

NIBBLE
CPU



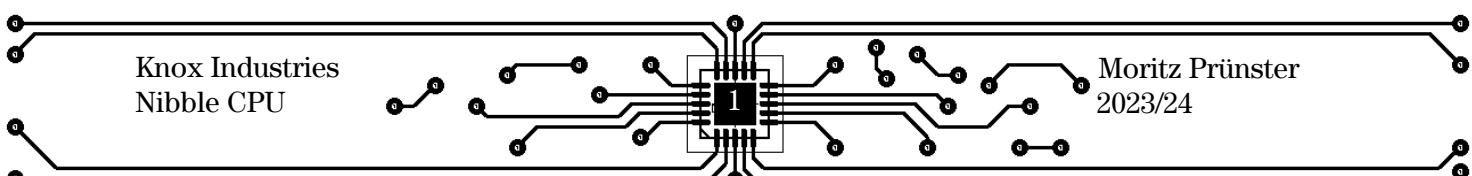
Moritz Prünster

Klasse 5B TFO
Elektronik

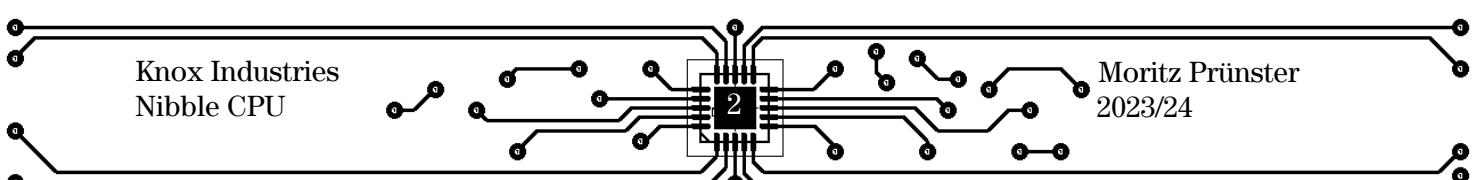
Schuljahr 2023/24

Inhalt

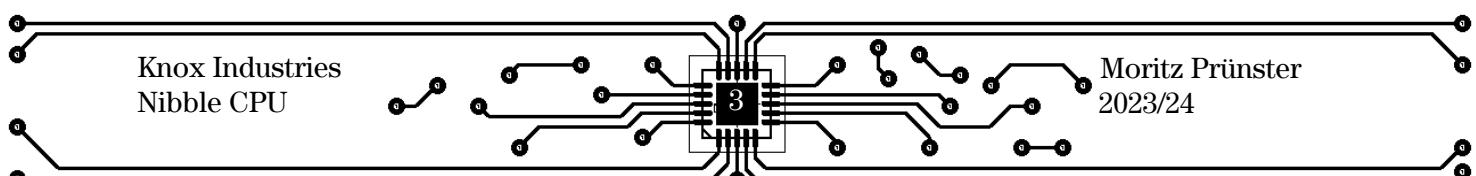
Inhalt.....	1
1. Eigenständigkeitserklärung.....	5
2. Vorwort.....	6
3. Projektbeschreibung.....	7
4. Astratto.....	8
5. Abstract.....	9
6. Motivation.....	10
7. Projektmanagement.....	11
7.1. Projektantrag.....	11
7.2. Pflichtenheft.....	12
7.3. Working Packages.....	13
7.4. Gantt-Diagramm.....	14
7.5. Wöchentlicher Projektbericht.....	15
7.6. Beraterstunden.....	15
8. Vorwissen.....	16
8.1. CPU.....	16
8.2. Binär.....	16
8.3. Assembler.....	17
8.4. Bauteile.....	18
8.4.1 NOT.....	18
8.4.2 OR.....	19
8.4.3 AND.....	19
8.4.4 XOR.....	20
8.4.5 JK-Flipflop.....	21
8.4.6 Tri-State Buffer.....	22
8.4.7 FT232.....	23
8.4.8 M48Z08.....	24
9. Benutzeranleitung.....	25
9.1. Installation.....	25
9.2. Programmierung.....	26
9.3. Assemblieren und Hochladen.....	27
10. Hardware.....	28
10.1 Steuereinheit.....	29
10.1.1 Pinboard.....	31
10.1.1.1 Die Clock.....	32
10.1.1.2 Die 4 Pins.....	33
10.1.1.3 Der Pulsgenerator.....	34
10.1.1.4 USB-Port.....	35



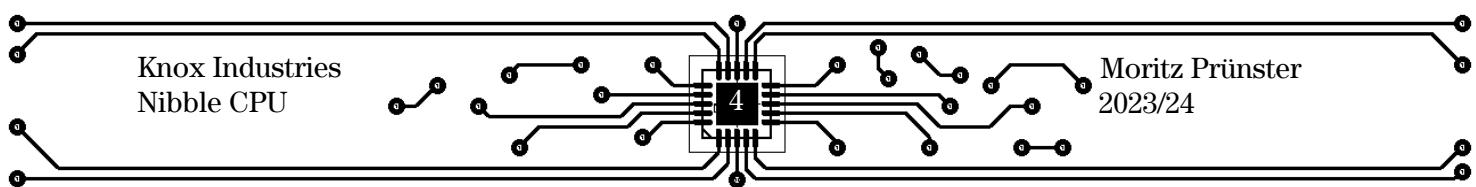
10.1.1.5 PC-Schleuse.....	36
10.1.2 Programmcounter(PC).....	37
10.1.2.1 PC CLock.....	37
10.1.2.2 PC Counter.....	38
10.1.2.3 PC Set.....	38
10.1.3 Serielle (SE).....	39
10.1.3.1 FT232.....	39
10.1.3.2 SE Timer.....	40
10.1.3.3 SE Clock.....	41
10.1.3.4 Auslese Strecke.....	41
10.1.3.5 SE Reset.....	42
10.1.3.6 SE Kurz Speicher.....	42
10.1.3.7 SE Ausgang.....	43
10.1.3.8 SE Kontrolle.....	44
10.1.4 Branch (BRA).....	45
10.1.4.1 BRA Befehlserkennung.....	46
10.1.4.2 Nullerkennung.....	47
10.1.4.3 Datenschleuse.....	47
10.1.4.4 BRA Schleuse.....	48
10.1.4.5 Zusammenfasser.....	49
10.1.4.6 BRA Counter.....	50
10.1.4.7 BRA Kurz Speicher.....	51
10.1.5 Speicher.....	52
10.1.5.1 M48Z08.....	53
10.1.5.2 Speicher Leser.....	54
10.2 Arithmetic Logic Unit(ALU).....	55
10.2.1 ALU Pinboard.....	57
10.2.1.1 Nummer/Adresse erkennung.....	58
10.2.1.2 Ergebnis abspeicherung.....	58
10.2.1.3 ALU Datenzusammenfasser.....	59
10.2.1.4 ALU Befehlsschleuse.....	59
10.2.1.5 Akku Befehle.....	60
10.2.2 ALU Befehlserkennung.....	61
10.2.3 Akkumulatoren.....	62
10.2.4 Ergebnis Abspeicherung.....	63
10.2.5 Addierer.....	64
10.2.6 Subtrahierer.....	65
10.2.7 Multiplizierer.....	66
10.2.8 Vergleicher.....	68
10.3 Register (Reg).....	69



10.3.1 Adressen Auswerter.....	70
10.3.2 Speicher.....	71
11. Platinen.....	74
12. NAL.....	76
12.1 Marker.....	76
12.2 Makros.....	76
12.3 Nummern.....	76
12.4 Start.....	77
12.5 Branch Commands(BCOM).....	77
12.5.1 BRA.....	77
12.5.2 BEQ.....	77
12.5.3 BCY.....	77
12.5.4 RE0.....	78
12.5.5 RE1.....	78
12.5.6 RE2.....	78
12.5.7 RE3.....	78
12.6 Arithmetic Commands(ACOM).....	79
12.6.1 REA.....	79
12.6.2 NOP.....	79
12.6.3 LDA.....	79
12.6.4 LDN.....	79
12.6.5 STA.....	80
12.6.6 ADN.....	80
12.6.7 ADA.....	80
12.6.8 SUN.....	80
12.6.9 SUA.....	81
12.6.10 MA1.....	81
12.6.11 MA2.....	81
12.6.12 MN1.....	81
12.6.13 MN2.....	82
12.6.14 CCF.....	82
12.6.15 ASH.....	82
12.6.16 ITS.....	82
12.6.17 CSF.....	83
13. NALO.....	83
14.Kostenberechnung.....	84
14.1 Materialkosten.....	84
14.2 Beraterstunden kosten.....	85
14.3 Arbeitsstunden.....	86
14.4 Prototype Kosten.....	87



14.5 Serien Produktionskosten.....	88
14.5.1 Bauteile.....	88
14.5.2 Zusammenrechnung.....	89
15. Anhang.....	91
16. Quellenverzeichnis.....	95



1. Eigenständigkeitserklärung

Hiermit bestätige ich, Moritz Prünster, geboren am 11.07.2005 in Meran, dass die vorliegende Dokumentation meiner Maturarbeit und das zugehörige Projekt für das Jahr 2023/24 mit dem Titel "Nibble CPU" eigenständig von mir verfasst bzw. erarbeitet wurden.

Ich versichere, dass ich die Dokumentation selbstständig erstellt habe. Jegliche Informationen, die nicht aus eigener Arbeit stammen, sind klar als solche gekennzeichnet und den entsprechenden Quellen zugeordnet.

Die Erstellung der Dokumentation und des Projekts erfolgte im Rahmen meines Abschlussjahres an der Technologischen Fachoberschule (TFO) "Oskar von Miller" in Meran und dient als Nachweis meiner eigenen Leistung.

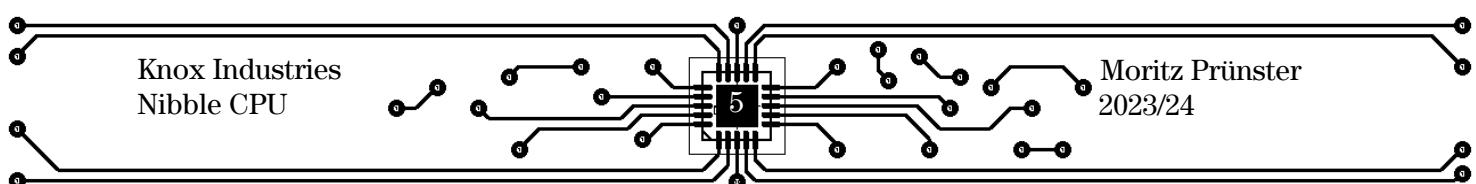
Projektleiter:

Der Direktor:

Die Tutoren/Lehrpersonen:

Ivan Huber

Martin De Tomaso



2. Vorwort

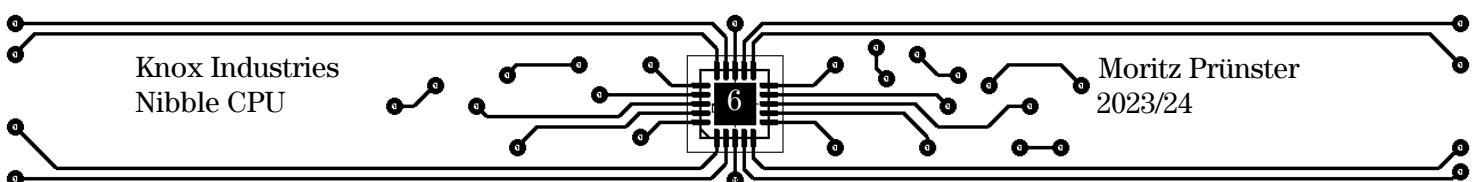
Sehr geehrte Leserinnen und Leser.

Die folgenden Seiten beinhalten eine technische Dokumentation eines Projekts, das wie jedes Jahr von den Maturanten der Technologischen Fachoberschule Meran "Oskar von Miller" im Schwerpunkt Elektronik und Elektrotechnik erarbeitet wurde.

Jedes Jahr wählen die Maturanten der Technologischen Fachoberschule Meran ein Projekt im Fach TPS (Technologie und Projektierung elektronischer Systeme) aus. Das Ziel dieser Projektarbeit besteht darin, die erlernten Inhalte des Trienniums praktisch anzuwenden, sie zu vertiefen und neue Fähigkeiten wie Dokumentation und Zeitmanagement zu erwerben.

Bei der Wahl des Projekts mussten die Schüler verschiedene Vorgaben und Aspekte berücksichtigen. Das Projekt sollte innerhalb der zur Verfügung stehenden Zeit realisierbar sein, die Verwendung eines Mikrocontrollers sowie die Erstellung einer dazugehörigen Leiterplatte für die benötigten Komponenten waren Pflicht. Außerdem sollten die Maturanten eine Software passend zum Projekt entwickeln. Dennoch war es unter bestimmten Umständen möglich, die Erlaubnis der Lehrpersonen zu erhalten, auch wenn nicht alle Vorgaben erfüllt wurden, wie es beispielsweise bei diesem Projekt der Fall war.

Die Ausarbeitung sollte möglichst selbstständig erfolgen, jedoch standen den Schülern bei Problemen und Fragen immer Lehrpersonen zur Verfügung. Die Projektarbeit wurde teilweise in der Schule in den Fächern TPS und FÜLA (Fachübergreifende Laborarbeiten) und teilweise zu Hause in der Freizeit der Schüler durchgeführt. Normalerweise wurde den Schülern an Montagen und Mittwochen die Möglichkeit gegeben, nachmittags im Elektroniklabor unter Aufsicht einer Lehrperson zu arbeiten. Leider konnte dies in diesem Jahr aufgrund fehlender Lehrkräfte nicht immer umgesetzt werden.



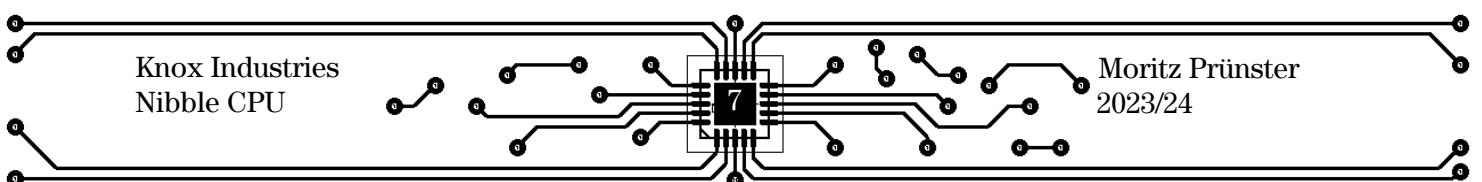
3. Projektbeschreibung

In meinem Maturaprojekt wurde eine 4-Bit (Nibble) Central Processing Unit (CPU) entwickelt. Diese CPU verfügt über eine eigene Architektur, die ausschließlich auf Logikgattern und Flip-Flops basiert. Zusätzlich ist die CPU in der Lage, mit einer speziell entwickelten Assemblersprache namens Nibble Assembly Language (NAL) programmiert zu werden. Diese Sprache ermöglicht es, die Funktionalität der CPU präzise zu steuern und komplexe Operationen auszuführen.

Die CPU unterstützt insgesamt 28 verschiedene Befehle, darunter grundlegende arithmetische Operationen wie Subtraktionen, Multiplikationen, Additionen sowie Vergleiche, die für bedingte Anweisungen entscheidend sind. Diese vielfältige Befehlssatzarchitektur ermöglicht eine breite Palette von Anwendungen und Nutzungsszenarien. Mit einer beeindruckenden Verarbeitungsgeschwindigkeit von 2400 Befehlen pro Sekunde ist die CPU in der Lage, komplexe Aufgaben schnell und effizient zu bewältigen.

Der Nibble Assembly Language Operator (NALO) ist ein eigens entwickelter Assembler, der in C++ geschrieben wurde. Er verfügt über die Fähigkeit, bis zu 13 verschiedene Programmierfehler zu erkennen und zu korrigieren. Der NALO wird auf einem PC ausgeführt und ist derzeit ausschließlich für die GNU/Linux-Plattform verfügbar. Um die Entwicklungsumgebung zu unterstützen, werden neben dem Assembler auch eine Programmieranleitung sowie eine Syntax-Hervorhebung für VIM bereitgestellt.

Das Hauptziel dieses Projekts war die Entwicklung einer voll funktionsfähigen CPU, die programmierbar und einsatzbereit ist. Zusätzlich wurde das Projekt ins Leben gerufen, um ein tiefergehendes Verständnis der grundlegenden Funktionsweisen einer CPU zu erlangen.



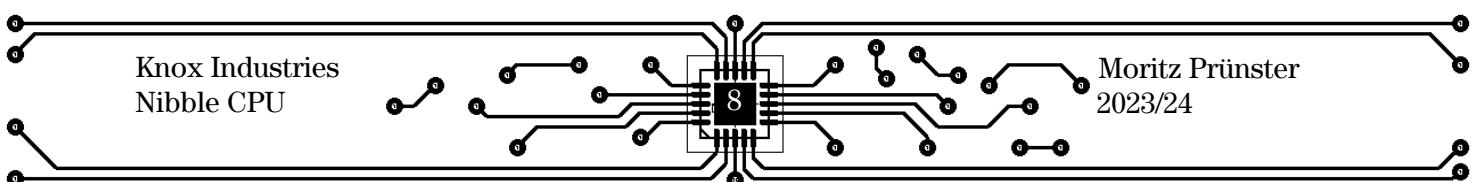
4. Astratto

Nel mio progetto di maturità è stata sviluppata un'unità di elaborazione centrale (CPU) a 4 bit (Nibble). Questa CPU presenta la sua propria architettura basata esclusivamente su porte logiche e flip-flop. Inoltre, la CPU è in grado di essere programmata utilizzando un linguaggio di assemblaggio appositamente sviluppato chiamato Nibble Assembly Language (NAL). Questo linguaggio consente un controllo preciso della funzionalità della CPU ed esecuzione di operazioni complesse.

La CPU supporta un totale di 28 istruzioni diverse, incluse operazioni aritmetiche di base come sottrazioni, moltiplicazioni, addizioni e confronti, che sono cruciali per le istruzioni condizionali. Questa diversificata architettura del set di istruzioni consente una vasta gamma di applicazioni e scenari d'uso. Con una velocità di elaborazione impressionante di 2400 istruzioni al secondo, la CPU può gestire rapidamente ed efficientemente compiti complessi.

Il Nibble Assembly Language Operator (NALO) è un assembler sviluppato su misura scritto in C++. Ha la capacità di rilevare e correggere fino a 13 errori di programmazione diversi. Il NALO funziona su un PC ed è attualmente disponibile esclusivamente per la piattaforma GNU/Linux. Per supportare l'ambiente di sviluppo, una guida alla programmazione e l'evidenziazione della sintassi per Vim sono fornite insieme all'assembler.

L'obiettivo principale di questo progetto era sviluppare una CPU completamente funzionale che fosse programmabile e pronta per l'uso. Inoltre, il progetto è stato avviato per acquisire una comprensione più profonda delle operazioni fondamentali di una CPU.



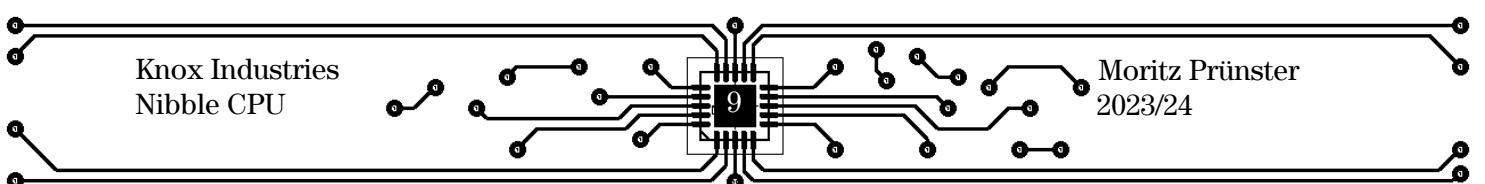
5. Abstract

In my high school graduation project, a 4-bit (Nibble) Central Processing Unit (CPU) was developed. This CPU features its own architecture based solely on logic gates and flip-flops. Additionally, the CPU is capable of being programmed using a specially developed assembly language called Nibble Assembly Language (NAL). This language enables precise control of the CPU's functionality and execution of complex operations.

The CPU supports a total of 28 different instructions, including basic arithmetic operations such as subtraction, multiplication, addition, and comparisons, which are crucial for conditional statements. This diverse instruction set architecture allows for a wide range of applications and usage scenarios. With an impressive processing speed of 2400 instructions per second, the CPU can handle complex tasks quickly and efficiently.

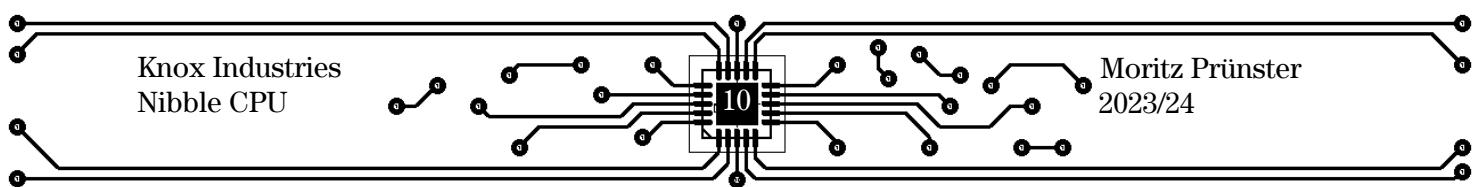
The Nibble Assembly Language Operator (NALO) is a custom-developed assembler written in C++. It has the ability to detect and correct up to 13 different programming errors. The NALO runs on a PC and is currently exclusively available for the GNU/Linux platform. To support the development environment, a programming guide and syntax highlighting for Vim are provided alongside the assembler.

The main goal of this project was to develop a fully functional CPU that is programmable and ready for use. Additionally, the project was initiated to gain a deeper understanding of the fundamental operations of a CPU.



6. Motivation

Ich habe dieses Projekt begonnen, um das Funktionsprinzip einer CPU zu verstehen. Bisher erhielt ich stets die Antwort "es funktioniert mit Einsen und Nullen", wenn ich nach dem Funktionsprinzip der CPU fragte. In der dritten Klasse haben wir Digitaltechnik behandelt, was mich meinem Ziel etwas näher brachte. Jedoch wurde meine Frage noch nicht vollständig beantwortet. Deshalb beschloss ich in der 4. Klasse, mein Füla-Projekt der CPU zu widmen und es auch als Maturaprojekt weiterzuführen. Zusätzlich haben mir die Kenntnisse aus der 4. Klasse bei diesem Projekt geholfen, da wir die DZ60 Assemblersprache gelernt und angewendet haben.



7. Projektmanagement

Um das Projekt reibungslos umzusetzen, erhielten wir von den Lehrkräften die Aufgabe, verschiedene Dokumente anzufertigen. Vom Projektantrag bis zum Gantt-Diagramm waren wir dazu angehalten, eine saubere und strukturierte Arbeitsweise beizubehalten.

7.1. Projektantrag

Als wir im September mit dem Projekt begannen, war es erforderlich, einen Projektantrag zu verfassen. In diesem Antrag haben wir eine kurze Beschreibung unseres Maturaprojekts angegeben und mit den Lehrkräften abgestimmt, welche Aspekte für die Bewertung wesentlich sind.

Projektantrag

Seite 1/2

Ansuchen für ein Maturaprojekt, Fassung September 2023

Projekttitel:	Nibble CPU
Projektleiter:	Moritz Prünster
Klasse und Schuljahr:	SBEL Schuljahr 2023 / 2024

Logo des Projektes:



Eine funktionierende CPU nur aus Logik Gates, die als Mikrocontroller agieren kann, mit einem eigenen Speicher und einer seriellen Schnittstelle, um Programme hinaufzuladen. Die CPU kann keinen Input lesen.
Als Basis dient die ALU aus der vierten Klasse, neu hinzukommt eine selbstgebaute SIO (Serial Input Output/ROM) und eine Steuereinheit, die somit die ALU, Speicher und SIO verbindet und eine vollständige CPU daraus macht.
Das Ziel ist es, ein selbstgeschriebenes Programm (in der eigenen Assembler Sprache) auf diese CPU hinaufzuladen und eine LED zum Blinken zu bringen.
Zusätzlich wird auf dem PC ein Programm geschrieben, das den Code über die serielle Schnittstelle zur selbst gebauten CPU schickt, sprich, ein Uploader wird erstellt.

Benötigung Relevante Meilensteine des Projektes:

- Meilenstein 1: Funktionierende ALU (Repariert, wurde beschädigt) (PW7)
- Meilenstein 2: funktionierende Testschaltung zur SIO/Abspeicherung (PW9)
- Meilenstein 4: Abgabe PCB Platine seriellen Schnittstellen (PW11)
- Meilenstein 3: funktionierende Platine zur Abspeicherung (PW15)
- Meilenstein 5: Funktionierendes PCB serielle Schnittstelle / PCB gelötet (PW21)
- Meilenstein 6: Schreiben der Seriellen Übermittlungsprogramms (PW22)
- Meilenstein 7: Abgabe des Projektes (PW32)

Abb. 1: Projektantrag scan Vorderseite

Projektantrag

Seite 2/2

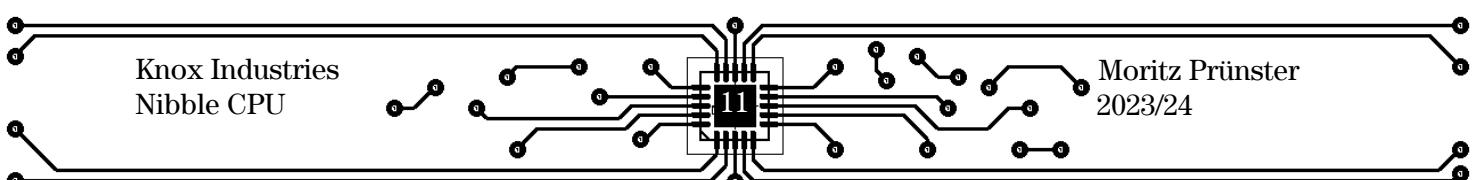
Anhänge:	<input type="checkbox"/> JA () Seiten	<input checked="" type="checkbox"/> NEIN
Projekt genehmigt:	<input checked="" type="checkbox"/> JA	<input type="checkbox"/> NEIN
Maximal erreichbare Note bei 100% Funktionalität: (unter Berücksichtigung der unten genannten Präzisierungen)	Zehn (10)	

Präzisierungen des PAG:
Fehlende SIO: - 1 Note
Fehlendes Steuerung: - 4 Noten

PAG 1: Martin De Tomaso
PAG 2: Ivan Huber
Der Projektleiter
Moritz Prünster



Abb. 2: Projektantrag scan Rückseite



7.2. Pflichtenheft

Im Pflichtenheft waren wir verpflichtet, eine detaillierte Erklärung abzugeben, die sowohl die Funktionsweise als auch die Umsetzung unseres Projekts umfasst. Diese Erklärung muss umfangreicher sein als der Projektantrag und alle Aspekte des Projekts einschließlich seiner Funktionalität, Implementierungsmethoden und Ziele genau beschreiben.

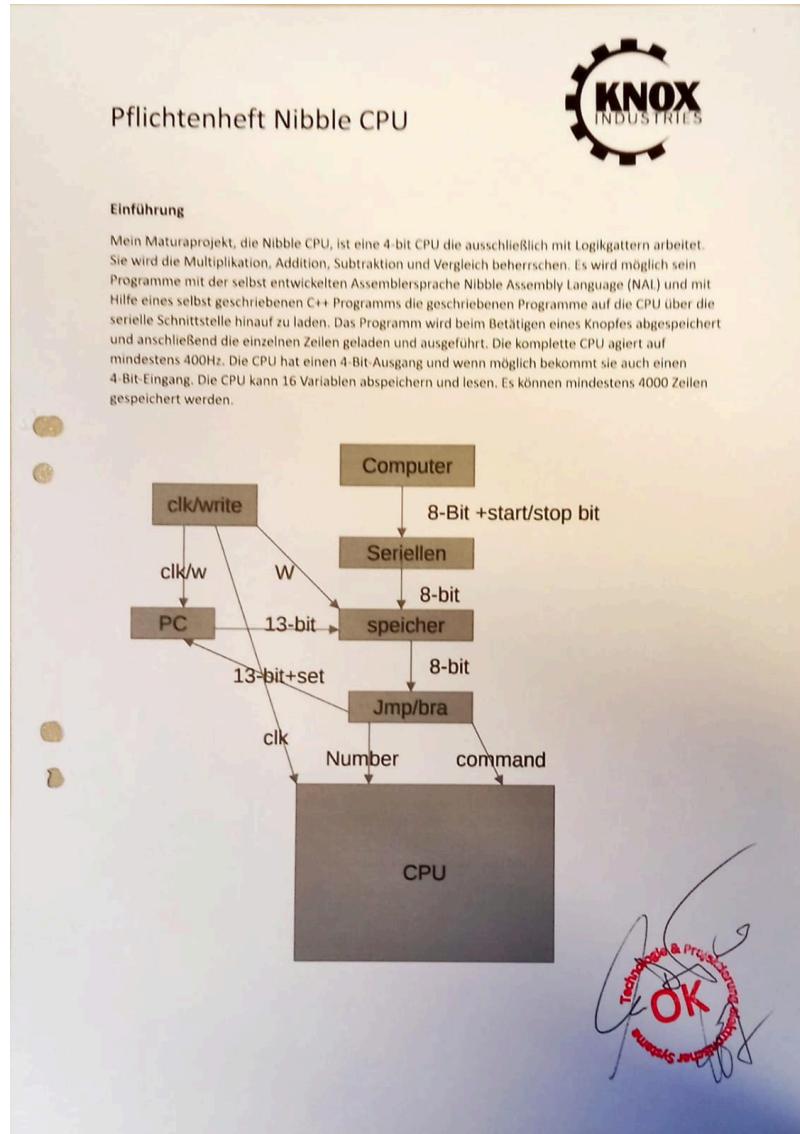
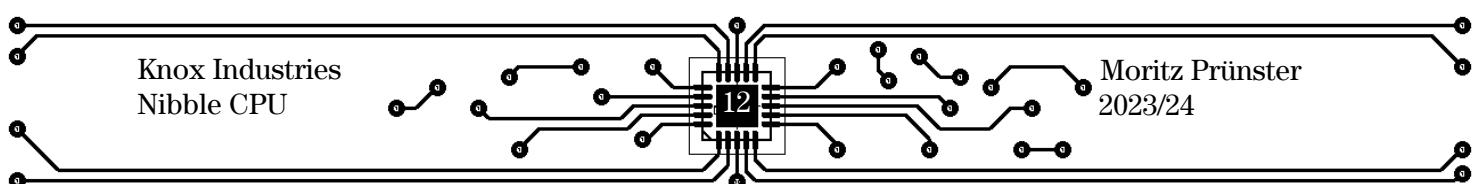


Abb. 3: Pflichtenheft



7.3. Working Packages

Die Working Packages dienen dazu, das Projekt besser zu strukturieren und einen klaren Überblick zu erhalten. Dazu wird das große Projekt in mehrere kleinere Teile unterteilt.

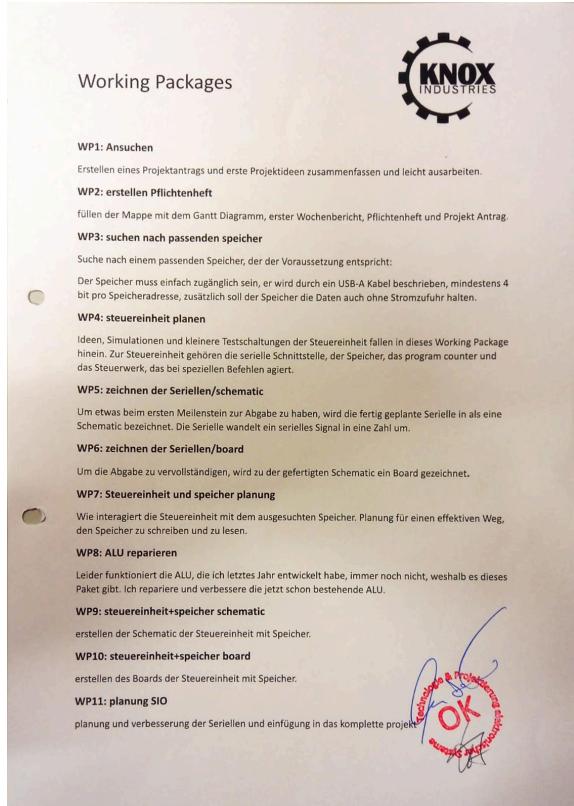
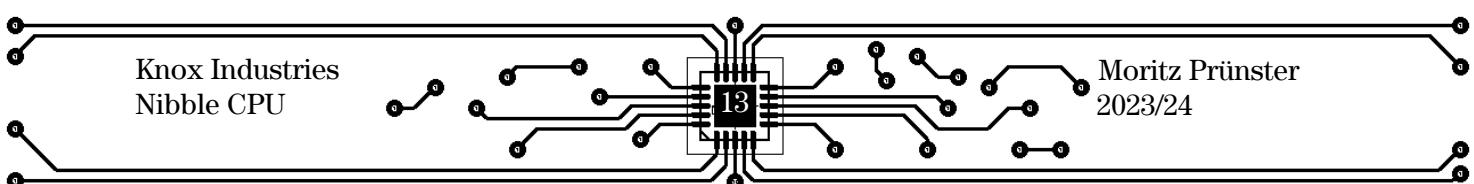


Abb. 4: Working Packages Vorderseite



Abb. 5: Working Packages Rückseite



7.4. Gantt-Diagramm

Im Gantt-Diagramm erstellt man eine grafische Darstellung, um zu visualisieren, wie viel Zeit man für ein Working Package hat und ob man im Zeitplan liegt oder Verzögerungen sind.



Abb. 6: Gantt Diagramm Seite 1

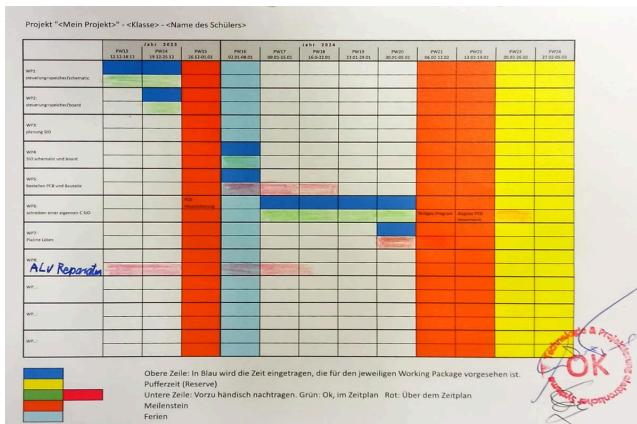
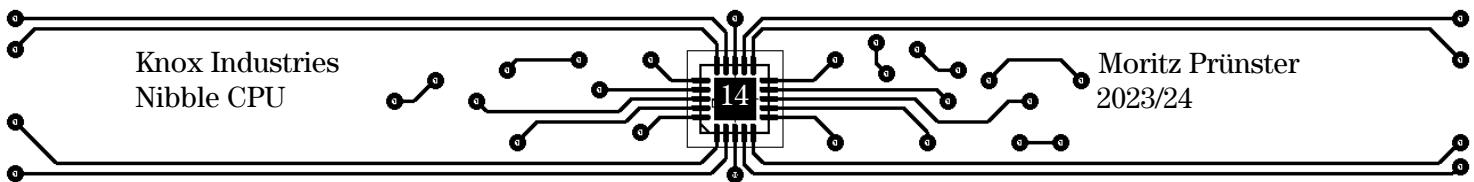


Abb. 7: Gantt Diagramm Seite 2



Abb. 8: Gantt Diagramm Seite 3



7.5. Wöchentlicher Projektbericht

Die Wochenberichte dienen dazu, den persönlichen Fortschritt am Projekt zu verfolgen und ermöglichen es den Lehrkräften, den Stand des Projekts zu überprüfen.

Wöchentlicher Projektbericht

KNOX
INDUSTRIES

Projekttitle:	Nibble CPU	
Projektleiter:	Moritz Prünster	
Projektwoche / Datum:	PW: 20	29.01.2024
		04.02.2024

Kurzbericht der erledigten Arbeiten in der Projektwoche:

• Weiter reparatur der ALU
Löten der Bauteile auf die platine
erster testlauf
programmierung des assemblierer

Im Zeitplan laut GANTT-Diagramm: JA NEIN

Wenn NEIN, welche Working Packages (WP) liegen in Verzug?
WP8: reparatur der ALU

Vorgeschlagene Maßnahmen, um den Zeitplan wieder aufzuholen:
neue programm schreiben um die ALU schneller durchzutesten

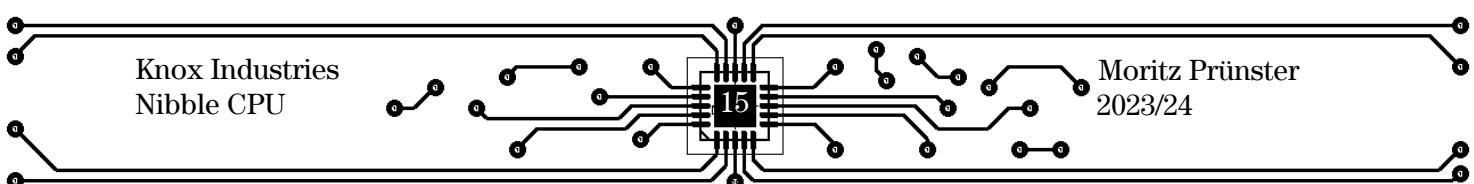
Wurden die Maßnahmen aus letzter JA NEIN Woche planmäßig umgesetzt?

Abb. 9: Wochenbericht

7.6. Beraterstunden

Wenn wir die Hilfe von Personen oder Lehrkräften in Anspruch nehmen, müssen wir die von ihnen für unser Projekt aufgewendete Zeit als Beraterstunden erfassen und die damit verbundenen Kosten entsprechend einbeziehen.

Person	Stunden
Lorenzo di Cello	2,25
Martin de Tomaso	9,5
Vladislav Yegerov	3
Ivan Huber	2



8. Vorwissen

Um ein gutes Verständnis über die Funktionsweise der Nibble CPU zu haben, verlangt es nach einem Vorwissen. In dem nun folgenden Unterkapitel wird einmal erklärt, was genau eine CPU ist und auch was eine Architektur ist, im nächsten Unterkapitel wird kurz erläutert, was ein Assembler funktioniert und im letzten Unterkapitel wird erklärt, was Binär ist und wie man es lesen verwenden kann.

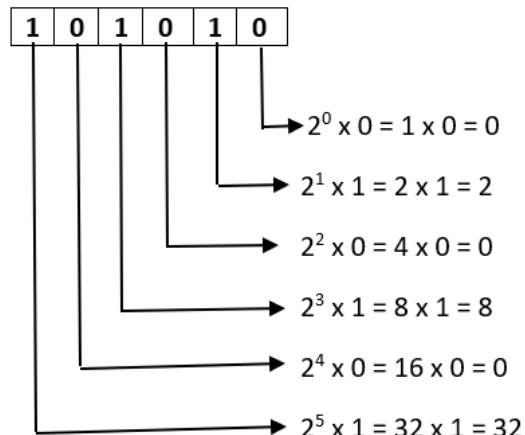
8.1. CPU

Eine CPU(Central Process Unit), oder zentrale Verarbeitungseinheit, ist das Gehirn eines Computers. Es ist das Herz des Computers, das dafür verantwortlich ist, Anweisungen auszuführen und Berechnungen durchzuführen. Die CPU nimmt Daten aus dem Speicher des Computers, führt die Operationen aus und gibt dann die Ergebnisse zurück. Kurz gesagt, die CPU ist der Hauptteil des Computers, der alles steuert und koordiniert, was der Computer macht.

8.2. Binär

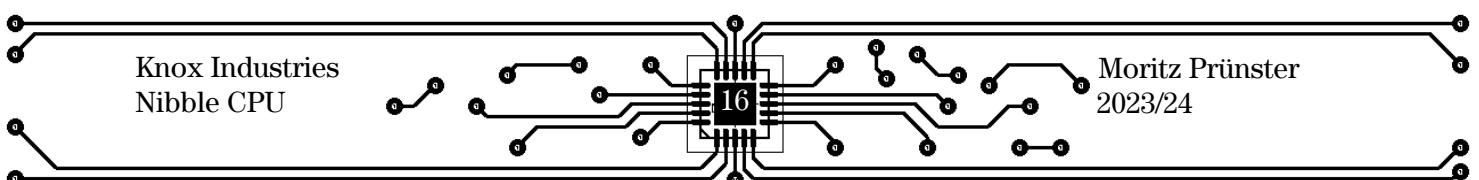
Im Dezimalsystem verwenden wir Zahlen von 0 bis 9, weil wir zehn Finger haben. Wenn wir bis neun zählen, beginnen wir wieder bei Null und fügen an der nächsten Stelle eine weitere Zahl hinzu, also kommt nach neun die Zahl Zehn.

