurch sich dieses Ergebnis bestätigt. Alle anderen Befehle werden in sechs Taktzyklen ausgeführt. Die Schritte *decode* und *execute* (ausgenommen `load` und `store`) nehmen jeweils nur einen Taktzyklus ein. Die aus zwei Controller-Zuständen bestehende *fetch*-Phase verzögert sich um zwei Taktzyklen aufgrund der bereits erwähnten Register auf insgesamt vier Taktzyklen.

6.1.3 Validierung des minimalen Nucleus

Die Ausführung des „Hello World!“-Programms mit korrekter Ausgabe auf der Konsole ist ein gutes Indiz dafür, dass der minimale Nucleus eine fehlerfreie `RV32I`-Implementierung ist. Im Laufe Programms wurden 2846 *branch*-Befehle ausgeführt (siehe Tabelle 6.1). Falsche Sprünge würden zur fehlerhaften Ausführung des Programms führen, ausgenommen, dass falsche Sprünge durch Zufall zur korrekten Ausführung führen. Das korrekte Sprungverhalten von *branch*-Befehlen ist indirekt von der korrekten Ausführung aller zuvor ausgeführten Befehle abhängig, da sie durch andere Befehle im Regfile abgespeicherte Werte nutzen. Es wird argumentiert, dass das korrekte Ausführen des ganzen Programms auf die korrekte Ausführung der *branch*-Befehle schließen lässt. Weiterhin: die korrekte Ausführung der *branch*-Befehle lässt auf die korrekte Ausführung aller zuvor ausgeführten Befehle schließen.

Es ist jedoch möglich, dass edge-case-Befehle existieren, die das „Hello World“-Programm nicht abdeckt und vom minimalen Nucleus fehlerhaft ausgeführt werden. Zur vollständigen Validierung bedürfe es also eines systematischen Ansatzes, welcher alle möglichen edge-cases abdecken kann.

Das `RISCOF`-Testframework [14] bietet einen weiteren Weg, RISC-V Prozessoren validieren. Das Framework führt Assembly-Programme der „RISC-V Architecture Test SIG“ [9] auf dem zu testendem Prozessor und einem RISC-V Referenzmodell aus. Nach der Ausführung der Programme wird ein Speicherabbild beider verglichen. Der zu testende Prozessor besteht die Tests, wenn die Speicherabbilder identisch sind.

Im Rahmen der Arbeit am minimalen Nucleus wurde versucht, dieses Framework auch für diesen zu nutzen. Dieser Versuch ist jedoch an der Komplexität des Frameworks und der Installation der dem Framework zugrunde liegenden Software gescheitert.

6.1.4 Analyse der Syntheseergebnisse

In diesem Abschnitt werden die Ergebnisse der YOSYS-Synthese Anhand der Logikzellen-Statistiken auf ihre Plausibilität geprüft.

Um die SystemC-Hardwarebeschreibung des minimalen Nucleus zu synthetisieren, wurde diese zuerst mit ICSC in äquivalenten SystemVerilog-Code konvertiert. Die Synthese wurde mit YOSYS für das Lattice Semiconductor ECP5 FPGA durchgeführt.

Um einen Synthesebericht zu erhalten wurden die folgenden Schritte ausgeführt:

- Konvertierung eines Designs zu Verilog-Code mit ISCS. Hierzu wurde für den minimalen Nucleus in den Ordner `/piconut/hw/piconut/nuclei/minimalnucleus` navigiert. Dann wurde das Kommando `make build-verilog` aufgerufen.
- Danach wurde in den Ordner `/piconut/systems/refdesign/hw/v_out` navigiert. Hier liegt nun eine Verilog-Datei.
- Mit dem Befehl `yosys <dateiname>.v` ruft man das Yosys-Syntheseframework auf.
- Nun kann mit `synth_ecp5` eine Synthese gestartet werden.
- Die Ausgabe des Syntheseberichts erfolgt auf der Konsole.

Ähnlich können auch die einzelnen Submodule des minimalen Nucleus synthetisiert werden. Hierzu muss `make build-verilog` in den Testbench-Ordnern der Module aufgerufen werden (z.B `../minimalnucleus/alu_tb`).

