Entwicklung eines eigenen Microcontrollers auf RV32i Basis mithilfe von litex auf dem Tang-Nano-9k FPGA

Und der Implementierung platformspezifischer Codes

Autor: Daniel Hohmann, Erik Donath

Schule: Theodor-Litt-Schule

Datum: 31. Dezember 2025

Lehrer: Elisabeth Engel

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Vorwort

Vorab möchten wir uns bei Frau Engel entschuldigen. Wir wussten, dass die Dokumentation maximal 15 Seiten lang sein sollte. Doch bei unserem umfangreichen Projekt, das deutlich aus dem Ruder gelaufen ist und am Ende ein enormes Maß an Komplexität erreicht hat, war es schlicht nicht mehr möglich, alles, was passiert ist, sowie die vollständige Logik innerhalb von 15 Seiten unterzubringen. Wir hoffen dennoch, dass alle, die dies lesen, ihren Spaß haben und vielleicht etwas lernen. Dies ist eine Dokumentation voller Verzweiflung und Hoffnung.

1.2 Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Dokumentation selbstständig erstellt habe. Der Inhalt stammt ausschließlich von mir, und es wurden keine anderen als die angegebenen Quellen und Hilfsmittel verwendet. Dieses Dokument wurde lediglich durch eine KI zur Korrektur (z. B. Grammatik und Stil) gelesen; der gesamte Inhalt basiert jedoch allein auf meiner eigenen Arbeit. Alle wörtlichen und sinngemäßen Zitate sind gekennzeichnet, und die gedruckte sowie die elektronische Version stimmen überein. Mir ist bekannt, dass falsche Angaben strafrechtliche Konsequenzen nach § 156 StGB haben können.

1.3 Einleitung

Das Ziel unseres Projekts war es, einen RISC-V RV32I für den Tang Nano 9K zu implementieren. Die HDL (Hardware Description Language) unserer Wahl war VHDL, da sie robuster und typsicherer ist als Verilog.
Vorab dieses Projekt mussten wir jedoch aufgeben, nachdem wir nach zwei Wochen erkannt hatten, wie aufwändig es tatsächlich ist. In dieser Zeit hätten wir es nicht geschafft, ein Ergebnis zu erzielen, das sowohl praktisch vorzeigbar gewesen wäre als auch unseren eigenen

Ansprüchen genügt hätte.
Also entschieden wir uns, das Projekt zu vereinfachen, indem wir ein Framework nutzten, das uns viel Arbeit abnahm. Statt alles manuell zu schreiben, mussten wir nur noch definieren, was wir haben wollten, ergänzt um etwas Logik. Dieses Framework übersetzte unseren Code anschließend automatisch in Verilog.

1.4 Projektumfeld

Das Projekt fand im Rahmen des IT-Unterrichts im Unterrichtsfeld Prozessautomatisierung statt. Es wurde sowohl in der Schule als auch im privaten Umfeld aktiv daran gearbeitet.

Kapitel 2

Analyse

2.1 Zielsetzung / Produktskizze

Die gesteckten Projektziele waren in zwei Hauptkriterien unterteilt. Das erste Kriterium bestand in der Entwicklung eines funktionsfähigen System-on-Chip (SoC) auf Basis der RISC-V-Architektur (RV32I). Der entwickelte SoC sollte in der Lage sein, eigenständig Software auszuführen, ähnlich wie ein Mikrocontroller, beispielsweise auf einem Arduino-System. Ziel war es, Programme direkt auf dem SoC auszuführen und über dessen GPIO-Pins externe Hardwarekomponenten, wie LEDs oder Sensoren, anzusteuern und auszulesen. Damit sollte eine einfache Schnittstelle zwischen Software und Hardware gewährleistet werden.

Auf architektonischer Ebene war vorgesehen, dass der Prozessor grundlegende CPU-Komponenten enthält, die die effiziente Ausführung von Programmen ermöglichen. Dazu zählen eine Pipeline-Architektur für parallele Befehlsverarbeitung, ein Interrupt-Handling-System sowie optionale Module wie eine Multiply-Divide-Unit (für RV32IM-Erweiterungen), eine Floating-Point-Unit (FPU) für Gleitkommaoperationen und ein Timer-Subsystem. Darüber hinaus sollte das SoC-Design über eine klar definierte Speicherhierarchie verfügen, bestehend aus einem Instruction und einem Data-Memory-Bereich sowie einem Bus-System zur Anbindung externer Peripherie. Zusätzlich sollten Taktgenerierung und Reset-Logik modular realisiert sein, beispielsweise durch die Nutzung von MMCMs (Mixed-Mode Clock Manager), um verschiedene Taktfrequenzen für CPU-Kern und Peripherie bereitzustellen.

Das zweite Hauptkriterium bezog sich auf die physische Implementierung: Der entwickelte SoC sollte nach erfolgreicher Simulation auch auf realer Hardware lauffähig sein. Konkret sollte der Chip auf dem gewählten FPGA-Board Tang Nano 9K implementiert und getestet werden. Dies erforderte die Integration der Hardware in die LiteX-Umgebung, das Generieren der entsprechenden Bitstream-Dateien sowie die erfolgreiche Übertragung und Inbetriebnahme des SoCs auf dem FPGA. Hierbei war insbesondere sicherzustellen, dass alle Funktionen, wie

Takterzeugung, Speicheranbindung, GPIO-Steuerung und Programm-Upload – sowohl in der Simulation als auch auf der Zielhardware konsistent funktionieren.

2.2 Begründung der Entscheidung

Das Projekt wurde aus der Intention heraus ausgewählt, eine echte Herausforderung zu schaffen, mit einem Vorhaben, das für die vorgesehene Zeit und das in der Schule vermittelte Wissen eigentlich nicht erreichbar war.
Ein weiterer Grund für diese Entscheidung war die Motivation, das eigene Wissen zu vertiefen und zu erweitern. Es war die Wissbegierde, zu verstehen, wie ein Mikrocontroller funktioniert und wie aufwendig oder einfach es ist, ein solches System selbst zu entwickeln.
Wir wollten kein Projekt umsetzen, das lediglich auf bestehenden Lösungen aufbaut. Stattdessen wollten wir die Grundlagen nachvollziehen und ein tieferes Verständnis dafür gewinnen, wie Hardware und Logik tatsächlich zusammenspielen.

2.3 Pflichtenheft

Das Pflichtenheft konkretisiert die in der Produktskizze beschriebenen Ziele für den RISC-V-RV32I-SoC auf dem Tang Nano 9K FPGA. Es legt die Architektur des CPU-Kerns, die 5-stufige Pipeline, das FPU-Design (Floating Point Unit – Gleitkomma-Einheit) sowie die Rolle des LiteX-Frameworks bei Integration, Takterzeugung und Peripherie-Anbindung fest. Die Implementierung erfolgt in synthesizierbarem SystemVerilog (Hardware-Beschreibungssprache); der SoC wird über das LiteX-Buildsystem erzeugt, das Bus-Infrastruktur (Wishbone – On-Chip-Bus), Clock/Reset-Management (Takt-/Zurücksetz-Logik), Speicher- und Peripherie-Mapping sowie den FPGA-spezifischen Build-Flow (Synthese und Place&Route) automatisiert bereitstellt.

Die CPU verwendet eine klassische 5-Stufen-Pipeline (Fetch, Decode, Execute, Memory, Writeback), wie in Abbildung 2.1 dargestellt. In der *Fetch*-Stufe erzeugt ein Program Counter (PC) die aktuelle Befehlsadresse, optional mit Unterstützung von Komponenten wie Branch Target Buffer (BTB – Sprungzielpuffer) und Branch History/Predictor (BHT, PHT – Sprungvorhersage), die in der Abbildung als separate Blöcke visualisiert sind. Die Instruktion wird aus dem Instruktionsspeicher gelesen und über das IfId-Pipeline-Register an die Decode-Stufe übergeben, wodurch eine klare zeitliche Trennung zwischen Adressberechnung und Befehlsdekodierung erreicht wird.