Das Binärsystem funktioniert nach einem ähnlichen Prinzip, jedoch mit dem Unterschied, dass es nur zwei Ziffern gibt: 1 und 0. Beim Zählen wird aus einer 0 eine 1 und aus einer 1 eine 0, und an der nächsten Stelle wird weitergezählt.



Resultant decimal number= $0+2+0+8+0+32 = 42$

Abb. 10: Diagramm über Binäres lesen



8.3. Assembler

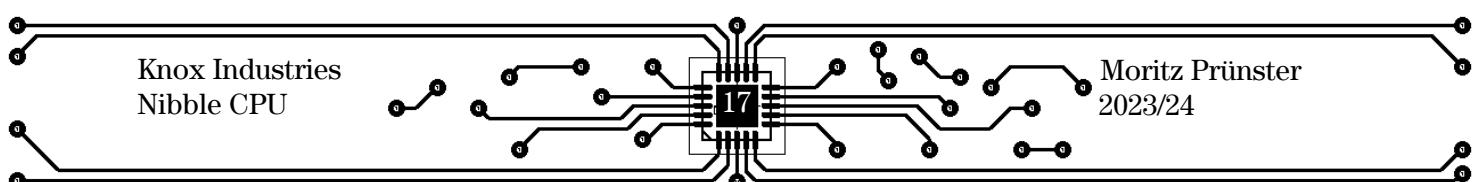
Assembler ist eine Programmiersprache, die eine niedrigere Ebene als Hochsprachen wie C++ oder Python hat. Es ist eine Art von Sprache, die direkt mit der Hardware des Computers kommuniziert. Im Gegensatz zu Hochsprachen, die für Menschen leichter zu lesen und zu verstehen sind, ist Assembler viel näher an der Maschinensprache des Computers, was bedeutet, dass die Befehle, die in Assembler geschrieben sind, direkt von der CPU des Computers ausgeführt werden können. Assembler-Programme bestehen aus Befehlen, die direkt die Aktionen der CPU steuern, wie beispielsweise das Laden von Daten aus dem Speicher oder das Ausführen von Berechnungen. Es ist eine leistungsstarke Sprache, aber auch komplexer und weniger intuitiv als Hochsprachen. Zum Beispiel würde Assembler code so aussehen:

```

ADC_Kanal_PB0
    lda      ADCRL
    sta      Analog_PB0
    lda      #9T
    sta      ADCSC1
    ldhx    #ADC_Kanal_PB1
    sthx    ADC_Pointer
    rts
ADC_Kanal_PB1
    lda      ADCRL
    sta      Analog_PB1
    lda      #10T
    sta      ADCSC1
    ldhx    #ADC_Kanal_PB2
    sthx    ADC_Pointer
    rts
ADC_Kanal_PB2
    lda      ADCRL
    sta      Analog_PB2
    lda      #11T
    sta      ADCSC1
    ldhx    #ADC_Kanal_PB3
    sthx    ADC_Pointer
    rts

```

Abb. 11: Assembler Beispiel

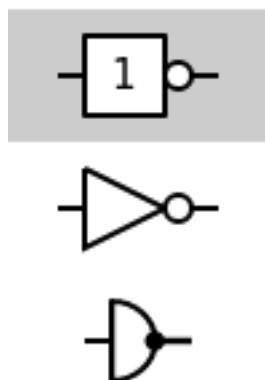


8.4. Bauteile

In diesem Abschnitt werden die verwendeten Bauteile erklärt. Grundsätzlich wurden neben Logikgattern und JK-Flip-Flops auch der FT232, der M48Z08 und Tri-State-Buffer verwendet. Alle Bauteile werden mit +5V betrieben, das bedeutet auch, dass HIGH oder 1 +5V und LOW oder 0 0V entsprechen.

8.4.1 NOT

NOT

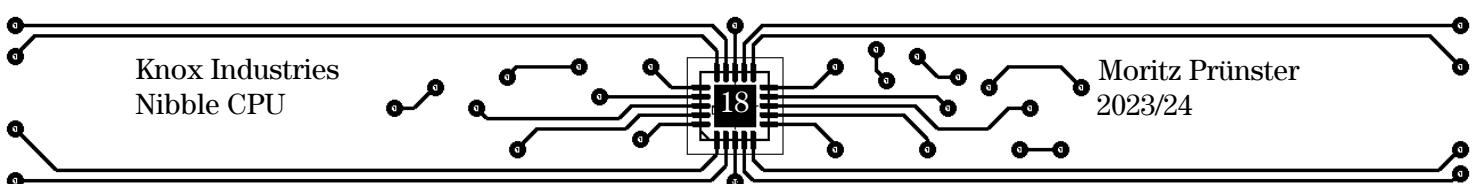


Ein NOT-Gatter ist ein logisches Gatter, das verwendet wird, um ein Signal zu invertieren. Das bedeutet, dass aus einem HIGH ein LOW und aus einem LOW ein HIGH wird. NOT-Gatter haben auch die Besonderheit, dass sie die Logik eines vorangehenden Logikgatters invertieren. Das bedeutet, dass ein OR-Gatter mit einem nachgeschalteten NOT-Gatter zu einem NOR-Gatter wird, ein AND-Gatter zu einem NAND-Gatter usw. Im Kapitel [10.1.1.1 Die Clock](#) wird auch von einem NOT-Gatter mit Schmitt-Trigger gesprochen. Dies ist ein NOT-Gatter, das zwei Schaltschwellen hat. Es bleibt so lange HIGH, bis es den oberen Schaltpunkt erreicht hat, und dann wieder LOW, sobald es den unteren Schaltpunkt erreicht hat.

Abb. 12: Not-Gatter

Wahrheitstabelle:

Eingang	Ausgang
0	1
1	0



8.4.2 OR

Ein OR-Gatter ist ein logisches Gatter, das zwei Eingänge und einen Ausgang besitzt. Dabei wird der Ausgang HIGH, wenn mindestens einer der Eingänge HIGH ist. Das OR-Gatter kann nur LOW am Ausgang haben, wenn beide Eingänge LOW sind. Zusätzlich wird in manchen Schaltungen ein OR-Gatter mit 4 Eingangspins verwendet.

Wahrheitstabelle:

Eingang 1	Eingang 2	Ausgang
0	0	0
0	1	1
1	0	1
1	1	1

OR

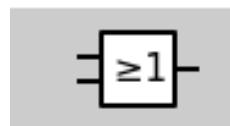


Abb. 13: OR

8.4.3 AND

Ein AND-Gatter ist ein logisches Gatter, das zwei Eingänge und einen Ausgang besitzt. Nur wenn beide Eingänge HIGH sind, ist der Ausgang HIGH. Man kann es auch als eine Art Schleuse verwenden, da das Signal A erst durchgeht, wenn das Signal B ebenfalls HIGH ist. Zusätzlich wird in manchen Schaltungen ein AND-Gatter mit 4 Eingangspins verwendet.

Wahrheitstabelle:

Eingang 1	Eingang 2	Ausgang
0	0	0
0	1	0
1	0	0
1	1	1

AND

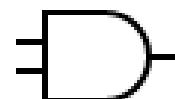
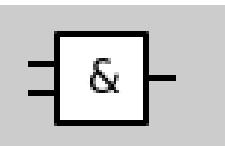
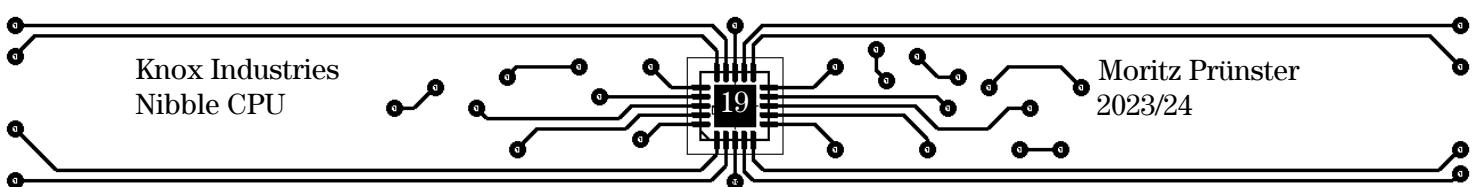


Abb. 14: AND-Gatter



8.4.4 XOR

Ein XOR-Gatter ist ein logisches Gatter, das zwei Eingänge und einen Ausgang besitzt. XOR steht für exklusives OR, da es ähnlich wie das OR-Gatter funktioniert, jedoch mit dem Unterschied, dass der Ausgang nur dann HIGH ist, wenn genau einer der beiden Eingänge HIGH ist. Wenn beide Eingänge HIGH sind oder beide LOW sind, ist der Ausgang LOW.

Wahrheitstabelle:

Eingang 1	Eingang 2	Ausgang
0	0	0
0	1	1
1	0	1
1	1	0

XOR

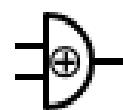
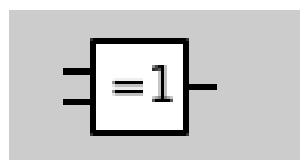
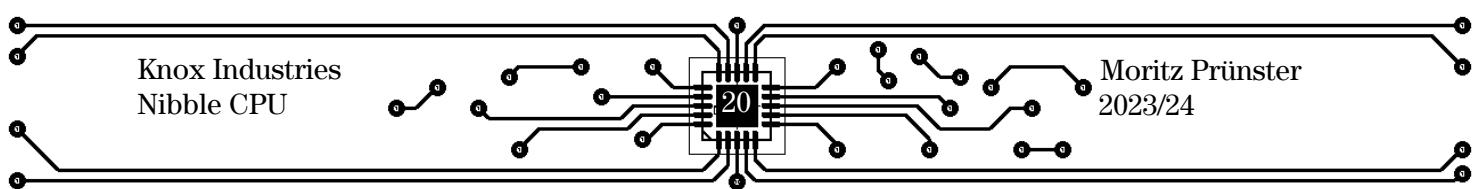


Abb. 15: XOR-Gatter



8.4.5 JK-Flipflop

Flip-Flops sind grundlegende elektronische Schaltungen, die in der digitalen Elektronik eingesetzt werden, um einen binären Zustand zu speichern. Sie dienen als grundlegende Speicherelemente und finden Anwendung in verschiedenen Anwendungen wie Speicherregistern, Zählern, Takterzeugungsschaltungen und sequenziellen Logikschaltungen. In diesem Projekt werden ausschließlich JK-Flip-Flops verwendet, da sie je nach Steuerung die Funktionalität jedes Flip-Flops übernehmen können.

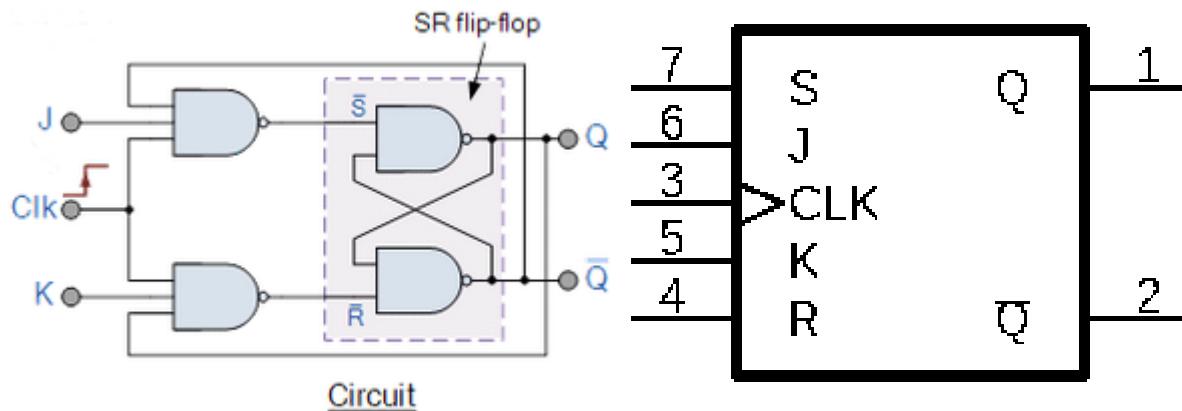
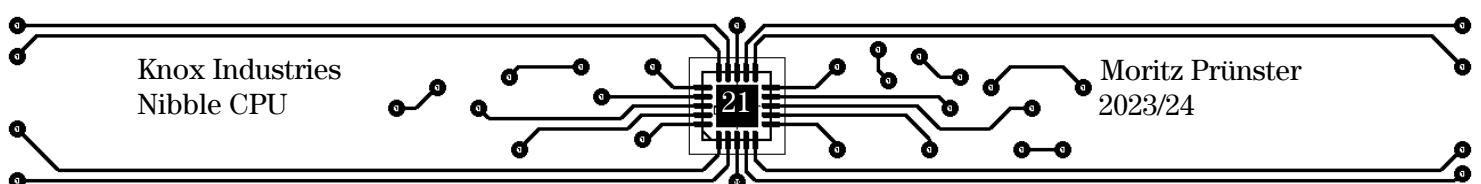


Abb. 16: JK-Flipflop aufbau

Abb. 17: JK-Flipflop symbol Eagle

clk	J	K	Q	Funktion
HIGH Flanke	0	0	n	Nichts
HIGH Flanke	1	0	1	Setzen
HIGH Flanke	0	1	0	Rücksetzen
HIGH Flanke	1	1	X	Toggeln

Wenn R ein HIGH Signal bekommt, wird es auf null gesetzt, unabhängig von den Zuständen der anderen Eingänge, das gleiche ist auch bei S, wenn es ein HIGH Signal bekommt, wird es auf HIGH gesetzt.



8.4.6 Tri-State Buffer

Ein Tri-State Buffer ist ein elektronisches Bauteil, das in digitalen Schaltungen verwendet wird. Es lässt Daten von A nach Y durch, wenn das OE Pin ein HIGH-Signal bekommt.

Zusätzlich ist der Ausgang Y hochohmig, wenn keine Signale kommen. Dies ist nützlich, um zum Beispiel die Daten aus dem Speicher und in den Speicher zu lesen, da man nur 8 bit hat und die Daten in beide Richtungen gehen.

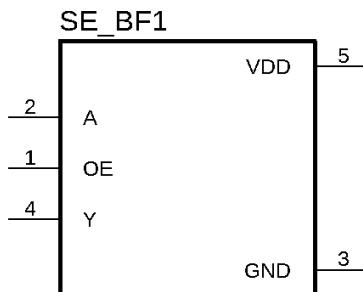


Abb. 18: Tri-state Buffer Eagle

8.4.7 FT232

Der FT232 ist ein USB-zu-Seriell-Wandler-Chip, der von FTDI (Future Technology Devices International) hergestellt wird. Er ermöglicht die Kommunikation zwischen einem USB-Anschluss und seriellen Geräten, indem er USB-Daten in serielle Signale umwandelt und umgekehrt. Dies macht ihn ideal für Anwendungen, bei denen ein Computer über USB mit Mikrocontrollern, Sensoren oder anderen seriellen Geräten verbunden werden soll. Der FT232 ist bekannt für seine Zuverlässigkeit und einfache Integration in verschiedene Projekte.

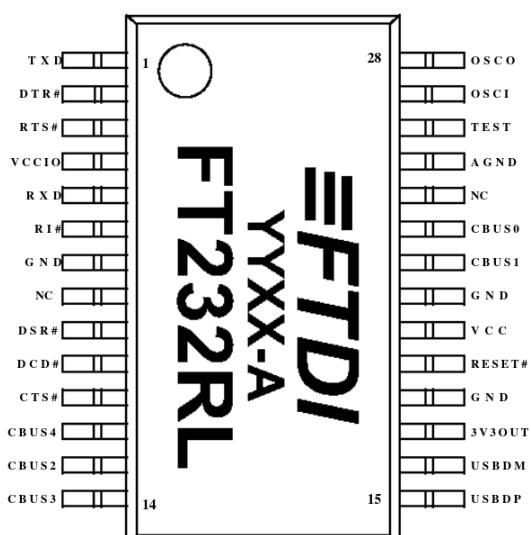


Abb. 19: FT232L Bauteildatenblatt

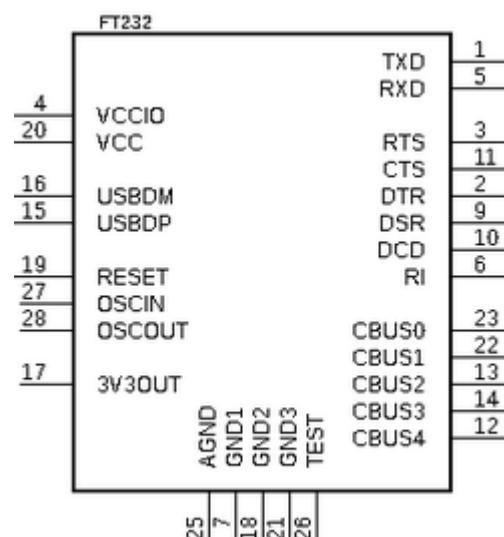
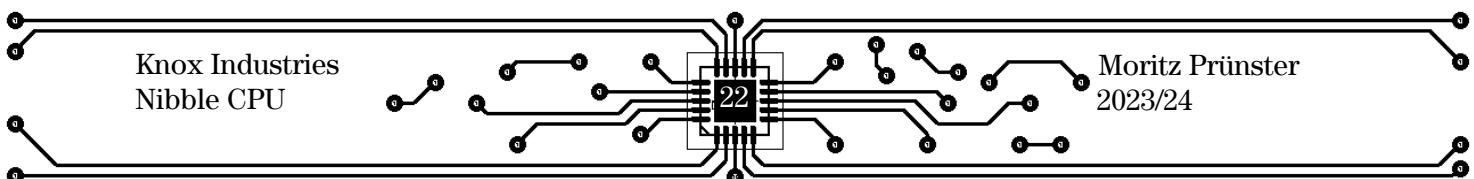


Abb. 20: FT232L Symbol Eagle



8.4.8 M48Z08

Der M48Z08 ist ein batteriebetriebenes RAM-Modul (NVRAM) von STMicroelectronics, das 64 Kilobit (8 KB) nichtflüchtigen Speicher bietet. Es kombiniert statisches RAM mit einer integrierten Batterie, die dafür sorgt, dass die gespeicherten Daten auch bei einem Stromausfall erhalten bleiben. Dieses Modul ist ideal für Anwendungen, die eine zuverlässige Speicherung von Daten über lange Zeiträume ohne externe Stromversorgung benötigen, wie etwa in Echtzeituhren Systemen oder sicherheitskritischen Datenaufbewahrung Systemen.

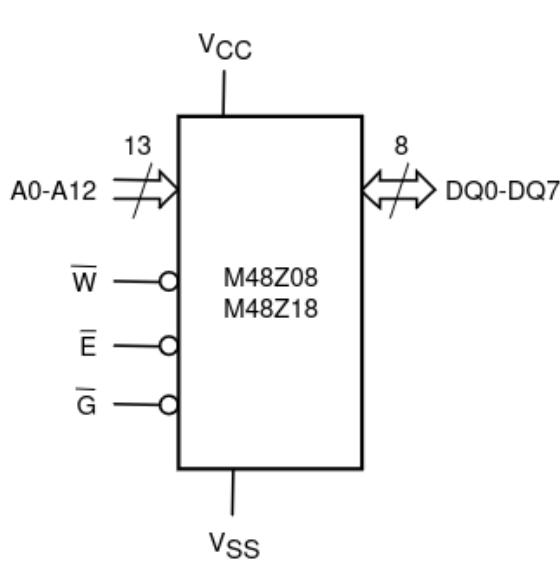
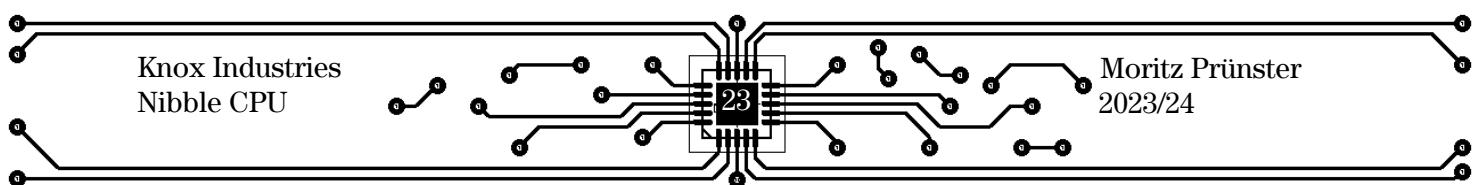


Abb. 21: M48Z08 Diagramm aus dem Datenblatt

NC	1	28	V _{CC}
A12	2	27	W
A7	3	26	NC
A6	4	25	A8
A5	5	24	A9
A4	6	23	A11
A3	7	M48Z08	22
A2	8	M48Z18	21
A1	9		A10
A0	10		20
DQ0	11		E
DQ1	12		19
DQ2	13		DQ7
VSS	14		18
			DQ6
			17
			DQ5
			16
			DQ4
			15
			DQ3

Abb. 22: M48Z08 Pin Layout Datenblatt



9. Benutzeranleitung

In diesem Kapitel wird erklärt, wie die Nibble CPU verwendet wird. Man benötigt die Nibble CPU, einen Computer und ein Mini USB Kabel.

9.1. Installation

Zuerst muss die erforderliche Software von GitHub heruntergeladen werden:

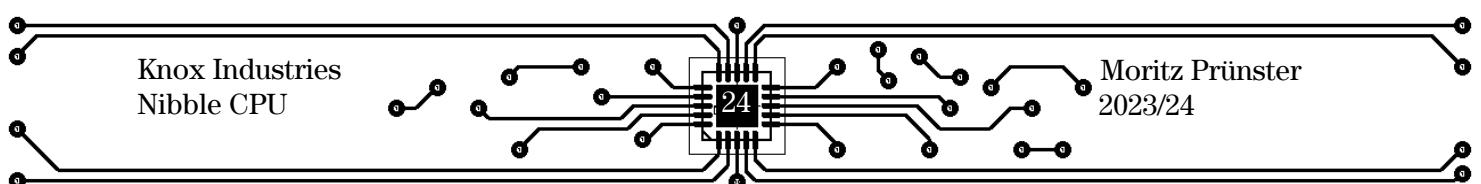
<https://github.com/Knoch1/NibbleCPU/>

Navigieren Sie dann in den heruntergeladenen Ordner und führen Sie die folgenden Befehle aus:

bash

```
sudo cp nalo /usr/local/bin/nalo  
cp nalo.1 /usr/share/man/man1  
cp nalo.vim ~/.config/nvim/syntax
```

Die Datei "nalo" enthält den kompilierten Assembler und "nalo.cpp" ist der Quellcode. Die Datei "nalo.1" ist das Handbuch, das mit dem Befehl `man nalo` geöffnet werden kann. Die Datei "nalo.vim" dient zur Syntaxhervorhebung in Neovim.



9.2. Programmierung

Nachdem "nalo" heruntergeladen und installiert wurde, können Sie die folgenden Befehle in der Befehlszeile eingeben, um die verfügbaren Befehle für die Programmierung anzuzeigen:

```
nalo -h
```

oder

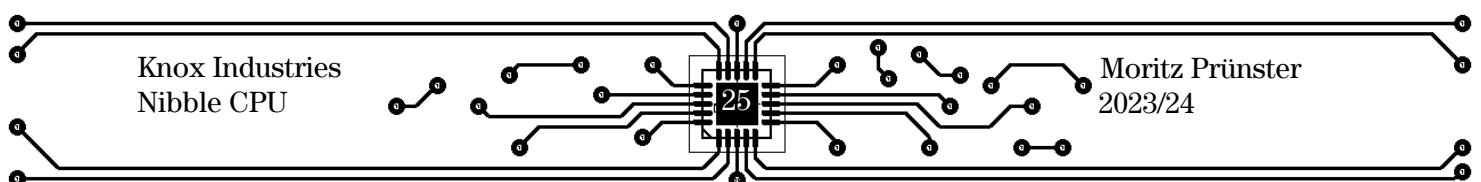
```
man nalo
```

Diese Befehle geben eine Liste der verfügbaren Befehle und Optionen aus, die beim Programmieren verwendet werden können.

Ein Programm würde mit der korrekten Text-Highlighting so aussehen.

```
1 ma:
2      test    0b0001
3 :me
4 #das ist ein kommentar
5 st:
6 Main
7      nop
8      lda     test
9      adn     1
10     sta    test
11     nop
12     bra    Main
13 :en
```

Abb. 23: Screenshot von einem Codeschnipsel



9.3. Assemblieren und Hochladen

Um das Programm hochzuladen, verbinden Sie die CPU über ein Kabel mit dem PC und verwenden Sie dann den folgenden Befehl:

```
nalo datei.nal usbport
```

Ersetzen Sie "datei" durch den Namen Ihrer Datei und "usbport" durch den entsprechenden Anschluss, an dem Sie das Programm hochladen möchten.

Der Assembler kann das Hochladen verweigern, wenn der Anschluss nicht gefunden wird oder ein Programmierfehler auftritt. Wenn Sie das Programm trotz Fehler hochladen möchten, fügen Sie "-f" (force upload) hinter den Dateinamen hinzu, etwa so:

```
nalo datei.nal -f usbport
```

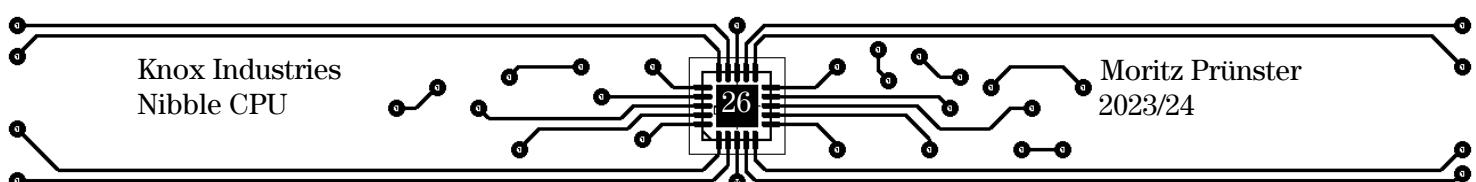
Es ist auch möglich, eine binäre Ansicht der hochgeladenen Datei zu erhalten, indem Sie einfach "-d"(debug) hinzufügen:

```
nalo datei.nal -f -d usbport
```

Sie können auch mehrere Dateien gleichzeitig zusammenstellen, indem Sie sie einfach nacheinander anhängen:

```
nalo datei1.nal datei2.nal -d -f usbport
```

Nach dem Hochladen müssen Sie einmal die Reset-Taste drücken, damit das Programm tatsächlich bei der ersten Zeile startet.



10. Hardware

In diesem Kapitel wird die Hardware genauer behandelt. Die Nibble CPU kann in drei Teile unterteilt werden: Steuereinheit, ALU(Arithmetic Logic Unit) und Register.

Diese drei Teile sind jeweils eine Platine, die dann mit Hilfe von Kabel miteinander verbunden sind.

Beim Ausführen eines Programms wird zunächst das gesamte Programm über die serielle

Schnittstelle vom Computer zur Steuereinheit gesendet. Diese speichert das Programm ab und ruft es dann nach dem Hochladen nacheinander auf. Dabei sendet die Steuereinheit die Daten an die ALU weiter oder führt bestimmte Befehle bereits aus, insbesondere wenn es sich um Branch-Befehle handelt. Die ALU führt die übermittelten Befehle aus. Dabei übernimmt sie nicht nur Berechnungen wie Addition, Subtraktion und Multiplikation, sondern auch Vergleiche, bei denen ein 1 Bit-Ergebnis zurückgegeben wird, das dann an die Steuereinheit gesendet wird.

Zusätzlich verarbeitet die ALU Befehle zum Speichern und Laden von Daten. Hierbei greift sie auf die Register zu, wobei sie die Adresse und die Nummer für das Abspeichern sendet. Je nach Befehl kann die ALU auch die abgespeicherte Nummer von einer speziellen Adresse erhalten. Das Register kann 16 verschiedene Variablen speichern, von denen jede 4 Bit groß ist. Wenn etwas an der Adresse 0000 gespeichert wird, wird dies direkt ausgelesen und an die Steuereinheit gesendet, die es dann an den entsprechenden 4 Ports ausgibt. Im gesamten Projekt wurden neben Logikgattern, Flip-Flops und Tristate-Puffern auch der M48Z08 (RAM mit Akku) als Speicher und der FT232 zur Auslesung der USB-Schnittstelle verwendet.

Das gesamte Projekt wurde mit circuitverse.org simuliert um das Projekt anzusehen folge diesem Link: https://circuitverse.org/users/143807/projects/nibble_cpu_v3

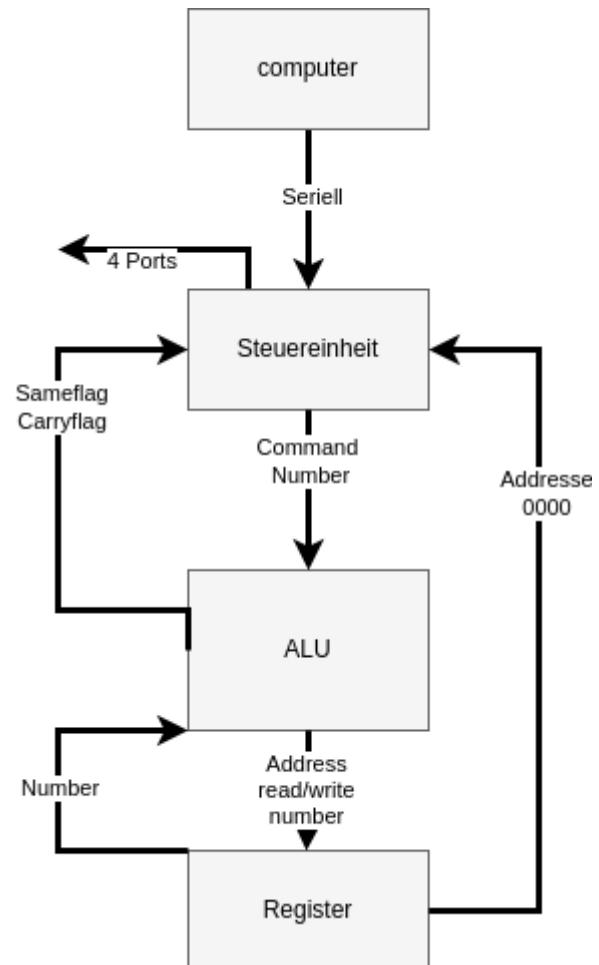
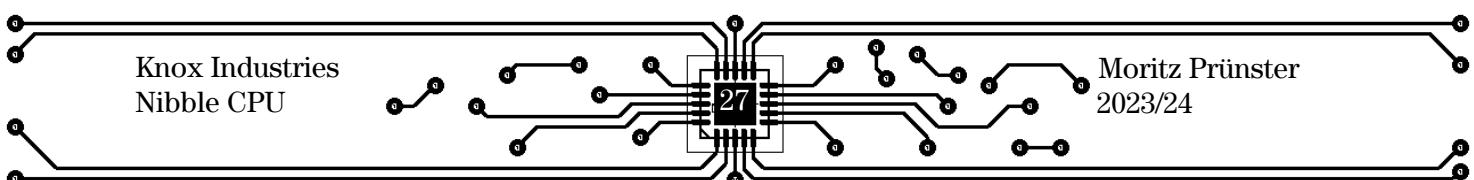


Abb. 24: Diagramm der Datenverteilung

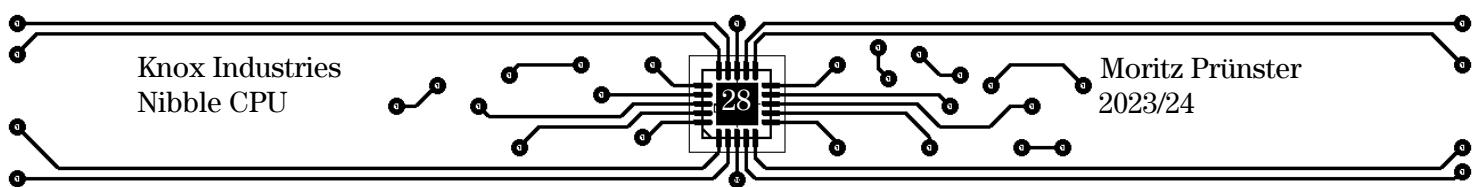


10.1 Steuereinheit

Die Steuereinheit speichert das hochgeladene Programm ab, liest es zeilenweise aus und sendet die Daten entweder an die ALU oder führt sie selbst aus. Die Befehle, die von der Steuereinheit ausgeführt werden, sind:

Die Steuereinheit ist, wie die anderen beiden Teile, in Blätter aufgeteilt. Hier werde ich jede Schaltung auf jedem Blatt einzeln erklären und besprechen. Die Steuereinheit ist in fünf Teile unterteilt: Pinboard, Programmzähler, Seriell, Branch und Speicher.

bra	branch
beq	branch if equal
bcy	branch if carry
rea	read the ports and send it to the ALU
re0	Branch if port 0 is HIGH
re1	Branch if port 1 is HIGH
re2	Branch if port 2 is HIGH
re3	Branch if port 3 is HIGH



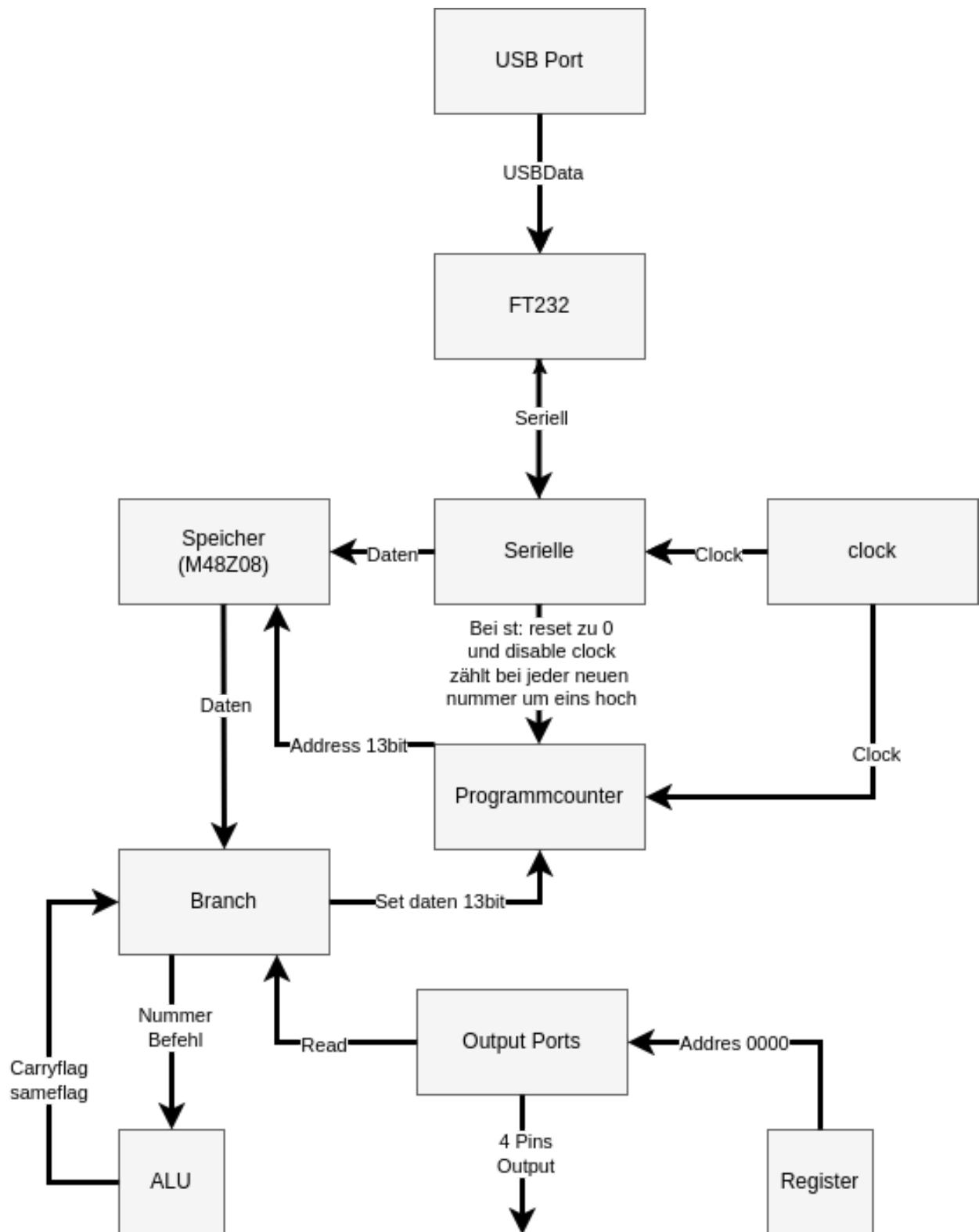
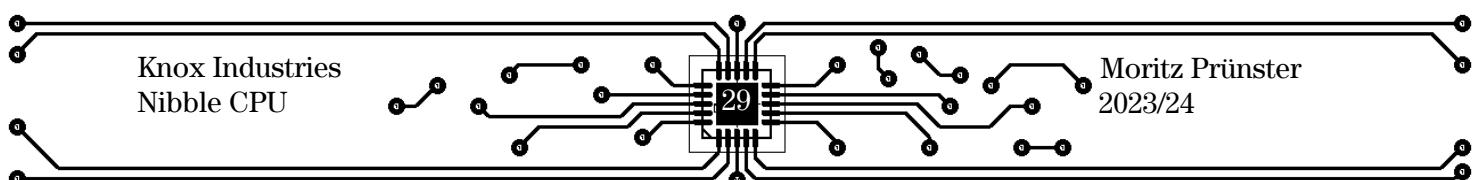


Abb. 25: Diagramm zur Datenverteilung in der Steuereinheit



10.1.1 Pinboard

Das Blatt "Pinboard" enthält kleine Schaltungen, die im gesamten Projekt verwendet werden. Auch die Verknüpfung zwischen den Blättern erfolgt meist über das Pinboard.