6.1.4.1 Synthesestatistik des minimalen Nucleus und der Submodule

Die Tabelle 6.2 zeigt eine Satistik der durch das Synthesetool gewählten FPGA-Zelltypen für dem gesamten minimalen Nucleus. Die Tabelle 6.3 zeigt wie viele Zellen die einzelnen Submodule jeweils benötigen.

Zelltyp	Anzahl	Beschreibung
CCU2C	84	2x LUT4 mit Carry-Logik
L6MUX21	783	2zu1 Multiplexer für LUT6-Zellen
LUT4	6204	4er LUTs
PFMUX	1855	2zu1 Multiplexer für LUT5-Zellen
TRELLIS_FF	1176	Flipflops
	10102	Summe

Tabelle 6.2: Synthesestatistik des minimalen Nucleus

Submodul	Anzahl der Zellen
ALU	961
Bysteselector	15
Controller	206
Datahandler	256
Extender	296
Immediate-Generator	66
IR	32
PC	76
Regfile	6434

Tabelle 6.3: Synthesestatistik der Submodule

Die Abbildung 6.1 zeigt den Logikverbrauch der Submodule in Form eines Diagramms.

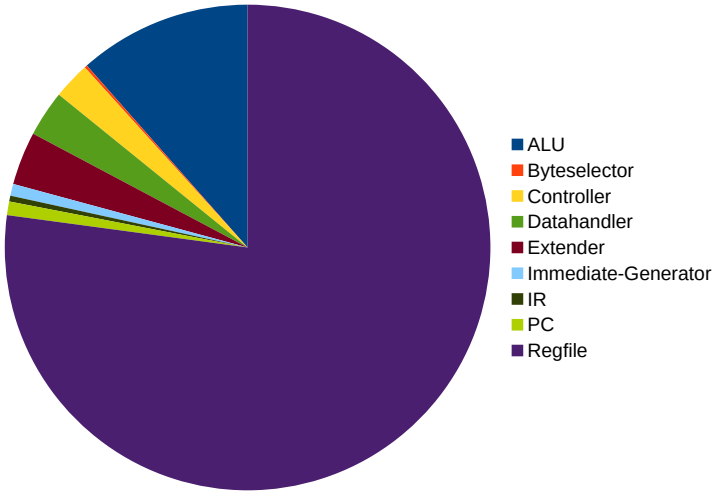

Abbildung 6.1: Logikverbrauch der Submodule

In Summe belegen alle Submodule zusammen 8342 Logikzellen. Dieser Wert unterscheidet sich von der Statistik des gesamten minimalen Nucleus (10102). Die verbleibenden 1760 Logikzellen müssen also von der im Abschnitt 5.5.10 beschriebenen Logik außerhalb der Submodule verbraucht werden.

6.1.4.2 Analyse des Regfile-Moduls

Anhand der Abbildung 6.1 ist zu erkennen, dass das Regfile das mit Abstand größte Submodul des minimalen Nucleus ist. Es macht 63% des gesamten Logikverbrauches aus. Deshalb soll dieses hier genauer analysiert werden. Die Tabelle 6.4 zeigt die Synthesestatistik des Moduls.

Zelltyp	Anzahl	Beschreibung
`CCU2C`	6	2x LUT4 mit Carry-Logik
`L6MUX21`	322	2zu1 Multiplexer für LUT6-Zellen
`LUT4`	3864	4er LUTs
`PFMUX`	1250	2zu1 Multiplexer für LUT5-Zellen
`TRELLIS_FF`	992	Flipflops
	6434	Summe

Tabelle 6.4: Synthesestatistik des Regfile-Moduls

Der hohe Logikverbrauch des Regfiles entsteht durch die Auswahllogik an dessen Ein- und Ausgängen. Am Eingang wird zwischen 31 (ohne `x0`) möglichen Zielregistern gewählt. An den beiden Ausgängen wird jeweils aus 32 Quellregistern gewählt. Diese

Multiplexer-Strukturen wählen jeweils zwischen 32 (31) 32-bit Werten (1024 1-bit Signalen) und benötigen dementsprechend viele Logikzellen.