In der *Decode*-Stufe wird die Instruktion dekodiert und über den Decoder-Block in Steuersignale, Registeradressen und Immediate-Werte (unmittelbare Operanden) aufgeteilt; gleichzeitig liest das Register File (REGFILE) die Operanden `rs1` und `rs2`. Ein Load-Use-Block

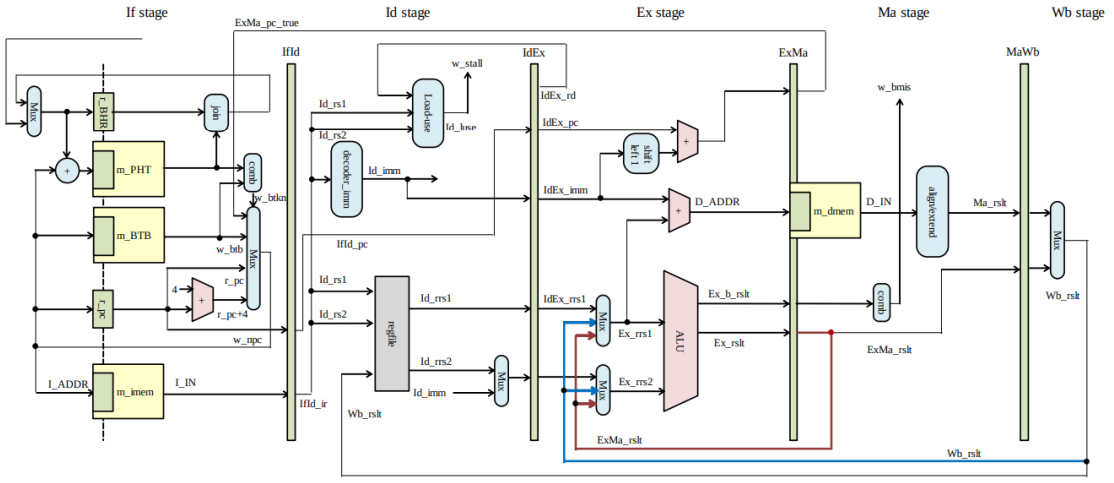

Abbildung 2.1: Fünfstufige Pipeline bestehend aus Fetch-, Decode-, Execute-, Memory- und Writeback-Stufe. Die Abbildung zeigt PC-Logik, Branch-Vorhersage, Register-File, ALU, Datenpfad sowie die Pipeline-Register zwischen den Stufen. [Miy+20]

überwacht Datenabhängigkeiten zwischen Load-Instruktionen und nachfolgenden Instruktionen und kann bei Bedarf die Pipeline anhalten (Stall) oder weiterlaufen lassen, wie durch die Rückkopplungssignale in Abbildung 2.1 angedeutet. Alle relevanten Signale werden im IdEx-Pipeline-Register zwischengespeichert, das in der Abbildung den Übergang zur Execute-Stufe markiert und so für deterministische Signalübergaben sorgt.

In der *Execute*-Stufe berechnet die ALU (Arithmetic Logic Unit – Rechen- und Logikeinheit) Zieladressen für Speicherzugriffe und Ergebnisse arithmetisch-logischer Operationen; Forwarding-Multiplexer (Datenweiterleitungs-MUX) führen Ergebnisse aus späteren Pipeline-Stufen zurück auf die ALU-Eingänge, um Data-Hazards (Datenkonflikte) zu vermeiden. Bei Sprungbefehlen entscheidet die ALU oder eine separate Branch-Logik über die Sprungbedingung, und ein PC-Multiplexer wählt das nächste PC-Ziel (normaler Sequenz-PC oder Sprungadresse), was in der IF-Logik von Abbildung 2.1 über die Signale für neuen PC sichtbar ist. Das ExMa-Pipeline-Register trennt Execute- und Memory-Stufe und leitet die berechneten Adressen und Daten an den Speicherpfad weiter.

In der *Memory*-Stufe werden Speicherzugriffe über den Datenpfad (Adress-, Daten- und Steuerleitungen) realisiert; ein Alignment-/Byte-Select-Block sorgt für korrekt ausgerichtete Byte-, Halfword- und Word-Zugriffe. Gleichzeitig können in dieser Stufe weitere Steuersignale für nachfolgende Interrupt- oder Ausnahmebehandlung gesammelt werden, etwa Fehlzugriffe oder Miss-Signale. Das MaWb-Pipeline-Register übergibt schließlich die Daten an die *Write-back*-Stufe, in der über einen Abschlussmultiplexer ausgewählt wird, ob ALU-Ergebnis oder geladene Daten in das Register File zurückgeschrieben werden; der geschlossene Ergebnisrückweg ist in Abbildung 2.1 als Wb_rslt-Signal markiert.

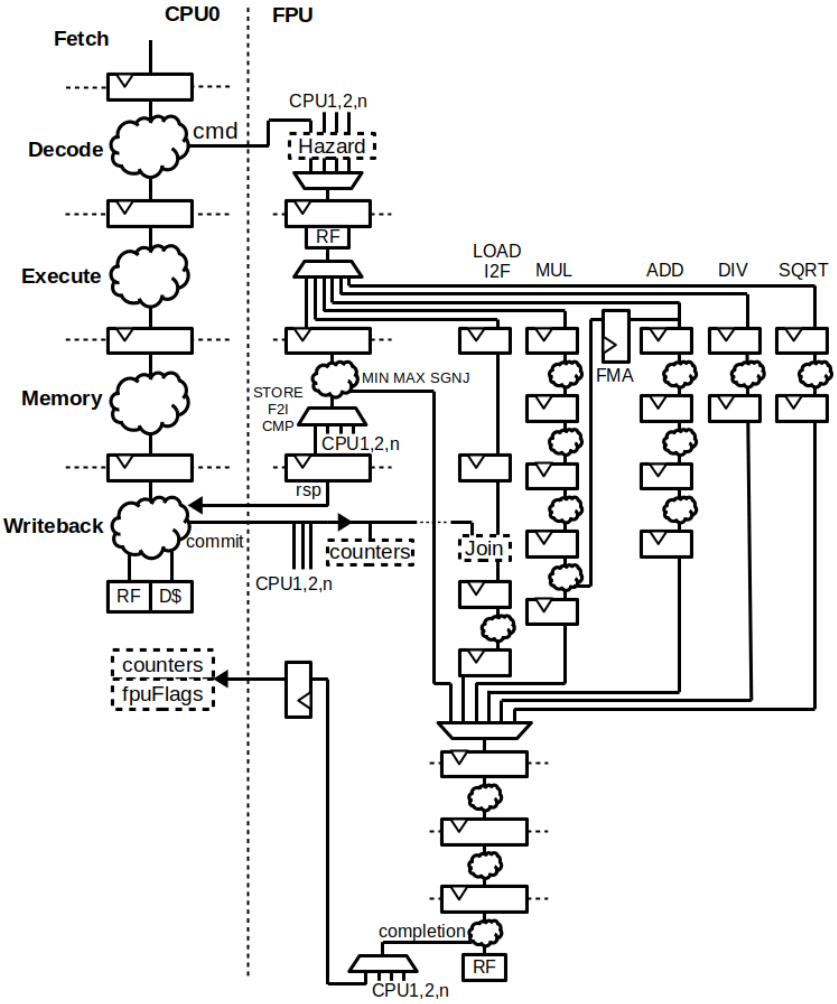

Abbildung 2.2: FPU-Design mit separatem Register-File, Hazard-Logik und spezialisierten Pipelines für Load/Convert, Multiplikation, Addition, Division, Quadratwurzel und FMA. Die Join- und Completion-Logik koppelt die FPU an die Writeback-Stufe der CPU an. [Spi25]

Das FPU-Design in Abbildung 2.2 zeigt eine entkoppelte Gleitkomma-Pipeline, die eng mit der CPU verbunden ist, aber eigene Pipeline-Stufen und Hazard-Logik besitzt. Auf CPU-Seite werden FPU-Befehle in der Decode-Stufe erkannt und als Kommando (`cmd`) an die FPU übergeben; der Hazard-Block am Eingang der FPU stellt sicher, dass neue FPU-Operationen nur akzeptiert werden, wenn interne Ressourcen frei sind und keine strukturellen Konflikte vorliegen. Die FPU verfügt über ein eigenes Register File (RF) für Gleitkommaregister und mehrere spezialisierte Funktionspfade, die in der Abbildung als separate Pipelines für `LOAD/I2F` (Integer-zu-Float-Konvertierung), `MUL` (Multiplikation), `ADD` (Addition), `DIV` (Division) und `SQRT` (Quadratwurzel) dargestellt sind.