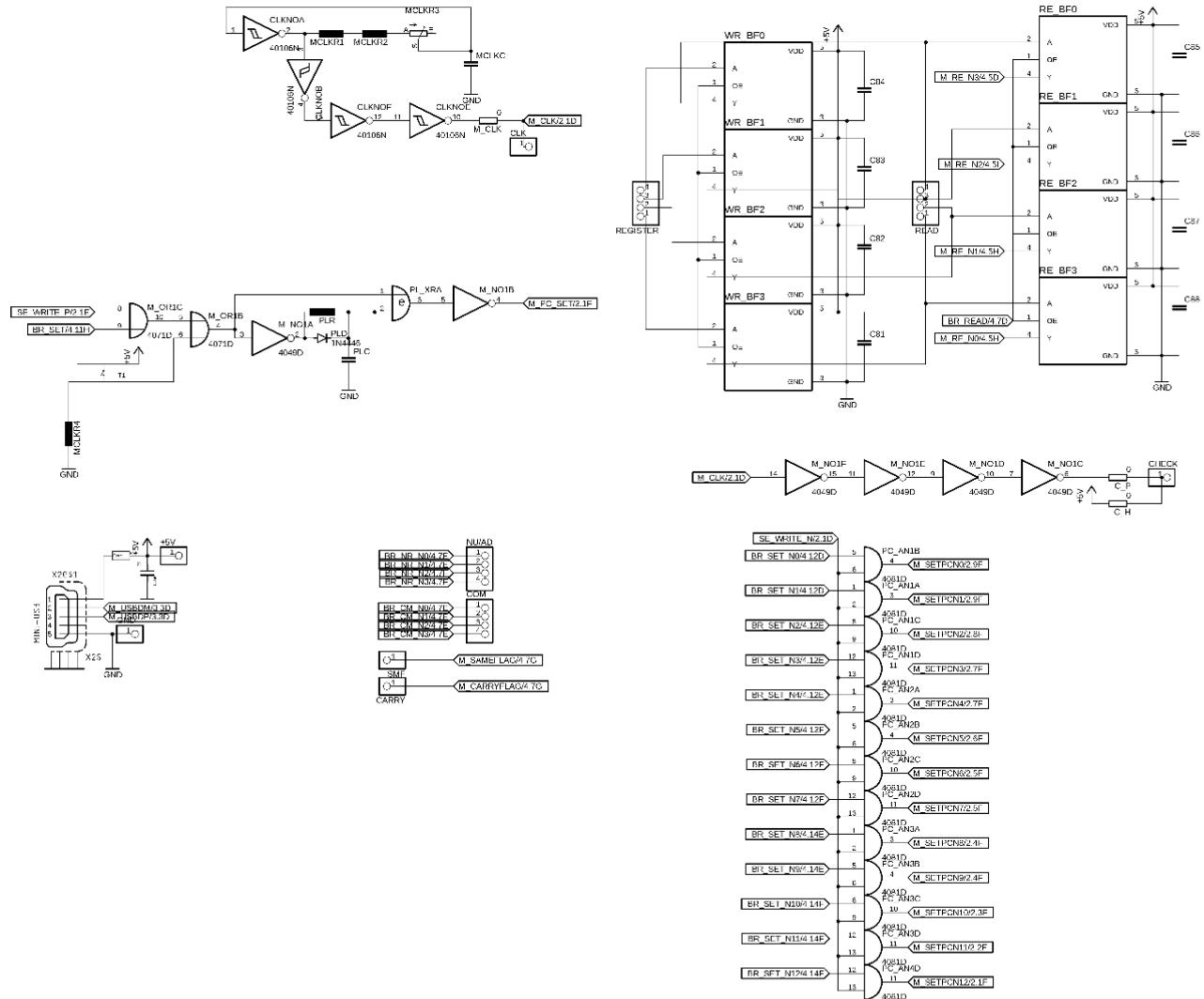
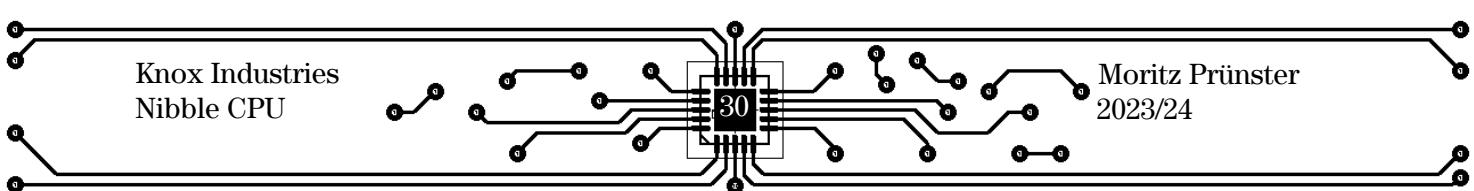


Abb. 26: Pinboard Schaltung Eagle



10.1.1.1 Die Clock

Die Steuereinheit verfügt über einen Taktgeber, der auf 2400 Hz eingestellt ist. Ursprünglich wurde eine Rechteckgenerator-Schaltung mit einem NOT-Schmitt-Trigger verwendet, jedoch erwies sich diese Schaltung aufgrund von Frequenzänderungen durch die Wärme des Bauteils als unzuverlässig. Daher wurde ein externer Taktgeber mit dem NE555 hinzugefügt.

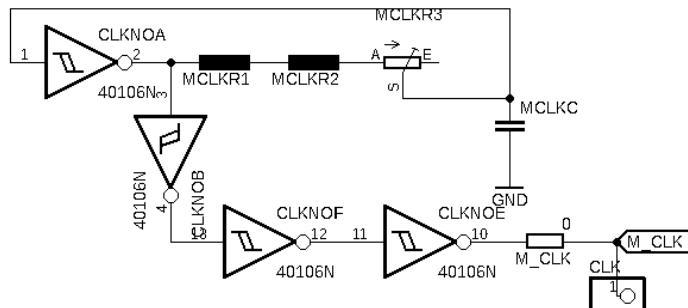


Abb. 27: NOT-Schmitttrigger Clock Schematic Eagle

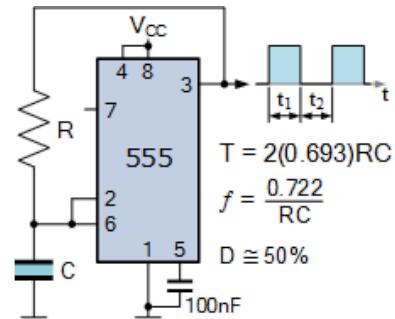
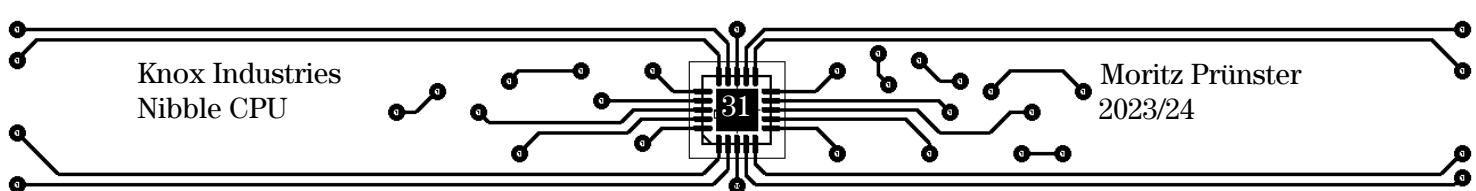


Abb. 28: NE555 Clock



10.1.1.2 Die 4 Pins

Die nächste Schaltung ist für die Verwaltung der 4 Pins zuständig. Die dort verwendeten Bauteile sind Tristate-Buffer, die einen gewissen Schutz ermöglichen. Logikgatter sollten keinen Strom am Ausgang haben, da dies ihre Funktionsweise beeinträchtigen kann. Daher habe ich hier zwei Sets von je vier Tristate-Buffer eingebaut. Die einen geben die Daten aus, die vom Register bei der Adresse 0000 ausgelesen werden, während die anderen vier Buffer die 4 Pins auslesen.

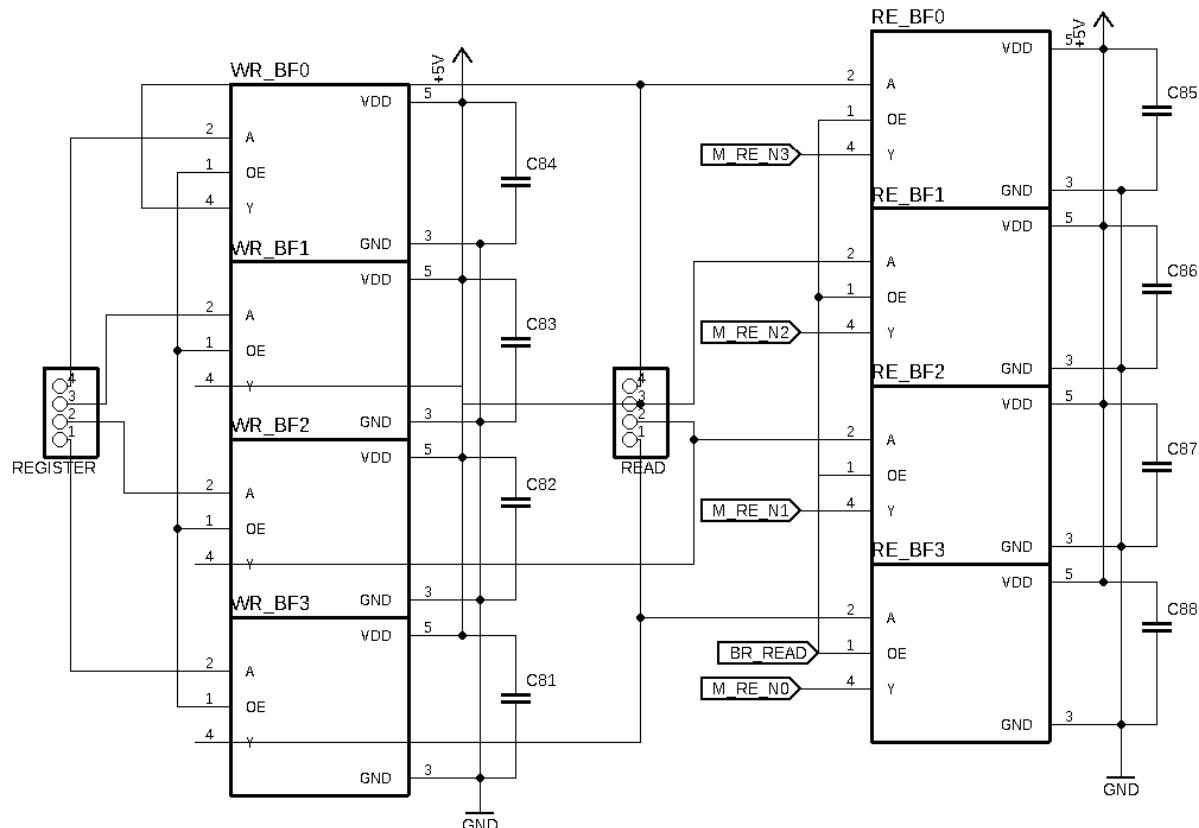
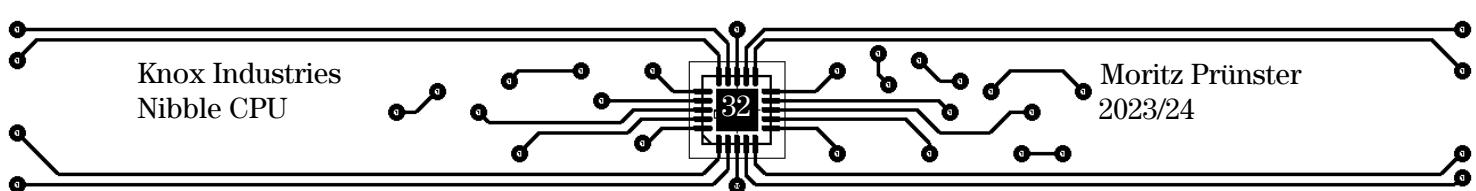


Abb. 29: Read schaltung Schematic Eagle



10.1.1.3 Der Pulsgenerator

Um den Programmzähler zurückzusetzen, benötigt man ein kurzes HIGH-Signal. Wenn dieses Signal zu lange dauert, besteht die Möglichkeit, dass der Zähler stecken bleibt und nicht mehr weiter zählt. Um dies zu verhindern, ist ein Impulsgenerator eingebaut. Dieser erzeugt aus einem langen Signal einen kurzen Impuls, um den Programmzähler zurückzusetzen. Das XNOR-Gatter wurde später durch ein AND-Gatter ersetzt, da der Impuls nur bei positiven Flanken gegeben werden sollte. Die Schaltung funktioniert wie folgt: Das Signal durchläuft einen OR-Gatter zum NOT-Gatter, der das Signal invertiert und das RC-Glied entlädt. Dadurch bleibt das Signal kurzzeitig über der oberen Schaltschwelle des XNOR/AND-Gatters, was dazu führt, dass beide Eingänge HIGH sind und ein kurzes Signal ausgegeben wird.

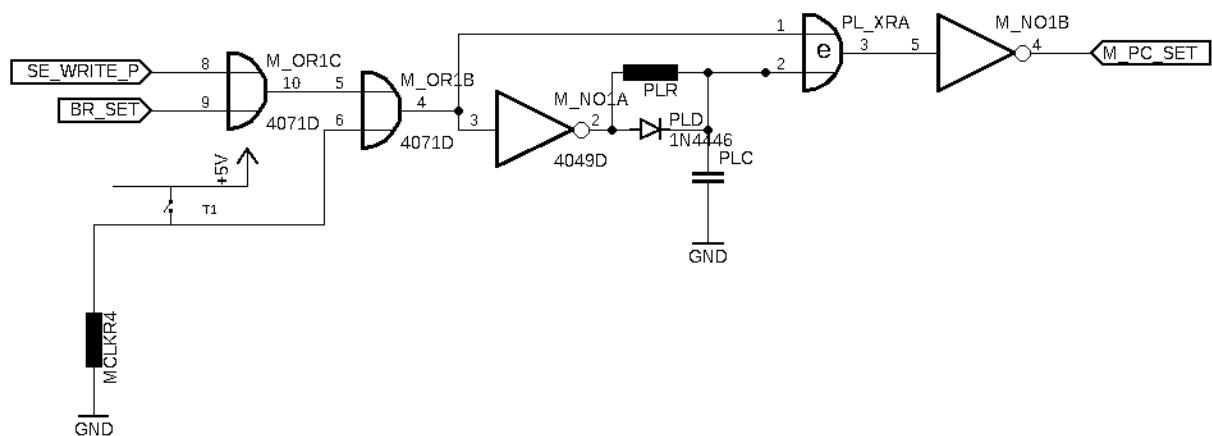
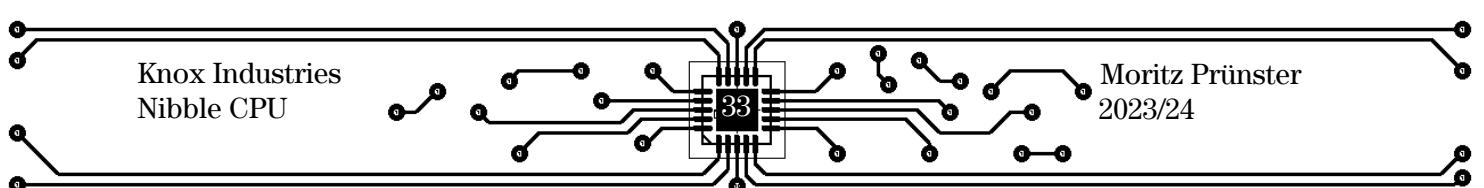


Abb. 30: Pulsgenerator Schematic Eagle



10.1.1.4 USB-Port

Die CPU bezieht ihren Strom ausschließlich über den USB-Port. Es wird ein Mini-USB-Port verwendet, da er einfacher anzuschließen ist. Dieser wird direkt auf der Platine angelötet. Zusätzlich zu den Pins für die Stromversorgung (+5V) werden auch die beiden anderen Pins für die Datenübertragung genutzt.

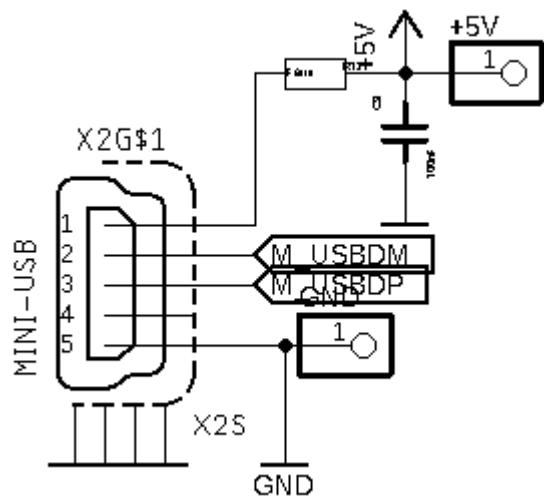
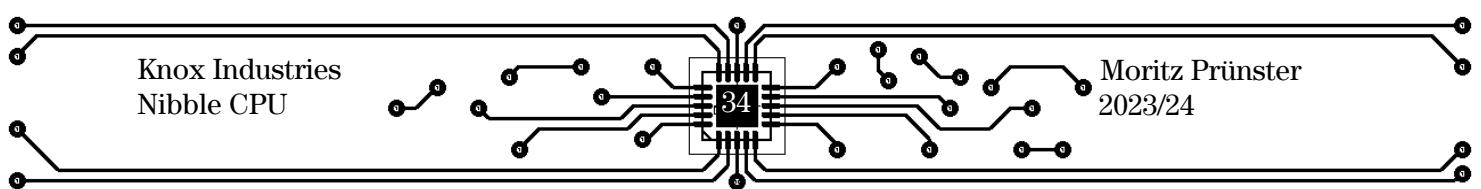


Abb. 31: USB-Port Schematic Eagle



10.1.1.5 PC-Schleuse

Die AND-Gatter sind dafür verantwortlich, die Reset-Daten für den Programmzähler zu steuern. Wenn kein Programm hochgeladen wird, ist "SE_WRITE_N" HIGH, und daher werden alle Daten durchgelassen. Wenn jedoch ein Programm hochgeladen wird, fällt "SE_WRITE_N" auf LOW, und der Ausgang ist immer 0, unabhängig davon, was vor den AND-Gattern liegt.

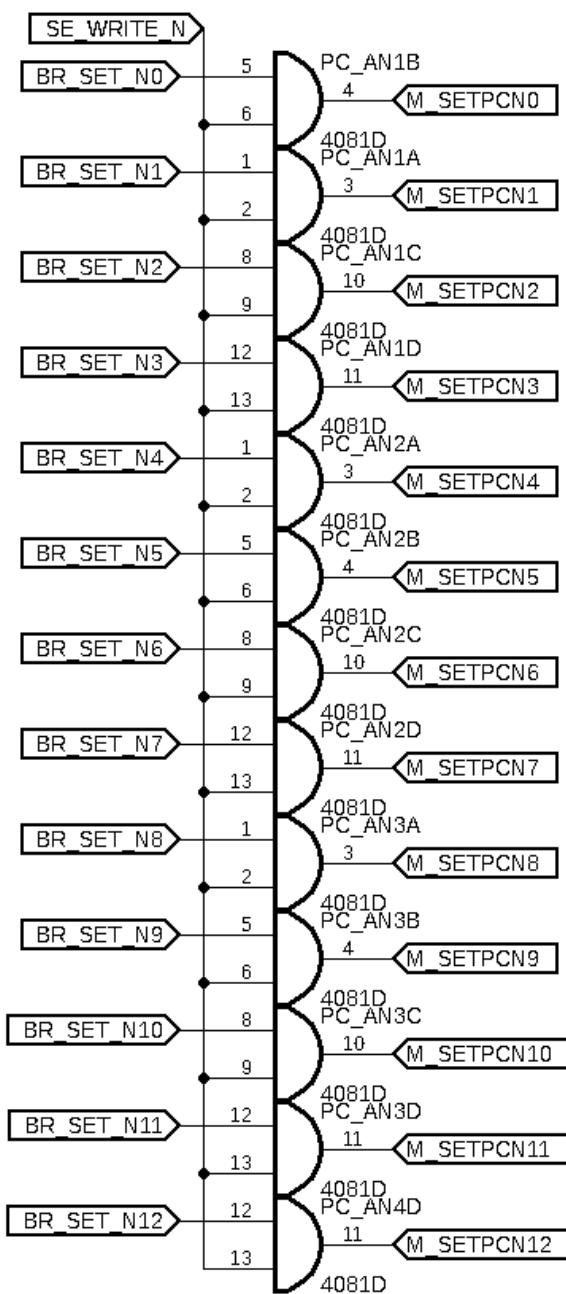
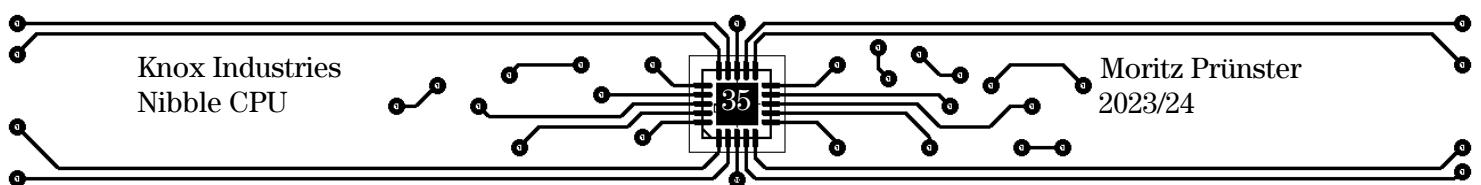


Abb. 32: PC-schleuse Schematic Eagle



10.1.2 Programmcounter(PC)

Der Programmzähler zählt bei jeder positiven Flanke des Taktsignals einen Schritt nach oben. Er verwendet dabei 13 Bits zum Zählen, was bedeutet, dass er von 0 bis 8191 zählen kann. Bei einer Taktfrequenz von 2400 Hz benötigt der Programmzähler ungefähr 3,4 Sekunden, um komplett durchgezählt zu werden und von Neuem anzufangen. Der Programmzähler wird verwendet, um durch die Zeilen des Programms zu zählen. Zusätzlich kann er durch den Reset-Knopf zurückgesetzt oder mit einem Branch-Befehl auf einen bestimmten Wert gesetzt werden.

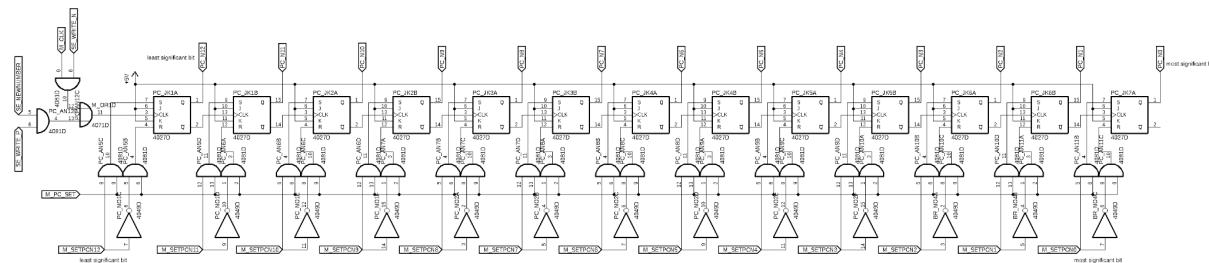


Abb. 33: Programmcounter Schematic Eagle

10.1.2.1 PC CLock

Die Schaltung wurde so konzipiert, dass der Programm Counter nicht immer mit 2400 Hz zählt. Wenn Daten vom Computer zur CPU gesendet werden, ist SE_WRITE_P HIGH und SE_WRITE_N LOW. Dadurch wird das Signal SE_NEWWNUMBER durchgelassen und bei jeder neuen Nummer ein kurzes Signal erzeugt. Auf der anderen Seite verhindert SE_WRITE_N, dass das Haupt-Taktsignal durchläuft, wenn Daten gesendet werden. Wenn keine Daten gesendet werden, zählt der Programmzähler mit den normalen 2400 Hz nach oben.

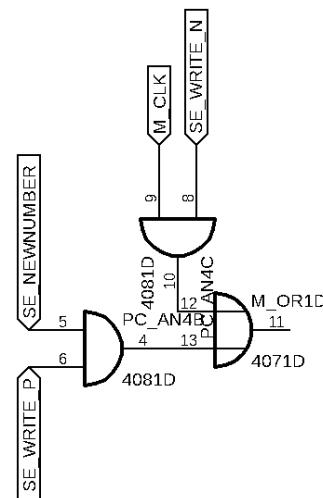


Abb. 34: PC CClock Schematic Eagle

10.1.2.2 PC Counter

Der Zähler besteht einfach aus JK-Flip-Flops, die in Reihe geschaltet sind, wobei der invertierte Ausgang eines Flip-Flops in den Clock-Eingang des nächsten führt. Die Eingänge J und K sind permanent auf HIGH geschaltet. Beim Ausführen des Programms nimmt der erste JK-Flip-Flop das Signal mit 2400 Hz und wechselt kurz auf HIGH und dann wieder auf LOW. Der Ausgang dieses Flip-Flops hat dann eine Frequenz von 1200 Hz, der nächste Flip-Flop 600 Hz, und so weiter. Die Eingänge R und S sind mit PC-set verbunden, sie dienen dazu, den Program Counter auf eine bestimmte Zahl zu setzen oder einen Reset zu machen.

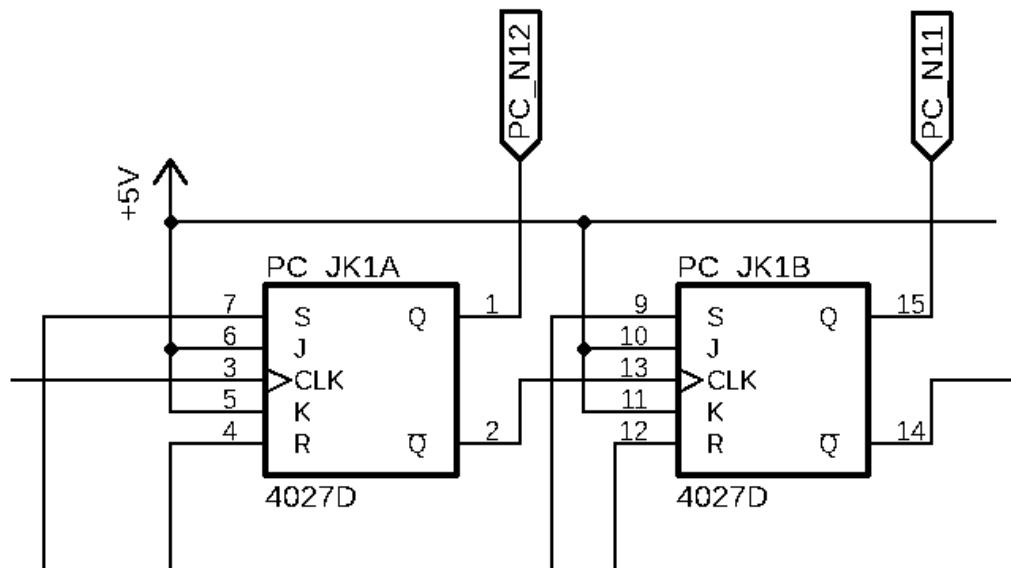


Abb. 35: PC Counter schematic Eagle

10.1.2.3 PC Set

Dieser Teil der Schaltung ermöglicht es den ProgramCounter auf eine bestimmte Adresse zu setzen dabei nehmen wir die daten aus der PC-Schleuse und geben jedes einzelne bit einmal invertiert und nicht invertiert weiter wenn dann das kurze signal vom Pulsgenerator werde die bits durchgelassen dabei ist der invertierte bit mit dem R des JK flipflops verbunden und das nicht invertierte mit dem S des JK flipflops.

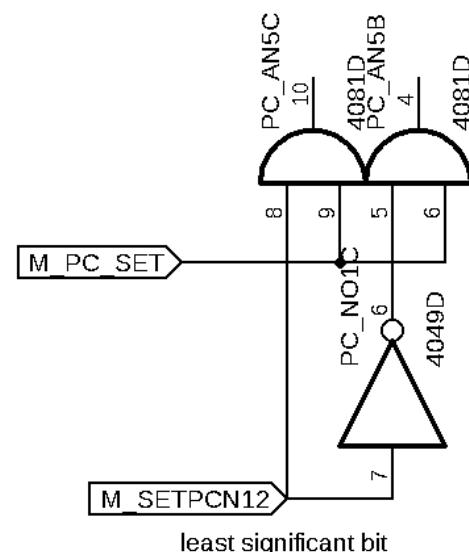
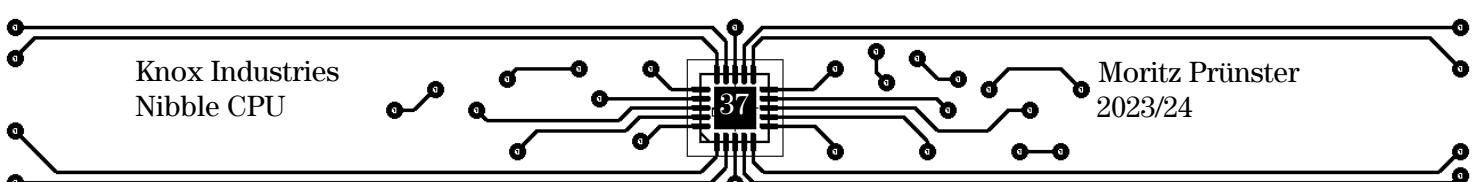


Abb. 36: PC Set schematic Eagle



10.1.3 Serielle (SE)

Die serielle liest den USB-Port und die gesendeten seriellen Daten aus. Die Daten werden dann kurzzeitig gespeichert, bevor sie im Hauptspeicher gespeichert werden. Dies ist auch der Grund weshalb die CPU mit 2400 Hz läuft und zwar wird die Frequenz mit Hilfe von JK-Flipflops geachtet. $2400 \text{ Hz} / 8 = 300 \text{ Hz}$ Diese 300 Hz werden zum Auslesen verwendet. Weshalb es auch wichtig ist eine Baudrate von 300 zu verwenden, da bei einer zu schnellen oder zu langsamen Baudrate die Daten nicht richtig gelesen werden können. Die Frequenz kann eine maximale Abweichung von ± 5 Hz haben.

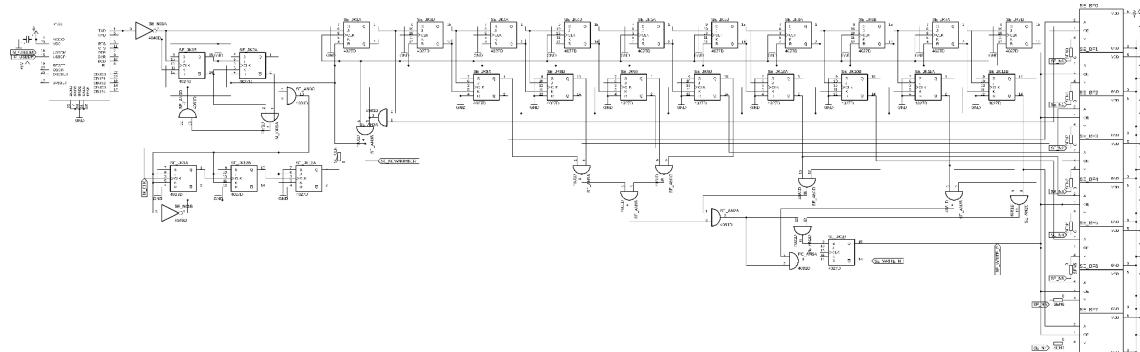


Abb. 37: Serielle Schematic Eagle

10.1.3.1 FT232

Dieser Teil der Schaltung liest die USB-Daten USB_Port aus und gibt serielle Daten weiter zum SE_Timer, dabei wird ein FT232 verwendet.

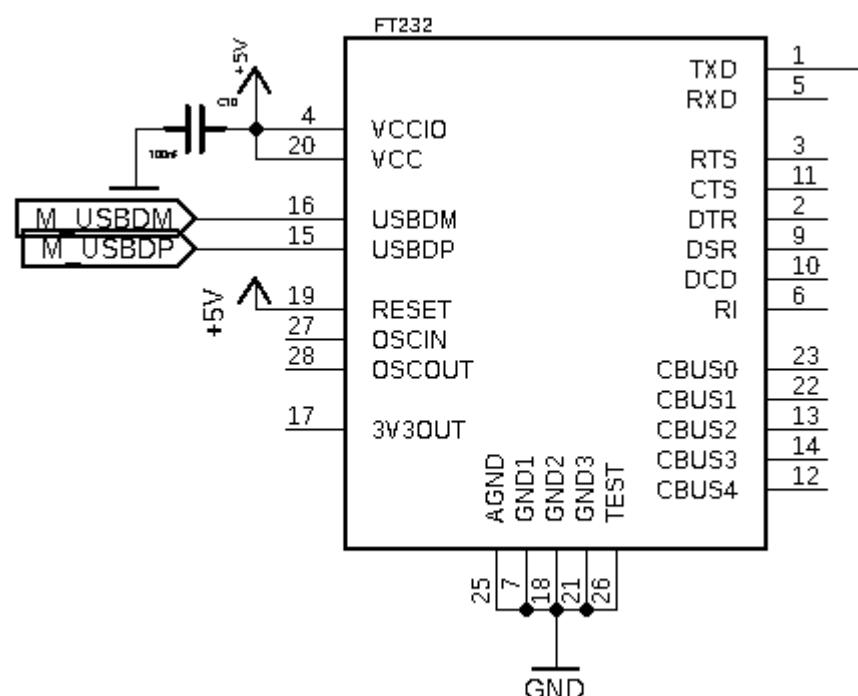
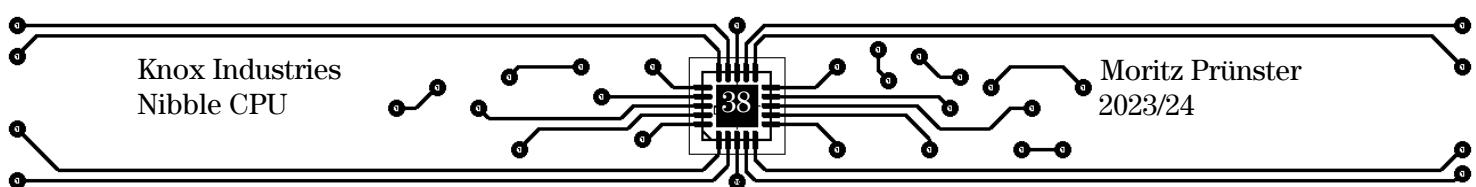


Abb. 38: FT232 Eagle



10.1.3.2 SE Timer

Dieser Teil sorgt dafür, dass beim Auslesen eine kleine Verzögerung stattfindet. Das serielle Signal wird zuerst invertiert und dann an die beiden JK-Flipflops weitergegeben. Wenn beide HIGH sind, bedeutet das, dass Daten ankommen, und das AND-Gatter gibt das Signal weiter an die serielle Clock. Zusätzlich wird die Clock durch das AND-Gatter blockiert, wodurch die Daten in beiden JK-Flipflops erhalten bleiben. Die R-Pins der JK-Flipflops sind mit einem Signal verbunden, das erst dann HIGH wird, wenn die gesamten 10 Bit (inklusive Start- und Stopp-Bit) ausgelesen werden.

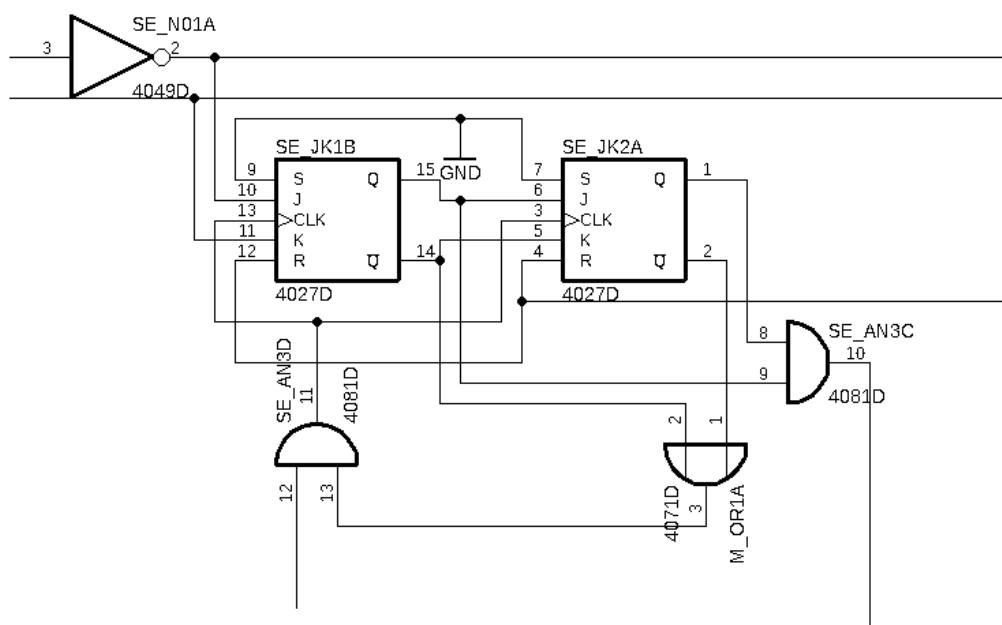
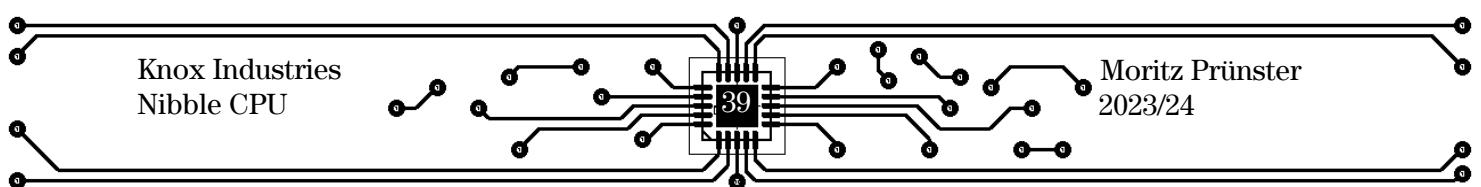


Abb. 39: SE Timer Schematic Eagle



10.1.3.3 SE Clock

Die SE_Clock kann mit einem Signal vom SE_Timer aktiviert und deaktiviert werden. Wenn das Signal HIGH ist, beginnt die Clock zu arbeiten und nimmt die 2400 Hz von der Hauptclock, um daraus 300 Hz zu machen, die dann weitergegeben werden. Wenn das Signal LOW ist, wird die gesamte Clock zurückgesetzt.

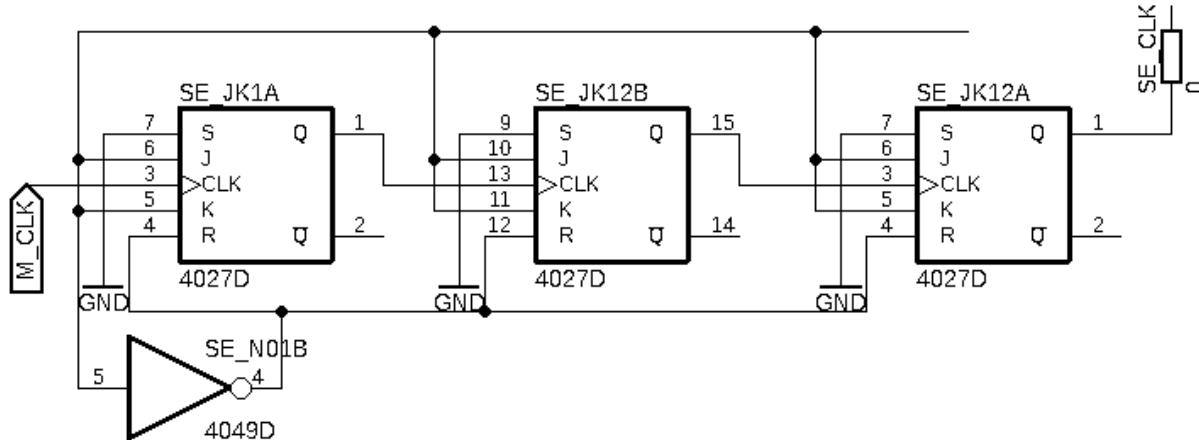


Abb. 40: SE Clock Schematic Eagle

10.1.3.4 Auslese Strecke

Die Auslese Strecke besteht aus 10 JK-Flipflops, die hintereinander geschaltet sind, sodass das invertierte serielle Signal nacheinander durch die einzelnen JK-Flipflops geht, ähnlich wie bei einem Schieberegister. Die dafür verwendete Clock sind 300 Hz von der SE_Clock. Wenn dann am Anfang eine 0 (Stopppbit) und am Ende eine 1 (Startbit) anliegt, wird die Schaltung durch SE_Reset zurückgesetzt.

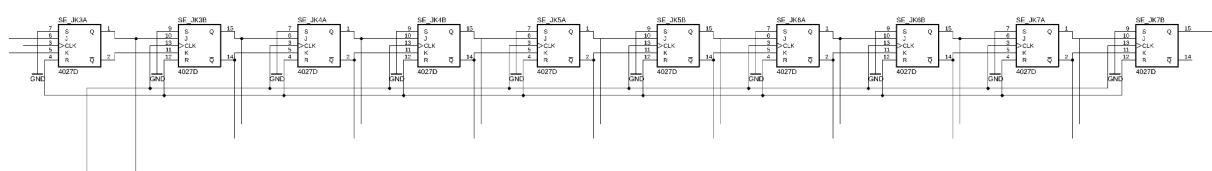
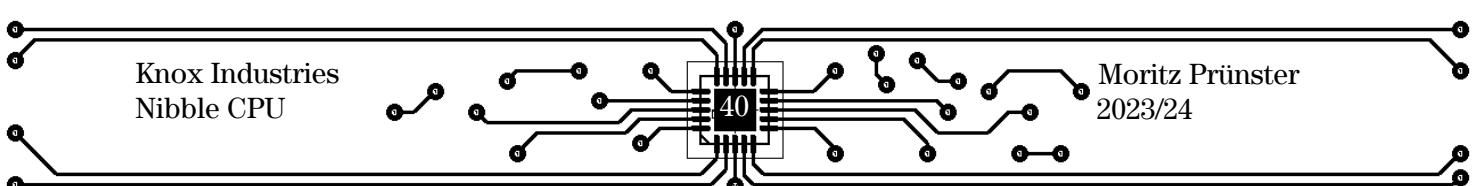


Abb. 41: Auslesestrecke Schematic Eagle.