Das Regfile enthält 992 Flipflops, entgegen der zu erwartenden Anzahl von 1024 (32 32-bit Register). Das Register `x0` hat nach der RISC-V-Spezifikation [22] den konstanten Wert `0x00000000` und kann deshalb durch ein Konstantes Signal mit dem gleichen Wert ersetzt werden. Diese Optimierung wurde bereits vor der Synthese von ICSC erkannt und durchgeführt.

6.1.4.3 Verteilung der Flipflops im minimalen Nucleus

Der minimale Nucleus enthält insgesamt 1176 Flipflops (siehe Tabelle 6.2). Die einzelnen Submodule des minimalen Nucleus enthalten jedoch nach deren Syntheseberichte in Summe 1075 Flipflops. Es existieren also 101 Flipflops außerhalb der Moodule. Wie im Abschnitt 5.4.1 beschrieben, sind die Ausgänge des minimalen Nucleus mit Registern versehen. Die Summe der Flipflops dieser Register ergibt zusammen mit denen der Submodule die Gesamtanzahl aus dem Synthesebericht des gesamten minimalen Nucleus.

Die Tabelle 6.5 zeigt die Verteilung aller Flipflops des minimalen Nucleus.

Anzahl	Quelle
992	Regfile
32	IR
30	PC (siehe 5.5.3)
21	Controller
32	DPort wdata-Ausgang
32	IPort `adr`-Ausgang
30	DPort `adr`-Ausgang (siehe 5.4.1)
4	DPort `bsel`-Ausgang
2	DPort `stb` und IPort `stb`
1	DPort `we`-Ausgang

Tabelle 6.5: Verteilung der Flipflops im minimalen Nucleus

Die Anzahl der Flipflops aus dem Controller-Modul gibt auch Aufschluss darüber, wie das Synthesetool dessen Zustände kodiert hat. Der Controller hat 22 Zustände und die Synthese resultierte in 21 zugewiesenen Flipflops. Dies lässt auf eine one-hot-Kodierung der Zustände schließen. Die SystemC-Hardwarebeschreibung des Moduls nutzt eine Gray-Kodierung.

6.2 Dokumentation des minimalen Nucleus

Neben der Hardwarebeschreibung des minimalen Nucleus wurde auch eine englischsprachige Dokumentation für diesen angefertigt. Sie ist in Markdown geschrieben und nutzt `Sphinx` zum automatischen Erstellen einer `hmtl`-Ansicht. Das generieren einer `pdf`-Datei ist ebenfalls möglich.

Sie beschreibt den Aufbau und die Funktionsweise des minimalen Nucleus und soll neben dieser Arbeit und dem zugehörigen Quellcode als zusätzliche Quelle zur Einarbeitung in das PicoNut-Projekt dienen.

6.3 Vergleich mit dem VISCY-V-CPU

Wie bereits im Kapitel 3 beschrieben, ist der VISCY-V-CPU ein in vielerlei Hinsicht mit dem minimalen Nucleus vergleichbarer RISC-V-CPU. Eine Synthesestatistik beider Lösungen ist in der Tabelle 6.6 dargestellt. Die Synthese des VISCY-V-CPU wurde ebenfalls mit YOSYS und für das Lattice Semiconductor ECP5 FPGA durchgeführt.

	VISCY-V-CPU	**Minimaler PicoNut-Nucleus**
Anzahl verbrauchter Logikzellen	6642	10102
Davon Flipflops	1075	1176

Tabelle 6.6: Logikverbrauch des VISCY-V-CPU und des minimalen PicoNut-Nucleus

Der Minimale Nucleus enthält genau 101 Flipflops mehr als der VISCY-V-CPU. Dies erklärt sich durch die Flipflops an den Ausgängen des minimalen Nucleus (siehe 6.1.4.3). Beide Lösungen enthalten also die selbe Anzahl an Flipflops, wenn man die zusätzlichen Flipflops an den Ausgängen des minimalen Nucleus vernachlässigt. Der minimale Nucleus verbraucht 65,7% mehr Logikzellen als der VISCY-V-CPU.