Die zentrale FMA-Einheit (Fused Multiply-Add – kombinierte Multiplikations-Addierer-Einheit) bildet das Herz der FPU-Pipeline und kann je nach Befehl Multiplikation, Addition

oder kombinierte Operationen ausführen; vorgeschaltete Pipeline-Register erlauben hohe Taktfrequenzen bei mehreren parallel aktiven FPU-Befehlen. Weitere Blöcke implementieren Operationen wie `MIN`, `MAX`, `SGNJ` (Vorzeichenmanipulation), `F2I` (Float-zu-Integer-Konvertierung) und `CMP` (Vergleich), während die Join-Logik Ergebnisse aus den unterschiedlichen Funktionspfaden zusammenführt und geordnet an die Completion-Phase weitergibt. Über das in Abbildung 2.2 eingezeichnete Commit-Interface werden FPU-Ergebnisse in der Writeback-Stufe der CPU in die Integer- oder FPU-Register geschrieben; parallel aktualisiert die FPU Status- und Fehlerflags (fpuFlags) sowie Performance-Counter, die der CPU zur Auswertung zur Verfügung stehen.

Die Instruktionsimplementierung folgt der [msy19] (RISC-V Instruction Set Architecture – Befehlssatz-Architektur) mit U-, I-, R-, S-, B- und J-Formaten; die Dekodierung erfolgt in der Decode-Stufe der Pipeline und steuert die in den Abbildungen gezeigten Pfade für ALU, Speicherzugriffe, Branch-Logik und FPU-Anbindung. LiteX übernimmt im Hintergrund die Rolle des SoC-Baukastens und sorgt dafür, dass die in den Abbildungen dargestellten CPU- und FPU-Pipelines nahtlos in ein vollständiges System eingebettet werden, indem es Wishbone-Schnittstellen generiert, Speicher und Peripherie an die Memory-Stufe bindet, Clock- und Reset-Netze für alle Pipeline-Register bereitstellt und den Build-Flow bis hin zum Bitstream automatisiert.

Kapitel 3

Planung

3.1 Personal

Das Projektteam bestand aus zwei Mitgliedern: Daniel und Erik. Die Aufgaben waren gleichmäßig aufgeteilt. Da nur ein FPGA zur Verfügung stand, wurde regelmäßig gewechselt. Während der eine am FPGA arbeitete und den Code entwickelte, kümmerte sich der andere um die Dokumentation und die Recherche. Durch diesen fortlaufenden Wechsel konnten beide Teammitglieder in allen Bereichen, sowohl in der praktischen Hardwarearbeit als auch in der theoretischen Ausarbeitung, gleichmäßig Erfahrung sammeln und zum Fortschritt des Projekts beitragen.

3.2 Material

Als Material kamen der FPGA *Tang Nano 9K*, unsere Laptops sowie ein USB-C-Kabel zum Einsatz.

3.3 Zeit

Das Zeitmanagement war von Beginn an schwer einzuschätzen, da anfangs unklar war, ob wir das Projekt überhaupt in der geplanten Form umsetzen können, selbst mit der Entscheidung, nicht alles von Grund auf neu zu entwickeln. Auch zum derzeitigen Stand dieser Dokumentation (31. Dezember 2025) hat die verfügbare Zeit bei weitem nicht ausgereicht. Wir verfügen zwar über ein funktionsfähiges SoC, jedoch ohne PlatformIO-Integration und ohne RTOS-Unterstützung. Das Projekt befindet sich weiterhin im Aufbau, und im Verlauf sind noch mehrere neue Ideen hinzugekommen, die jedoch zeitlich nicht mehr umsetzbar waren.

Abbildung 3.1 zeigt den zeitlichen Verlauf der Entwicklung des Basis-SoCs.

Zum Diagramm 3.1 ist anzumerken, dass unser ursprünglicher Projektansatz darin bestand, den SoC komplett in VHDL von Grund auf zu entwickeln. Diese frühe Phase und die dort

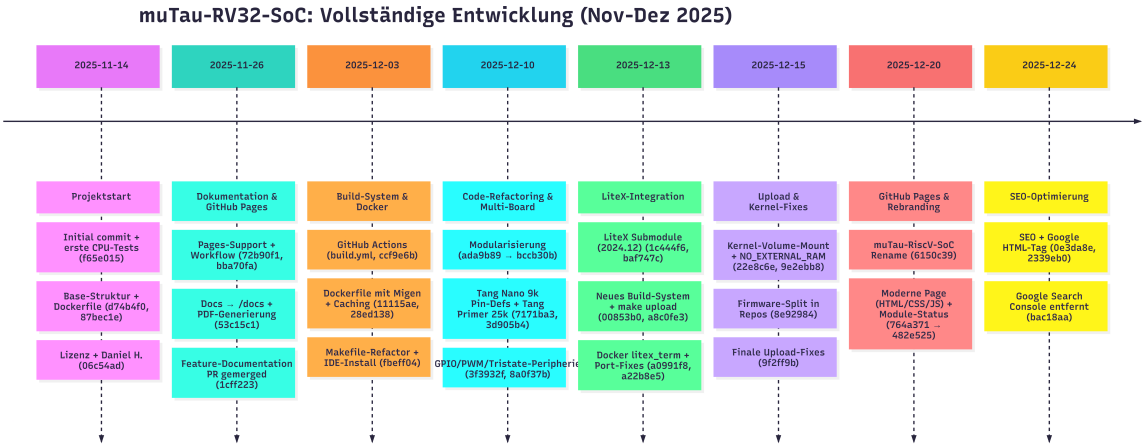

Abbildung 3.1: Zeitleiste (erstellt mit Mermaid), welche die Commits und den ungefähren Fortschritt des Projekts visualisiert.

erzielten Fortschritte sind im gezeigten Zeitverlauf nicht mehr enthalten.

Wie der Zeitleiste zu entnehmen ist, hat insbesondere das Aufsetzen des Docker-Containers unerwartet viel Zeit in Anspruch genommen, insgesamt mehr als drei Wochen(ist nicht ganz aus dem Diagramm zu entnehmen da wir am DockerContainer schon vorher aufgesetzt haben). Nach Abschluss dieser Aufgabe traten weitere Verzögerungen auf: Unser ursprüngliches Ziel war es, den Bitstream mithilfe einer Open-Source-Toolchain zu synthetisieren, zu platzieren und zu routen. Wie sich jedoch später herausstellte, waren die verfügbaren Open-Source-Tools für unsere Anforderungen nicht ausreichend optimiert, was uns letztlich etwa zwei zusätzliche Wochen kostete. Nachdem dieses Problem behoben war, konnten wir endlich den bis dahin entwickelten Code auf der physischen Hardware testen.