10.1.3.5 SE Reset

Diese Schaltung überprüft, ob das letzte Bit der Auslesestrecke 1 und das erste Bit 0 ist. Wenn dies der Fall ist, wird das AND-Gatter HIGH. Das Signal geht dann weiter zum zweiten AND-Gatter, das als Schleuse für die 300 Hz von der SE_Clock dient. Wenn beide Signale HIGH sind, gibt es einen kurzen Impuls, der über die Leiterbahn namens "SE_NEWWNUMBER" verläuft. Dieser Impuls setzt den SE_Timer und die Auslesestrecke zurück und kann unter bestimmten Bedingungen auch den Programcounter erhöhen.

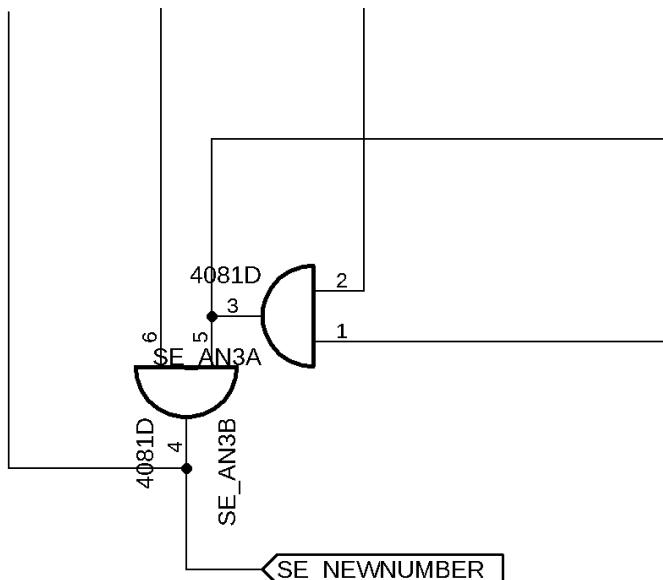


Abb. 42: SE Reset Schematic Eagle

10.1.3.6 SE Kurz Speicher

Diese Schaltung speichert die Daten aus der Auslesestrecke (außer die Start- und Stoppbits). Sie verwendet dazu das Signal aus dem ersten AND-Gatter des SE_Reset. Zusätzlich gibt sie die Daten invertiert aus, um sie wieder in den ursprünglichen Zustand zu bringen, da die seriellen Daten am Anfang invertiert ausgelesen wurden.

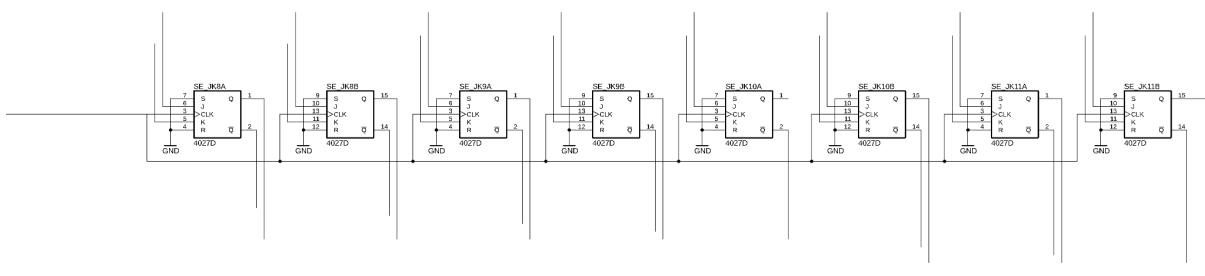
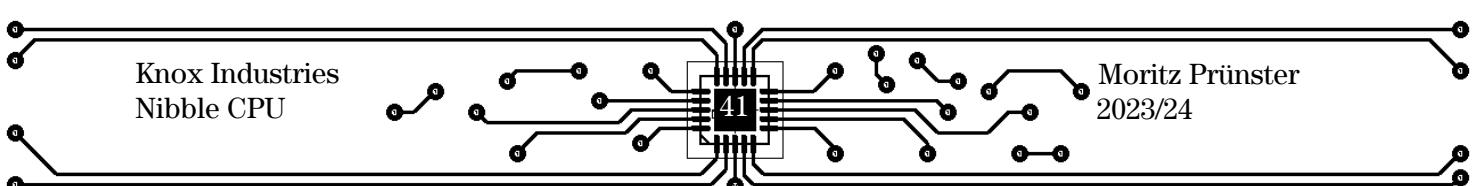


Abb. 43: Se Kurz Speicher Schematic Eagle



10.1.3.7 SE Ausgang

Dieser Teil gibt die Daten vom Kurzspeicher weiter zum Speicher, dabei werden Tri-State Buffer verwendet, wenn sie ein HIGH-Signal bei dem OE Pin haben, gehen die Daten weiter zum Speicher. Wenn es LOW ist, ist der Ausgang hochohmig, weshalb diese Schaltung als Schutz dient.

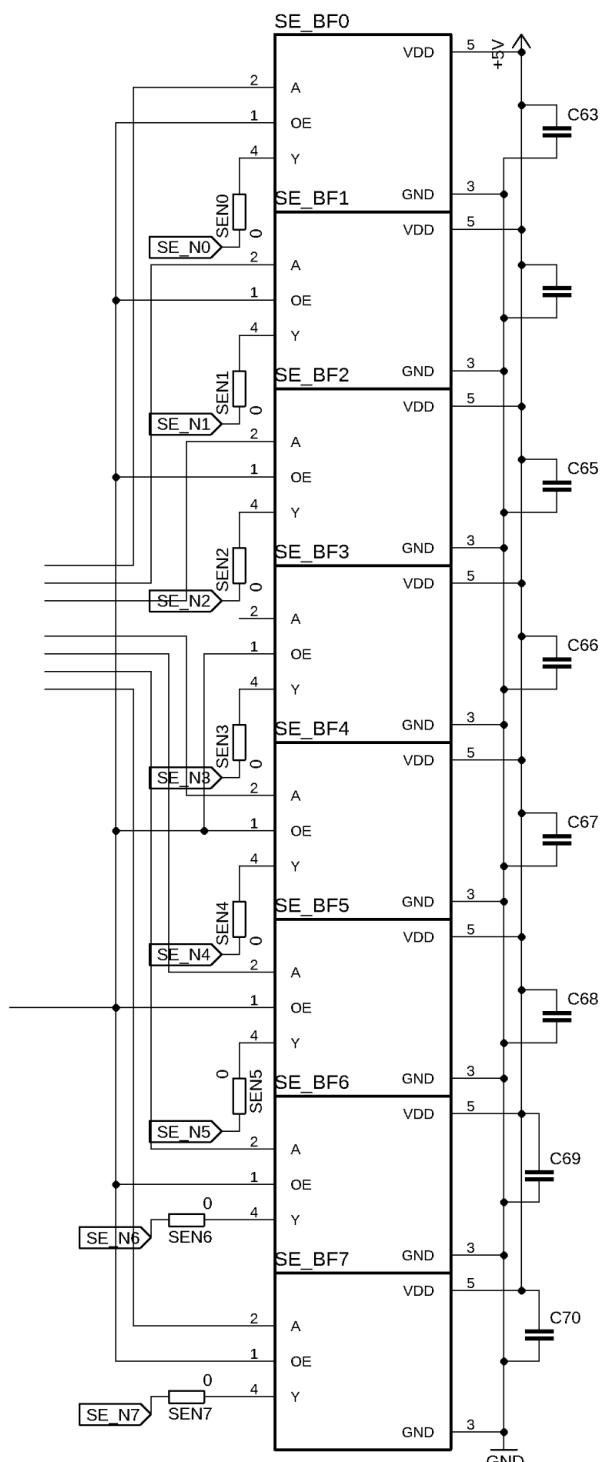
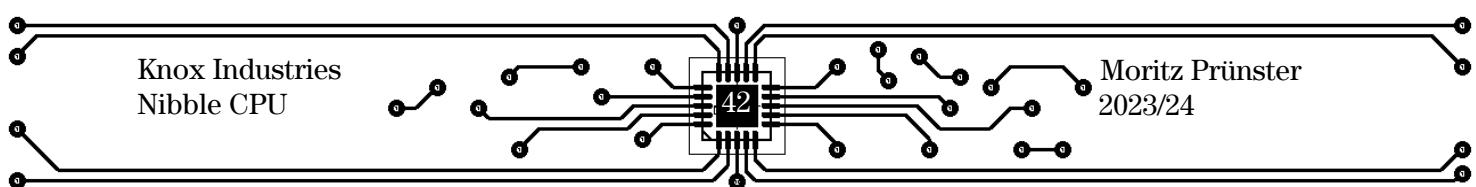


Abb. 44: SE Ausgang Schematic Eagle



10.1.3.8 SE Kontrolle

Damit die CPU weiß, ob sie die Daten speichern soll oder nicht, wird diese Schaltung verwendet. Sie erkennt, ob im Kurzspeicher die Daten 0000 1001 oder 0000 1010 abgespeichert sind. Wenn 0000 1001 gespeichert ist, wird das JK-Flipflop auf HIGH gesetzt. Wenn 0000 1010 gespeichert ist, wird es auf LOW gesetzt. Ist das JK-Flipflop HIGH, bedeutet dies, dass alle nachfolgenden Daten gespeichert werden sollen, da ein Programm hochgeladen wird. Die Daten werden dann weitergegeben auf den Leiterbahnen "SE_WRITE_P" und "SE_WRITE_N".

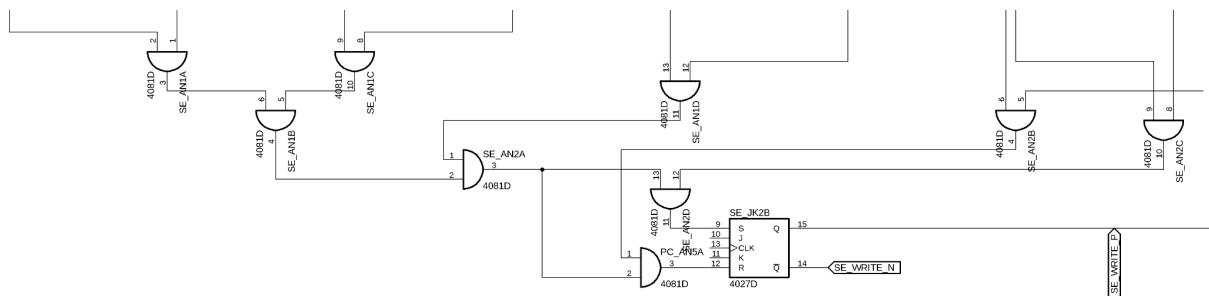
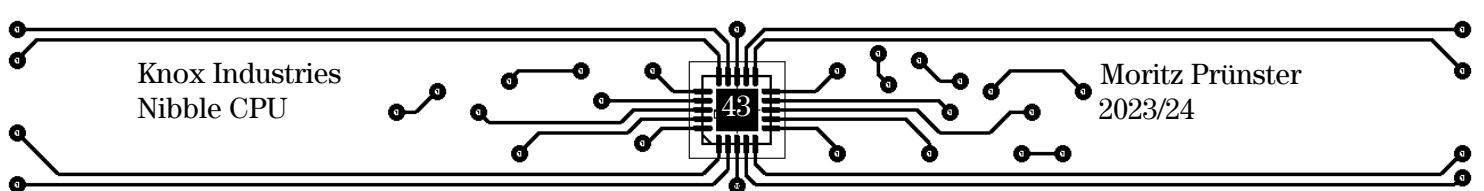


Abb. 45: SE Kontrolle Schematic Eagle



10.1.4 Branch (BRA)

Dieser Teil der Schaltung überprüft, ob es sich bei dem Befehl um einen Branch-Befehl handelt. Ist dies der Fall, werden die zwei darauf folgenden Daten nicht an die ALU weitergegeben, da diese zwei Bytes die Adresse enthalten, zu der der Programmzähler beim Ausführen des Branch-Befehls springen muss. Zusätzlich gibt die Schaltung ein Signal an den Pulsgenerator, um den Programmzähler zurückzusetzen.

Neben Branch-Befehlen werden hier auch Lese-Befehle ausgeführt. Die Schaltung erhält die 8-Bit-Daten aus dem Speicher und teilt sie in der Mitte in zwei Stränge auf: die oberen 4 Bits sind die Befehle und die unteren 4 Bits sind die Nummern oder Adressen, die an die ALU weitergeleitet werden. Zusätzlich erhält die Schaltung 2-Bit-Daten von der ALU: ein Bit ist das SameFlag und das andere Bit ist das CarryFlag. Diese Flags ermöglichen das Ausführen von bedingten Befehlen wie BEQ (Branch if Equal) und BCY (Branch if Carry).

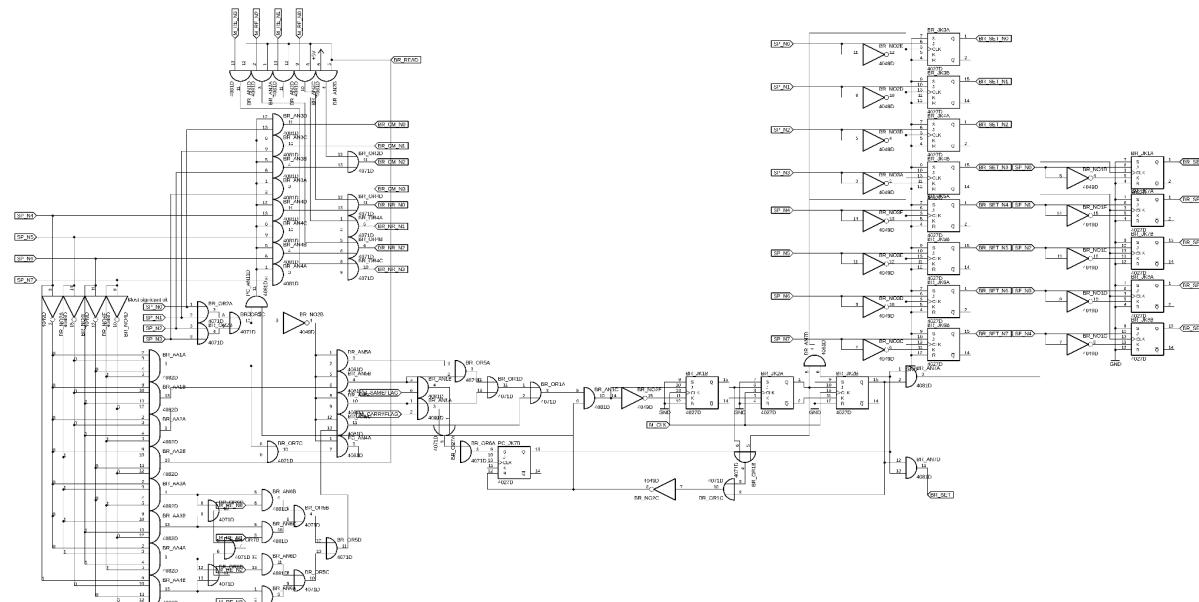
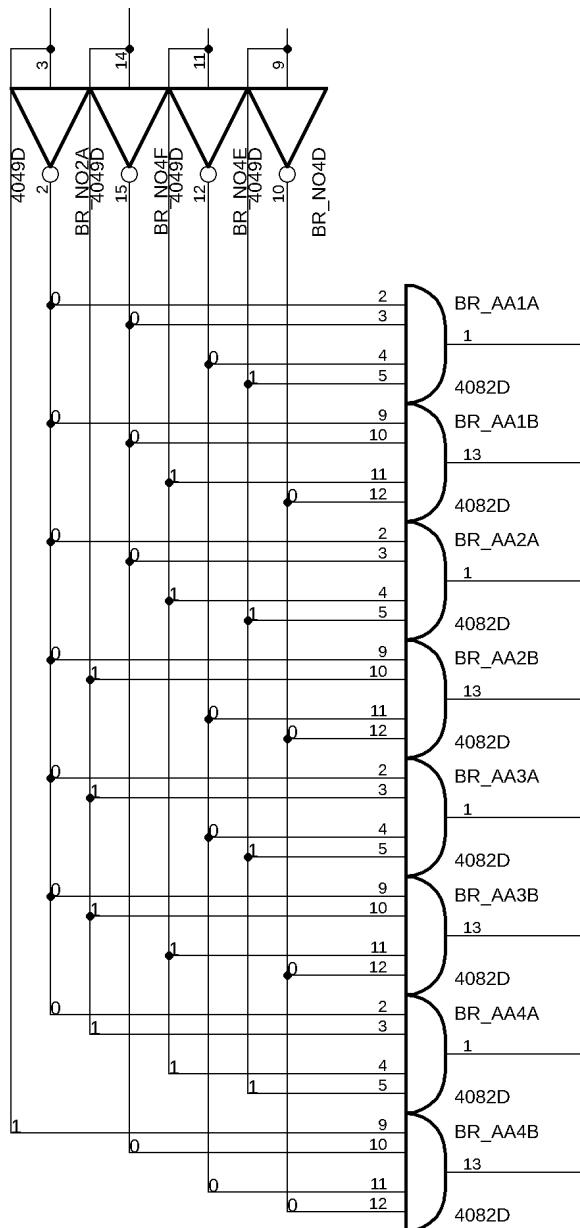


Abb. 46: Branch Schematic Eagle

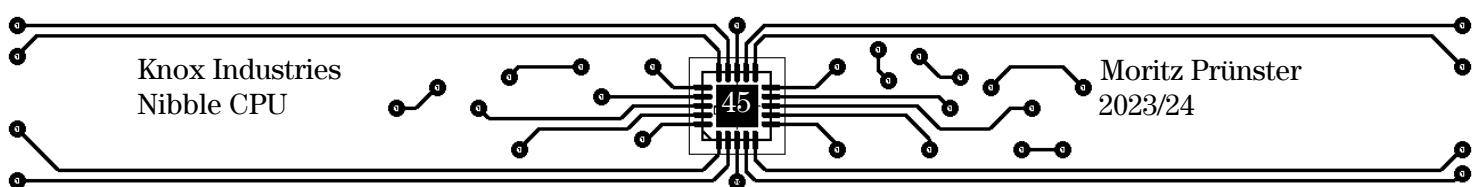
10.1.4.1 BRA Befehlserkennung

Diese Schaltung dient dazu, den jeweiligen Befehl zu erkennen. Sie ähnelt einem Multiplexer und kann je nach Kombination der letzten 4 Bits einen der 8 AND-Gatter-Ausgänge auf HIGH setzen.



Befehl	Kombination
bra	0001
beq	0010
bcy	0011
rea	0100
re0	0101
re1	0110
re2	0111
re3	1000

Abb. 47: Bra Befehlserkennung Schematic Eagle



10.1.4.2 Nullerkennung

Diese Schaltung überprüft, ob die ersten 4 Bits 0 sind oder nicht. Sind sie nicht 0, wird das Signal einmal an die Datenschleuse gesendet und invertiert an die Brancheschleuse weitergeleitet. Das Signal zur Datenschleuse kann blockiert werden, wenn nicht alle JK-Flipflops beim Branch Counter auf LOW sind.

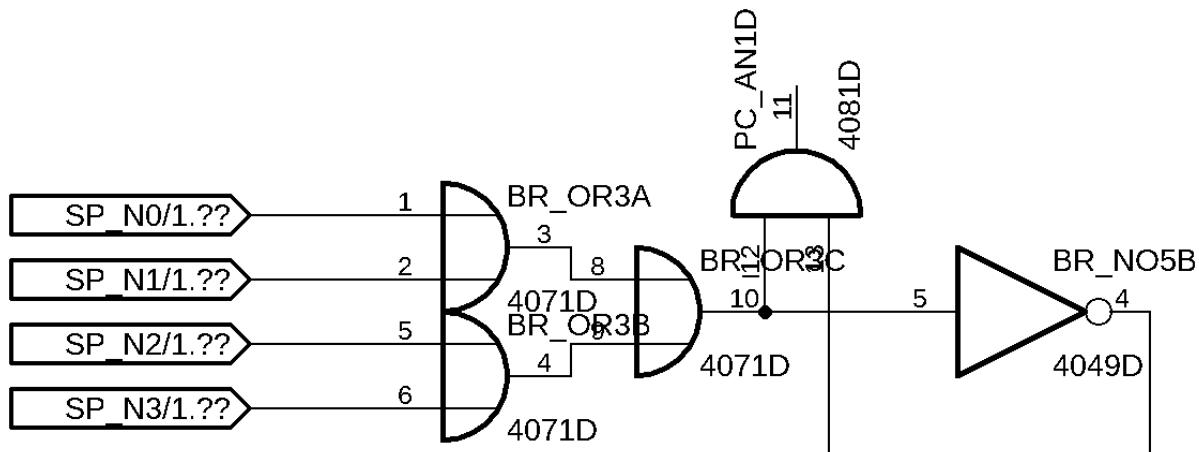


Abb. 48: Nullerkennung Schematic Eagle

10.1.4.3 Datenschleuse

Diese Schaltung dient als Schleuse, damit keine Befehle zur ALU gelangen, wenn sie deaktiviert ist.

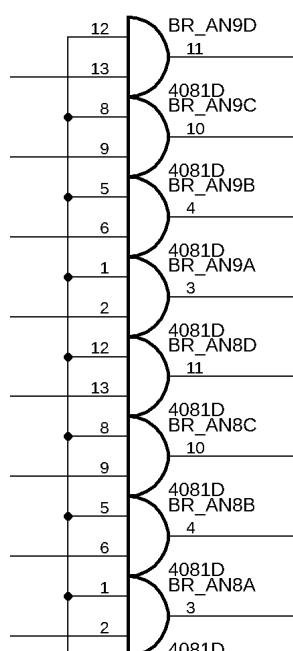
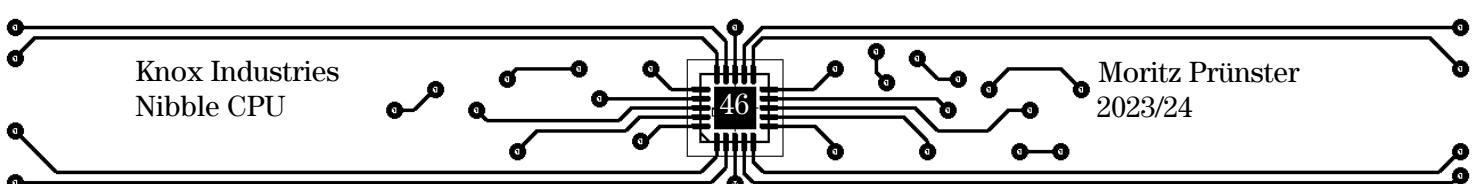


Abb. 49: Datenschleuse Schematic Eagle



10.1.4.4 BRA Schleuse

Diese Schaltung dient dazu, die Daten der Befehlserkennung nur dann durchzulassen, wenn die vorherigen 4 Bits ebenfalls null sind. Das bedeutet, dass 0000 0001 erkannt wird, während 0100 0001 nicht erkannt wird. So sind nun auch die Befehle vollständig. Zusätzlich kann die Schaltung auch erkennen, ob bei den 4 Read-Befehlen ein Pin auf HIGH ist. Dazu dienen die OR-Gatter und AND-Gatter in der unteren Hälfte der Schaltung.

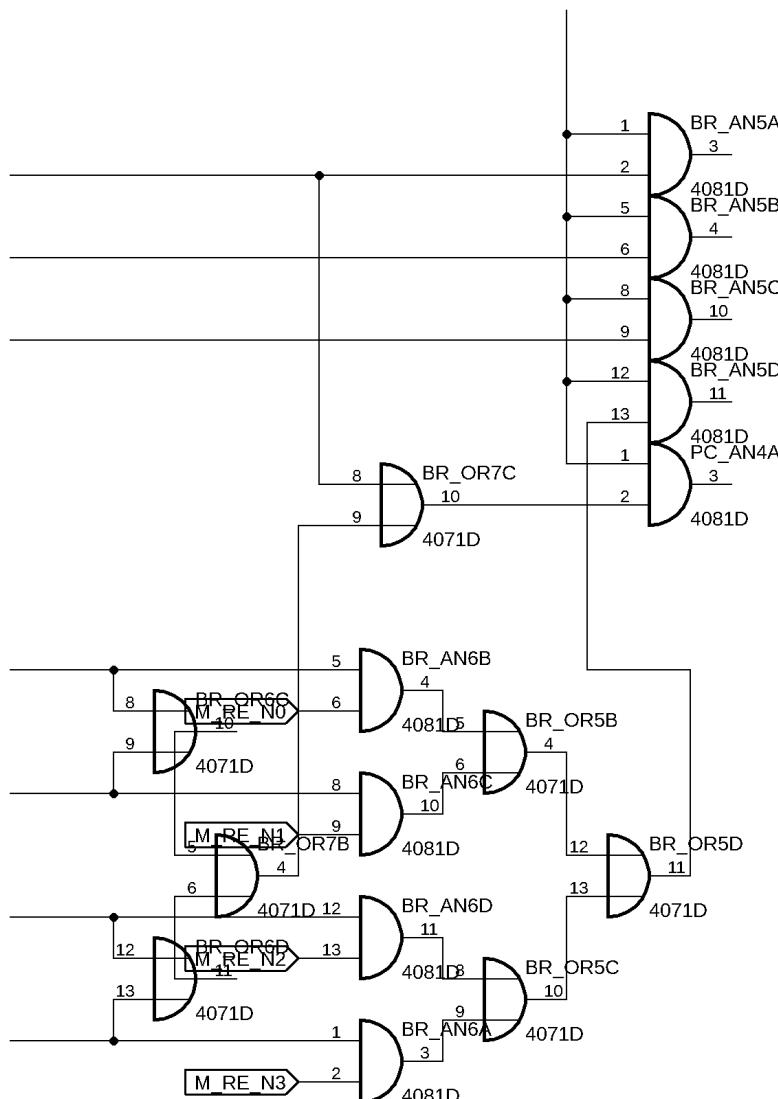
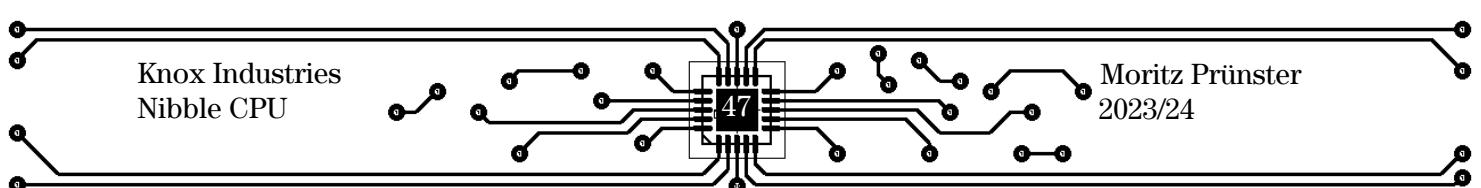


Abb. 50: Bra Schleuse Schematic Eagle



10.1.4.5 Zusammenfasser

Diese Schaltung fasst die Daten aus der Datenschleuse zu einem Signal zusammen, das dann weitergeleitet wird. Zusätzlich überprüft sie bei den Befehlen BEQ (Branch if Equal) und BCY (Branch if Carry), ob die jeweiligen Flags HIGH sind. Wenn dies der Fall ist, werden diese Signale ebenfalls zusammengefasst. Außerdem wird, wenn ein Signal HIGH ist, das JK-Flipflop auf HIGH gesetzt und kann nur vom Branch Counter zurückgesetzt werden. Wenn der Ausgang des JK-Flipflops HIGH ist, bedeutet dies, dass ein Branch-Befehl aktiv ist und der Programmzähler (PC) auf die gesendete Adresse gesetzt werden kann, da ein HIGH-Signal vorliegt. Mit diesem Prinzip wird der PC nicht gesetzt, wenn ein BEQ-Befehl ausgeführt wird, aber das SameFlag nicht HIGH ist, wodurch der Befehl ungültig ist.

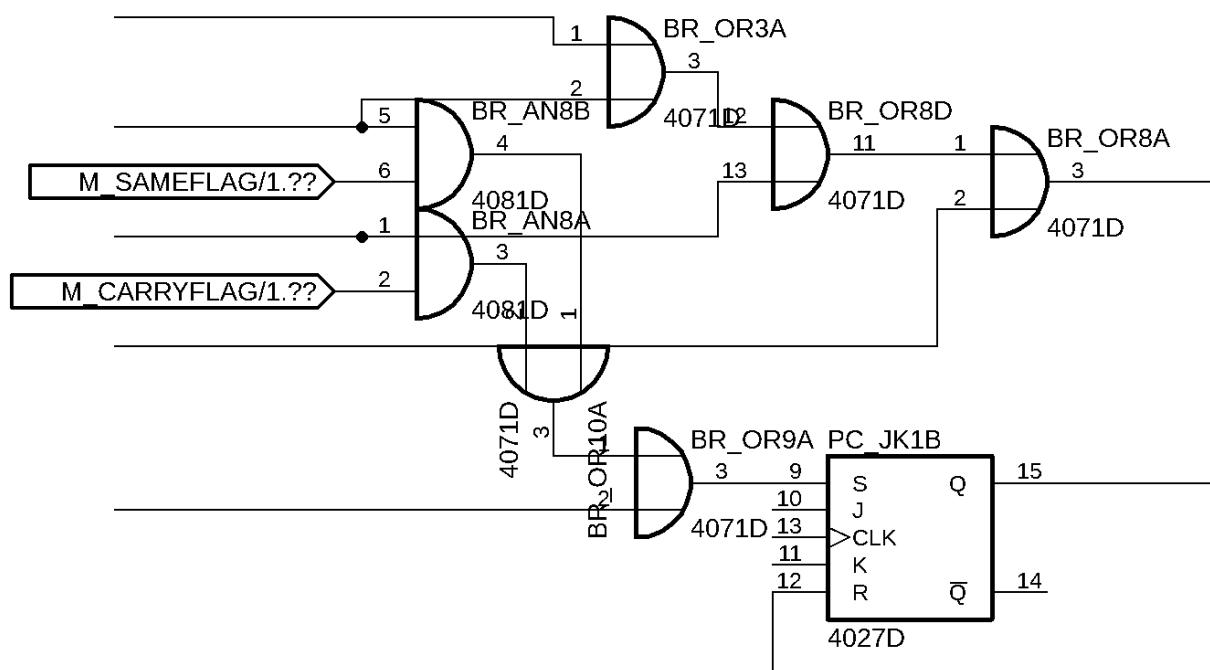
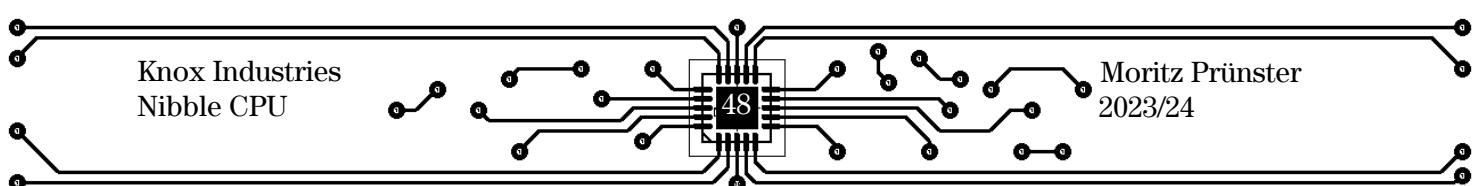


Abb. 51: Zusammenfasser Schematic Eagle



10.1.4.6 BRA Counter

Diese Schaltung gewährleistet eine Verzögerung von zwei Taktzyklen. Ihr Hauptzweck besteht darin sicherzustellen, dass die beiden Bytes nach dem Branch-Befehl nicht an die ALU gesendet werden, da es sich dabei um PC-Adressen handelt. Gleichzeitig werden die Daten im Branch-Kurzspeicher abgelegt. Die Funktionsweise der Schaltung ist wie folgt: Sie empfängt ein Signal vom Zusammenfasser, das das erste JK-Flipflop auf HIGH setzt. Anschließend wird das erste Datenbyte übertragen, wobei auch das zweite JK-Flipflop auf HIGH gesetzt wird. Der Ausgang dieses Flipflops wird an ein AND-Gatter weitergeleitet, das die Daten passieren lässt, wenn das JK-Flipflop des Zusammenfassers auf HIGH ist. Wenn dies der Fall ist, wird das Signal zum Branch-Kurzspeicher gesendet. Der gleiche Vorgang wiederholt sich beim letzten JK-Flipflop, wobei zusätzlich ein Signal an den Pulsgenerator gesendet wird, um den PC zu setzen. Auf diese Weise werden die beiden Bytes nach dem Branch-Befehl korrekt als PC-Adressen behandelt und gespeichert, ohne an die ALU weitergeleitet zu werden. Zusätzlich kann das erste JK-Flipflop nur gesetzt werden, wenn alle JK-Flipflops auf LOW sind, was mit Hilfe von OR- und NOT-Gattern realisiert wird. Das Signal wird auch an die Nullerkennung gesendet, um zu verhindern, dass die Daten zur ALU weitergeleitet werden.

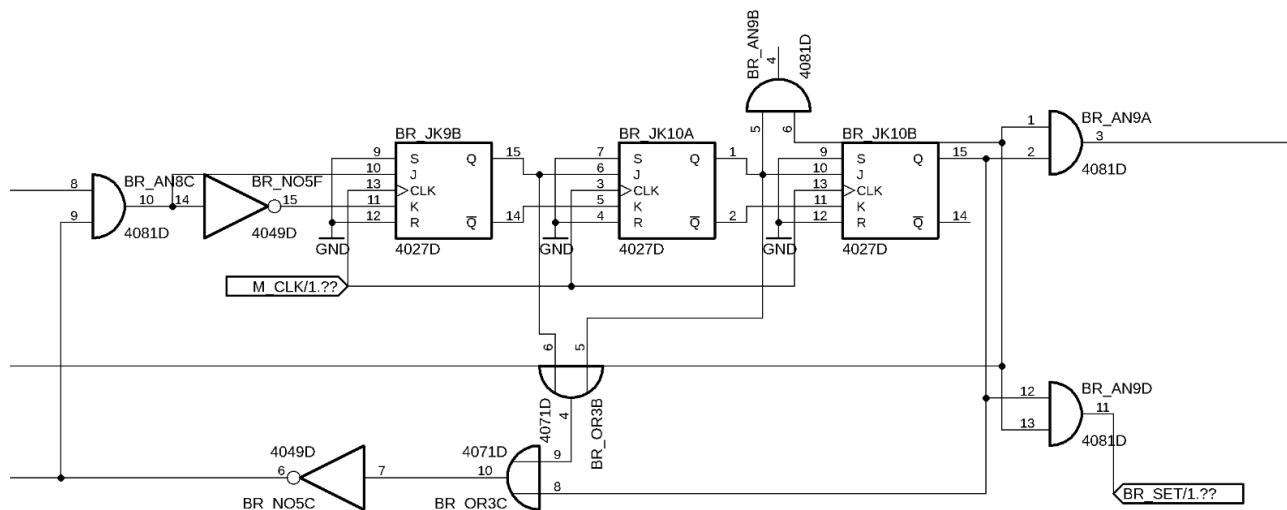


Abb. 52: Bra Counter Schematic Eagle

10.1.4.7 BRA Kurz Speicher

Die Schaltung besteht aus zwei Teilen: einem großen Speicher, der 8 Bit speichern kann, und einem kleinen Speicher, der nur 5 Bit speichern kann.

Sie dient dazu, die Branch-Adresse für einen kurzen Moment zu speichern, bevor sie vom PC übernommen wird. Dabei wird das erste Byte vollständig gespeichert, während vom zweiten Byte nur die ersten fünf Bits gespeichert werden. Die letzten drei Bits des zweiten Bytes werden nicht gespeichert, da der PC nur 13-Bit-Adressen benötigt.

Um eine genaue Abspeicherungszeit zu gewährleisten und sicherzustellen, dass beide Speicher nicht dieselben Daten erhalten, wird ein Signal vom Branch-Counter verwendet. Beim zweiten JK-Flipflop wird dieses Signal im großen Speicher abgespeichert, während es beim dritten JK-Flipflop im kleinen Speicher abgelegt wird.

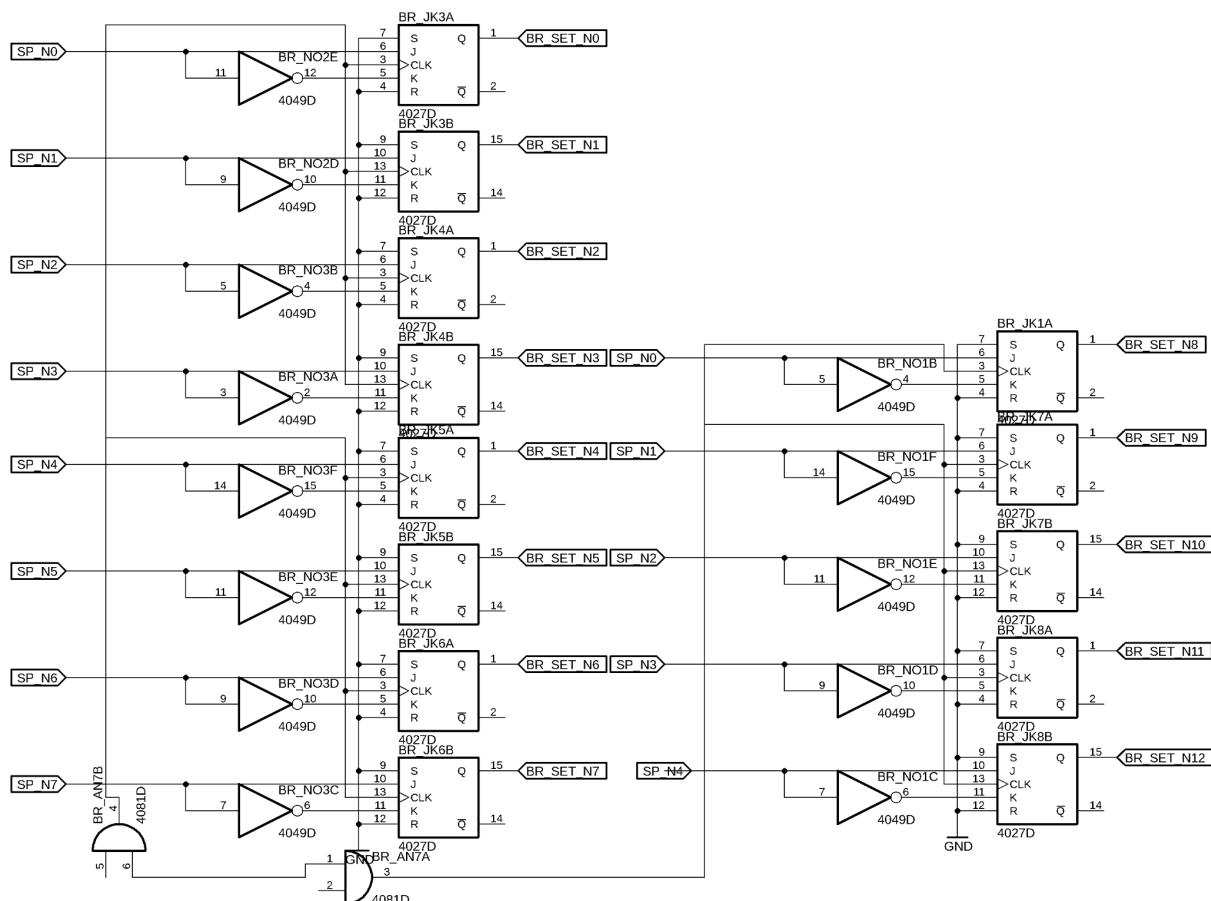


Abb. 53: Bra Kurzspeicher Schematic Eagle

10.1.5 Speicher

Diese Schaltung speichert die Daten aus der seriellen Schnittstelle ab und ist direkt mit dem Programmcounter verbunden. Dies ist auch der Grund, warum es den Programmcounter gibt, da er die Adresse des Speichers hinaufzählt. Wenn ein Programm hochgeladen wird, wird es in diesem Speicher gespeichert und während des Programmablaufs nacheinander geladen und ausgelesen.