7 Fazit

7.1 Zusammenfassung

Im Laufe dieser Arbeit ist ein funktionierender, der `RV32I`-Spezifikation entsprechender und minimaler PicoNut Nucleus entwickelt worden. Zusammen mit der `sim-only`-MemU konnte ein Simulationsmodell des ersten PicoNut-Prozessors erstellt werden. Die Anforderung, ein mit GCC kompiliertes C-Programm auszuführen konnte durch Nutzung des Simulationsmodells in einer Simulationsumgebung erfüllt werden. Der entstandene Simulator ist voll funktionsfähig und kann in Zukunft für die weitere Entwicklung am PicoNut-Projekt für Debugging und die Validierung von anderen Nucleus-Varianten genutzt werden. Eine Analyse des vom Simulator generierten Trace-Files hat ergeben, dass dieser Befehle in durchschnittlich 7,94 Taktzyklen ausführt.

Die SystemC-Hardwarebeschreibung ist synthetisierbar und das Syntheseergebnis ist plausibel. Sie ist ausführlich dokumentiert und folgt einer einheitlichen Code- und Namenskonvention.

7.2 Ausblick

Ein Hardware-Demonstrator konnte nicht erstellt werden. Zum Zeitpunkt der Abgabe dieser Arbeit stand noch keine funktionierende, synthetisierbare MemU für die Verwendung auf einem FPGA zur Verfügung. Die Fertigstellung einer synthetisierbaren MemU für die Erprobung des minimalen Nucleus auf einem FPGA sollte der nächste Schritt in der Entwicklung des PicoNut-Projekts sein. Ein funktionierender Hardware-Demonstrator stellt einen großen Erfolg für jedes Hardwareentwicklungsprojekt dar und sollte mit hoher Priorität verfolgt werden.

Darüber hinaus ist die Erweiterung des minimalen Nucleus um CSR-Register und die `A`-Extension ein sinnvoller nächster Schritt in Richtung Linux-Unterstützung. Der minimale Nucleus muss dafür beides unterstützen.

Neben der Unterstützung für Linux könnte auch auf die Verbesserung des PicoNut-Systems insgesamt hingearbeitet werden. Sinnvoll wäre die Unterstützung der `M`- und `F`-Extensions.

Literaturverzeichnis

[1] Accellera Systems Initiative Inc.: *SystemC Synthesizable SubsetVersion 1.4.7*. 2016. (Besucht am 20. 10. 2024) (siehe S. 14).

[2] Arlt, Patrick: *Entwicklung einer minimalen RISC-V-CPU zuLehrzwecken*. 2023. (Besucht am 30. 09. 2024) (siehe S. 3, 16, 20).

[3] Avelar, Julio: *RISCO 5*. 2024. URL: https://github.com/JN513/Risco-5 (besucht am 20. 10. 2024) (siehe S. 17).

[4] Bahle, Alexander et al.: *The ParaNut/RISC-V Processor - An Open, Parallel,and Highly Scalable Processor Architecture forFPGA-based Systems*. 2020. (Besucht am 30. 09. 2024) (siehe S. 17).

[5] Bhasker, J.: *A SystemC Primer*. Star Galaxy Publishing, 2010. ISBN: 9780984629206 (siehe S. 14).

[6] Das Europäische Parlament, Rat der europäischen Union: *Verordnung (EU) 2023/1781 des Europäischen Parlaments und des Rates vom 13. September 2023 zur Schaffung eines Rahmens für Maßnahmen zur Stärkung des europäischen Halbleiter-Ökosystems und zur Änderung der Verordnung (EU) 2021/694 (Chip-Gesetz)*. 2023. URL: https://eur-lex.europa.eu/legal-content/DE/TXT/?uri=CELEX:32023R1781 (besucht am 13. 10. 2024) (siehe S. 1).

[7] EES-Forschungsgruppe: *Forschungsgruppe Effiziente Eingebettete Systeme*. 2024. URL: https://ees.tha.de/index.html (besucht am 01. 10. 2024) (siehe S. 2, 7).