Während des gesamten Projekts haben wir parallel an mehreren Teilbereichen gearbeitet. Jeden Montag fand eine Gruppenbesprechung statt, in der alle Teammitglieder ihren aktuellen Stand präsentierten. Gemeinsam entschieden wir dann, welche Aufgaben weiterverfolgt oder vorerst pausiert werden sollten. Feste Zeitpläne haben wir bewusst vermieden, da sich im Verlauf oft zeigte, dass diese in der Praxis kaum realistisch waren.

Ab einem gewissen Zeitpunkt hat sich Erik vom Hauptprojekt abgekoppelt, um an der Integration von PlatformIO zu arbeiten. Ziel war es, das Projekt zu einem wirklich nutzbaren Produkt weiterzuentwickeln. Der zeitliche Verlauf dieser Arbeiten ist in Abbildung 3.2 dargestellt.

Die Diagramme enthalten ausschließlich zeitlich relevante Aspekte, jedoch keine detaillierten Ablaufbeschreibungen oder Programmierfortschritte.

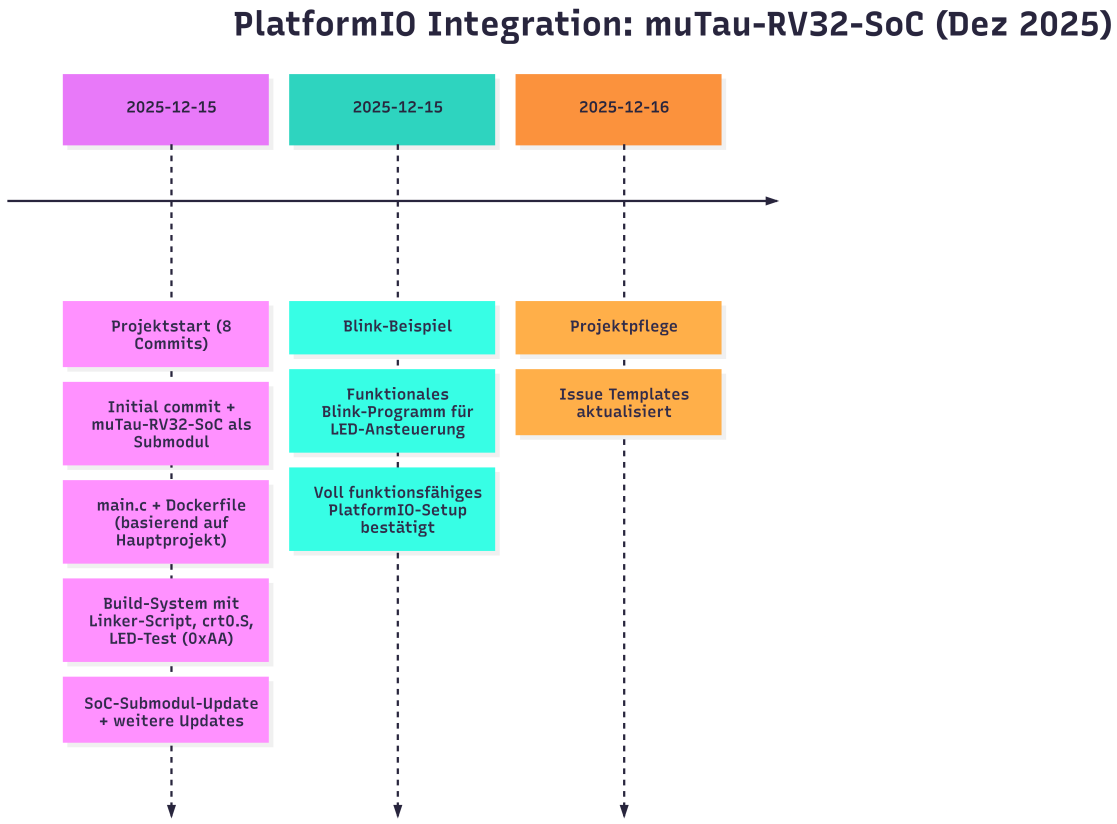

Abbildung 3.2: Zeitleiste (erstellt mit Mermaid) mit den Commit-Zeiten und Arbeitsschwerpunkten der PlatformIO-Integration.

Kapitel 4

Durchführung

4.1 Bau des Modells

Beim Bau des Modells beziehen wir uns auf unser Endprodukt, nicht auf das vorherige, aufgegebene Projekt.

4.1.1 Docker

Um überhaupt mit der Entwicklung beginnen zu können, mussten wir zunächst eine Entwicklungsumgebung schaffen, die die folgenden Anforderungen erfüllt:

- Sie muss plattformunabhängig und reproduzierbar sein (damit beide dieselbe Entwicklungsumgebung nutzen können).
- Alle benötigten Tools müssen enthalten und miteinander kompatibel sein (um Abhängigkeitsfehler zu vermeiden).
- Für das Endprodukt soll sie einfach zu bedienen und möglichst fehlerunanfällig sein, damit sie von jedem problemlos verwendet werden kann.

Die Entscheidung fiel uns zunächst nicht leicht, wir standen zwischen **Nix** und **Docker**. Nix hätte den Vorteil gehabt, schneller und einfacher zu bauen, jedoch fehlten viele Pakete der Toolchain als Nix-Pakete. Daher entschieden wir uns für eine Kombination aus Docker und Makefile. So kann das Projekt mit einfachen CLI-Kommandos gebaut werden, ohne dass man sich mit Docker weiter beschäftigt haben muss.
Wir entwickelten einen Docker-Container auf Basis von *debian-bookworm-slim*, um Bloatware zu vermeiden und den Container schneller zu bauen. Innerhalb des Containers installierten wir alle benötigten Tools, darunter Python (mit einer virtuellen Umgebung), LiteX, liteX-boards sowie einige Build-Abhängigkeiten wie `meson` und `jinja2`. Anfangs dauerte der Build-Prozess des Containers jedoch sehr lange, pro Änderung bis zu 30 Minuten. Um dies zu optimieren, implementierten wir ein Caching-System, teilten den Container in fünf Stages auf und nutzten Docker-Args, um verschiedene Stage-Kombinationen gezielt ausführen zu können.

4.1.2 Toolchain

Der ursprüngliche Plan war, eine Open-Source-Toolchain zu verwenden 3.3, mit `yosys` als Routing-Tool sowie `nextpnr-himbaechel` als Building-Tool für den Bitstream, in Kombination mit `apycula` und `openFPGALoader` zum Flashen des Designs.
Es zeigte sich jedoch schnell, dass die Open-Source-Tools nicht optimal mit den LUT-Ressourcen des Tang Nano 9K umgehen. Dadurch kam es zu fehlerhaften Synthesen, obwohl das Design grundsätzlich kompatibel war. Wir entschieden uns daher für die alternative, geschlossene Toolchain **GOWIN EDA**. Das Problem dabei war jedoch, dass für die Installation persönliche Daten angegeben werden mussten, was nicht im Sinne unseres Projekts war.
Nach längerer Recherche und Experimenten mit der Open-Source-Toolchain analysierten wir schließlich den Download-Prozess der GOWIN-Webseite und stellten fest, dass der eigentliche Download-Link Statisch war. Diese Schwachstelle nutzten wir, um mithilfe des CLI-Tools `curl` die Installationsdateien direkt herunterzuladen, ganz ohne Anmeldung. Dieses Verfahren funktionierte erfolgreich, und wir waren gespannt mit der GOWIN EDA die ersten SoC-Tests für den Tang Nano 9K durchführen.