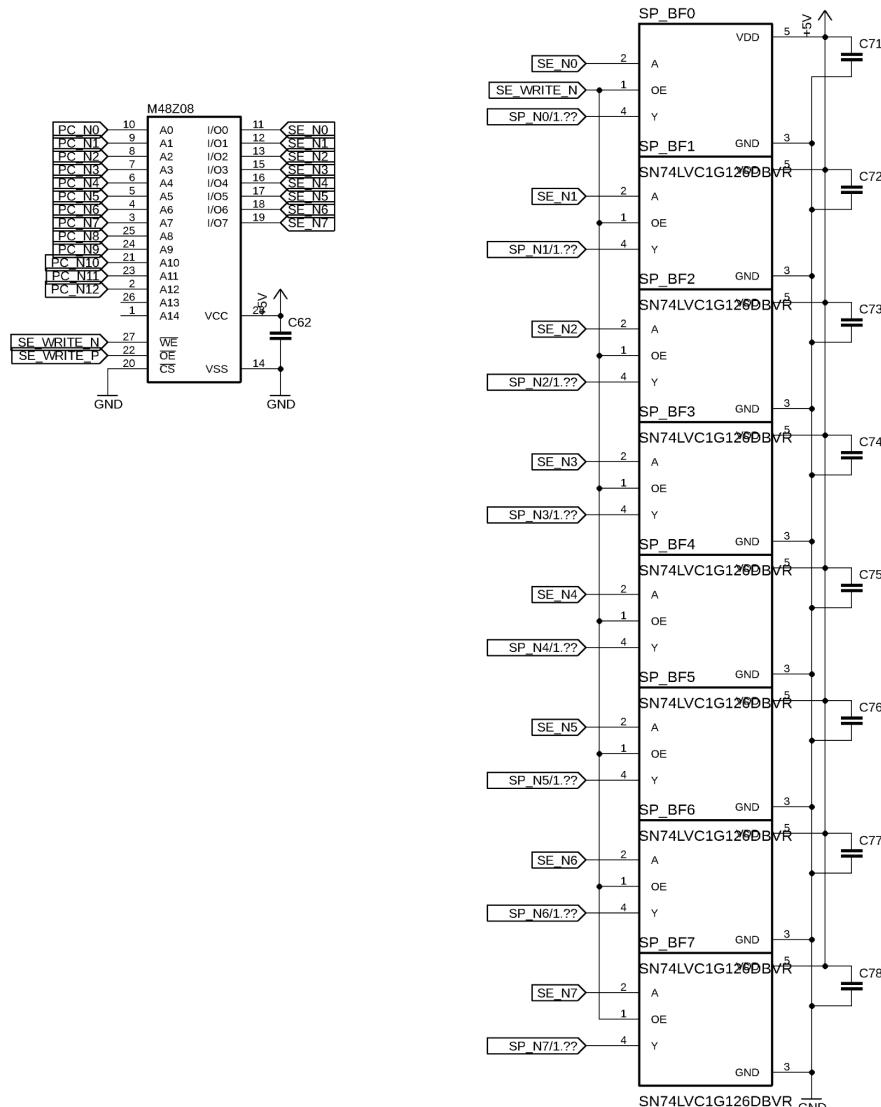
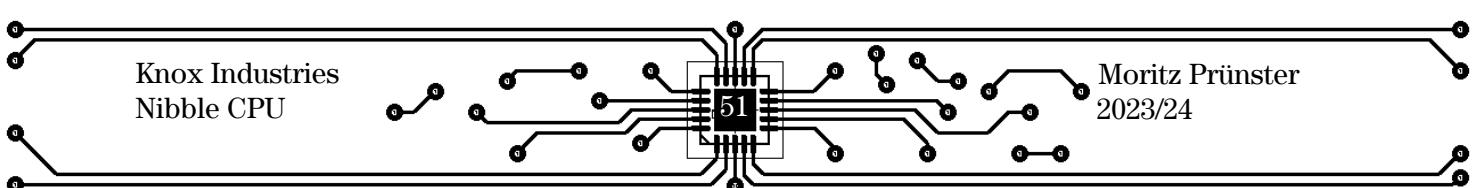


Abb. 54: Speicher Schematic Eagle



10.1.5.1 M48Z08

M48Z08 speichert Daten, wenn "SE_WRITE_P" HIGH ist, und gibt sie aus, wenn sie LOW sind. Dabei verwendet es die Pins 11 bis 19 zum Lesen und Schreiben der Daten, weshalb auch Tri-State-Buffer verwendet wurden. Diese ermöglichen das Lesen und Schreiben der Daten in beide Richtungen. Die linken Pins sind die Adressen, die mit dem PC verbunden sind und sich ständig ändern.

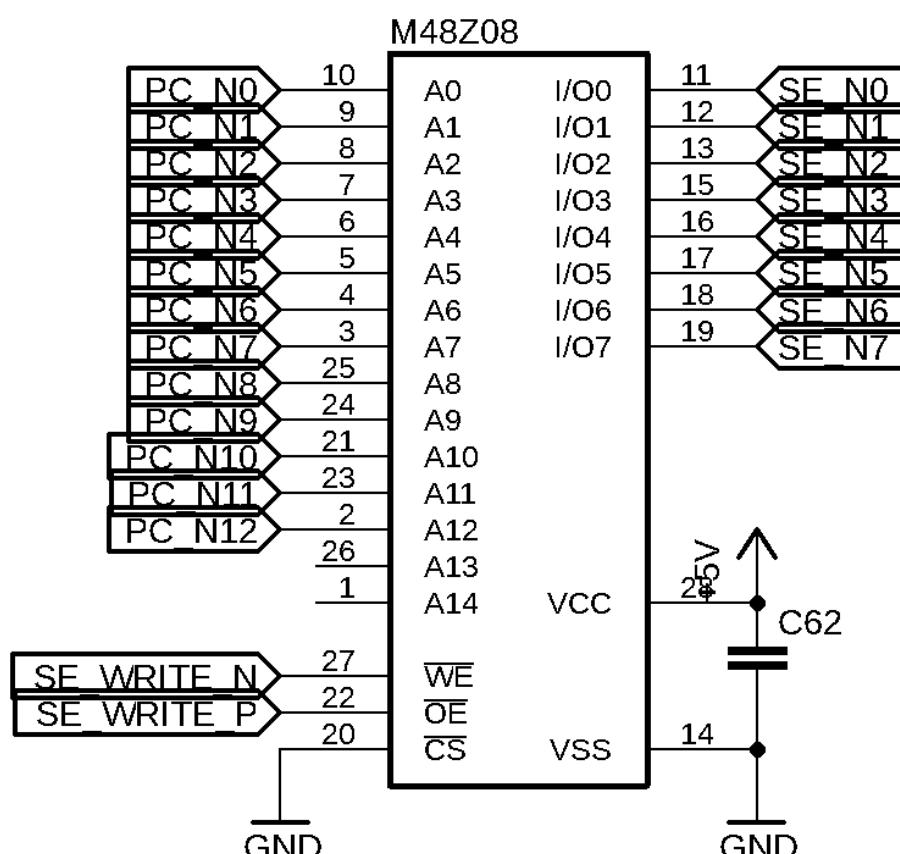
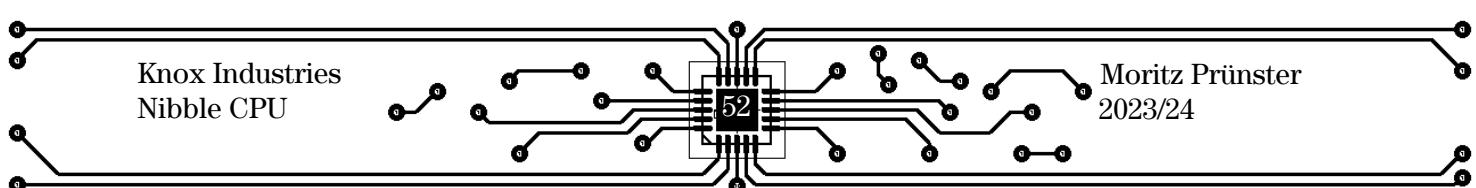


Abb. 55: M48Z08 Schematic Eagle



10.1.5.2 Speicher Leser

Tri-State-Buffer werden verwendet, um sicherzustellen, dass der Speicher ordnungsgemäß ausgelesen wird. Dadurch wird verhindert, dass die Daten während des Schreibprozesses gelesen werden und die CPU das Programm bereits ausführt, während es noch geladen wird.

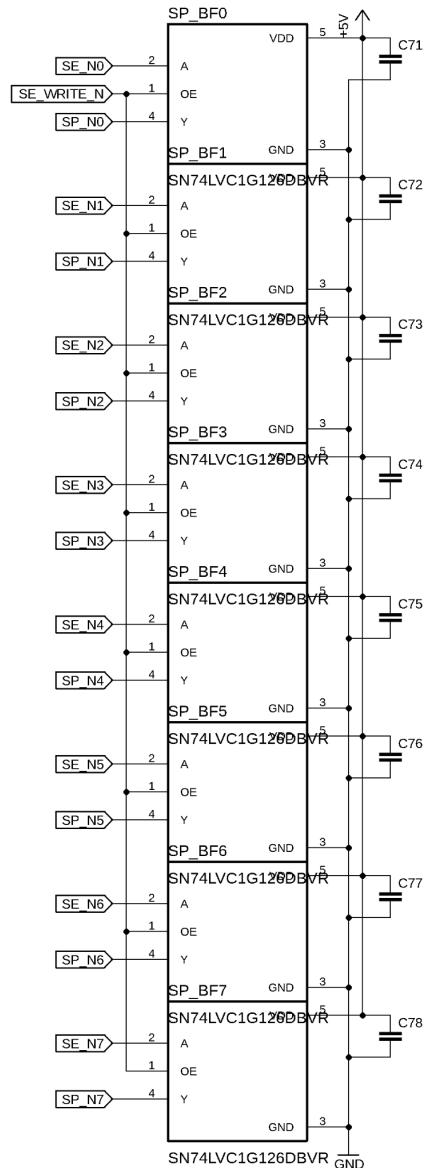
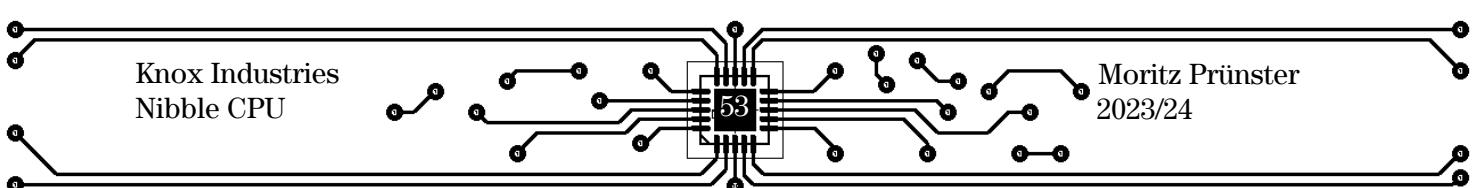


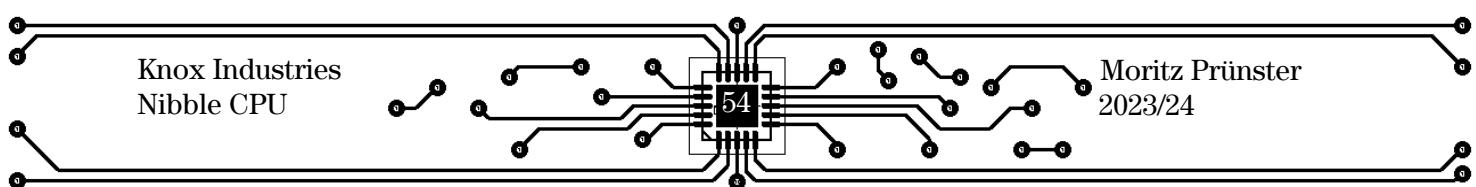
Abb. 56: Speicher leser Schematic Eagle



10.2 Arithmetic Logic Unit(ALU)

Die ALU (Arithmetic Logic Unit) ist der zweite Teil der CPU und ist für die Durchführung von Berechnungen wie Addition, Subtraktion, Multiplikation und Vergleiche zuständig. Im Wesentlichen schreibt man Programme, die direkt für die ALU ausgeführt werden. Die ALU kann auch auf das Register zugreifen, um Daten zu lesen und zu speichern. Die Befehle die mit der ALU ausgeführt werden sind:

Befehl	Bedeutung
nop	nope
lda	load address
ldn	load number
sta	store at address
adn	add number
ada	add address
sun	subtract number
sua	subtract address
ma1	multiplicate with address part 1
ma2	multiplicate with address part 2
mn1	multiplicate with number part 1
mn2	multiplicate with number part 2
ccf	clear carry flag
ash	akkumulator shift
its	if it's the same as number
csf	clear same flag



Um einen kleinen Einblick in die Funktionsweise der ALU zu geben: Die ALU erhält zunächst 8 Bits von der Steuereinheit, wobei 4 Bits Befehle und die anderen 4 Bits Nummern oder Adressen darstellen. Die 4 Bits für Befehle werden zur Befehlerkennung gesendet, wo sie in 16 Bits umgewandelt werden, sodass jedes Bit einem bestimmten Befehl entspricht. Diese Daten gelangen weiter zur Einheit für Adressen oder Nummern, die entscheidet, ob die zweiten 4 Bits von der Steuereinheit als Nummer oder als Adresse interpretiert werden, abhängig vom Befehlstyp.

Handelt es sich um eine Adresse, wird diese zum Register gesendet und ausgelesen. Die dort gespeicherte Nummer wird zurückgegeben und an das Pinboard gesendet. Handelt es sich um eine Nummer, wird diese ohne Änderungen an das Pinboard gesendet. Am Pinboard werden die Daten sowohl an den Akkumulator 1 (Akku 1) als auch an den Rechner gesendet.

Wenn der Befehl LDA (Load Address) oder LDN (Load Number) lautet, werden die Daten im Akku 1 gespeichert. Handelt es sich um Berechnungsbefehle wie ADN (Add Number), SUA (Subtract Address) oder MA1 (Multiply Address part 1), wird das Ergebnis der Berechnung zwischen dem Inhalt des Akku 1 und der Adresse oder Nummer, je nach verwendetem Berechnungsbefehl, weiter an den Akku 2 gesendet und dort gespeichert. Möchte man das Ergebnis der Berechnung verwenden, muss man ASH (Akkumulator Shift) verwenden, wodurch die Daten von Akku 2 in Akku 1 geladen werden.

Zum Speichern in das Register verwendet man den Befehl STA (Store at Address), wobei auch hier die 4 Bits von der Steuereinheit als Adresse interpretiert werden. Zuletzt noch kurz zu den Flags: Das Carry-Flag und das Same-Flag können vom Rechner auf HIGH gesetzt werden. Um sie zurückzusetzen, verwendet man die Befehle CCY (Clear Carry) oder CSF (Clear Same Flag).

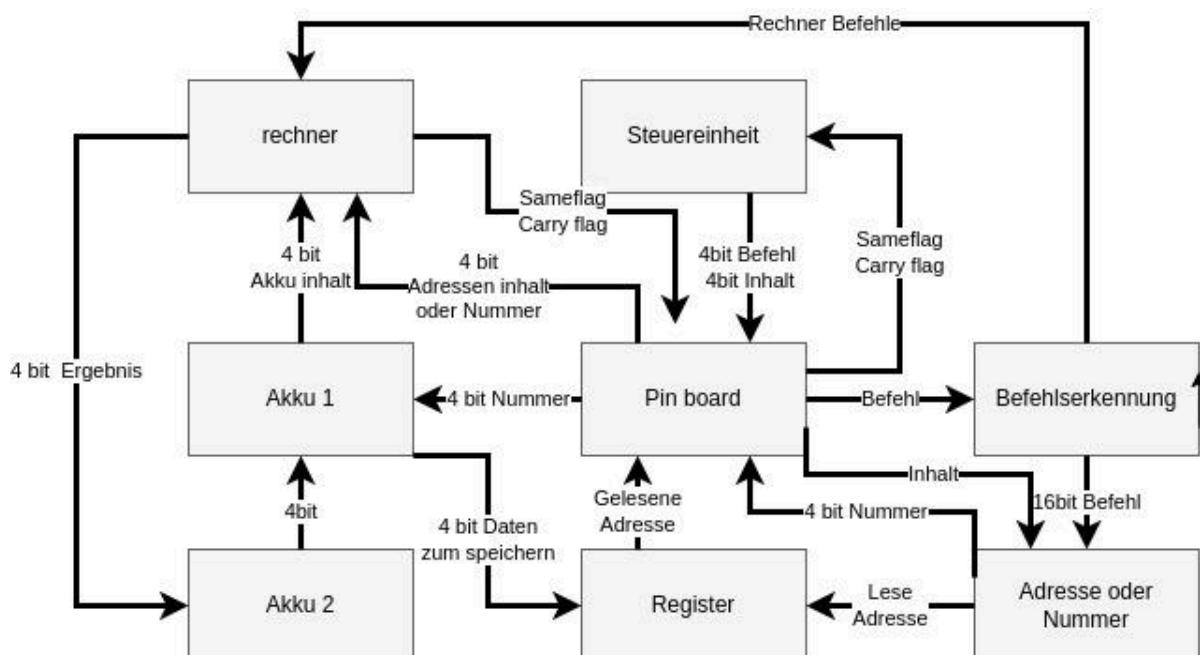
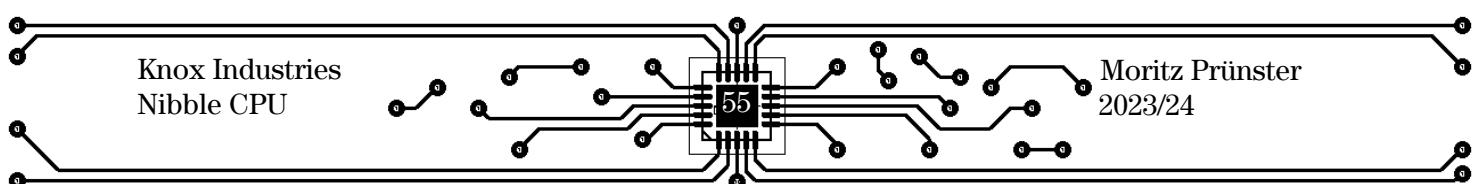


Abb. 57: ALU Daten verteilungsdiagramm



10.2.1 ALU Pinboard

Diese Schaltung enthält alle Pins, die in der ALU verwendet werden. Er dient dazu, die Daten an den richtigen Stellen durchzulassen und eine Verbindung zwischen den verschiedenen Schaltungen in der ALU herzustellen.

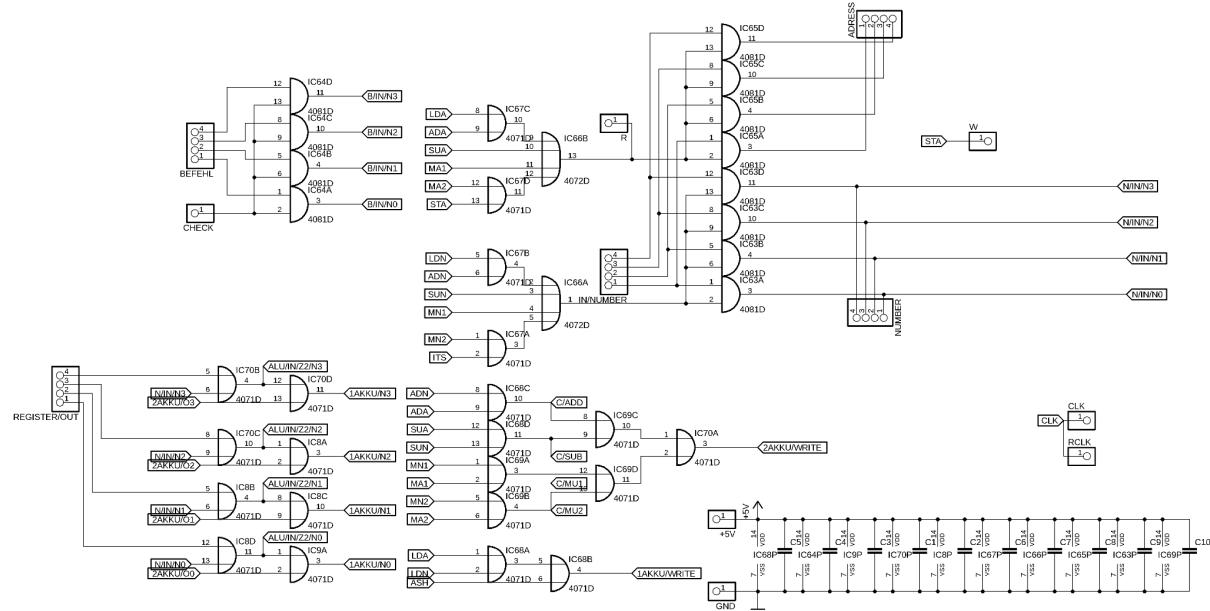
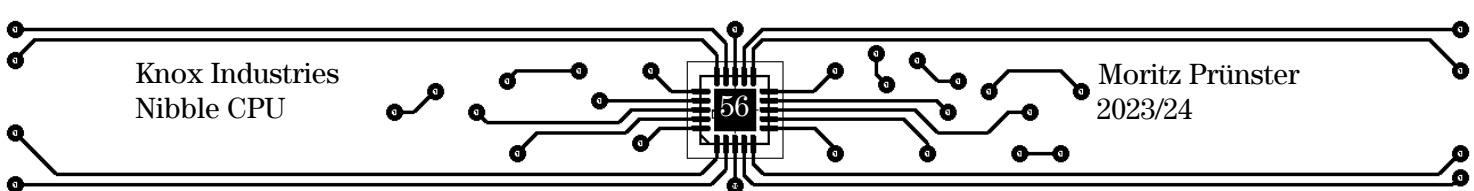


Abb. 58: ALU Pinboard Schematic Eagle



10.2.1.1 Nummer/Adresse erkennung

Dieser Teil der Schaltung überprüft, um welchen Befehlstyp es sich handelt. Falls es ein Befehl ist, der eine Adresse verlangt, wie LDA (Load Address) oder ADA (Add Address), werden die letzten 4 Bits als Adresse interpretiert. Handelt es sich hingegen um Befehle wie ADN (Add Number) oder LDN (Load Number), dann werden die 4 Bits als Nummer interpretiert und entsprechend weitergegeben.

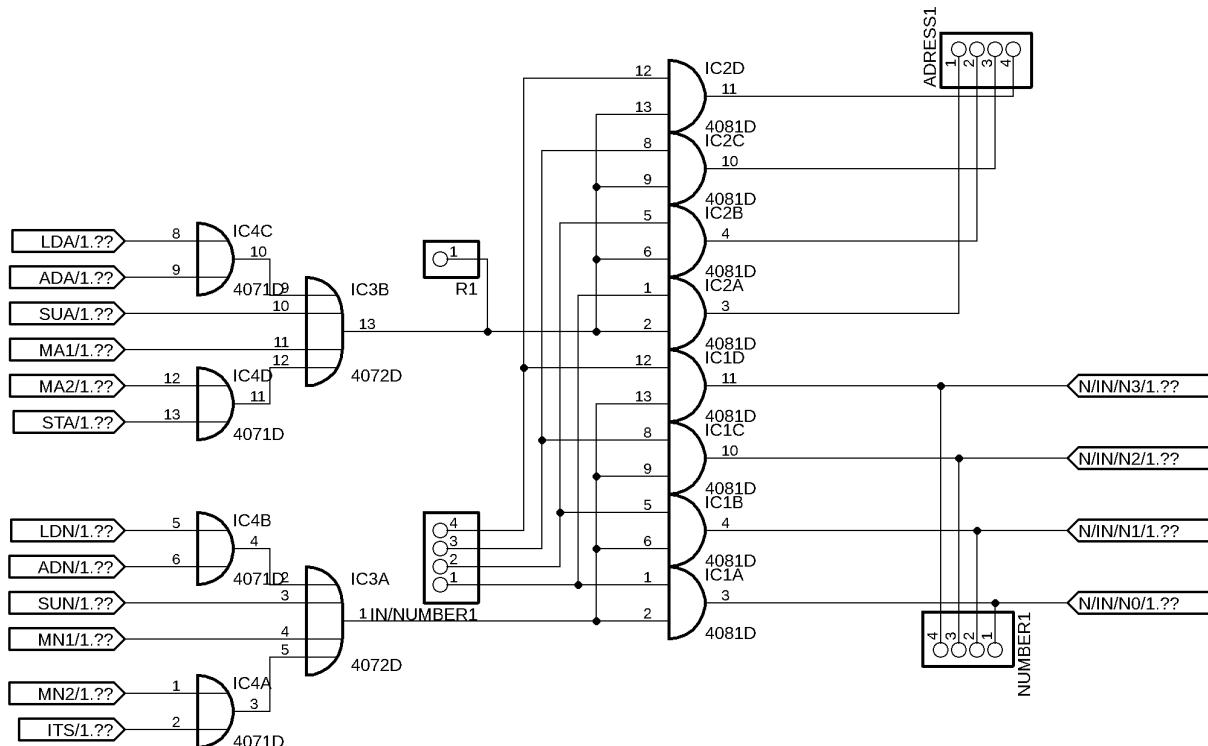


Abb. 59: Nummer/Adressen erkennung Schematic Eagle

10.2.1.2 Ergebnis abspeicherung

Diese Schaltung erkennt, ob es sich bei dem Befehl um einen Berechnungsbefehl handelt. Wenn dies der Fall ist, wird ein Signal an den zweiten Akkumulator gesendet. Zusätzlich fasst sie die unterschiedlichen Befehle wie ADN (Add Number) und ADA (Add Address) zusammen und leitet ein Signal an den C/Add-Ausgang weiter.

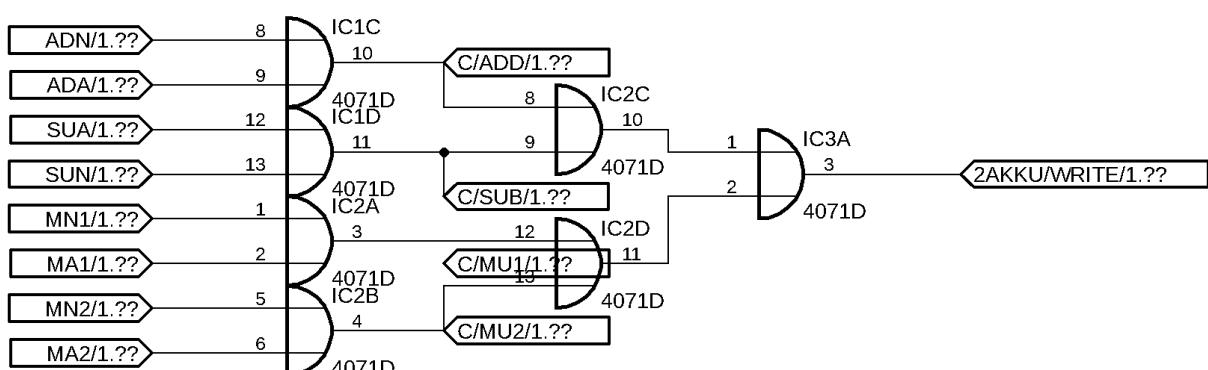
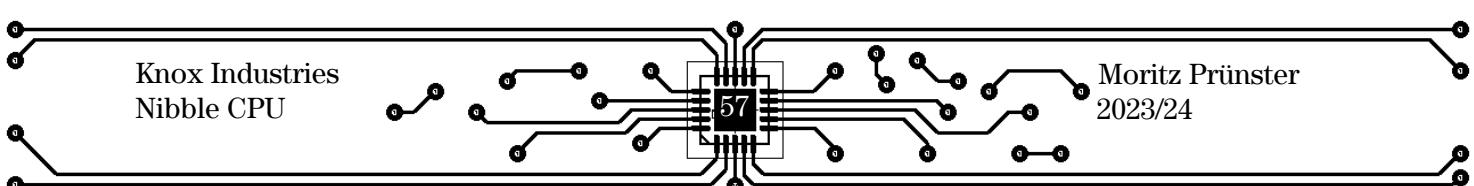


Abb. 60: Ergebnis abspeicherung Schematic Eagle



10.2.1.3 ALU Datenzusammenfasser

Diese Schaltung fasst die Daten aus dem Register und aus den Nummer-Daten der Nummer/Adressen-Erkennung zusammen. Diese 4 Bits werden dann zur Berechnung weitergeleitet. Zusätzlich wird der Ausgang vom zweiten Akkumulator hinzugefügt, wodurch die Daten nun zum ersten Akkumulator gelangen.

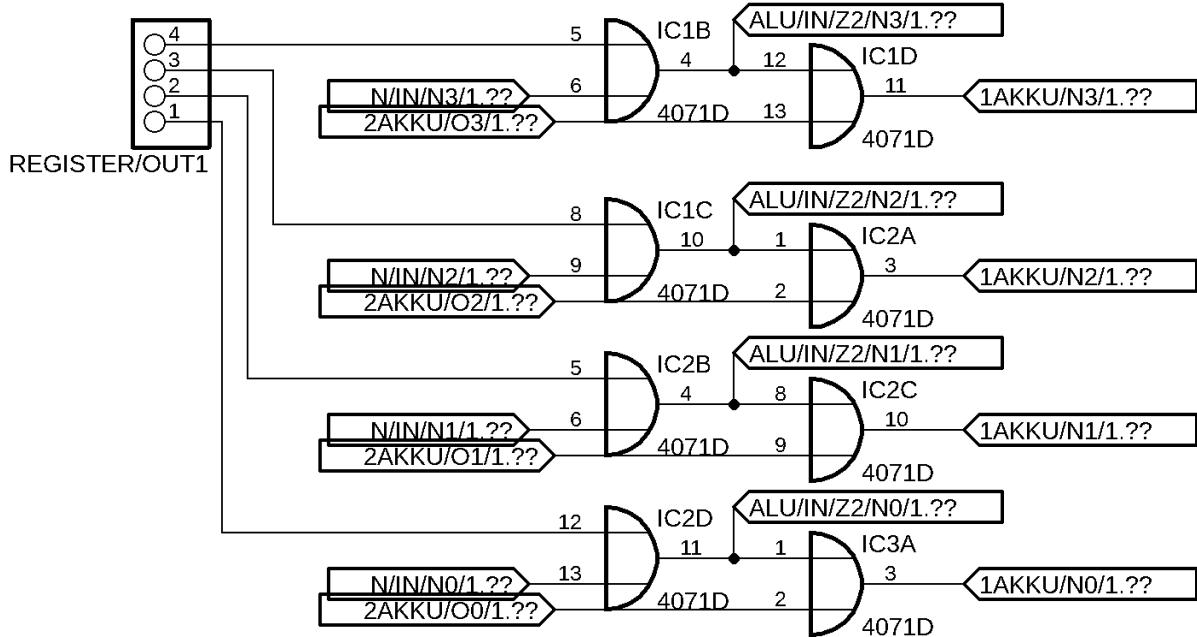
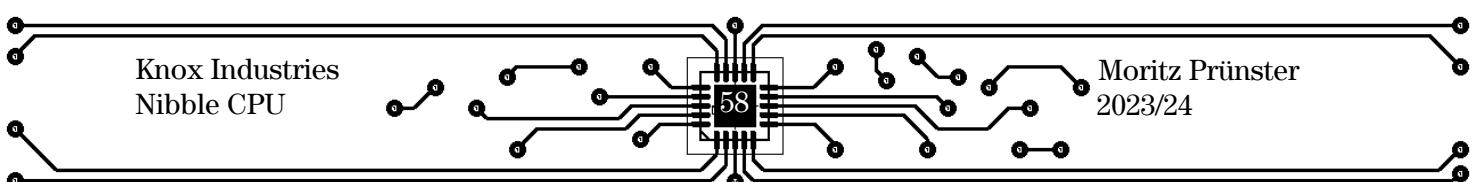
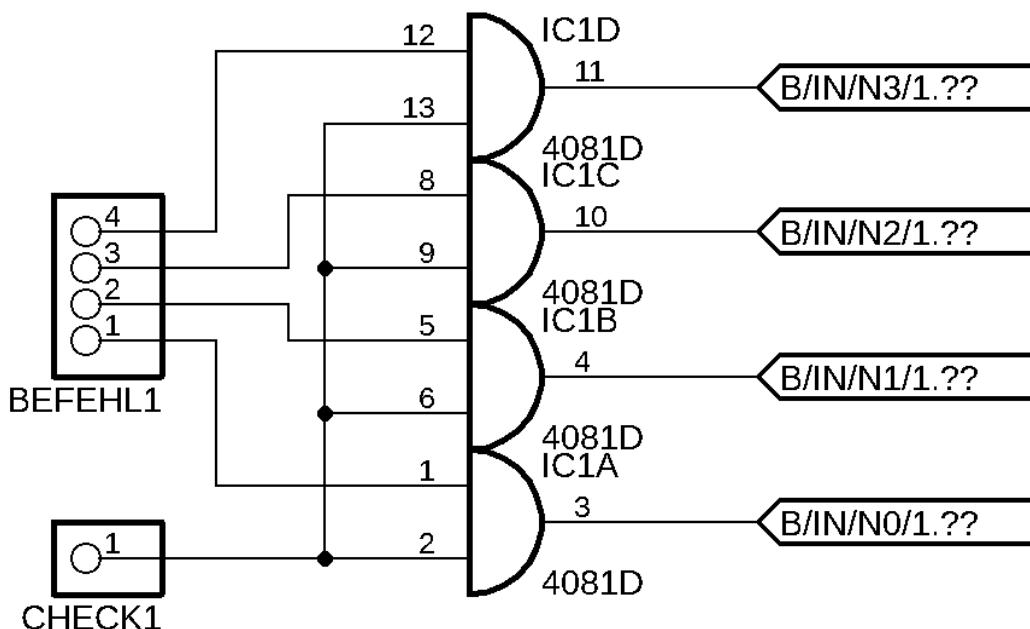


Abb. 61: ALU Datenzusammenfassung Schematic Eagle

10.2.1.4 ALU Befehlsschleuse

Diese Schaltung dient dazu, dass nur, wenn Check HIGH ist, die 4 bit weiter zur Verarbeitung gegeben werden.

Abb 92: Befehlschleuse Schematic Eagle



10.2.1.5 Akku Befehle

Diese Schaltung überprüft die Befehle, die den ersten Akkumulator ansteuern. Sie leitet die Clock von der Steuereinheit an das Register weiter und sendet das Write-Signal an das Register, wenn der Befehl STA (Store at Address) aktiv ist.

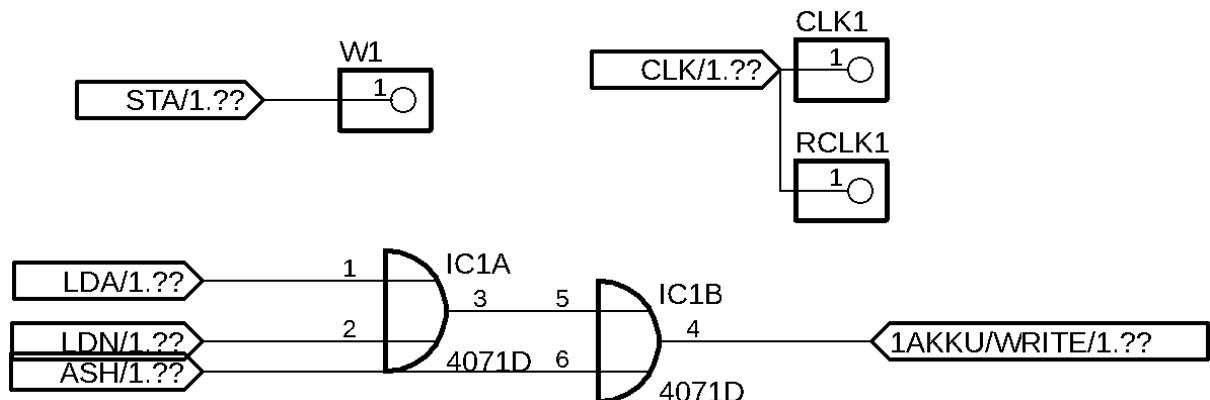
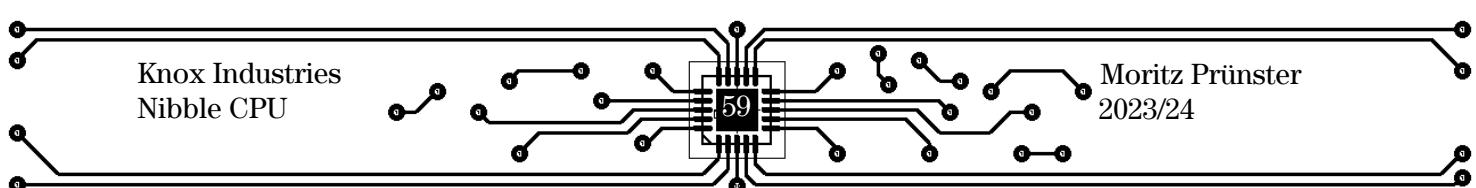
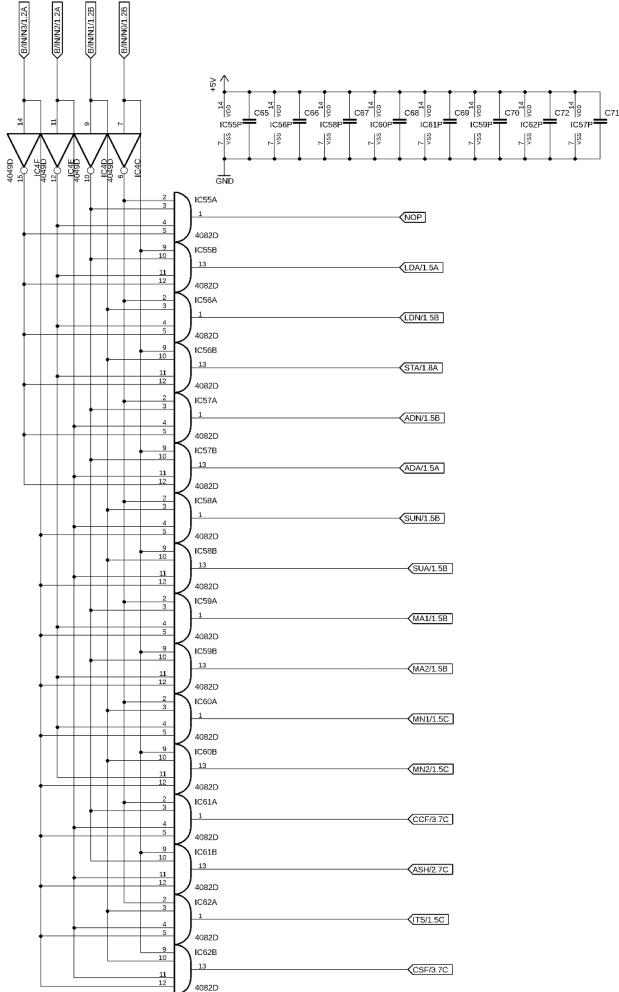


Abb. 63: Akku Befehle Schematic Eagle



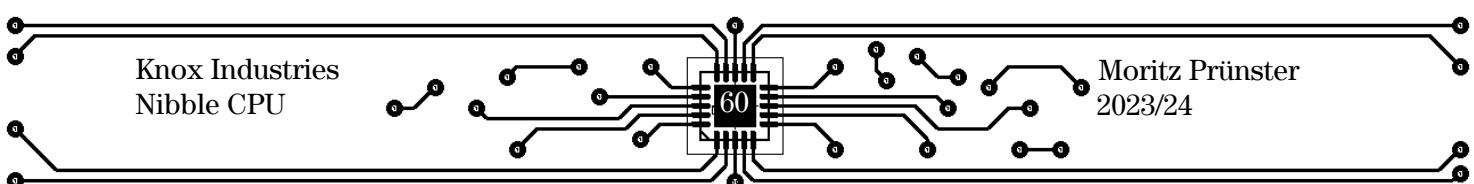
10.2.2 ALU Befehlserkennung

Diese Schaltung verwendet die ersten 4 Bits und funktioniert ähnlich wie ein Multiplexer. Sie kann 16 verschiedene Kombinationen ausgeben, wobei jede Kombination einem spezifischen Befehl entspricht. Dabei werden die 4 Bits jeweils invertiert und weiter gegeben, dann mit den 4 input AND-Gatter jede Kombination durchprobiert.



Befehl	Kombination
nop	0000
lda	0001
ldn	0010
sta	0011
adn	0100
ada	0101
sun	0110
sua	0111
ma1	1000
ma2	1001
mn1	1010
mn2	1011
ccf	1100
ash	1101
its	1110
csf	1111

Abb. 64: ALU Befehlserkennung Schematic Eagle



10.2.3 Akkumulatoren

Diese Schaltung besteht aus zwei Akkumulatoren, die jeweils als temporärer Speicher für 4-Bit-Daten dienen. Der erste Akkumulator, der sich unten links befindet, fungiert als Haupt-Akkumulator, in dem alle geladenen Daten gespeichert werden. Der zweite Akkumulator dient als Zwischenspeicher für Berechnungsergebnisse.