[8] Fraunhofer-Institut für Mikroelektronische Schaltungen und Systeme IMS: "RISC-V Ökosystem: Status und Potenzial". In: (2023) (siehe S. 1, 2, 17).

[9] Gala, Neel; Karasek, Marc: *RISC-V Architecture Test SIG*. URL: https://github.com/riscv-non-isa/riscv-arch-test (besucht am 18. 10. 2024) (siehe S. 44).

[10] Haramboure, Antton et al.: "Vulnerabilities in the semiconductor supply chain". In: (2023). DOI: https://doi.org/https://doi.org/10.1787/6bed616f-en. URL: https://www.oecd-ilibrary.org/content/paper/6bed616f-en (siehe S. 1).

[11] Höller, Roland et al.: "Open-Source RISC-V Processor IP Cores for FPGAs — Overview and Evaluation". In: *2019 8th Mediterranean Conference on Embedded Computing (MECO)*. 2019. DOI: 10.1109/MECO.2019.8760205 (siehe S. 17).

[12] IEEE: *IEEE Xplore*. 2024. URL: https://ieeexplore.ieee.org/Xplore/home.jsp (besucht am 21. 10. 2024) (siehe S. 17).

[13] "IEEE Standard for Standard SystemC® Language Reference Manual". In: *IEEE Std 1666-2023 (Revision of IEEE Std 1666-2011)* (2023). DOI: 10.1109/IEEESTD.2023.10246125 (siehe S. 9).

[14] InCore Semiconductors: *RISCOF*. 2019. URL: https://riscof.readthedocs.io/en/stable/intro.html (besucht am 18. 10. 2024) (siehe S. 44).

[15] *Intel® Compiler for SystemC*. URL: https://github.com/intel/systemc-compiler (besucht am 01. 10. 2024) (siehe S. 9).

[16] Kiefer, Gundolf: *Entwurf digitaler Systeme II*. Forschungsgruppe Effiziente Eingebettete SystemeFakultät für InformatikTechnische Hochschule Augsburg. 2024. URL: https://www.hs-augsburg.de/homes/kiefer/eds_ii/material/eds_ii-v-show.pdf (siehe S. 14).

[17] Nolting, Stephan: *The NEORV32 RISC-V Processor*. 2020. URL: https://github.com/stnolting/neorv32 (besucht am 20. 10. 2024) (siehe S. 17).

[18] RISC-V International: *RISC-V Exchange*. 2024. URL: https://riscv.org/exchange/?_sft_exchange_category=core,cores (besucht am 19. 10. 2024) (siehe S. 17).

[19] RISC-V International: *About RISC-V*. URL: https://riscv.org/about/ (besucht am 02. 10. 2024) (siehe S. 1).

[20] RISC-V International: *Specifications*. URL: https://riscv.org/technical/specifications/ (besucht am 01. 10. 2024) (siehe S. 4).

[21] Semiconductor Industry Association: *State of the U.S. Semiconductor Indstry 2024*. Sep. 2024. URL: https://www.semiconductors.org/wp-content/uploads/2024/09/SIA_State-of-Industry-Report_2024_final_091124.pdf (besucht am 12. 10. 2024) (siehe S. 1).

[22] Waterman, Andrew; Asanović, Krste: *The RISC-V InstructionSet Manual, Volume I: User-Level ISA, Document Version 20191214-draft*. Abgerufen am: 30.09.2024. 2019. URL: https://github.com/riscv/riscv-isa-manual/releases/download/20240411/unpriv-isa-asciidoc.pdf (besucht am 30. 09. 2024) (siehe S. VII, 2, 4–6, 20, 21, 25, 29, 39, 47, a).

[23] WATERMAN, Andrew et al.: *The RISC-V Instruction SetManual: Volume II: Privileged Architecture*. 2024. URL: https://github.com/riscv/riscv-isa-manual/releases/download/20240411/priv-isa-asciidoc.pdf (siehe S. 4).

[24] *Wilkommen beim PicoNut-Projekt*. 2024. URL: https://ees.tha.de/piconut/index.html (besucht am 01. 10. 2024) (siehe S. 7).