4.1.3 SoC

Für den SoC begaben wir uns zunächst in eine intensive Lernphase. Wir beschäftigten uns eingehend mit **LiteX** (das auf **Migen** basiert) und den internen Strukturen, um zu verstehen, wie ein solches Projekt aufgebaut ist, welche Möglichkeiten es bietet und wie wir es am sinnvollsten nutzen können.
Schnell stellten wir fest, dass LiteX nur sehr spärlich dokumentiert ist. Es existieren zwar Beispiele, diese sind jedoch oberflächlich, unkommentiert und bieten keine tiefere Hilfe zum Verständnis der Architektur. Außerdem waren die Beispielprojekte unstrukturiert, alles befand sich meist in einer einzigen Datei, was wir für unser Projekt vermeiden wollten.
Daher entwickelten wir unsere eigene, logisch aufgebaute Ordnerstruktur, die nach unserer Einschätzung übersichtlicher und wartungsfreundlicher ist (siehe 5.0.1).
Nachdem wir die Basis für den SoC geschaffen, diesen mit Verilator simuliert und erfolgreich getestet hatten, wagten wir den ersten vollständigen Flash-Versuch auf dem FPGA mit dem LiteX-BIOS. Zunächst schien alles in Ordnung, keine Fehler wurden angezeigt. Beim Versuch, sich via serieller Verbindung mit dem FPGA zu verbinden, traten jedoch Kommunikationsfehler auf (siehe Kapitel 4.1.4). Diese Probleme konnten wir nach einiger Zeit beheben.
Damit hatten wir einen voll funktionsfähigen SoC, also quasi einen Mikrocontroller, Entwickelt, genau den Punkt, mit dem andere Projekte normalerweise beginnen. Bald stellten wir jedoch fest, dass das Programmieren dieses SoCs komplexer war als erwartet. Gleichzeitig erkannten wir das große Potenzial des Projekts. Unsere nächste Zielsetzung war daher, daraus ein vollwertiges Produkt zu machen, mit **PlatformIO**-Integration, sodass jeder den SoC leicht programmieren konnte. Außerdem wollten wir ermöglichen, ein RTOS darauf auszuführen, was

weitere vorteile für die konsumenten bieten würde.
Ab diesem Punkt war das ursprüngliche Projekt zwar abgeschlossen, doch wir wollten mehr. Wir starteten zwei neue Teilprojekte: 1. Ein Repository mit Beispielen und Dokumentation, wie man auf dem SoC unser RTOS Installiert. 2. Eine Integration für PlatformIO (*work in progress*), da PlatformIO ein Industriestandard ist.
Für das RTOS entschieden wir uns für **Zephyr** [Zep25a]. Damit der SoC mit dessen Anforderungen kompatibel ist, passten wir das Design entsprechend an. Mithilfe eines kaum dokumentierten LiteX-internen Tools gelang es uns schließlich, Konfigurationsdateien für Zephyr automatisch zu generieren. Ein vollwertiges Image ist derzeit noch nicht komplett fertiggestellt.
Darüber hinaus spezialisierten wir unseren SoC auf Signalverarbeitung. Die Idee: Ein FPGA-Design arbeitet schneller als Software, die darauf ausgeführt wird. Daher integrierten wir eine FFT-(Fast Fourier Transform)-Bibliothek direkt in das SoC-Design.
Das Ergebnis ist ein System, das für zeitkritische Anwendungen, etwa in der Forschung, extrem geringe Latenzzeiten erreicht, nahezu vergleichbar mit einem ASIC.

4.1.4 Serielle Verbindung

Das Problem mit der seriellen Kommunikation war, dass wir keine Verbindung mit den gewählten Tools herstellen konnten. Die Fehlersuche dauerte einige Zeit, bis wir die Ursache erkannt und das Problem behoben hatten.
Nachdem die Verbindung schließlich funktionierte, erschien beim Start das LiteX-BIOS (siehe Abbildung 4.1).

4.1.5 Website und Marketing

Wie bereits erwähnt, wollten wir unser Projekt zu einem vollwertigen Produkt weiterentwickeln, das jeder einfach nutzen oder als Grundlage für eigene Vorhaben verwenden kann – mit den in Kapitel 4.1.3 beschriebenen Features.
Um das Projekt bekannter zu machen und Interessierten einen Eindruck über dessen Entstehung zu vermitteln, entwickelten wir eine Website. Diese gestalteten wir SEO-konform, sodass sie von Suchmaschinen wie Google leichter gefunden und auch von KI-Systemen über die bereitgestellte `sitemap.xml` erkannt und indexiert werden kann.
Die Website zum Projekt ist unter folgendem Link erreichbar: https://erik-donath.github.io/muTau-RV32-SoC/.

4.1.6 Git, Branches, Reviews und GitHub-Workflow

Um Fehler einfach korrigieren zu können und jederzeit auf den aktuellen Stand des Projekts zugreifen zu können, entschieden wir uns für **Git** als Versionsverwaltungssystem. Zur Synchronisation der Projektstände, für kollaboratives Arbeiten und als Cloud-Backup verwendeten wir

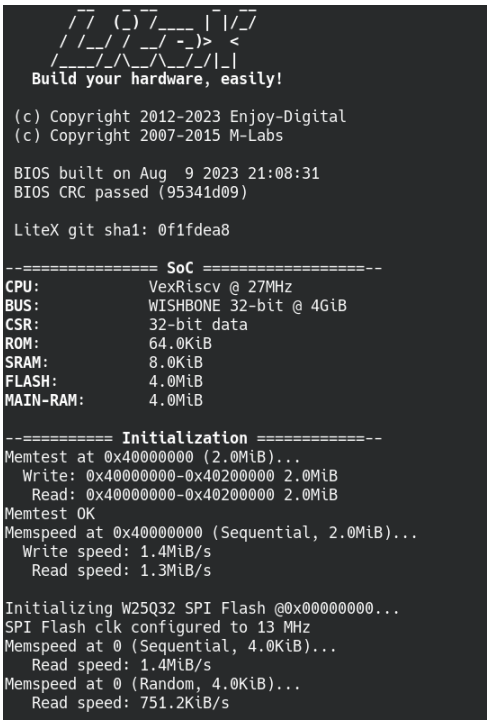

Abbildung 4.1: Das BIOS, das angezeigt wird, wenn man sich via Serial verbindet und den Befehl `reboot` eingibt.

zusätzlich **GitHub**. Dadurch war sichergestellt, dass jederzeit eine konsistente und aktuelle Version des Projekts online verfügbar war.

Durch die Nutzung von Branches hielten wir den *main*-Branch stets stabil und funktionsfähig. Für jeden neuen Entwicklungsschritt oder jedes neue Feature wurde eine eigene Branch erstellt. Diese Vorgehensweise verhinderte, dass experimentelle Änderungen die stabile Hauptversion beeinträchtigten, und schuf eine klare Trennung zwischen produktiver Entwicklung und neuen Funktionen.

Sobald eine Feature-Branch als funktionsfähig galt, nutzten wir das GitHub-Review-System (siehe z. B. dieses Beispiel). In diesem Workflow stellte Erik jeweils einen *Pull Request* mit einer Beschreibung der Änderungen, und Daniel überprüfte den Code, testete die Funktionalität und gab abschließend die Freigabe zum Merge. Auf diese Weise entstand ein klar strukturiertes, nachvollziehbares und dauerhaft wartbares Projekt. Zudem ermöglichte diese Arbeitsweise, dass beide Teammitglieder parallel an unterschiedlichen Bereichen arbeiten konnten, ohne dass Fortschritte verloren gingen oder Missverständnisse über den Projektstand entstanden.

Um das Projekt weitgehend zu automatisieren, richteten wir zusätzlich mehrere **GitHub Actions Workflows** ein. Diese führen diverse Aufgaben automatisch aus:

- **Build:** Baut das Projekt automatisch bei jedem Commit oder Release und generiert die

Bitstream-Dateien.

- **Pages:** Veröffentlicht die Projektwebseite automatisch über GitHub Pages.
- **CodeQL:** Führt eine statische Codeanalyse durch, um potenzielle Sicherheitslücken oder Fehler frühzeitig zu erkennen.