Der Grund, warum das Ergebnis von Berechnungen nicht direkt im ersten Akkumulator gespeichert wird, liegt darin, dass die Daten des ersten Akkumulators für die Berechnung verwendet werden. Wenn die Daten im ersten Akkumulator geändert würden, würde dies das Berechnungsergebnis beeinflussen, was wiederum die Daten im ersten Akkumulator ändern würde. Dies würde zu einem Endlosschleifen Problem führen. Um dies zu vermeiden, wird das Ergebnis der Berechnung zunächst im zweiten Akkumulator gespeichert, bevor es gegebenenfalls, mit dem Befehl ASH(Akkumulator shift), in den ersten Akkumulator übertragen wird.

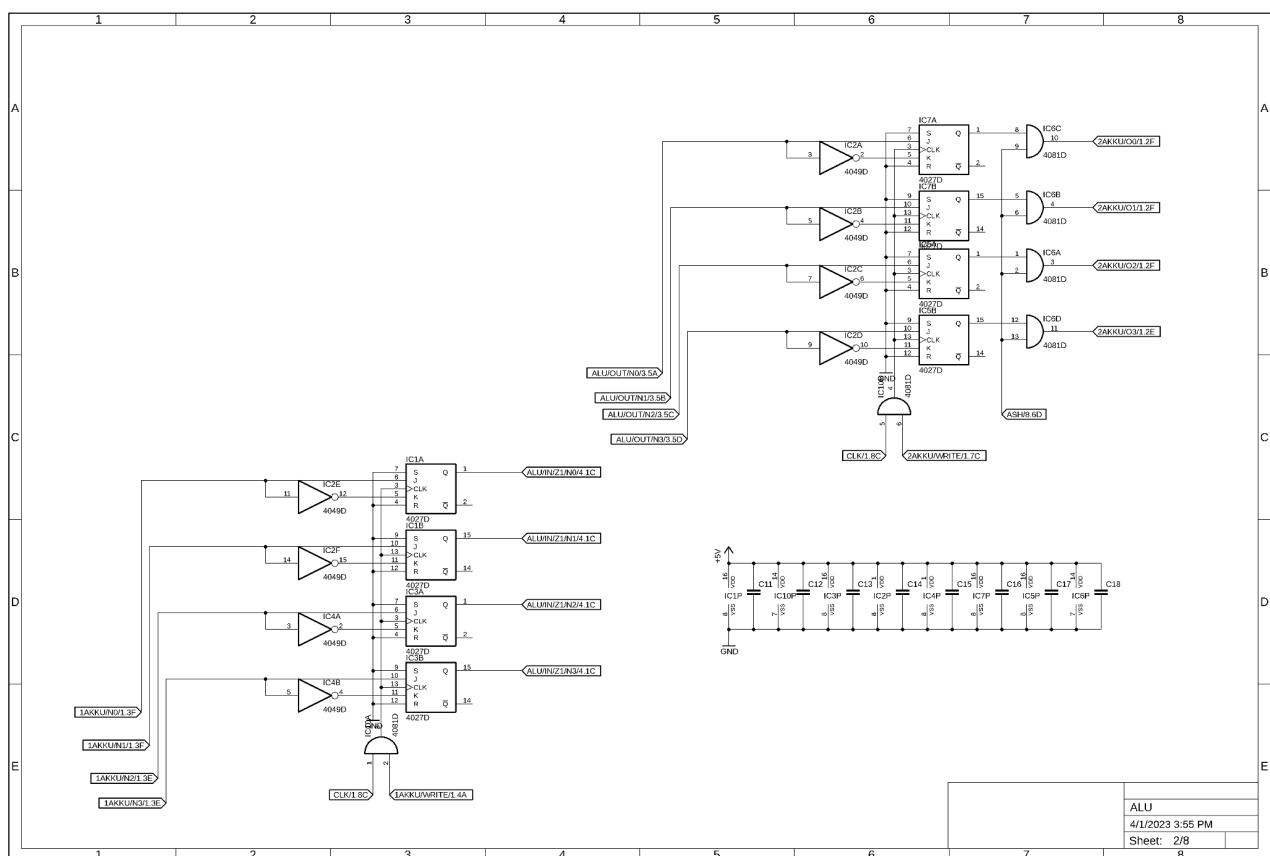
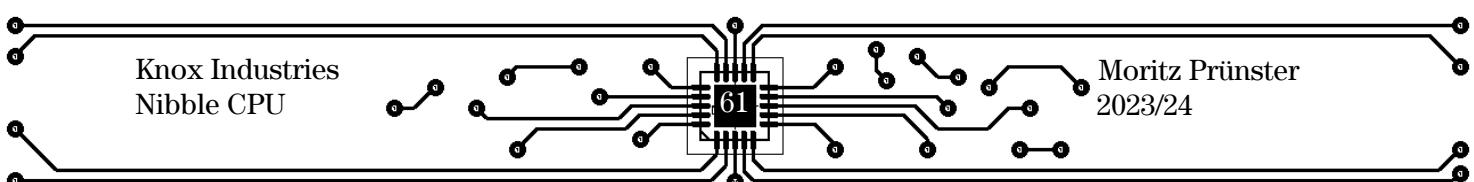


Abb. 65: Akkumulatoren Schematic Eagle



10.2.4 Ergebnis Abspeicherung

Dieser Teil der Schaltung empfängt die Ergebnisse vom Addierer, Subtrahierer und Multiplizierer und leitet diese nur dann an den zweiten Akkumulator weiter, wenn der jeweilige Befehl aktiviert ist. Zusätzlich sind in diesem Abschnitt zwei JK-Flipflops integriert, die das Same-Flag und das Carry-Flag speichern. Diese Flipflops werden bei den entsprechenden Befehlen wie CCF und CSF auch zurückgesetzt.

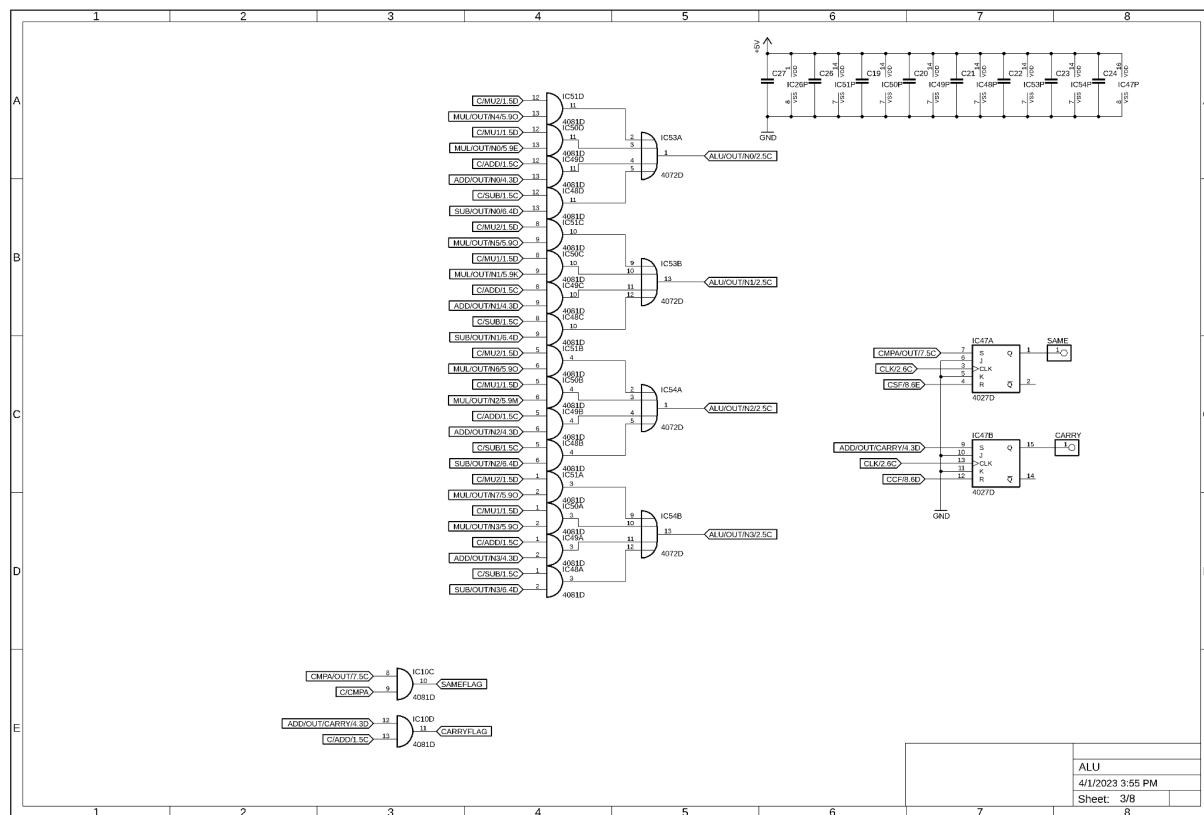
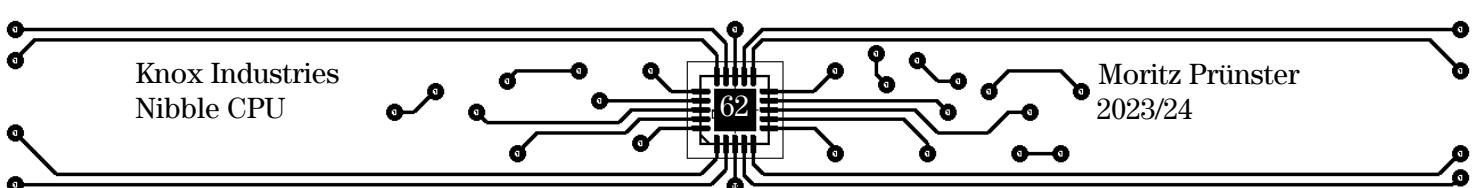


Abb. 66: Ergebnis Abspeichern Schematic Eagle



10.2.5 Addierer

Der 4-Bit-Addierer addiert zwei 4-Bit-Zahlen zusammen und funktioniert folgendermaßen: Er besteht aus vier Volladdierern, die jeweils eine 1-Bit-Addition durchführen. Jeder Volladdierer hat drei Eingänge (Z1, Z2, Carry-In) und zwei Ausgänge (Q, Carry-Out). Um mehrere Volladdierer miteinander zu verbinden, verbinden wir das carry-out mit dem carry-in des nächsten Volladdierers.

Z1	Z2	carry in	Q	carry out
0	0	0	0	0
1	0	0	1	0
1	1	0	0	1
1	1	1	1	1

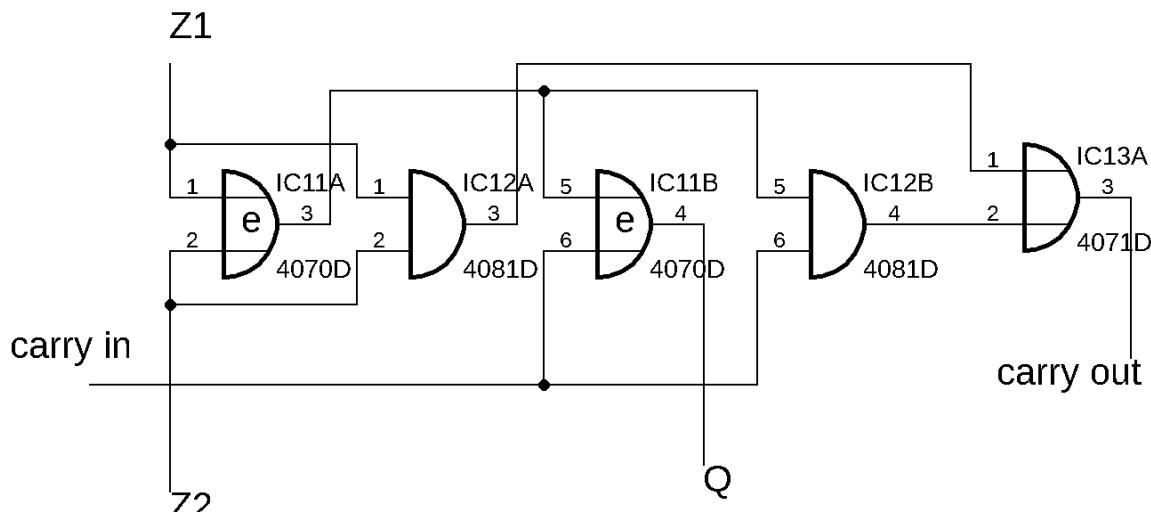


Abb. 67: ein bit Volladdierer Schematic Eagle

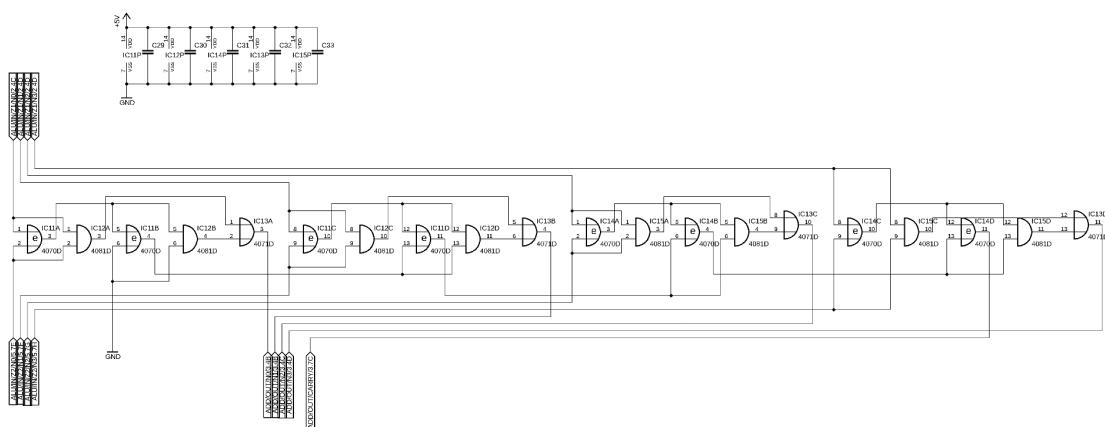
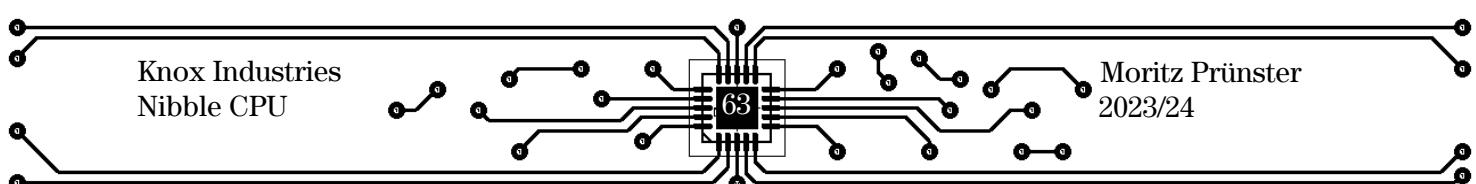


Abb. 68: 4 bit Addierer Schematic Eagle



10.2.6 Subtrahierer

Der Subtrahierer verwendet zwei Addierer, um eine Subtraktion durchzuführen. Dabei wird der Subtrahend zunächst invertiert und dann mit eins addiert, um die Zweierkomplement-Darstellung zu erhalten. Diese negative Zahl wird anschließend mit dem Minuend addiert. Das Ergebnis der zweiten Addition ist das Gesamtresultat der Subtraktion. Zusätzlich, falls das Carry-Bit bei der zweiten Addition HIGH ist, bedeutet das, dass das Ergebnis positiv ist. Andernfalls ist das Ergebnis negativ.

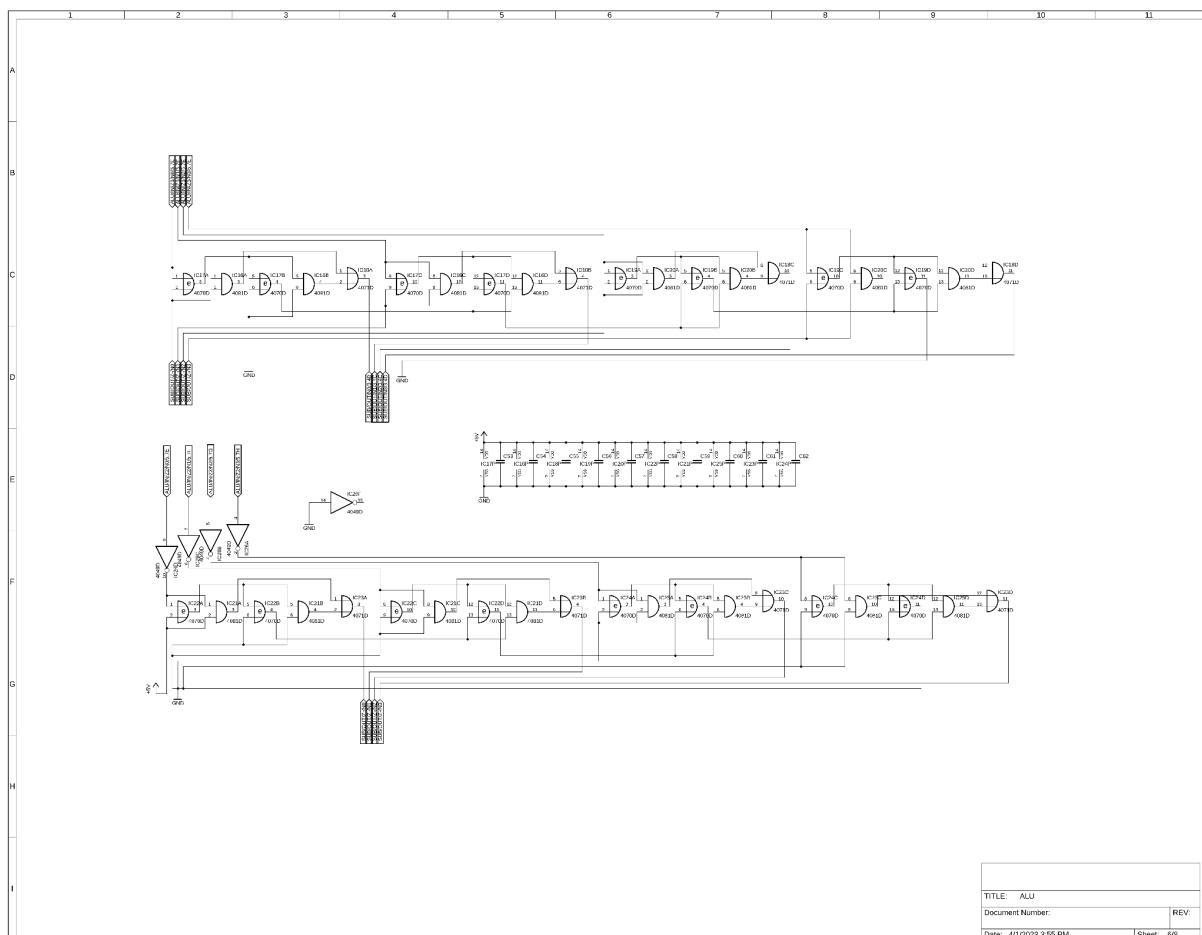
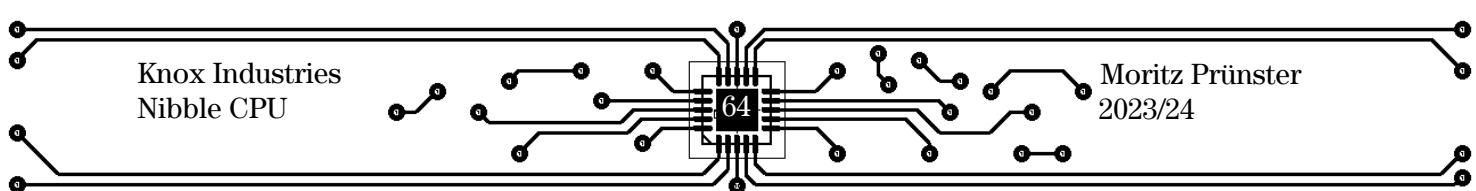


Abb. 69: Subtrahierer Schematic Eagle



10.2.7 Multiplizierer

Der Multiplizierer funktioniert ähnlich wie das schriftliche Multiplizieren auf Papier. Zuerst haben wir zwei 4-Bit-Zahlen: den Multiplikanden und den Multiplikator. Wir beginnen mit dem Multiplikanden und überprüfen jedes Bit nacheinander, ob es 1 (HIGH) ist oder nicht. Wenn es 1 ist, wird der Multiplikator weitergeleitet; wenn nicht, wird 0000 weitergegeben. Dann werden die Daten nach jedem Bit um eine Stelle verschoben und die Ergebnisse zusammenaddiert.

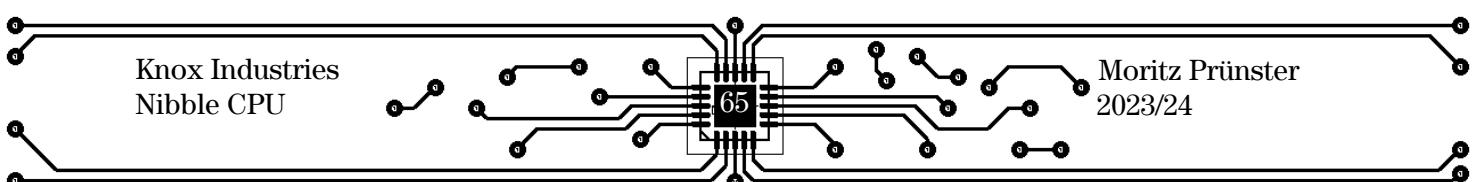
Um das zu verdeutlichen, betrachten wir ein Beispiel: Wir möchten 1010 (10 in Dezimal) mit 1100 (12 in Dezimal) multiplizieren.

wir nehmen nun den Multplikant 1010 her und überprüfen das erste bit es ist 0 weshalb wir 0000 anschreiben das zweite bit ist 1 weshalb wir den Multiplikator anschreiben also 1100 das machen wir nun auch für die anderen zwei bits herausbekommen sollen wir 0000,1100,0000,1100 nun schreiben wir es so an das es jeweils um eine ziffer verschoben ist also:

$$\begin{array}{r}
 0000 \\
 1100 \\
 0000 \\
 +1100 \\
 \hline
 1111000
 \end{array}$$

Nun können wir die Zahlen normal addieren. Dies ist auch der Grund, warum wir drei Addierer für den Multiplikator benötigen, da drei Additionen durchgeführt werden, um die vier Zahlen zusammen zu berechnen. Am Ende erhalten wir 1111000, was 120 im Dezimalsystem entspricht. Das ist das Ergebnis von 12 mal 10.

Spätestens jetzt können wir erkennen, warum es bei der Multiplikation zwei Teile gibt. Das Ergebnis der Operation kann 8 Bits groß sein, aber da wir nur 4 Bits verwenden und speichern können, wird das Ergebnis einfach in zwei Teile geteilt.



Nibble CPU

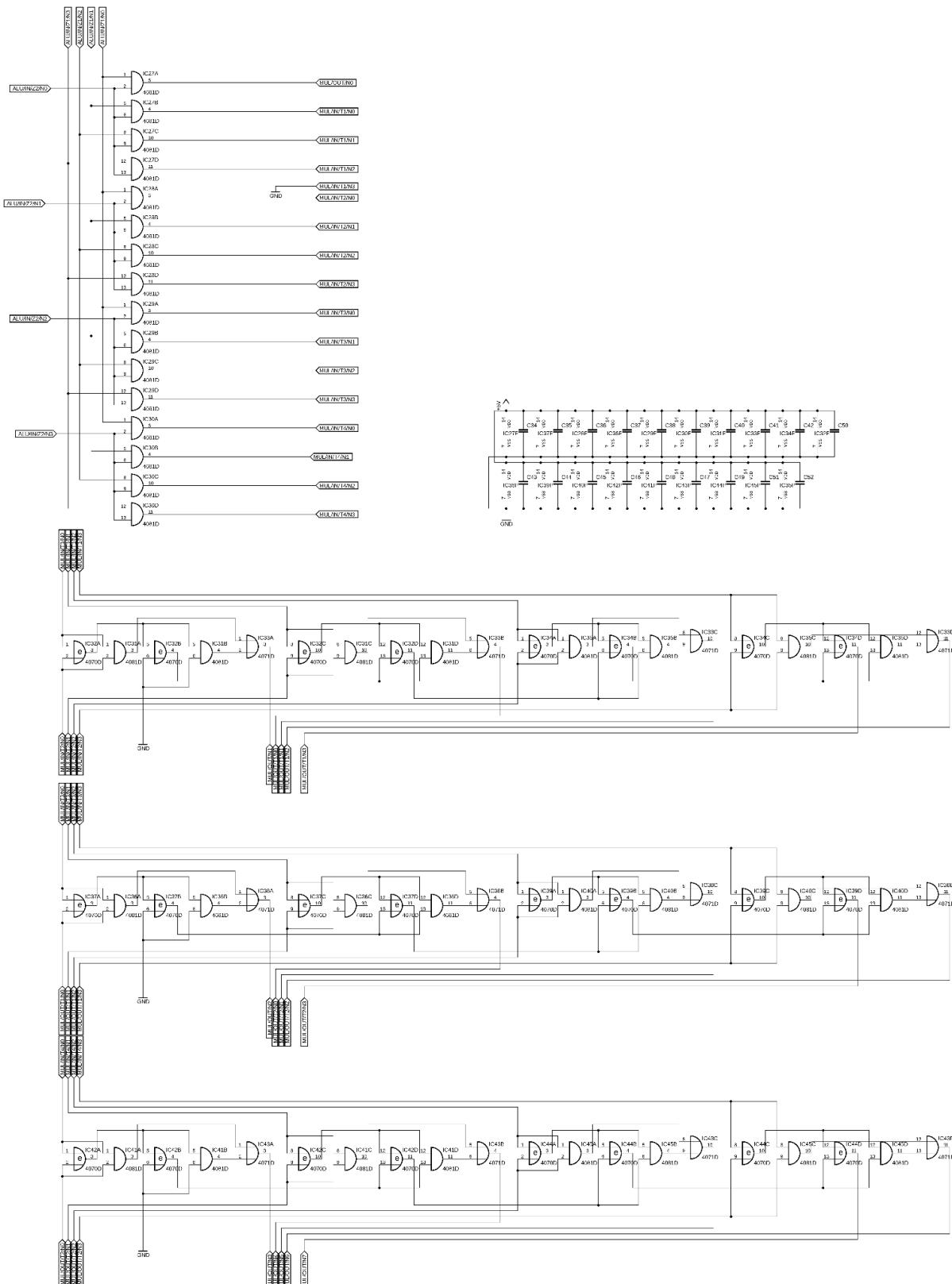
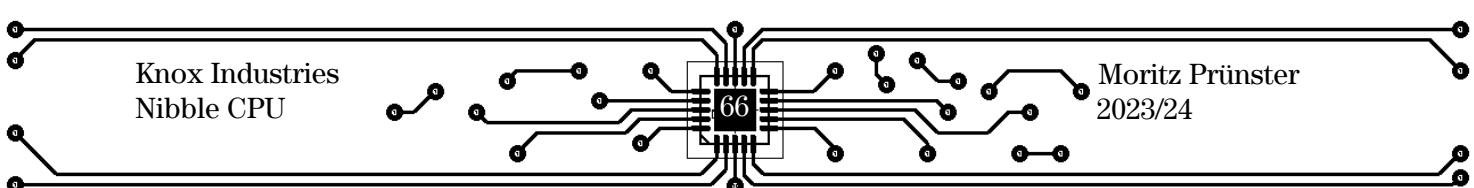


Abb. 70: Multiplizierer Schematic Eagle



10.2.8 Vergleicher

Diese Schaltung vergleicht einfach nur zwei Daten miteinander: wenn sie gleich sind, gibt es ein HIGH-Signal an das 10.2.4 Ergebnis Abspeicherung, wo es dann als Same Flag gesetzt wird.

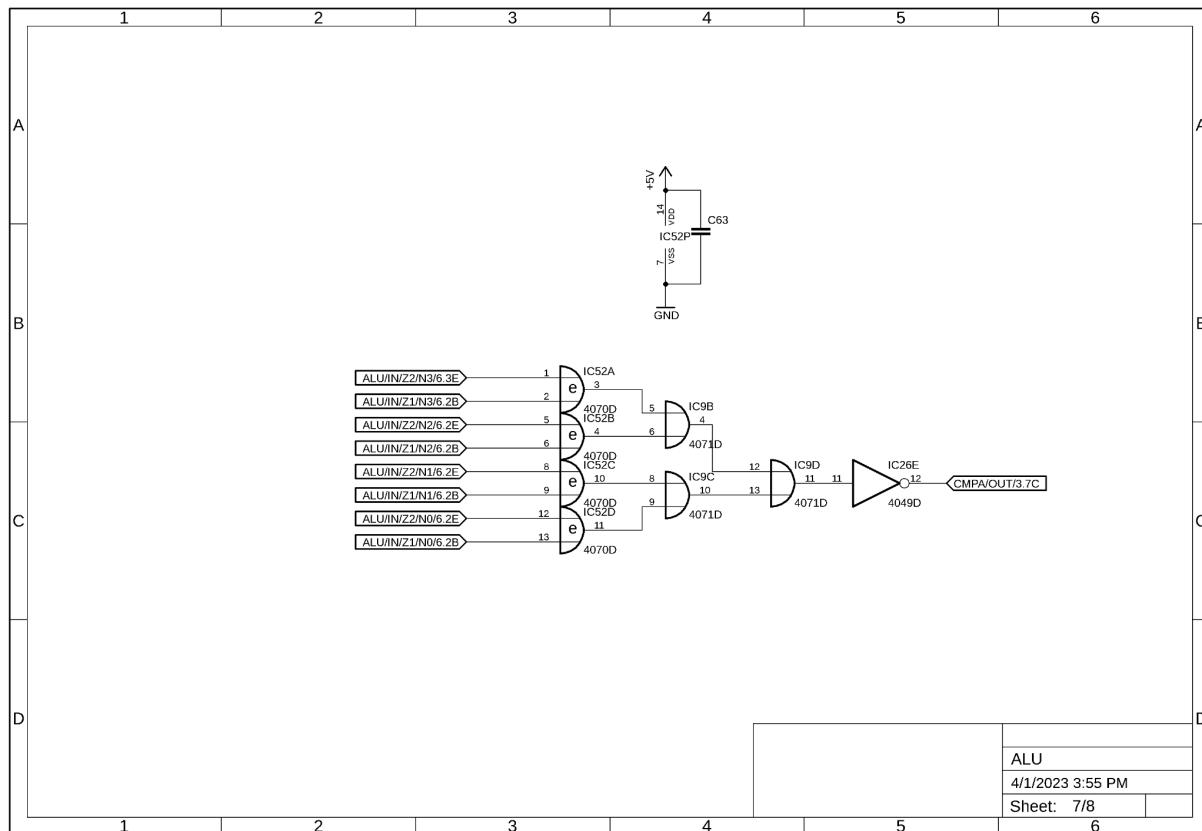
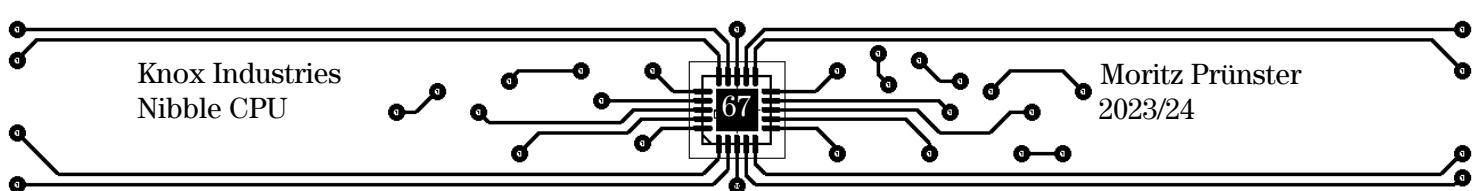


Abb. 71: Vergleicher Schematic Eagle



10.3 Register (Reg)

Das Register speichert 16 verschiedene 4-Bit-Daten ab und kann diese auch auslesen. Wenn Daten von der ALU kommen, können sie gespeichert werden, aber nur, wenn die ALU ein HIGH-Signal am Write-Pin ausgibt. Beim Lesen verhält es sich ähnlich: Nur wenn an der Read-Pin ein HIGH-Signal anliegt, können Daten ausgelesen werden. Die Adresse allein reicht nicht aus; es muss auch ein HIGH-Signal am Read-Pin vorhanden sein, um Daten auszulesen.

Wenn Daten gelesen oder geschrieben werden sollen, wird die Adresse zunächst an den Adressenauswerter gesendet, der die entsprechende Speicherstelle ermittelt und zurückgibt, welche Speicherzelle angesteuert werden soll. Anschließend können die Daten an der jeweiligen Adresse gelesen oder geschrieben werden. Zusätzlich wird der Ausgang der Speicherzelle permanent an die Steuereinheit gesendet, wobei der Wert "0000" repräsentiert wird.

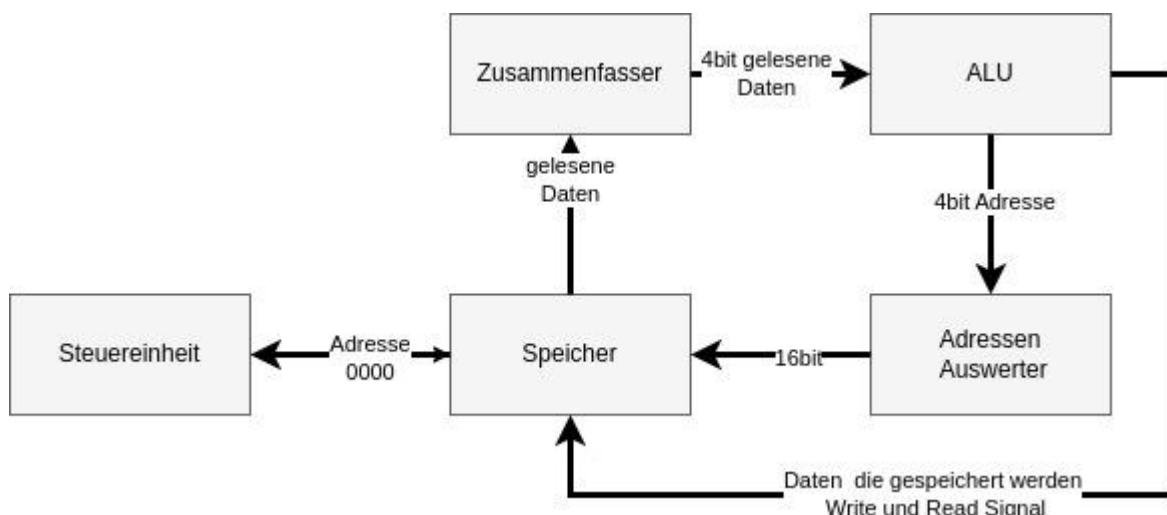
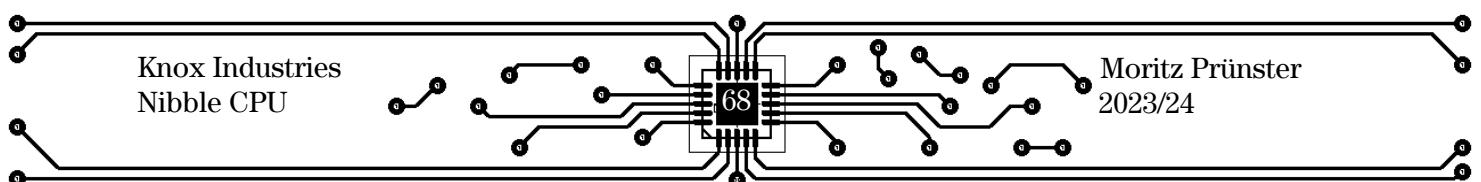


Abb. 72: Datenverteilungsdiagramm Register



10.3.1 Adressen Auswerter

Diese Schaltung dient dazu, die richtige Speicherzelle anzusteuern. Sie funktioniert ähnlich wie ein Multiplexer. Die vier Eingangs Bits werden zunächst invertiert und anschließend wird jede mögliche Kombination durch ein 4-Input-AND-Gatter geleitet. So wird sichergestellt, dass nur die gewünschte Speicherzelle aktiviert wird.

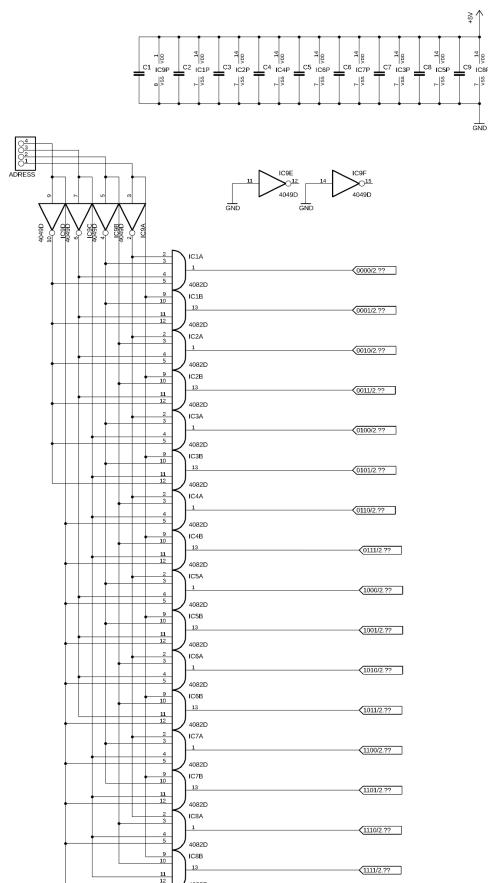
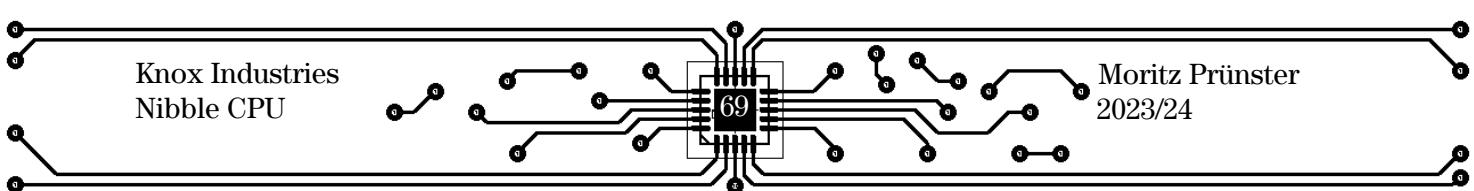


Abb. 73: Adressen Auswerter Schematic Eagle



10.3.2 Speicher

Der Speicher besteht aus 16 Speicherzellen, die jeweils wie folgt aufgebaut sind

Zuerst kommt ein Signal von Adressen-Auswerten zur jeweiligen Speicherzelle, die die gleiche Adresse besitzt. Dieses Signal geht dann in zwei AND-Gatter am Anfang der Speicherzelle. Wenn ein Write-Signal ankommt, wird der Ausgang des ersten AND-Gatters HIGH und das Signal wird zur Clock-Pin der JK-Flipflops weitergeleitet. Wenn das Read-Signal aktiv ist, wird der Ausgang des zweiten AND-Gatters HIGH. Dieses Signal öffnet eine Schleuse, die die gespeicherten 4-Bit-Daten durchlässt.