[25] WOLF, Claire: *PicoRV32 - A Size-Optimized RISC-V CPU*. 2019. URL: https://github.com/YosysHQ/picorv32 (besucht am 20. 10. 2024) (siehe S. 17).

[26] *yosys – Yosys Open SYnthesis Suite*. URL: https://yosyshq.net/yosys/ (besucht am 01. 10. 2024) (siehe S. 9).

A Anhang

A.1 RISC-V RV32I Befehlsreferenz

Das `RV32I` BIIS spezifiziert die in der Tabelle A.1 aufgelisteten Befehle [22].

A.2 „Hello World“-Testprogramm

```
1        #include <stdio.h>
2        #include <unistd.h>
3
4        int main () {
5            int n;
6            char hello[] = "Hello World!";
7
8            for (n = 1; n <= 10; n++)
9            printf ("%2i. %s\n", n, hello);
10
11           return 0;
12           }
```

Listing A.1: Quellcode des „Hello World“-Testprogramms

Befehl	Beschreibung	Format	Opcode	funct3	funct7	Operation
add	Addtion	R-Typ	0110011	0x0	0x0	$rd = rs1 + rs2$
sub	Subtraktion	R-Typ	0110011	0x0	0x20	$rd = rs1 - rs2$
xor	Logisches Exklusiv-Oder	R-Typ	0110011	0x4	0x0	$rd = rs1 \oplus rs2$
or	Logisches Oder	R-Typ	0110011	0x6	0x0	$rd = rs1 \mid rs2$
and	Logisches Und	R-Typ	0110011	0x7	0x0	$rd = rs1 \& rs2$
sll	Logische Linksschiebung	R-Typ	0110011	0x1	0x0	$rd = rs1 \ll rs2[4:0]$
srl	Logische Rechtsschiebung	R-Typ	0110011	0x5	0x0	$rd = rs1 \gg rs2[4:0]$
sra	Arithmetische Rechtsschiebung	R-Typ	0110011	0x5	0x20	$rd = rs1 \gg rs2[4:0]$
slt	„Set less than“	R-Typ	0110011	0x2	0x0	$rd = (rs1 < rs2)?1:0$
sltu	„Set less than (unsigned)“	R-Typ	0110011	0x3	0x0	$rd = (rs1 < rs2)?1:0$
addi	Addieren mit Direktwert	I-Typ	0010011	0x0	0x0	$rd = rs1 + imm$
xori	Logisches Exklusiv-Oder mit Direktwert	I-Typ	0010011	0x4	0x0	$rd = rs1 \oplus imm$
ori	Logisches Oder mit Direktwert	I-Typ	0010011	0x6	0x0	$rd = rs1 \mid imm$
andi	Logisches Und mit Direktwert	I-Typ	0010011	0x7	0x0	$rd = rs1 \& imm$
slli	Logische Linksschiebung mit Direktwert	I-Typ	0010011	0x1	0x0	$rd = rs1 \ll imm[4:0]$
srli	Logische Rechtsschiebung mit Direktwert	I-Typ	0010011	0x5	0x0	$rd = rs1 \gg imm[4:0]$
srai	Arithmetische Rechtsschiebung mit Direktwert	I-Typ	0010011	0x5	0x20	$rd = rs1 \gg imm[4:0]$
slti	„Set less than“ mit Direktwert	I-Typ	0010011	0x2	0x0	$rd = (rs1 < imm)?1:0$
sltiu	„Set less than (unsigned)“ mit Direktwert	I-Typ	0010011	0x3	0x0	$rd = (rs1 < imm)?1:0$
lw	Laden eines 32-bit Wertes aus dem Speicher	I-Typ	0000011	0x2	0x0	$rd = M[rs1 + imm]$
lh	Laden eines 16-bit Wertes aus dem Speicher	I-Typ	0000011	0x1	0x0	$rd = M[rs1 + imm][0:15]$
lb	Laden eines 8-bit Wertes aus dem Speicher	I-Typ	0000011	0x0	0x0	$rd = M[rs1 + imm][0:7]$
lbu	Laden eines 8-bit Wertes aus dem Speicher ohne Vorzeichen	I-Typ	0000011	0x4	0x0	$rd = M[rs1 + imm][0:7]$
lhu	Laden eines 16-bit Wertes aus dem Speicher ohne Vorzeichen	I-Typ	0000011	0x5	0x0	$rd = M[rs1 + imm][0:15]$
sw	Speichern eines 32-bit Wertes im Speicher	S-Typ	0100011	0x2	0x0	$M[rs1 + imm] = rs2$
sh	Speichern eines 16-bit Wertes im Speicher	S-Typ	0100011	0x1	0x0	$M[rs1 + imm][0:15] = rs2[0:15]$
sb	Speichern eines 8-bit Wertes im Speicher	S-Typ	0100011	0x0	0x0	$M[rs1+imm][0:7] = rs2[0:7]$
lui	Laden eines 20-bit Direktwertes in die oberen 20 Bit	U-Typ	0110111	0x0	0x0	$rd = imm \ll 12$
auipc	20-bit Direktwert auf PC addieren	U-Typ	0010111	0x0	0x0	$rd = PC + imm \ll 12$
jal	Sprung	J-Typ	1101111	0x0	0x0	$rd = PC + 4; PC = PC + imm$
jalr	Indirekter Sprung	I-Typ	1100111	0x0	0x0	$rd = PC + 4; PC = rs1 + imm$
beq	Branch (Sprung) bei Gleichheit	B-Typ	1100011	0x0	0x0	$if(rs1 == rs2)PC+ = imm$
bne	Branch (Sprung) bei Ungleichheit	B-Typ	1100011	0x1	0x0	$if(rs1! = rs2)PC+ = imm$
blt	Branch (Sprung) wenn kleiner	B-Typ	1100011	0x4	0x0	$if(rs1 < rs2)PC+ = imm$
bge	Branch (Sprung) wenn größer oder gleich	B-Typ	1100011	0x5	0x0	$if(rs1 >= rs2)PC+ = imm$
bltu	Branch (Sprung) wenn kleiner (unsigned)	B-Typ	1100011	0x6	0x0	$if(rs1 < rs2)PC+ = imm$
bgeu	Branch (Sprung) wenn größer oder gleich (unsigned)	B-Typ	1100011	0x7	0x0	$if(rs1 >= rs2)PC+ = imm$
ecall	Systemaufruf (Betriebssystem)	I-Typ	1110011	0x0	0x0	
ebreak	Systemaufruf (Debugger)	I-Typ	1110011	0x0	0x1	
fence	Speicherbarriere	I-Typ	0001111	0x0	0x0	