Durch diesen automatisierten Prozess können die erzeugten Bitstreams direkt als GitHub-Releases bereitgestellt werden. Dadurch müssen Nutzer, die sich nur für die fertigen FPGA-Bitstreams interessieren, das Repository nicht klonen oder selbst bauen, sie können die fertigen Builds bequem herunterladen.

Kapitel 5

Implementierung

5.0.1 Architekturübersicht (Hardware)

Das Projekt ist modular aufgebaut und besteht im Wesentlichen aus folgenden Teilen:

- **boards/** – Board-spezifische Plattformdefinitionen (Pinout, Ressourcen).
- **cores/** – Wiederverwendbare Hardware-Cores (z. B. HyperRAM/HyperBus).
- **soc/** – SoC-Definitionen und Builder-Logik (Python, Migen/LiteX).
- **firmware/** – Bare-Metal-Firmware-Targets (z. B. BIOS, einfache Beispiele).
- **docs/**, **pages/** – Dokumentation / GitHub Pages.
- **docker/**, **Makefile** – Build-Umgebung und Automatisierung.

Die Hardwarebeschreibung wird mit Migen/LiteX in Python geschrieben; daraus erzeugt der Builder die FPGA-Bitstreams (Gowin-Toolchain in diesem Projekt). Die Firmware nutzt die von LiteX generierten Header (CSR-Definitionen), um auf die Peripherie zuzugreifen.

5.0.2 Wichtige Softwarekomponenten

Im Folgenden werden die zentralen Python-Module aus `soc/` erklärt, die das System orchestrieren.

SoC-Konfiguration (SoCConfig)

Die `SoCConfig`-Dataclass fasst die build- und board-spezifischen Parameter zusammen (Boardname, Takt, RAM-Optionen, CPU-Konfiguration, Pfade). Wichtige Aspekte:

- Standard-Board: `tang_nano_9k`
- Sys-Clock-Default: 27 MHz

- Wahl: externe RAM-Nutzung oder SRAM-only
- Output-Pfad: `build/<board>`

Beispiel (Auszug): → A.7.

Basis-SoC (BaseSoC)

`BaseSoC` (erbt von `litex.soc.integration.soc_core.SoCCore`) setzt das SoC zusammen:

- Initialisiert Clock-/Reset-Generator (CRG).
- Wählt CPU-Typ und -Variante (z. B. VexRiscv).
- Fügt Main Memory hinzu (falls extern nutzbar).
- Lässt das Board proprietäre Peripherie hinzufügen (GPIOs, UART, LEDs, etc.).

Ausschnitt: A.6.

Clocking / CRG

Die Clock- und Reset-Logik ist in `soc/clocking.py` implementiert. Für Gowin GW1N/GW1NR Devices wird ein GW1N-PLL genutzt, ansonsten ein einfacher Pass-Through (sofern die Eingangsfrequenz der Ziel-Frequenz entspricht).

Beispiel (Kernlogik) → A.5.

Builder / Build-Flow

Der Build-Entry-Point ist `soc/builder.py`. Er erzeugt ein `BaseSoC`-Objekt, bindet LiteX's `Builder` ein und bietet Optionen zum Bauen, Flashen und Laden (SRAM):

Wesentliche Funktionen:

- `builder.build()` erzeugt den FPGA-Bitstream (Gowin-Tooling).
- `prog.flash(...)` und `prog.load_bitstream(...)` zum Flashen bzw. Laden.
- Ausgabe der CSR-Map (`csr.csv`) in den Output-Ordner.

Ausschnitt → A.4.

5.0.3 Firmware / Software-Interaktion

LiteX generiert beim SoC-Build Header- und Linker-Dateien (unter `build/<board>/software/include/ger`
Diese enthalten CSR-Definitionen (Control and Status Registers) und Regions-Definitionen (z. B. `regions.ld`), die Firmware benötigt:

- `generated/csr.h`: Helper-Funktionen / Makros zum Lesen/Schreiben von Peripherie-Registers (z. B. LEDs, UART).
- `generated/regions.ld`: Memory-Map (SRAM, Flash) für den Linker.
- `variables.mak`: enthält CFLAGS, LDFLAGS, LIBS für Firmware-Build (wird von Firmware-Build-Skripten gelesen).

Der typische Firmware-Start besteht aus:

1. Start-up-Assembly (crt0) initialisiert Stack, BSS, ggf. kopiert .data.
2. `main()` benutzt die LiteX-generierten CSR-APIs, um Peripherie zu steuern.
3. Firmware wird als BIOS oder als Kernel über SerialBoot/Flash geladen.

5.0.4 Beispiel: minimale Firmware (aus dem Baseline-Repository)

Eine typische Demo-Firmware toggelt LEDs über die CSR-API → A.2.

Architekturübersicht (Software)

Die Firmware ist ein kleines Bare-Metal-Programm für einen RV32-Kern (VexRiscv über LiteX). Die wichtige Idee:

1. LiteX (in `soc/`) erzeugt Board-/SoC-spezifische Header und Linker-Regionen (unter `software/include/generated`).
2. Das Firmware-Projekt kompiliert gegen diese generierten Header (CSR-Definitionen, Adressen).
3. Die Startsequenz (crt0.S) richtet die Laufzeitumgebung ein (Stack, BSS, .data).
4. `main.c` verwendet die generierten CSR-Access-Funktionen (z.B. `leds_out_write`) um die Hardware zu steuern.
5. Linker-Skript (`linker.ld`) sorgt dafür, dass alle Segmente im SRAM liegen (keine externe Flash-Nutzung).

Start-Up: `src/crt0.S`

Wesentliche Aufgaben:

- Stackpointer initialisieren (typischer Stack-Top: SRAM-Start + Offset).
- BSS-Bereich mit Nullen füllen.
- Optional .data von ROM nach RAM kopieren (für SRAM-only Konfiguration evtl. bereits korrekt).
- `main` aufrufen; falls `main` zurückkehrt, Endlosschleife.

Beispiel (aus `src/crt0.S`) → A.3.

5.0.5 Anwendungs-Entry: `src/main.c`

Das kleine C-Programm demonstriert direkte Nutzung eines LiteX-generierten CSR-APIs. Es inkludiert `generated/csr.h` – diese Header wird von LiteX erzeugt und enthält Register-/Zugriffs-Makros/-Funktionen für die SoC-Peripherie.

Code (aus `src/main.c`) → A.2.

Erläuterung:

- `generated/csr.h` definiert Funktionen wie `leds_out_write(uint32_t)` zum Schreiben in die LED-Register (per LiteX generiert).
- Die Endlosschleife wechselt LED-Muster mit einfachen Software-Delays.
- In einem echten Projekt würden hier zusätzlich Initialisierung, Interrupt-Setup oder Peripheriebehandlung stattfinden.

5.0.6 Linker-Skript: `linker.ld`

Das Linker-Skript nutzt LiteX-generierte Definitionsdateien, um Adresse/Größe des SRAMs (und ggf. anderer Regionen) zu benutzen. Es legt alle Segment-Aliase auf SRAM fest, damit die Firmware komplett im RAM liegt.

Auszug → A.1

Wirkung:

- Alle Abschnitte werden an Adressen gemappt, die in `generated/regions.ld` definiert sind (von LiteX bereitgestellt).
- Dadurch ist ein nahtloses Zusammenspiel zwischen LiteX-Hardware-Map und Firmware möglich.

5.0.7 Build-Konfiguration: `CMakeLists.txt` und Toolchain

Hauptaufgaben von CMake:

- Einlesen der LiteX-Variablen (z. B. CFLAGS, LDFLAGS, LIBS) aus `variables.mak`.
- Zusammensetzen der Compiler-/Linker-Flags (z. B. `-march=rv32i -mabi=ilp32`).
- Einbinden des localen `linker.ld` und Verhindern von Standard-Libs (bare-metal).