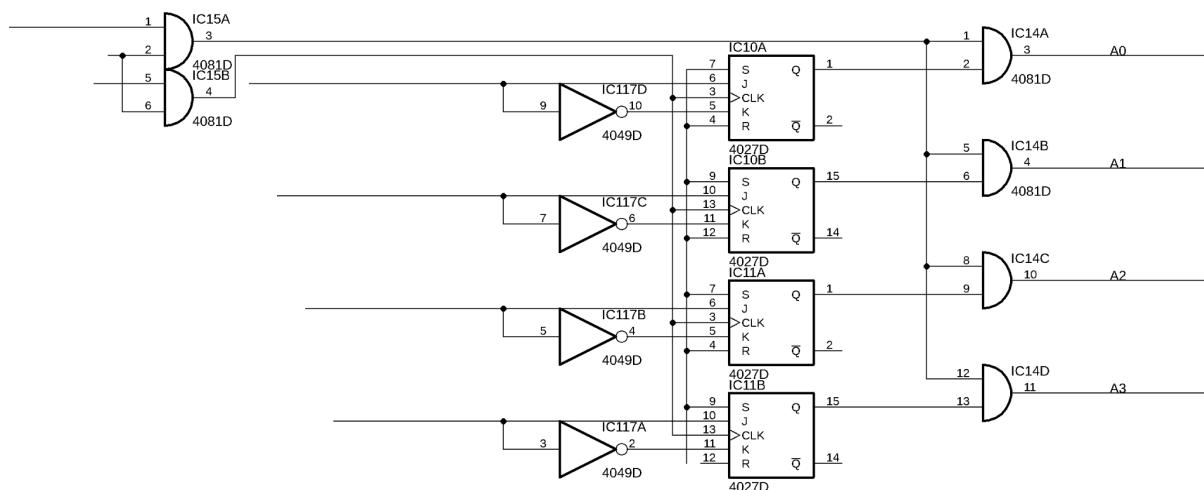
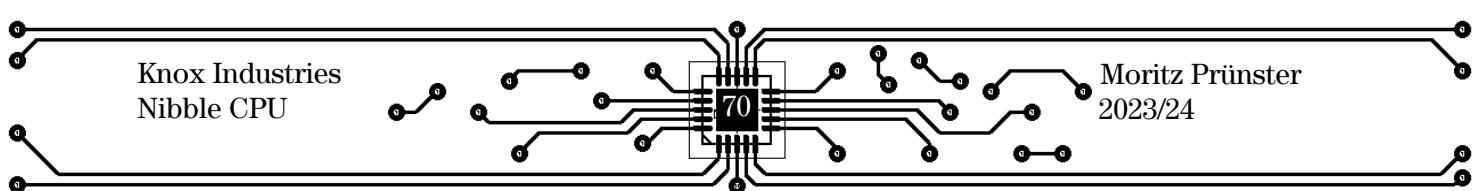


Abb. 74: Speicherzelle Schematic Eagle



Nibble CPU

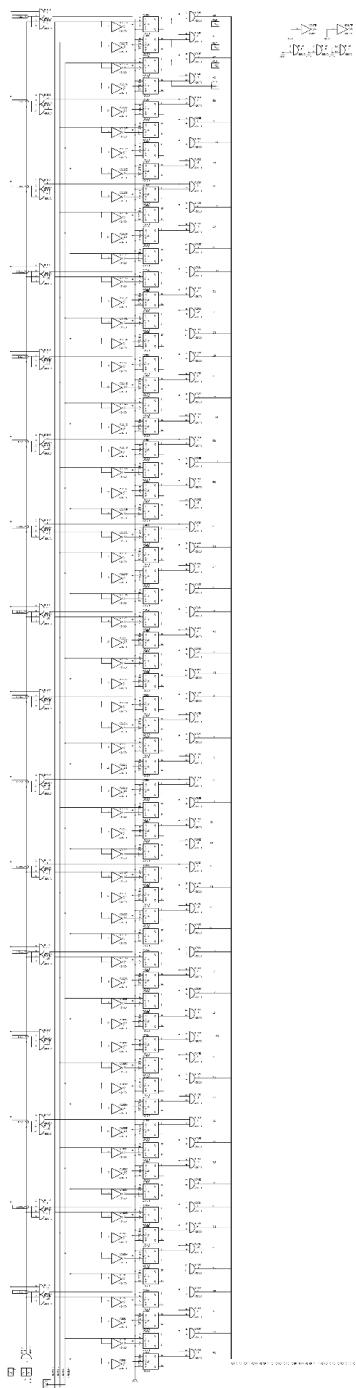
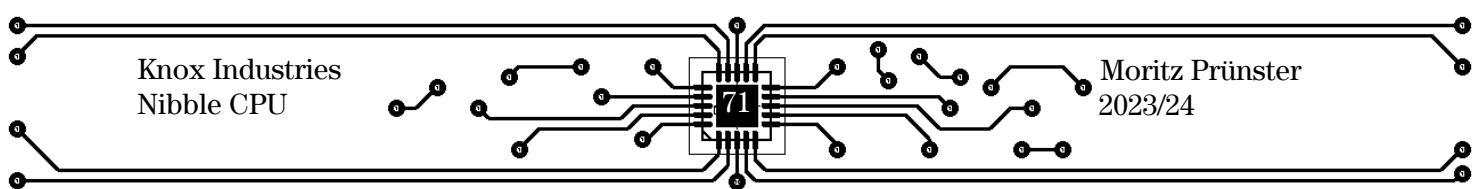


Abb. 75: komplette Speicher Schematic Eagle



10.3.3 Zusammenfasser

Diese Schaltung fasst die gesendeten Daten der 16 Speicherzellen in eine 4-Bit-Nummer zusammen. Da immer nur eine Speicherzelle ausgelesen wird, können die Ausgänge der Speicherzellen einfach mit OR-Gattern verbunden werden, um die 4-Bit-Daten zusammenzuführen.

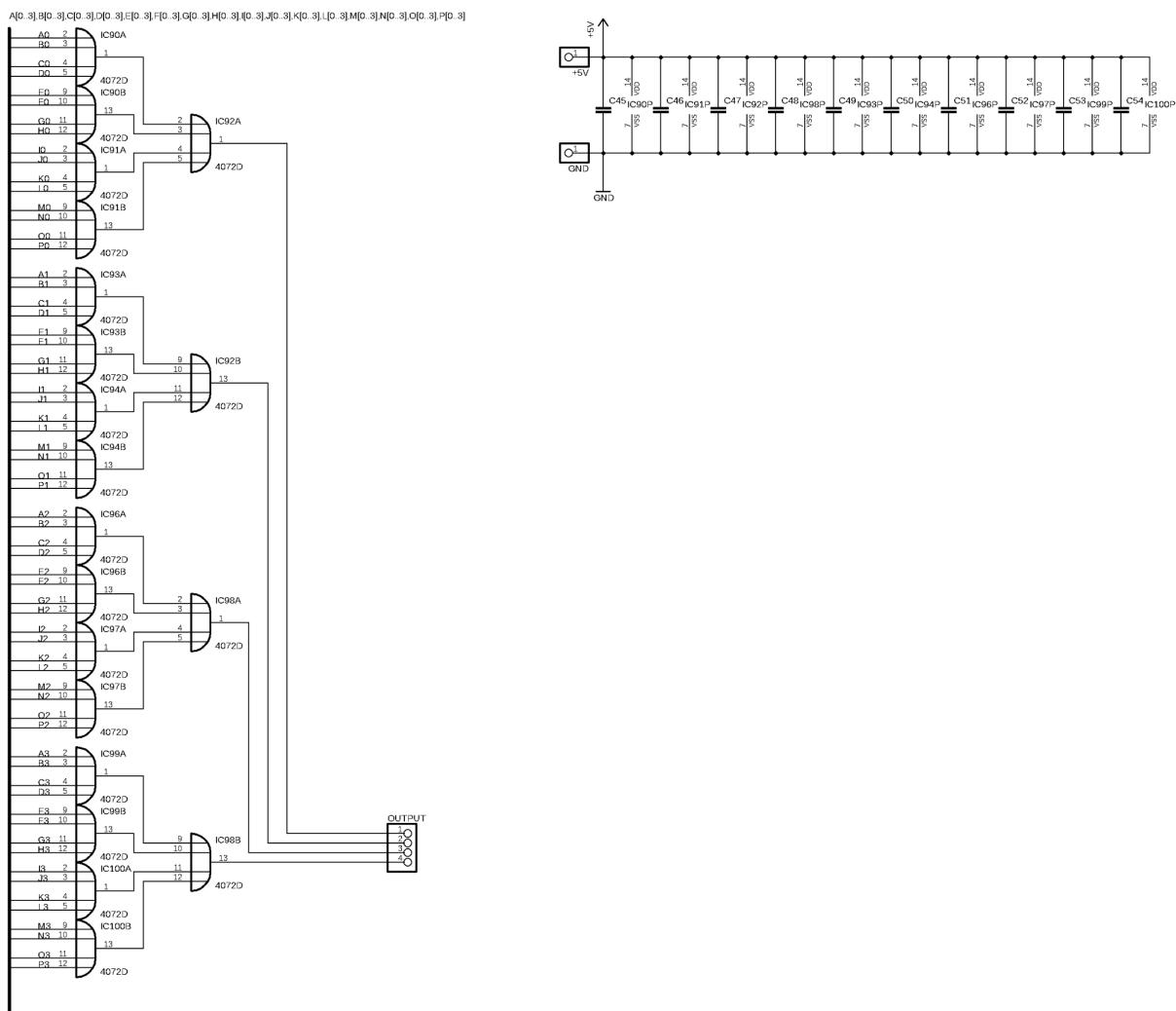
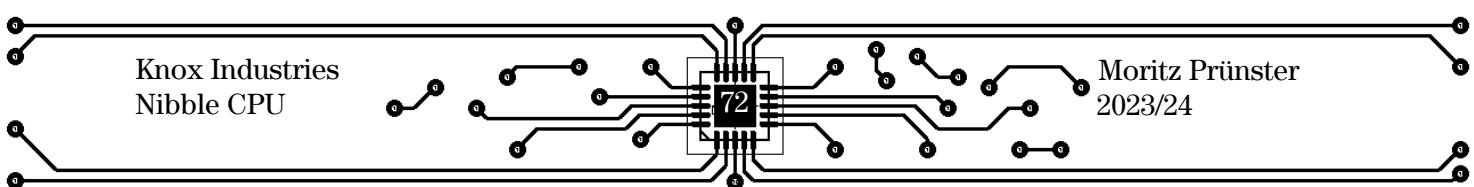


Abb. 76: Zusammenfasser Schematic Eagle



11. Platinen

Die Platinen wurden bei jlpcb.com bestellt. Dabei hatten die Platinen der ALU und des Registers 6 Layer, während die Platine der Steuereinheit nur 4 Layer hatte. Beim Design der Platinen wurden die jeweiligen Entkoppelkondensatoren auf die Unterseite der Platine gelegt, damit auf der Oberseite mehr Platz ist.

Steuereinheit:

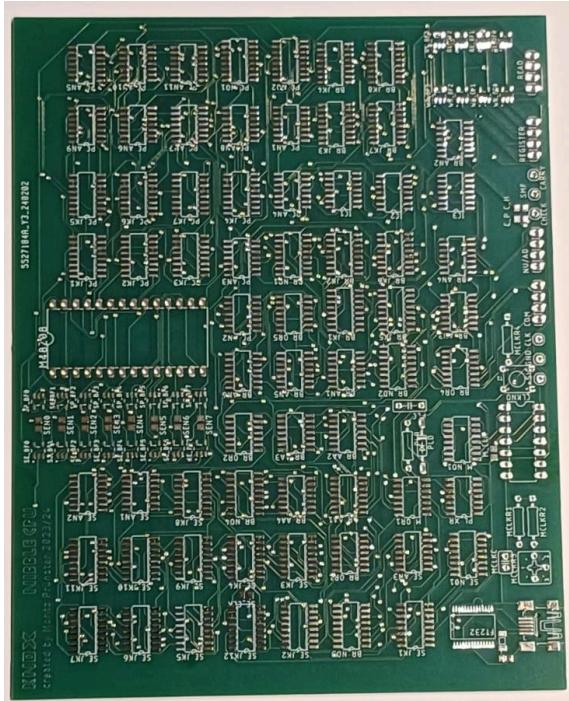


Abb. 77: Platine Steuereinheit vorne Foto

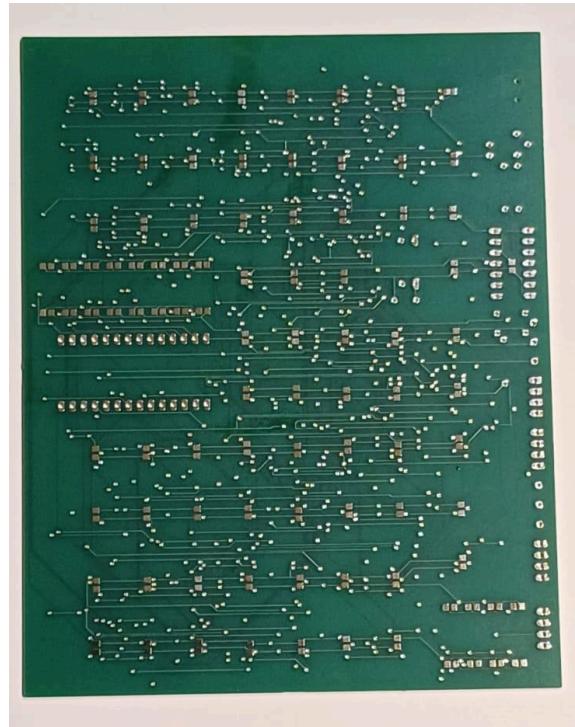
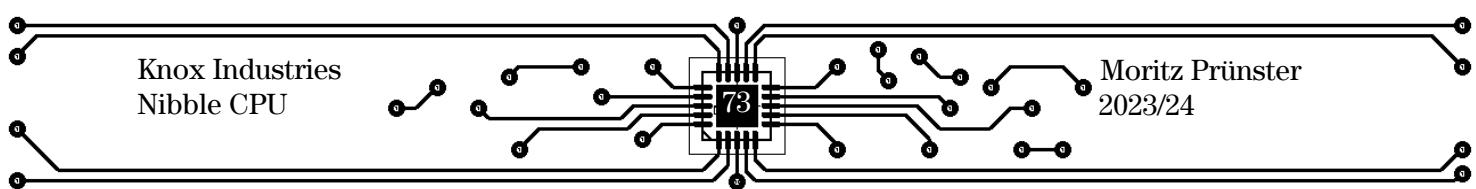


Abb. 78: Platine Steuereinheit hinten Foto



ALU:

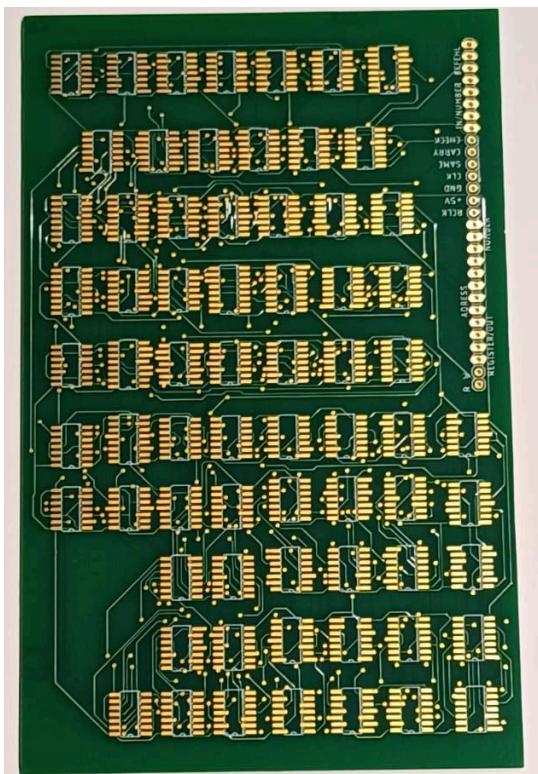


Abb. 79: Platine ALU vorne Foto

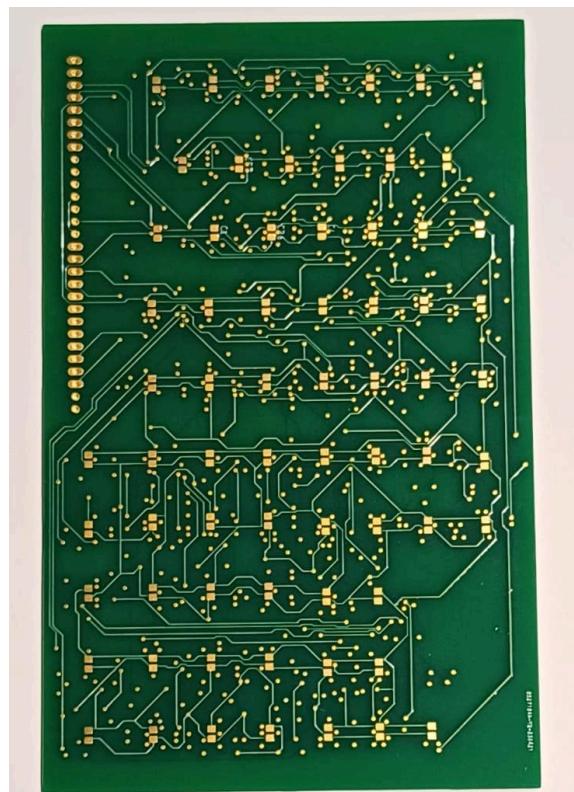


Abb. 80: Platine ALU hinten Foto

Register:

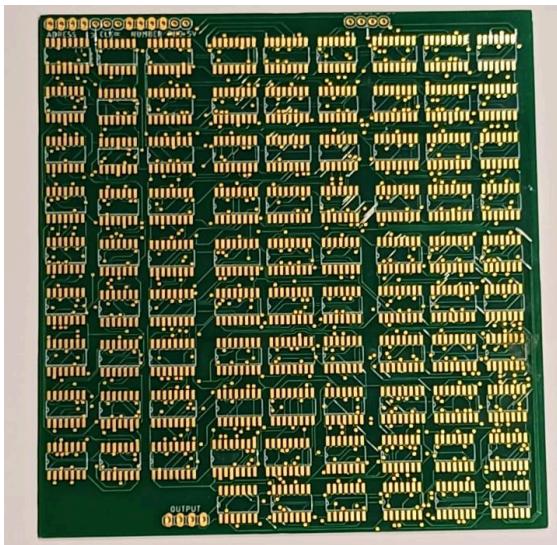


Abb. 81: Platine Register vorne Foto

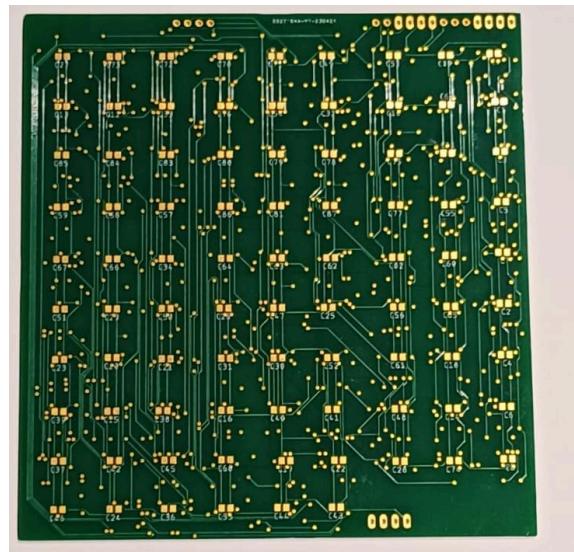
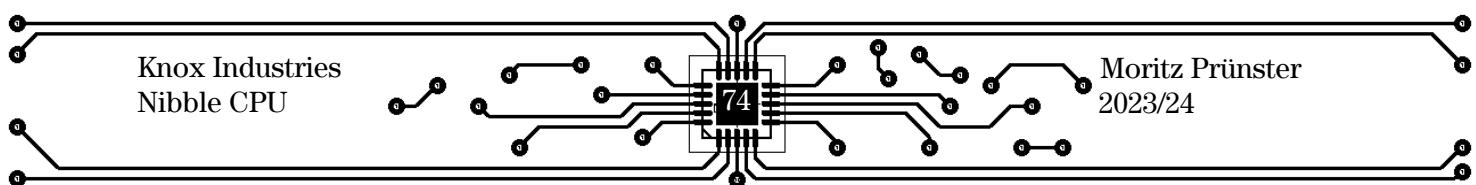


Abb. 82: Platine Register hinten Foto



12. NAL

NAL steht für Nibble Assembly Language und umfasst 28 Befehle. Diese setzen sich aus 17 ACOM (Arithmetic Commands), 7 BCOM (Branch Commands) und 4 SCOM (Software Commands) zusammen. Jeder Befehl ist durch einen OP-Code definiert, unter dem die CPU ihn versteht. Ein Befehl besteht aus genau 3 Zeichen. Alles, was länger als 3 Zeichen ist, wird als Marker oder Makro interpretiert.

12.1 Marker

Marker sind dazu da, um Punkte zu markieren, zu denen man springen möchte. Man erkennt Marker daran, dass sie länger als drei Zeichen sind und entweder allein am Rand des Textes stehen oder hinter einem Branch-Befehl.

12.2 Makros

Makros dienen dazu, den Code übersichtlicher zu gestalten, indem man anstelle von bloßen Nummern Wörter oder Text verwenden kann, auf die sich die Nummer bezieht. Um dies zu erreichen, gibt es zwei Befehle: "ma:" steht für den Makrostart und ist ein SCOM. Alles, was nach "ma:" kommt, wird als Makro interpretiert. Dies endet dann bei ":me", was für "Makro Ende" steht, dies ist auch ein SCOM. Das bedeutet, man könnte nun folgendes verwenden:

ma:

test 10

:me

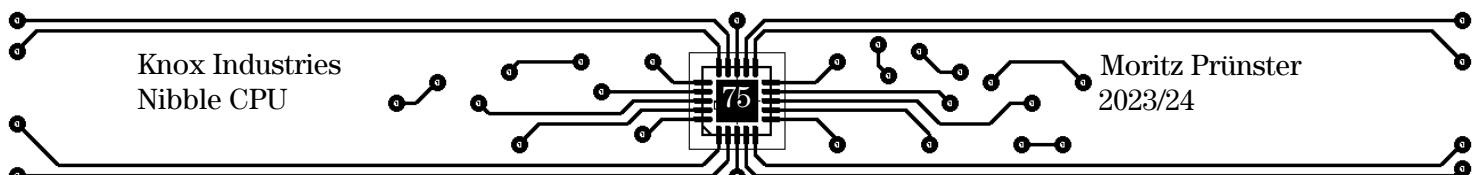
Durch diesen Codeschnipsel kann man im Code "test" statt "10" verwenden. Das bedeutet, man muss jetzt nicht mehr etwas an der Adresse 10 abspeichern, sondern bei "test". Das erleichtert die Lesbarkeit und Wartung des Codes, da nun der Zweck von "test" klarer ist als nur eine Nummer.

12.3 Nummern

In der Nibble Assembly Language kann man Zahlen auf drei verschiedene Arten schreiben: Dezimal, Hexadezimal und Binär.

- Dezimal: Man schreibt einfach die Zahl ohne Vorzeichen.
- Hexadezimal: Man verwendet "0x" vor der Zahl, um anzugeben, dass es sich um eine hexadezimale Zahl handelt.
- Binär: Man verwendet "0b" vor der Zahl, um anzugeben, dass es sich um eine binäre Zahl handelt.

Nummern haben die Farbe **#eba96c** beim Text-Highlighting.



12.4 Start

Um den Beginn und das Ende des Programms in der Nibble Assembly Language zu markieren, gibt es zwei Befehle: "st:" für den Start und ":en" für das Ende, bei beiden handelt es sich um SCOM. Alles, was zwischen diesen beiden Markierungen steht, wird in die CPU geladen und ausgeführt.

12.5 Branch Commands(BCOM)

Die folgenden Befehle werden von der Steuereinheit ausgeführt und dienen dazu, Sprünge zu Markern durchzuführen. Sie werden beim Text-Highlighting mit der Farbe **#f8c02d** dargestellt.

12.5.1 BRA

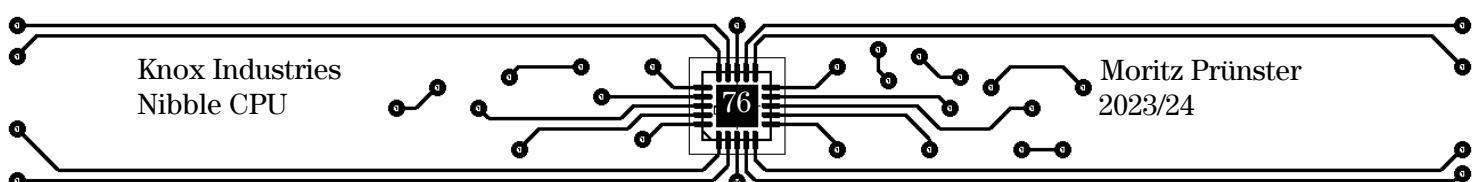
Volle Bezeichnung:	Branch
Opcode:	0000 0001
Funktion:	Springt immer auf den nebenstehenden Marker und kann von nichts daran gehindert werden.

12.5.2 BEQ

Volle Bezeichnung:	Branch if equal
Opcode:	0000 0010
Funktion:	Springt nur, wenn das Sameflag auf HIGH ist, auf den nebenstehenden Marker.

12.5.3 BCY

Volle Bezeichnung:	Branch if carry
Opcode:	0000 0011
Funktion:	Springt nur, wenn das carryflag auf HIGH ist, auf den nebenstehenden Marker.



12.5.4 RE0

Volle bezeichnung:	Branch if Read pin 0
Opcode:	0000 0101
Funktion:	Springt nur, wenn der Read Pin 0 auf HIGH ist, auf den nebenstehenden Marker.

12.5.5 RE1

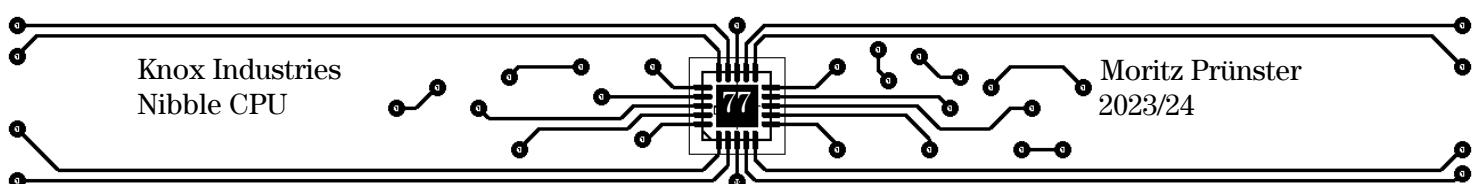
Volle bezeichnung:	Branch if Read pin 1
Opcode:	0000 0110
Funktion:	Springt nur, wenn der Read Pin 1 auf HIGH ist, auf den nebenstehenden Marker.

12.5.6 RE2

Volle bezeichnung:	Branch if Read pin 2
Opcode:	0000 0111
Funktion:	Springt nur, wenn der Read Pin 2 auf HIGH ist, auf den nebenstehenden Marker.

12.5.7 RE3

Volle bezeichnung:	Branch if Read pin 3
Opcode:	0000 1000
Funktion:	Springt nur, wenn der Read Pin 3 auf HIGH ist, auf den nebenstehenden Marker.



12.6 Arithmetic Commands(ACOM)

Die nun folgenden Befehle werden überall ausgeführt. Beim Text-Highlighting bekommen sie die Farbe #ff8f2e. Wenn der Opcode bei den letzten 4 Bits XXXX steht bedeutet dies, dass das die Inhaltsdaten sind, also die Adresse oder Nummer je nach Befehl.

12.6.1 REA

Volle bezeichnung:	Read
Opcode:	0000 0100
Funktion:	Es liest die Daten der Hauptpins ein und lädt sie in den ersten Akku, um es zu ermöglichen, sie zu verwenden
Ausführungsort:	Steuereinheit,ALU

12.6.2 NOP

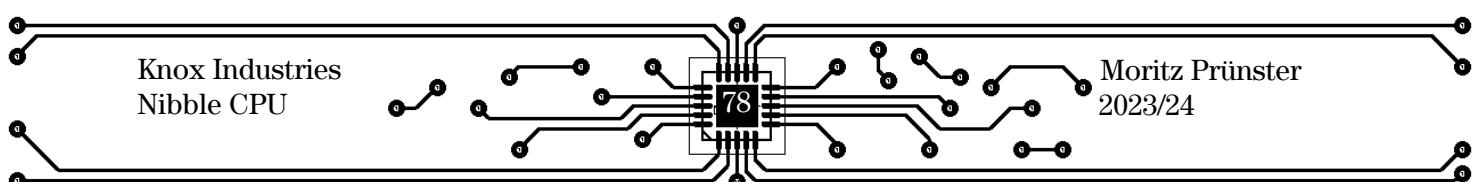
Volle bezeichnung:	NOPE
Opcode:	0000 0000
Funktion:	macht nichts, dient als Platzhalter.
Ausführungsort:	nirgendwo

12.6.3 LDA

Volle bezeichnung:	Load Address
Opcode:	0001 XXXX
Funktion:	Lädt den Inhalt der Adresse in den ersten Akku.
Ausführungsort:	ALU, Register

12.6.4 LDN

Volle bezeichnung:	Load Number
Opcode:	0010 XXXX
Funktion:	Lädt die Nummer in den ersten Akku.
Ausführungsort:	ALU



12.6.5 STA

Volle Bezeichnung:	Store at Address
Opcode:	0011 XXXX
Funktion:	Speichert den Inhalt vom ersten Akku in die angegebene Adresse
Ausführungsort:	ALU, Register

12.6.6 ADN

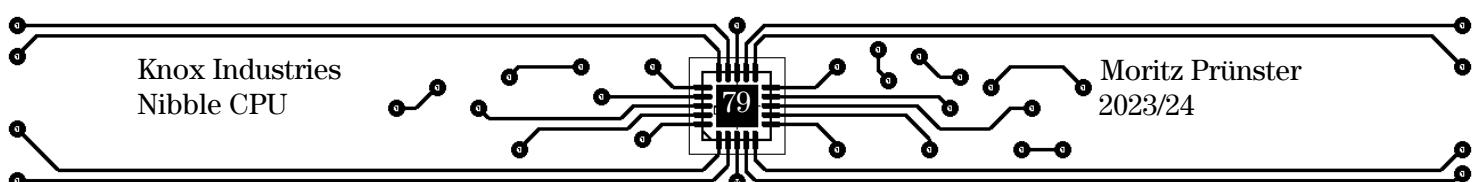
Volle Bezeichnung:	Add Number
Opcode:	0100 XXXX
Funktion:	Addiert den Inhalt des ersten Akku mit der angegebenen Nummer und speichert das Ergebnis im zweiten Akku.
Ausführungsort:	ALU

12.6.7 ADA

Volle Bezeichnung:	Add Address
Opcode:	0101 XXXX
Funktion:	Addiert den Inhalt des ersten Akku mit dem Inhalt der angegebenen Adresse und speichert das Ergebnis im zweiten Akku.
Ausführungsort:	ALU, Register

12.6.8 SUN

Volle Bezeichnung:	Subtract Number
Opcode:	0110 XXXX
Funktion:	Subtrahiert den Inhalt des ersten Akku mit der angegebenen Nummer und speichert das Ergebnis im zweiten Akku.
Ausführungsort:	ALU



12.6.9 SUA

Volle Bezeichnung:	Subtract Address
Opcode:	0111 XXXX
Funktion:	Subtrahiert den Inhalt des ersten Akku mit dem Inhalt der angegebenen Adresse und speichert das Ergebnis im zweiten Akku.
Ausführungsort:	ALU, Register

12.6.10 MA1

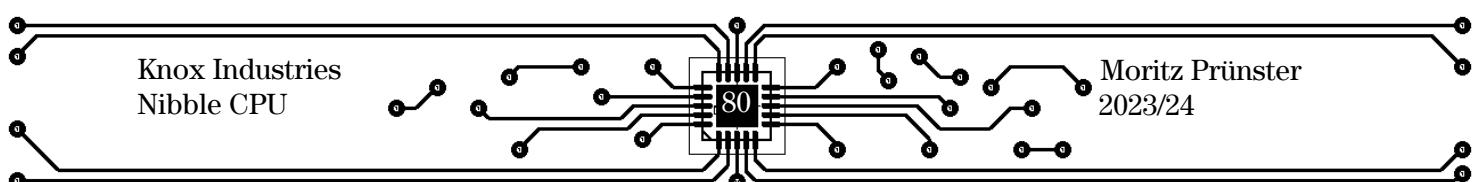
Volle Bezeichnung:	Multiply Address part 1
Opcode:	1000 XXXX
Funktion:	Multipliziert den Inhalt des ersten Akku mit dem Inhalt der angegebenen Adresse und speichert den ersten Teil des Ergebnis im zweiten Akku.
Ausführungsort:	ALU, Register

12.6.11 MA2

Volle Bezeichnung:	Multiply Address part 2
Opcode:	1001 XXXX
Funktion:	Multipliziert den Inhalt des ersten Akku mit dem Inhalt der angegebenen Adresse und speichert den zweiten Teil des Ergebnis im zweiten Akku.
Ausführungsort:	ALU, Register

12.6.12 MN1

Volle Bezeichnung:	Multiply Number part 1
Opcode:	1010 XXXX
Funktion:	Multipliziert den Inhalt des ersten Akku mit der angegebenen Nummer und speichert den ersten Teil des Ergebnis im zweiten Akku.
Ausführungsort:	ALU



12.6.13 MN2

Volle bezeichnung:	Multiply Number part 2
Opcode:	1011 XXXX
Funktion:	Multipliziert den Inhalt des ersten Akku mit der angegebenen Nummer und speichert den zweiten Teil des Ergebnis im zweiten Akku.
Ausführungsort:	ALU

12.6.14 CCF

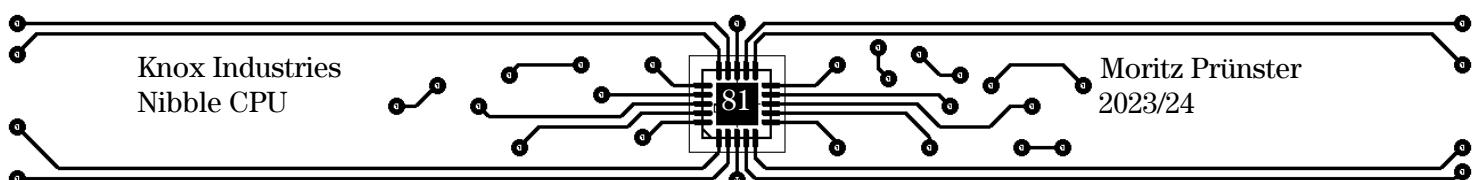
Volle bezeichnung:	Clear Carry Flag
Opcode:	1100 0000
Funktion:	Setzt das Carry Flag zurück.
Ausführungsort:	ALU

12.6.15 ASH

Volle bezeichnung:	Akkumulator Shift
Opcode:	1101 0000
Funktion:	Die Daten werden vom zweiten Akku in den ersten geladen und gespeichert.
Ausführungsort:	ALU

12.6.16 ITS

Volle bezeichnung:	If it's the same
Opcode:	1110 XXXX
Funktion:	Überprüft, ob die Daten aus dem ersten Akku die gleichen sind wie die angegebene Nummer. Wenn es übereinstimmt, wird das Same Flag auf HIGH gesetzt.
Ausführungsort:	ALU



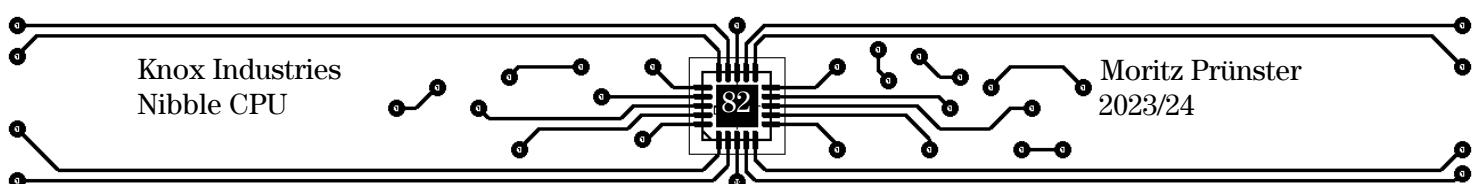
12.6.17 CSF

Volle Bezeichnung:	Clear Same Flag
Opcode:	1111 0000
Funktion:	Setzt das Same Flag zurück.
Ausführungsart:	ALU

13. NALO

"Nalo" steht für Nibble Assembly Language Operator. Dies ist das Programm, das man verwendet, um Programme auf die CPU hinzuzuladen. Das Programm ist in C++ geschrieben und funktioniert nur für Linux-basierte Computer. Es kann 13 unterschiedliche Fehlermeldungen ausgeben, um das Programmieren zu vereinfachen.

Das Programm selbst funktioniert wie folgt: Zunächst öffnet es die angegebene Datei und liest sie einmal von oben nach unten Zeile für Zeile durch. Dabei werden alle Zeilen gespeichert, die länger als drei Zeichen sind, da es sich dabei um Marker handeln könnte. Nachdem es einmal durchgelaufen ist, durchläuft es ein zweites Mal den Code. Diesmal werden alle Befehle beachtet und in die entsprechenden Opcodes umgewandelt. Anschließend wird das assemblierte Programm auf die CPU hochgeladen.



14. Kostenberechnung

In diesem Kapitel werden die Gesamtkosten des Prototypen und der Serienproduktion berechnet.

14.1 Materialkosten

Steuereinheit Materialkosten

Material	Kosten
Platine	113,23€
Mouser	53,07€
Mitbestellen bei Vladislav	50,00€
Summe	216,30€

ALU und Register Materialkosten

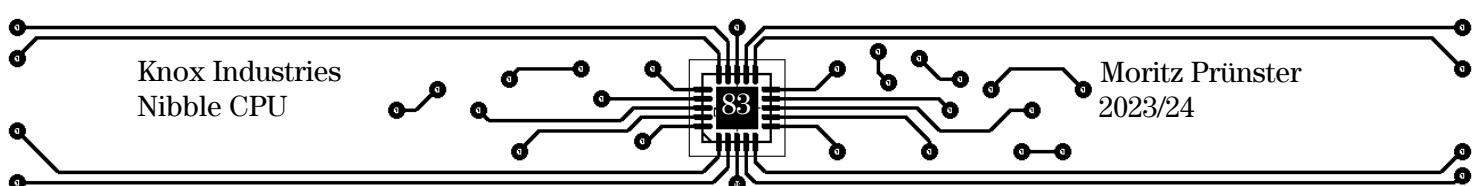
Material	Kosten
Platinen	220,41€
Mouser	16,80€
Farnell	95,70€
Summe	332,91€

Sonstige Materialkosten

Material	Kosten
Blechspiegel	16,95€
Holz	15,45€
Plexiglas groß	17,90€
Plexiglas klein	16,95€
Plexiglas klein	16,95€
Bodenstopper	3,95€
Summe	88,15€

Gesamte Materialkosten

Material	Kosten
ALU und Register	332,91€
Steuereinheit	216,30€
Sonstige	88,15€
Summe	637,36€

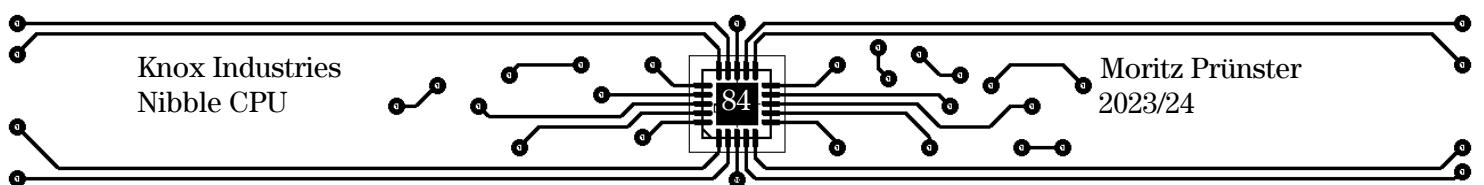


14.2 Beraterstunden kosten

Person	Stunden
Lorenzo di Cello	2.25
Martin de Tomaso	9.5
Vladislav Yegerov	3
Ivan Huber	2
Summe	16.75

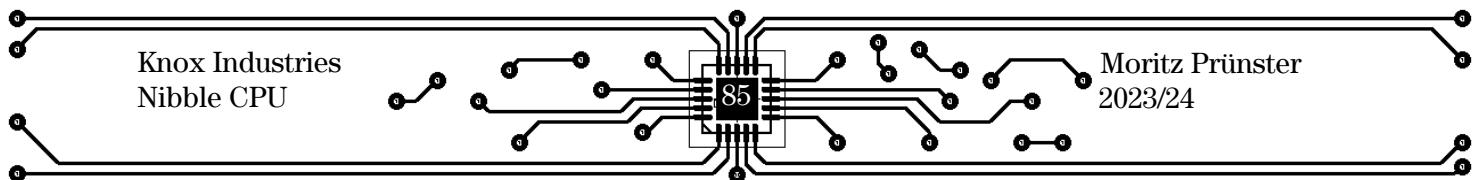
Die Beraterstunden werden ungefähr 100€ pro Stunde kosten, weshalb man nun die 16,75 Stunden mit 100 multiplizieren kann.

$$100 \times 16,75 = 1675\text{€}$$
 für Beraterstunden.