Tabelle A.1: RV32I Befehlsreferenz

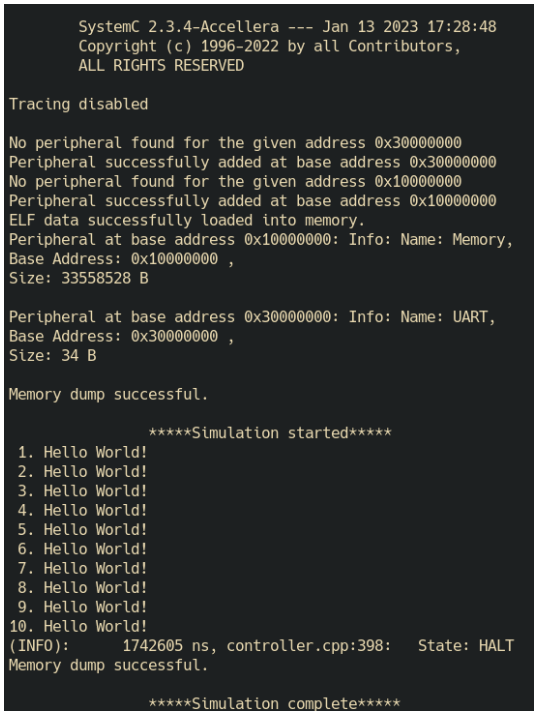

Abbildung A.1: Konsolenausgabe des „Hello World“-Testprogramms im Simulator