Toolchain (cmake/toolchain-riscv.cmake) setzt cross-compiler Pfade:

```
[
# cmake/toolchain-riscv.cmake
set(CMAKE_SYSTEM_NAME Generic)
set(CMAKE_SYSTEM_PROCESSOR riscv32)

set(CMAKE_C_COMPILER   /opt/riscv-toolchain/bin/riscv64-unknown-elf-gcc)
set(CMAKE_CXX_COMPILER /opt/riscv-toolchain/bin/riscv64-unknown-elf-g++)
...
```

5.0.8 Zusammenspiel mit LiteX (generated headers)

Wichtig ist, dass LiteX vor dem Firmware-Build ausgeführt wurde, damit die Dateien unter `LITEX_BUILD_DIR/software/include/generated` vorhanden sind. Diese generierten Dateien enthalten:

- `csr.h` / andere Header mit Register-Defines und Helper-Funktionen.
- `regions.ld`, `output_format.ld`, `variables.mak` für Linker- und Build-Parameter.

5.1 Test

Wie bereits in Abbildung 3.1 dargestellt, wurden mehrere Tests durchgeführt, um die korrekte Funktionsweise des SoCs sicherzustellen. Dazu zählten unter anderem einfache Schreibtests in CSRs (Control and Status Register) innerhalb der Verilatorsimulation, aber auch praktische Experimente direkt auf dem Board, etwa durch kleine Blinky-Programme, einfache Assembler-Befehle und verschiedene BIOS-Kommandos.

Kapitel 6

Projektabschluss

6.1 Fazit

Das ursprüngliche Ziel des Projekts haben wir nicht vollständig erreicht. Unser Plan, einen eigenen SoC bzw. eine CPU von Grund auf in VHDL zu entwickeln, ließ sich in der vorgesehenen Zeit nicht umsetzen. Wie jedoch aus dieser Dokumentation hervorgeht, war auch das alternative Projekt wesentlich umfangreicher, als zuvor angenommen. Wir haben uns anfangs deutlich überschätzt, konnten jedoch, trotz Zeitverzögerung – ein Ergebnis schaffen, das sich sehen lassen kann.

Am Ende entstand ein voll funktionsfähiges Produkt, das sich keineswegs hinter anderen Projekten verstecken muss. Mit etwas weiterer Entwicklung könnte es sogar marktreif sein, da unser SoC langfristig eine echte Alternative zu Mikrocontrollern wie Arduino oder gängigen MCUs darstellt (sobald die Integration vollständig abgeschlossen ist). Darüber hinaus haben wir unser Produkt durch eine integrierte FFT-Schaltung erweitert, die für Signalverarbeitung genutzt werden kann, direkt in Hardware, ohne zusätzliche Bibliotheken.

Ebenso ist es uns gelungen, das Projekt strukturiert und im Team organisiert anzugehen, eine ausführliche Dokumentation zu erstellen und sogar eine begleitende Website zu veröffentlichen. Trotz aller Erfolge müssen wir jedoch einräumen, dass wir uns mit diesem Vorhaben eindeutig übernommen haben. Wir haben unzählige Stunden außerhalb der Schule daran gearbeitet – und mittlerweile fühlt sich das Projekt fast an wie unser gemeinsames „Kind“, das wir über Monate hinweg großgezogen haben. Und das alles für gerade einmal eine Note in einem Nebenfach.

6.2 Ausblick

6.2.1 Weitere Ideen

Da unser Projekt bzw. Produkt bislang noch nicht zu 100 % fertiggestellt ist, wäre der erste Schritt, es vollständig abzuschließen. Anschließend gibt es jedoch zahlreiche weitere Ideen, die

wir gern umsetzen würden.

Da sich das Projekt besonders an Einsteiger und Hobbyanwender richtet, wäre ein wichtiger nächster Schritt, den Installationsprozess zu vereinfachen. Aktuell erfolgt das Flashen des FPGAs, indem dieser an einen Docker-Container gemountet wird. Über diesen Container wird das Board mit einem Makefile beschrieben, ein Prozess, der für Anfänger zu komplex sein kann. Eine sinnvolle Erweiterung wäre daher ein **Web-Installer**, der mittels **JTAG-WebUSB** direkt über den Browser die Bitstreams auf das FPGA lädt.

Eine weitere Idee ist die **Erweiterung der integrierten Signalverarbeitungstools**. Neben der FFT könnten zusätzliche Filter, etwa ein *Butterworth-Filter*, implementiert werden. Außerdem wäre ein weiterer SoC interessant, der sich speziell an Kryptographie-Anwendungen richtet, mit geeigneten Hardwaremodulen zur Unterstützung solcher Aufgaben.

Darüber hinaus könnten wir ein ganz neues Produkt entwickeln, ähnlich wie ein Xilinx-SoC. Dabei würden wir unser bestehendes Projekt erweitern, indem wir einen AXI-Bus an einige Pins anbinden, der zu einem weiteren FPGA führt. Über den Device Tree eines RTOS könnte dieser Bus mit den CSRs verbunden werden, wodurch es möglich wäre, einen zweiten FPGA dynamisch über HDL zu konfigurieren, während er gleichzeitig vom SoC gesteuert wird. Dies würde die Möglichkeiten unseres Systems erheblich erweitern.

Auch ein weiteres lohnenswertes Ziel wäre, den SoC so anzupassen, dass darauf **Linux** lauffähig ist, was ihn zu einem kleinen, vollwertigen Einplatinencomputer machen würde.

Zudem könnte unser aktuelles PWM-Protokoll verbessert werden, da es derzeit auf einem Workaround basiert und noch keine echte PWM-Implementierung darstellt.

6.3 Code

Da es aufgrund der Größe des Projekts und der vielen Submodule zu aufwendig wäre, den gesamten Code in eine einzelne ZIP-Datei zu packen, und dies außerdem die Speicherlimits von Moodle oder IServ überschreiten würde – stellen wir die Quellcodes stattdessen über GitHub zur Verfügung. Dort können die Projekte einzeln eingesehen oder als ZIP-Dateien heruntergeladen werden.

https://github.com/Erik-Donath/muTau-Zephyr
https://github.com/Erik-Donath/muTau-RV32-SoC
https://github.com/Erik-Donath/muTau-Barebone
https://github.com/Erik-Donath/muTau-PlatformIO

Anhang A

Anhang

A.1 Linker Script

```
/* linker.ld - SRAM-resident firmware for muTau-RV32-SoC */

/* Use LiteX-generated memory layout and output format. */
INCLUDE generated/regions.ld
INCLUDE generated/output_format.ld

/* Use sram (defined in regions.ld) for everything */
REGION_ALIAS("REGION_TEXT", sram);
REGION_ALIAS("REGION_RODATA", sram);
REGION_ALIAS("REGION_DATA", sram);
REGION_ALIAS("REGION_BSS", sram);
REGION_ALIAS("REGION_STACK", sram);

SECTIONS
{
    .text :
    {
        _ftext = .;
        KEEP(*(.init))
        *(.text .text.*)
        KEEP(*(.fini))
        _etext = .;
    } > REGION_TEXT

    ... (data, bss, stack) ...
```

```
}
```

A.2 main.c

```c
// src/main.c

#include <generated/csr.h>

int main(void) {
    while (1) {
        leds_out_write(0xAA);
        for(int i = 0; i < 1000000; i++);
        leds_out_write(0x55);
        for(int i = 0; i < 1000000; i++);
    }
    return 0;
}
```