14.3 Arbeitsstunden

Projektwoche	Schule(h)	offenes Labor(h)	Zuhause(h)
PW1	7	0	1
PW2	7	0	5
PW3	7	0	10
PW4	7	0	14
PW5	7	0	5
PW6	7	0	12
PW7	0	0	35
PW8	7	0	6
PW9	7	0	7
PW10	7	0	8
PW11	7	0	19
PW12	7	0	14
PW13	7	0	7
PW14	7	0	10
PW15	0	0	40
PW16	0	0	23
PW17	3	0	8
PW18	5	0	10
PW19	7	0	7
PW20	7	0	5
PW21	7	0	8
PW22	7	0	0
PW23	7	0	12



PW24	7	0	10
PW25	7	0	4
PW26	7	0	12
PW27	7	0	12
PW28	7	0	10
PW29	2	0	20
PW30	5	0	4
Summe	176	0	338

Insgesamt habe ich 514 Stunden im Schuljahr 2023/24 in das Projekt investiert. Wenn man dies mit 20€ pro Stunde multipliziert, ergibt das 10.280€ für das Jahr 2023/24.

Für das Jahr 2022/23 habe ich leider keine Daten darüber, wie lange ich insgesamt an dem Projekt gearbeitet habe. Deshalb nehme ich nur die Stunden, die ich zuhause gearbeitet habe, also 338 Stunden, was 6.760€ bedeutet.

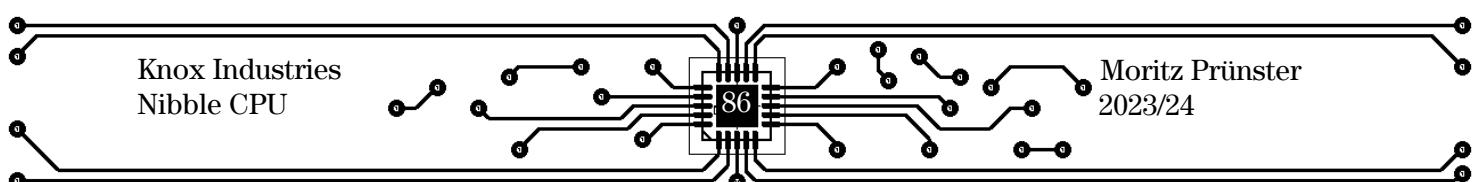
Wenn man diese beiden Beträge addiert, ergibt das:

$$10.280\text{€} + 6.760\text{€} = 17.040\text{€}.$$

Insgesamt habe ich also 17.040€ in das Projekt investiert.

14.4 Prototype Kosten

Materialkosten	637,36€
Beraterstunden	1.675€
Arbeitsstunden	17.040€
Summe	19.352,36€



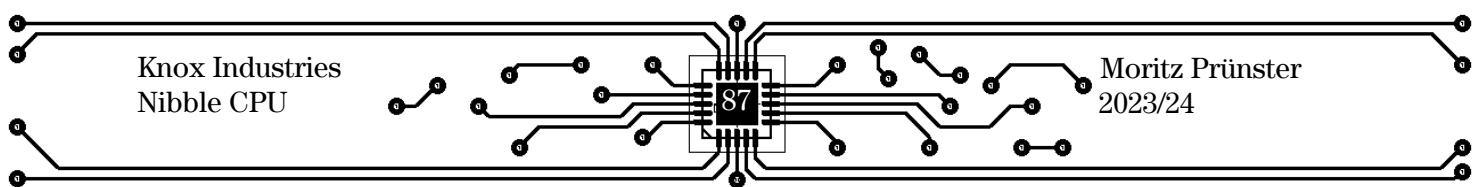
14.5 Serien Produktionskosten

Um zu sehen, wie viel ein Produkt in der Serienproduktion kosten würde, werden hier die Serienproduktionskosten berechnet, um einen angemessenen Preis zu ermitteln. Dabei werden die Bauteile bei [mouser.it](#) und die Platinen bei [jlpcb.com](#) bestellt.

14.5.1 Bauteile

In der unten dargestellten Tabelle werden alle Bauteile aufgelistet, die in einer Nibble CPU verbaut sind. Zusätzlich sind der Name, die Art des Bauteils und die Anzahl der Bauteile, die in jedem Teil der CPU verbaut sind, angegeben.

Bauteilname	Bauteilart	Steuereinheit	ALU	Register	Summe
M48Z08	RAM Speicher	1	0	0	1
FT232RNL-REEL	USB leser	1	0	0	1
HEF4027B-Q100J	JK-Flipflop	27	5	64	96
MC14081BDG	AND 2 Input	21	25	50	96
SN74LS21DR	AND 4 Input	4	8	8	20
CD4071BNSR	OR 2 Input	7	12	0	19
CD4072BM	OR 4 Input	0	2	20	22
HEF4070BT	XOR	1	13	0	14
NVL14049BDR2G	NOT	8	3	24	35
SN74LVC1G126DBVR	Tri-state Buffer	24	0	0	24
C0603X104K1RACTU	Stützkondensator	93	69	174	336
1734035-3	USB-port	1	0	0	1
HVR3700001003JR500	Widerstände	2	0	0	2
PTS645SM502LFS	Knopf	1	0	0	1
BLM18KG260TN1D	Ferrit	1	0	0	1



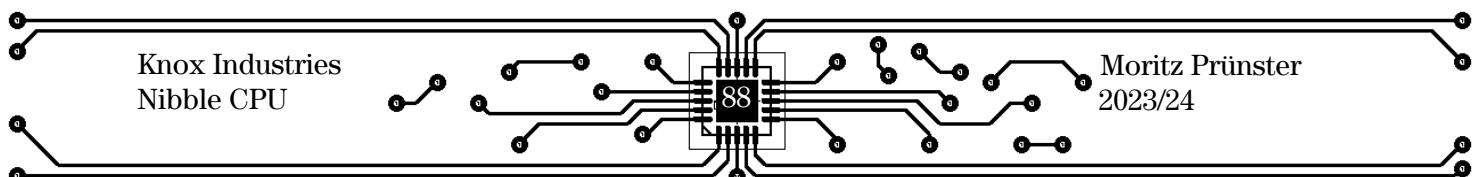
14.5.2 Zusammenrechnung

Nun nehmen wir die Kosten bei 1000 Bauteilen und die Kosten bei 1000 Platinen und berechnen diese zusammen. Dabei werden alle Bauteile bei [mouser.it](#) bestellt.

Teile	Anzahl bei 1000 Stück	Preis von 1000 Stück
RAM Speicher	1000	16.510,00 €
USB leser	1000	2.740,00 €
JK-Flipflop	96000	15.360,00 €
AND 2 Input	96000	19.680,00 €
AND 4 Input	20000	4.780,00 €
OR 2 Input	19000	4.636,00 €
OR 4 Input	22000	10.780,00 €
XOR	14000	1.792,00 €
NOT	35000	7.595,00 €
Tri-state Buffer	24000	936,00 €
Stützkondensator	336000	53.424,00 €
USB-port	1000	728,00 €
Widerstände	2000	148,00 €
Knopf	1000	1521,00 €
Ferrit	1000	33,00 €
Summe		139.294,00 €

Platinen preis bei 1000 Stück bei [jlcpcb.com](#)

Platine	Preis bei 1000 Stück
Steuereinheit	2.992,00 €
ALU	5.508,00 €
Register	2.770,00 €
Summe	11.270,00 €

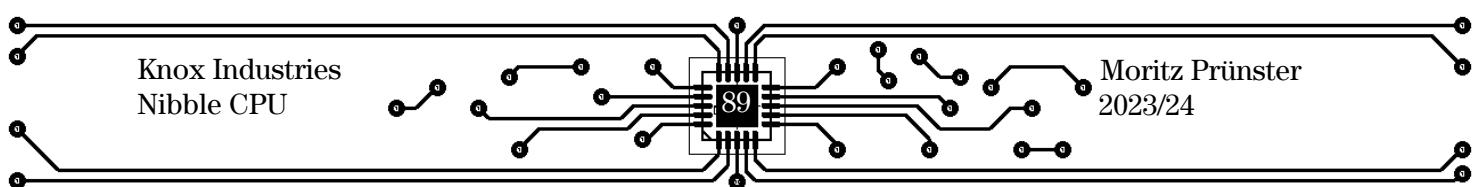


Nun Addieren wir die ganzen bauteile zusammen und rechnen auch noch 40€ hinzu für die Zusammensetzung der Bauteile da dies fast 2 Stunden dauern kann und falls man noch ein gehäuse haben möchte werden 88.15€ hinzu addiert für die Materialkosten vom Gehäuse.

Platinen	11.270,00 €
Bauteile	139.294,00 €
Summe	150.564,00 €
Materialkosten pro CPU	150,57 €

Die Materialkosten für eine Nibble CPU beträgt 150,57 € nun addieren wir die 40€ für die zwei Stunden Arbeit um alles hinauf zu löten und falls mit gehäuse zusätzlich die 88.15 €

Serien Nibble CPU kosten	190,57 €
Serien Nibble CPU kosten mit Gehäuse	278,72 €



15.Anhang

Hier werden zusätzliche Bilder, Dateien usw. angehängt, die nicht in sonstige Kapitel gepasst haben.

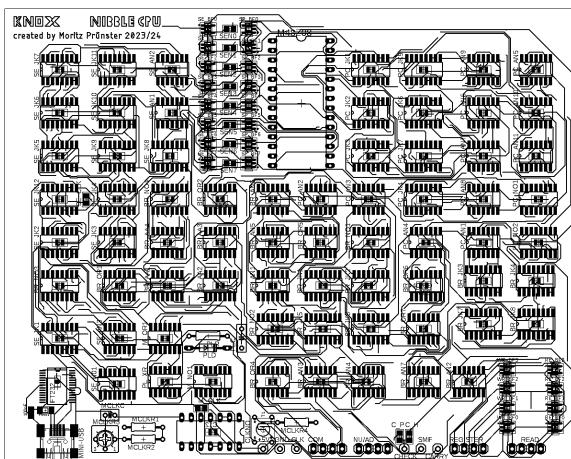


Abb. 83: Platine Steuereinheit Layer 1 Oben

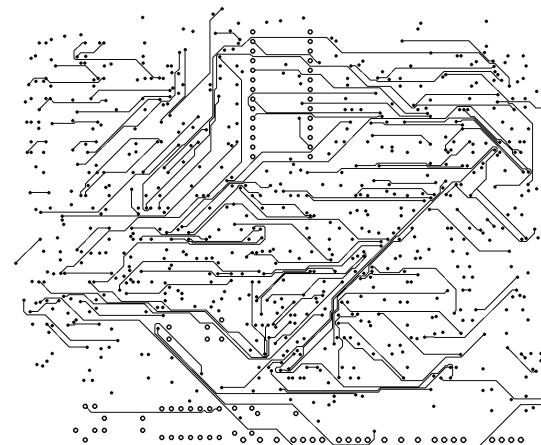


Abb. 84: Platine Steuereinheit Layer 2

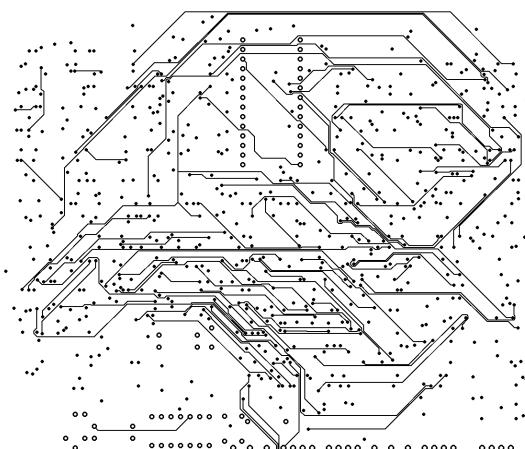


Abb. 85: Platine Steuereinheit Layer 2

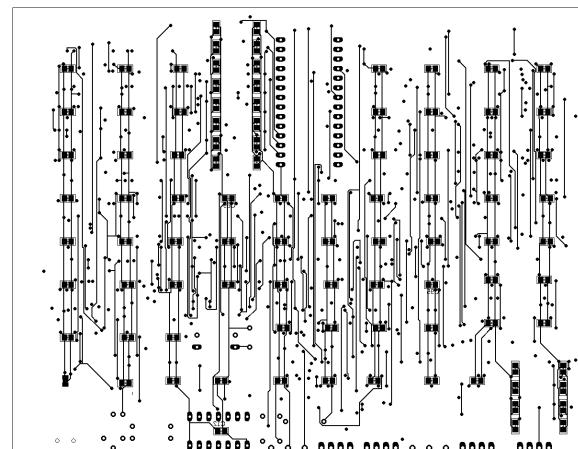
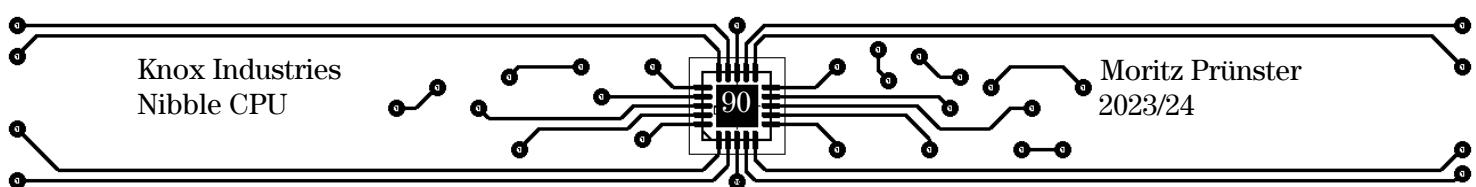


Abb. 86: Platine Steuereinheit Layer 4 Unten



Nibble CPU

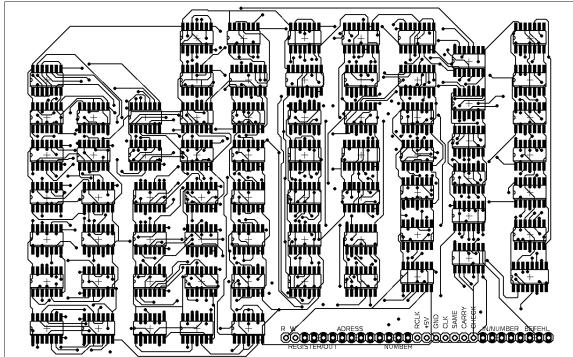


Abb. 87: Platine ALU Layer 1 Oben

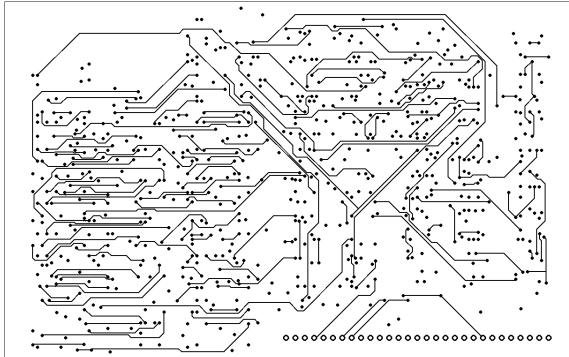


Abb. 88: Platine ALU Layer 2

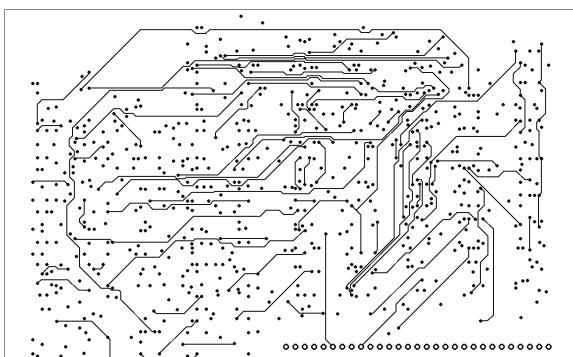


Abb. 89: Platine ALU Layer 3

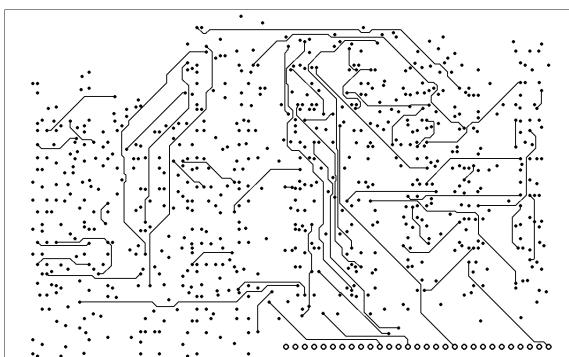


Abb. 90: Platine ALU Layer 4

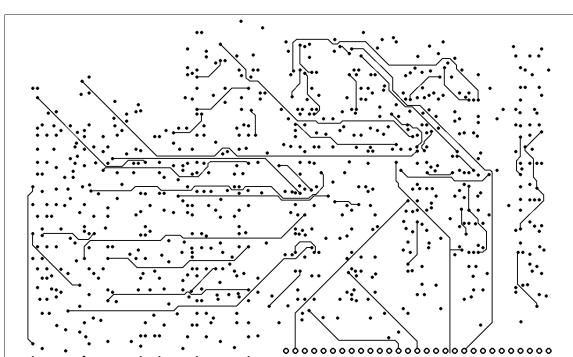


Abb. 91: Platine ALU Layer 5

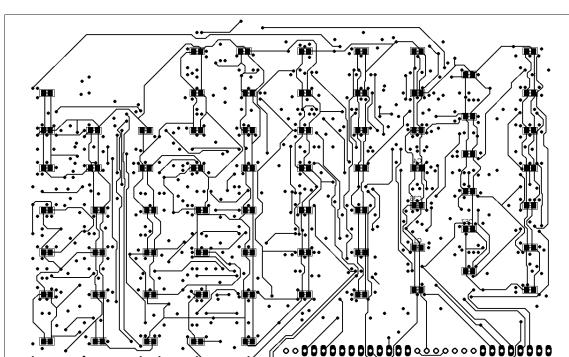
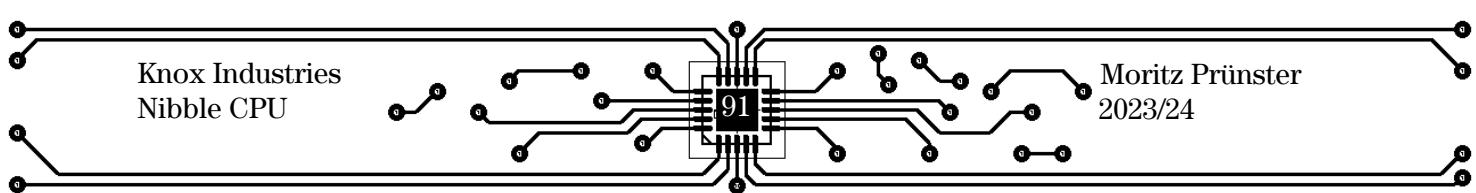


Abb. 92: Platine ALU Layer 6 Unten



Nibble CPU

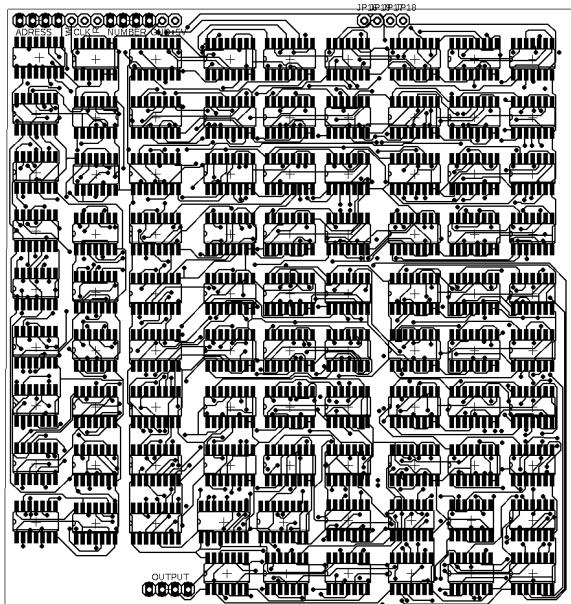


Abb. 93: Platine Register Layer 1 Oben

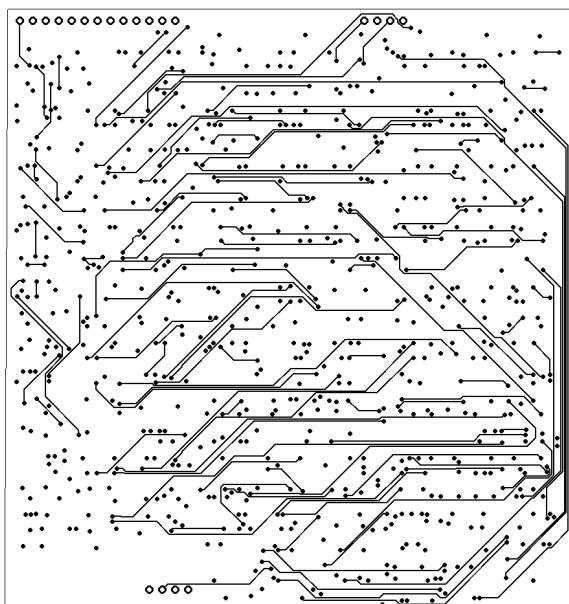


Abb. 94: Platine Register Layer 2

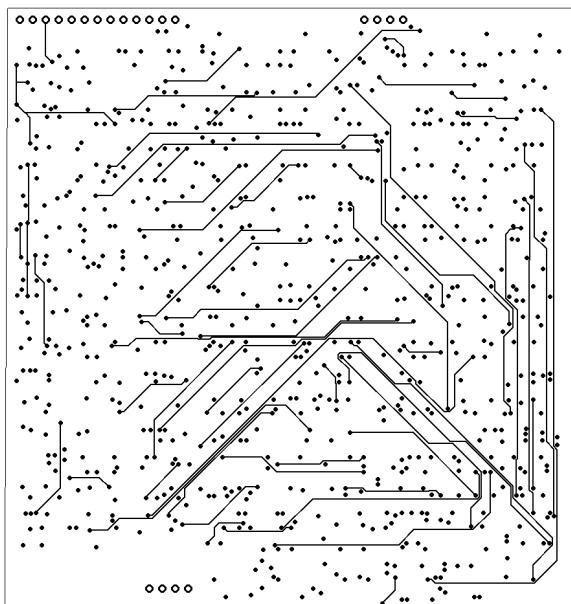


Abb. 95: Platine Register Layer 3

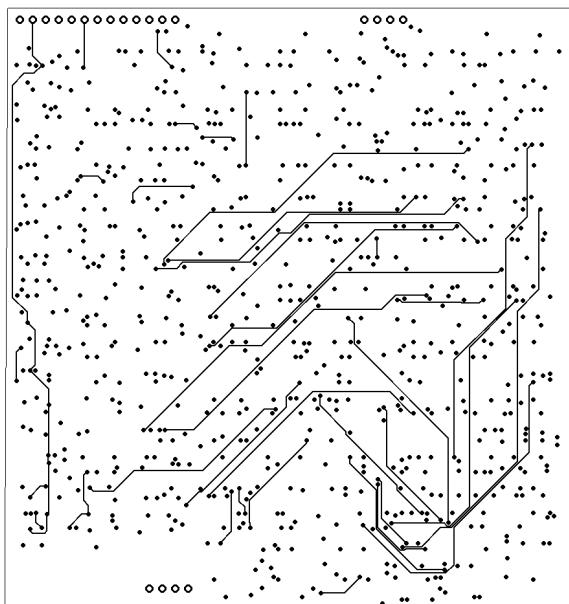
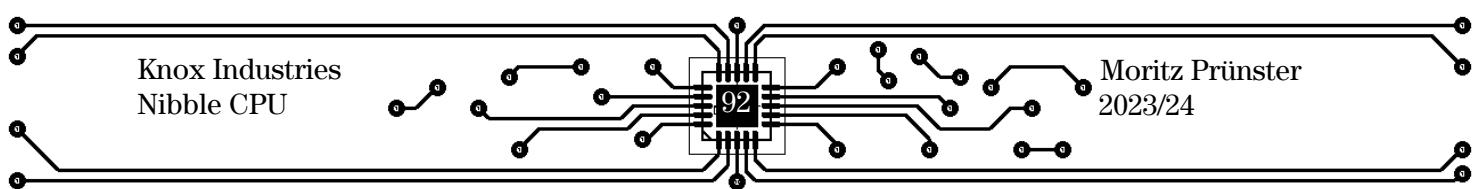


Abb. 96: Platine Register Layer 4



Nibble CPU

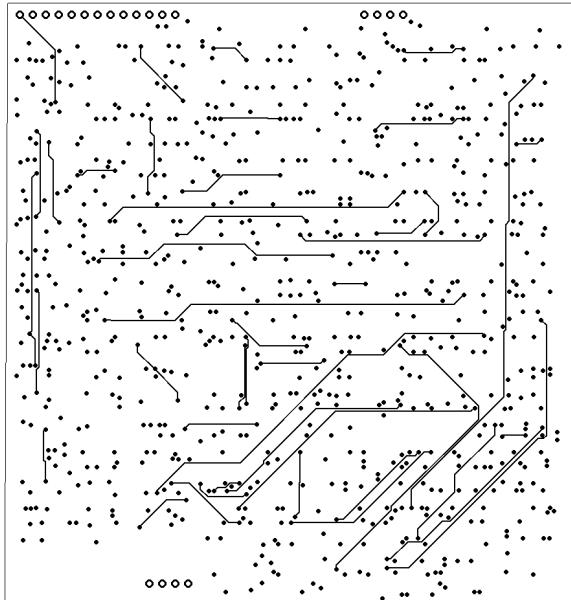
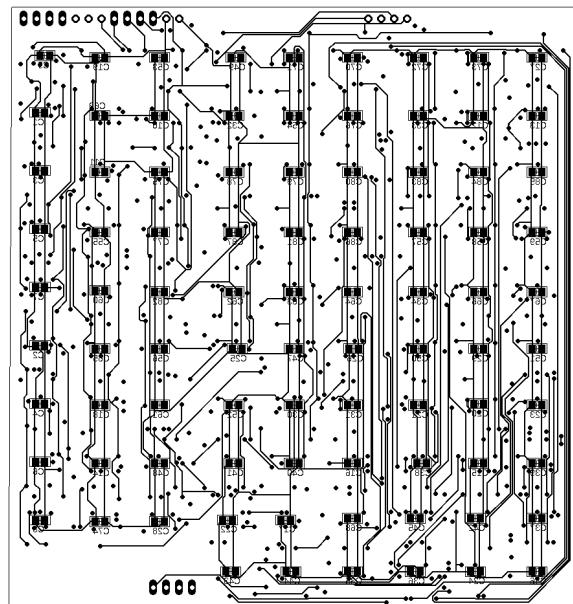
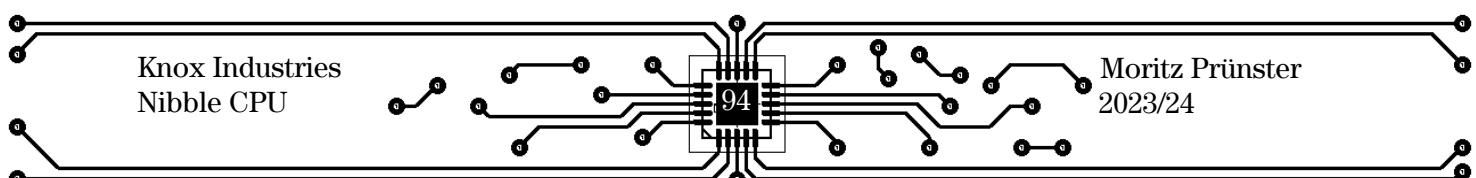


Abb. 97: Platine Register Layer 5



16. Quellenverzeichnis

Abbildung	Dargestellt	Quelle
Abb. 1	Projektantrag scan Vorderseite	Selbst kreiert Datei gescannt mit Microsoft Lens
Abb. 2	Projektantrag scan Rückseite	Selbst kreiert Datei gescannt mit Microsoft Lens
Abb. 3	Pflichtenheft scan	Selbst kreiert Datei gescannt mit Microsoft Lens
Abb. 4	Working Packages Vorderseite.	Selbst kreiert Datei gescannt mit Microsoft Lens
Abb. 5	Working Packages Rückseite.	Selbst kreiert Datei gescannt mit Microsoft Lens
Abb. 6	Die erste Seite des Gantt-Diagramms.	Selbst kreiert Datei gescannt mit Microsoft Lens
Abb. 7	Die zweite Seite des Gantt-Diagramms.	Selbst kreiert Datei gescannt mit Microsoft Lens
Abb. 8	Die dritte Seite des Gantt-Diagramms.	Selbst kreiert Datei gescannt mit Microsoft Lens
Abb. 9	Wochenbericht	Selbst kreiert Datei gescannt mit Microsoft Lens
Abb. 10	Diagramm zur Erklärung wie man Binär Zahlen liest.	https://kunduz.com/blog/binary-to-decimal-conversion-293995/
Abb. 11	Assembler Beispiel mit der DZ60 Assemblersprache.	Screenshot von emacs mit der geöffneten selbst kreierten Datei: ADC.inc
Abb. 12	Die unterschiedlichen Schaltungszeichen von einem NOT-Gatter.	Ausschnitt von einem Google Bild https://retrofixer.it/en/porte-logiche/
Abb. 13	Die unterschiedlichen Schaltungszeichen von einem OR-Gatter.	Gleches Bild, nur ein anderer Ausschnitt wie bei Abb 12.
Abb. 14	Die unterschiedlichen Schaltungszeichen von einem AND-Gatter.	Gleches Bild, nur ein anderer Ausschnitt wie bei Abb 12.
Abb. 15	Die unterschiedlichen	Gleches Bild, nur ein anderer



	Schaltungszeichen von einem XOR-Gatter.	Ausschnitt wie bei Abb 12.
Abb. 16	Schaltungs innere von einem JK-Flipflop.	https://www.electronics-tutorials.ws/d/e/sequentielle/jk-flipflop.html
Abb. 17	Schaltungszeichen von einem JK-Flipflop.	Eagle Library 40xx, Bauteil 4027.
Abb. 18	Schaltungszeichen von einem Tri-State Buffer.	Eagle Library SN74LVC1G126DBVR selbst hinzugefügt.
Abb. 19	Das Bauteil FT232L Pin Layout.	Aus dem Datenblatt: https://html.alldatasheet.com/html-pdf/144591/FTDI/FT232RL/708/7/FT232RL.html
Abb. 20	Schaltzeichen eines FT232L	Eagle Library deto2, selbst hinzugefügt
Abb. 21	M48Z08 Logik Diagramm	Aus dem Datenblatt: https://www.mouser.it/datasheet/2/389/m48z08-1849966.pdf
Abb. 22	M48Z08 Pin Layout	Aus dem gleichen Datenblatt wie Abb 21.
Abb. 23	Kleines vollständiges Beispielprogramm mit der Nibble Assambly Language(NAL) geschrieben.	Screenshot aus Neovim mit der geöffneten Datei example2.nal
Abb. 24	Diagramm zur Darstellung der Datenverteilung über die 3 Teile	Selbst mit dem Programm Drawio gezeichnet.
Abb. 25	Diagramm zur Darstellung der Datenverteilung in der Steuereinheit	Selbst mit dem Programm Drawio gezeichnet.
Abb. 26	Pinboard Schaltung	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 27	Not-Schmitt Trigger Clock	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 28	Ne555 Clock	https://www.electronics-tutorials.ws/waveforms/555-circuits-part-1.html
Abb. 29	Read Schaltung	Exportet image aus Eagle von der Steuereinheit Schematic.

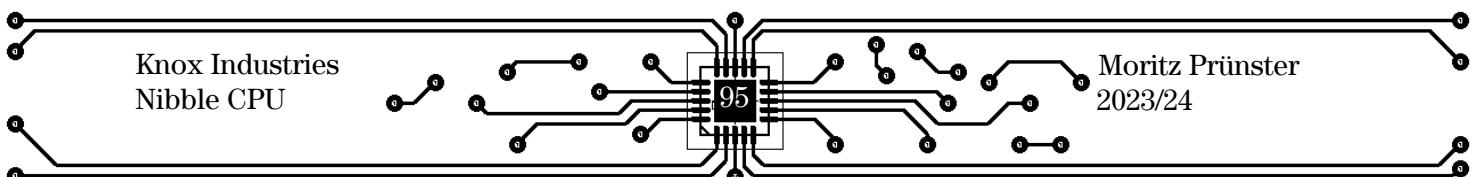


Abb. 30	Schaltung vom Pulsgenerator.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 31	Schaltung vom USB-Port.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 32	Schaltung von der PC-Schleuse.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 33	Programmcounter Schaltung.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 34	PC Clock Schaltung.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 35	Schaltung vom PC Counter.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 36	Schaltung vom PC SET.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 37	Schaltung der Seriellen	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 38	Schaltung mit dem FT232L	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 39	Schaltung vom seriellen Timer.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 40	Schaltung der SE Clock.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 41	Schaltung der Auslese Strecke.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 42	Schaltung des Seriellen Reset.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 43	Schaltung vom seriellen Kurzspeicher.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 44	Schaltung vom seriellen Ausgang.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 45	Schaltung von der seriellen Kontrolle.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 46	Schaltung vom Branch	Exportet image aus Eagle von der Steuereinheit Schematic.

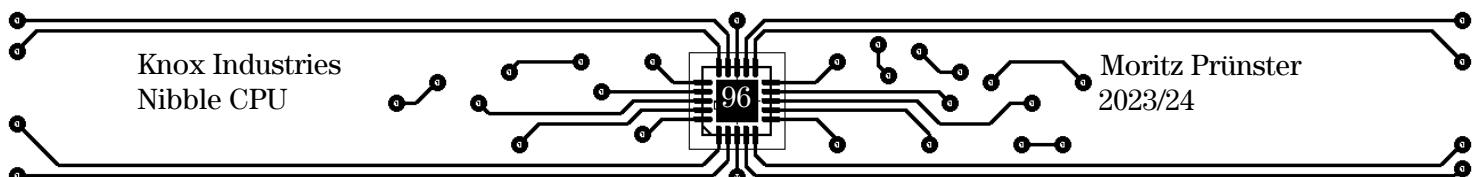


Abb. 47	Schaltung für Branch-Befehlserkennung.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 48	Schaltung für Nullerkennung.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 49	Schaltung der Branch Datenschleuse.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 50	Schaltung von der Branch Schleuse.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 51	Schaltung vom Branch Zusammenfasser.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 52	Schaltung vom Branch counter.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 53	Schaltung vom Branch Kurzspeicher.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 54	Schaltung vom Steuereinheit Speicher	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 55	Schaltung mit dem M48Z08.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 56	Schaltung vom Speicher-Leser.	Exportet image aus Eagle von der Steuereinheit Schematic.
Abb. 57	Diagramm zur Darstellung der Datenverteilung in der ALU	Selbst mit dem Programm Drawio gezeichnet.
Abb. 58	Schaltung vom ALU Pinboard	Exportet image aus Eagle von der Steuereinheit ALU.
Abb. 59	Schaltung zur Erkennung von Nummern und Adressen.	Exportet image aus Eagle von der Steuereinheit ALU.
Abb. 60	Schaltung zur Abspeicherung.	Exportet image aus Eagle von der Steuereinheit ALU.
Abb. 61	Schaltung zum Datenzusammenfassung.	Exportet image aus Eagle von der Steuereinheit ALU.
Abb. 62	Befehls Schleuse.	Exportet image aus Eagle von der Steuereinheit ALU.
Abb. 63	Schaltung füe Akku Befehle	Exportet image aus Eagle von der Steuereinheit ALU.

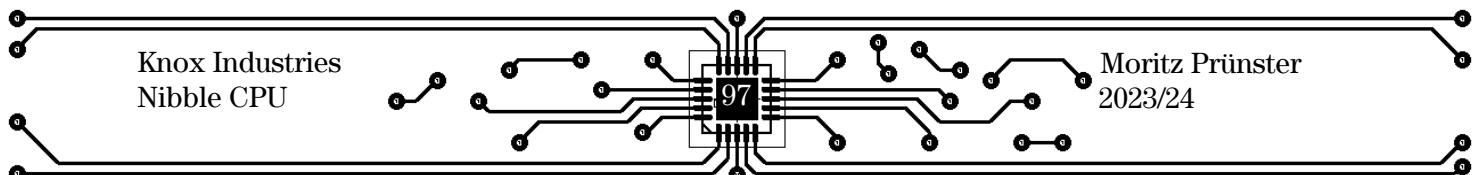


Abb. 64	Schaltung für ALU-Befehlserkennung.	Exportet image aus Eagle von der Steuereinheit ALU.
Abb. 65	Schaltung von zwei Akkumulatoren.	Exportet image aus Eagle von der Steuereinheit ALU.
Abb. 66	Schaltung zum Ergebnis speichern.	Exportet image aus Eagle von der Steuereinheit ALU.
Abb. 67	Schaltung von einem Volladdierer.	Exportet image aus Eagle von der Steuereinheit ALU.
Abb. 68	Schaltung von einem 4 bit Addierer.	Exportet image aus Eagle von der Steuereinheit ALU.
Abb. 69	Schaltung von einem Subtrahierer	Exportet image aus Eagle von der Steuereinheit ALU.
Abb. 70	Schaltung von einem Multiplizierer.	Exportet image aus Eagle von der Steuereinheit ALU.
Abb. 71	Schaltung von einem Vergleicher	Exportet image aus Eagle von der Steuereinheit ALU.
Abb. 72	Diagramm zur Darstellung der Datenverteilung in des Registers	Selbst mit dem Programm Drawio gezeichnet.
Abb. 73	Schaltung von Adressen auswerter.	Exportet image aus Eagle von der Steuereinheit Register.
Abb. 74	Schaltung von einer Speicherzelle.	Exportet image aus Eagle von der Steuereinheit Register.
Abb. 75	Schaltung vom kompletten Speicher.	Exportet image aus Eagle von der Steuereinheit Register.
Abb. 76	Schaltung vom Zusammenfasser.	Exportet image aus Eagle von der Steuereinheit Register.
Abb. 77	Platine Steuereinheit vorne.	Foto mit dem Smartphone.
Abb. 78	Platine Steuereinheit hinten.	Foto mit dem Smartphone.
Abb. 79	Platine ALU vorne.	Foto mit dem Smartphone.
Abb. 80	Platine ALU hinten.	Foto mit dem Smartphone.
Abb. 81	Platine Register vorne.	Foto mit dem Smartphone.
Abb. 82	Platine Register hinten.	Foto mit dem Smartphone.
Abb. 83	Platine Steuereinheit Layer 1	Eagle Board Image Layer 1.

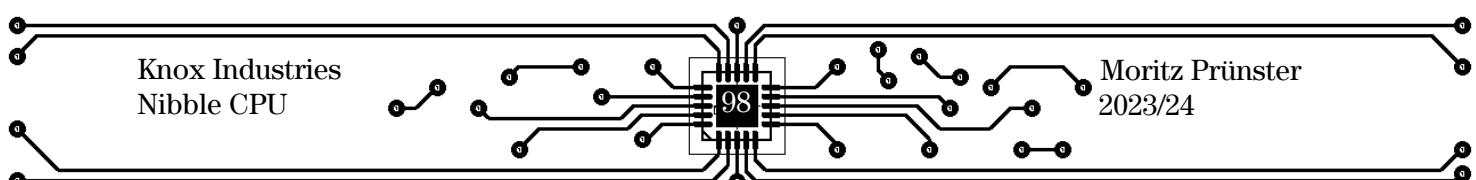


Abb. 84	Platine Steuereinheit Layer 2	Eagle Board Image Layer 2.
Abb. 85	Platine Steuereinheit Layer 3	Eagle Board Image Layer 3.
Abb. 86	Platine Steuereinheit Layer 4	Eagle Board Image Layer 4.
Abb. 87	Platine ALU Layer 1	Eagle Board Image Layer 1.
Abb. 88	Platine ALU Layer 2	Eagle Board Image Layer 2.
Abb. 89	Platine ALU Layer 3	Eagle Board Image Layer 3.
Abb. 90	Platine ALU Layer 4	Eagle Board Image Layer 4.
Abb. 91	Platine ALU Layer 5	Eagle Board Image Layer 5.
Abb. 92	Platine ALU Layer 6	Eagle Board Image Layer 6.
Abb. 93	Platine Register Layer 1	Eagle Board Image Layer 1.
Abb. 94	Platine Register Layer 2	Eagle Board Image Layer 2.
Abb. 95	Platine Register Layer 3	Eagle Board Image Layer 3.
Abb. 96	Platine Register Layer 4	Eagle Board Image Layer 4.
Abb. 97	Platine Register Layer 5	Eagle Board Image Layer 5.
Abb. 98	Platine Register Layer 6	Eagle Board Image Layer 6.
Abb. 99	Komplette Nibble CPU mit Deckel	Foto mit der Smartphone Kamera.
Abb. 100	Komplette Nibble CPU ohne Deckel	Foto mit der Smartphone Kamera.