A.3 crt0.S

```
/* src/crt0.S - Minimal startup code for VexRiscv on LiteX */

.section .init, "ax"
.global _start

_start:
    /* Set stack pointer to top of SRAM (0x10000000 + 0x2000 = 0x10002000) */
    li sp, 0x10002000

    /* Clear the BSS segment */
    la t0, _fbss
    la t1, _ebss
bss_loop:
    bgeu t0, t1, bss_done
    sw zero, 0(t0)
    addi t0, t0, 4
    j bss_loop
bss_done:
```

```
    /* Copy data section from ROM to RAM if needed (optional for SRAM-only) */
    la t0, _fdata
    la t1, _edata
    la t2, _fdata  /* In SRAM-only config, data is already in place */
data_loop:
    bgeu t0, t1, data_done
    lw t3, 0(t2)
    sw t3, 0(t0)
    addi t0, t0, 4
    addi t2, t2, 4
    j data_loop
data_done:

    /* Call main */
    call main

halt:
    j halt
```

A.4 load csr

```
# soc/builder.py
from litex.soc.integration.builder import Builder

def build_soc(config: SoCConfig, build=False, flash=False, load=
   False):
    soc = BaseSoC(config)
    builder = Builder(soc, output_dir=config.output_path, csr_csv
       =f"{config.output_path}/csr.csv")

    if build:
        builder.build()
    if flash:
        prog = soc.platform.create_programmer()
        bitstream = builder.get_bitstream_filename(mode="flash",
           ext=".fs")
        prog.flash(0, bitstream)
        bios = builder.get_bios_filename()
        prog.flash(0x40000, bios, external=True)
    if load:
```

```
        prog = soc.platform.create_programmer()
        bitstream = builder.get_bitstream_filename(mode="sram")
        prog.load_bitstream(bitstream)
    return builder

def main():
    # Argumentparsing: --board, --build, --flash, --load, ...
    # Erstellt SoCConfig und ruft build_soc auf
```

A.5 kernlogik

```
# soc/clocking.py
from litex.soc.cores.clock.gowin_gw1n import GW1NPLL
class ClockDomainGenerator(LiteXModule):
    def __init__(self, platform, sys_clk_freq, input_clk_name="clk27", input_clk_freq=27e6):
        self.rst = Signal()
        self.cd_sys = ClockDomain()
        clk_in    = platform.request(input_clk_name)
        reset_btn = platform.request("user_btn", 0)

        if hasattr(platform, "devicename"):
            self._create_gowin_pll(platform, clk_in, reset_btn, input_clk_freq, sys_clk_freq)
        else:
            raise NotImplementedError(...)

    def _create_gowin_pll(self, platform, clk_in, reset_btn, input_freq, output_freq):
        dev = platform.devicename
        if dev.startswith("GW1N") or dev.startswith("GW1NR"):
            self.pll = GW1NPLL(devicename=platform.devicename, device=platform.device)
            self.comb += self.pll.reset.eq(~reset_btn)
            self.pll.register_clkin(clk_in, input_freq)
            self.pll.create_clkout(self.cd_sys, output_freq)
        else:
            self.comb += self.cd_sys.clk.eq(clk_in)
            self.comb += self.cd_sys.rst.eq(~reset_btn)
```

A.6 BaseSoC

```python
# soc/base.py
from litex.soc.integration.soc_core import SoCCore
class BaseSoC(SoCCore):
    def __init__(self, config):
        self.soc_config = config
        board = get_board(config.board_name)
        platform = board.create_platform()

        self.crg = ClockDomainGenerator(
            platform=platform,
            sys_clk_freq=config.sys_clk_freq,
            input_clk_name=getattr(board, "input_clk_name",
               platform.default_clk_name),
            input_clk_freq=getattr(board, "input_clk_freq",
               config.sys_clk_freq),
        )

        SoCCore.__init__(
            self,
            platform,
            config.sys_clk_freq,
            cpu_type=config.cpu_type,
            cpu_variant=config.cpu_variant,
            cpu_reset_address=config.cpu_reset_address,
            integrated_rom_size=config.integrated_rom_size,
            integrated_sram_size=config.integrated_sram_size,
            ident=f"RISC-V␣SoC␣on␣{board.name}",
            ident_version=True,
        )

        if not self.integrated_main_ram_size and config.
           with_external_ram:
            board.add_main_memory(self, platform, config)

        board.add_peripherals(self, platform, config)
```

A.7 soconfig

```python
# soc/config.py
from dataclasses import dataclass
@dataclass
class SoCConfig:
    board_name: str = "tang_nano_9k"
    sys_clk_freq: float = 27e6
    with_external_ram: bool = True
    integrated_rom_size: int = 128 * 1024
    integrated_sram_size: int = 8 * 1024
    external_ram_size: int = 4 * 1024 * 1024
    cpu_type: str = "vexriscv"
    cpu_variant: str = "standard"
    build_name: str = "soc"
    output_dir: str = "build"
    # ...
    @property
    def output_path(self):
        return f"{self.output_dir}/{self.board_name}"
```

A.8 Quellen

Quellen

[EL25] Enjoy-Digital und LiteX developers. *LiteX: Build your hardware, easily!* https://github.com/enjoy-digital/litex. 2025. (Besucht am 30. 11. 2025).

[Geh23] Winfried Gehrke. *Digitaltechnik: Grundlagen, VHDL, FPGAs, Mikrocontroller*. 8. Aufl. 399 Abb., 3 Abb. in Farbe. Berlin: Springer Vieweg, 2023. ISBN: 978-3-662-63953-5. URL: https://www.lehmanns.de/shop/technik/57041214-9783662639535-digitaltechnik (besucht am 30. 12. 2025).

[lit24] litex-hub. *Zephyr on LiteX VexRiscv*. https://github.com/litex-hub/zephyr-on-litex-vexriscv. LiteX SoC builder for Zephyr on VexRiscv platform. GitHub, 2024. (Besucht am 30. 12. 2025).

[MM25] Mithro und Migen developers. *Migen Manual*. Read the Docs. 2025. (Besucht am 29. 11. 2025).

[Miy+20] Hiromu Miyazaki u. a. *An optimized RISC-V soft processor of five-stage pipelining*. 2020. URL: https://arxiv.org/pdf/2002.03568 (besucht am 30. 12. 2025).

[msy19] msyksphinz. *RV32I, RV64I Instructions*. 2019. URL: https://msyksphinz-self.github.io/riscv-isadoc/html/rvi.html (besucht am 29. 12. 2025).

[RS07] Juergen Reichardt und Bernd Schwarz. *VHDL-Synthese: Entwurf digitaler Schaltungen und Systeme*. 4. Aufl. Muenchen: Oldenbourg, 2007. ISBN: 978-3-486-58192-8. URL: https://www.lehmanns.de/shop/mathematik-informatik/7923972-9783486581928-vhdl-synthese (besucht am 01. 12. 2025).

[rik23] riktw. *Getting started with Litex on a Tang Nano 9K*. 2023. URL: https://justanotherelectronicsblog.com/?p=1263 (besucht am 10. 12. 2025).

[Spi25] SpinalHDL. *VexRiscv*. 2025. URL: https://github.com/SpinalHDL/VexRiscv?tab=readme-ov-file#vexriscv-architecture (besucht am 12. 12. 2025).

[Unk24] Unknown. *Zephyr on LiteX VexRiscv Demo*. https://www.youtube.com/watch?v=mTJ_vKlMS_4. YouTube Video Demonstration. 2024. (Besucht am 30. 12. 2025).

[Zep25a] Zephyr Project. *Getting Started with Zephyr*. https://docs.zephyrproject.org/latest/develop/getting_started/index.html. Zephyr Project, 2025. (Besucht am 30. 12. 2025).

[Zep25b] Zephyr Project. *LiteX VexRiscv Documentation*. `https://github.com/zephyrproject-rtos/zephyr/blob/main/boards/enjoydigital/litex_vexriscv/doc/index.rst`. Zephyr Project, 2025. (Besucht am 30. 12. 2025).