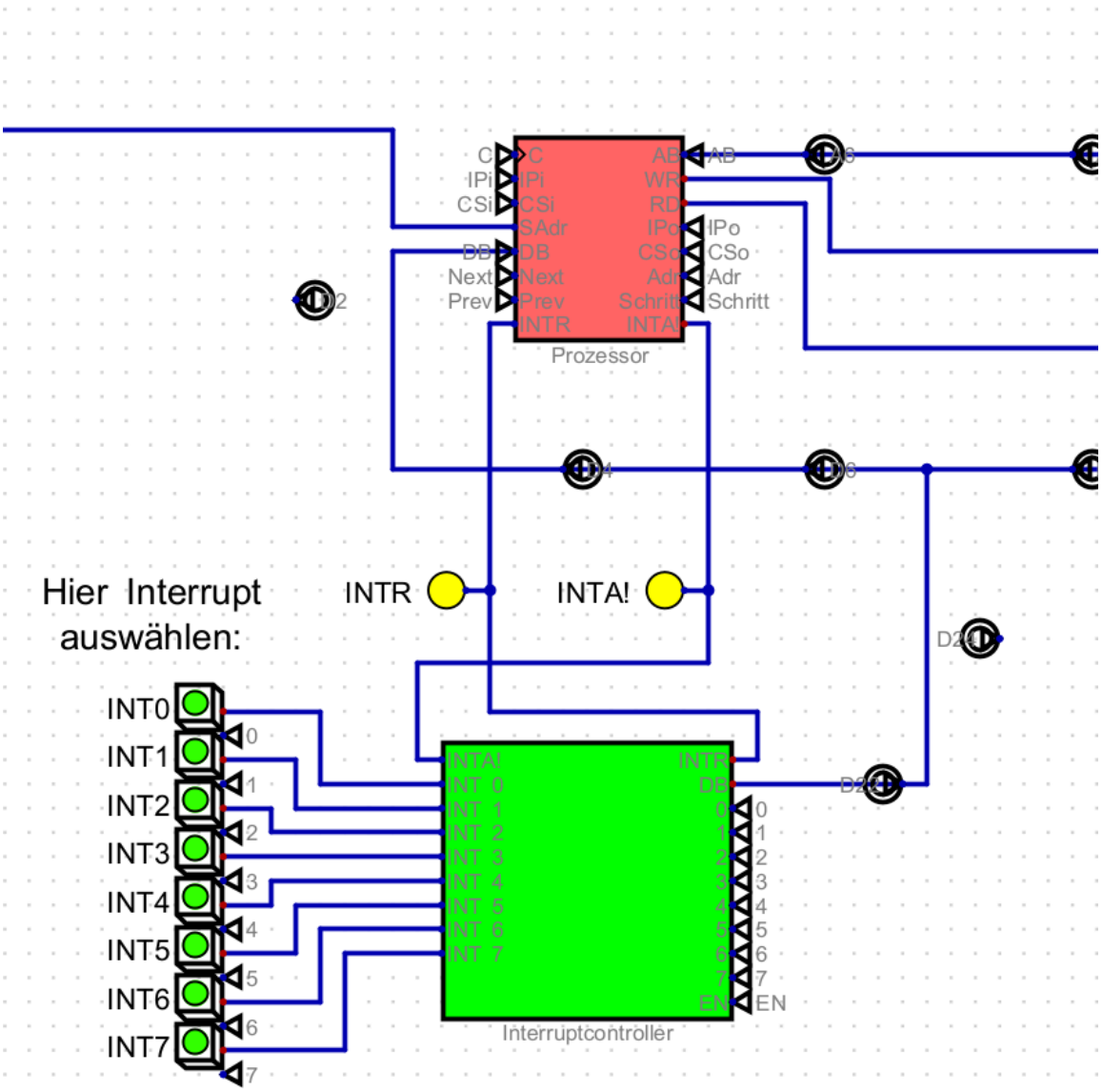
# Simulation

# Interruptverarbeitung

# Intel 8086



**Modul T3_3100**

**Studienarbeit 5. Theoriesemester**

des Studiengangs Elektrotechnik

mit der Studienrichtung Automation

an der Dualen Hochschule Baden-Württemberg (DHBW) Mannheim

von

Daniel Hermes

Abgabedatum: 25.03.2022

| | |
|---|---|
| Bearbeitungszeitraum | 17.01.2022 - 25.03.2022 |
| Matrikelnummer | 6470600 |
| Kurs | TEL19AT2 |
| Ausbildungsfirma | Tesla Automation, Prüm |
| Betreuer | Dipl.-Ing. Gerhard Lehmann, DHBW |

Unterschrift Betreuer: _____

# Selbstständigkeitserklärung

Ich versichere hiermit, dass ich meine Studienarbeit mit dem Thema: **Simulation Interruptverarbeitung Intel 8086** selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

| Mannheim | 25.03.2022 | |
|---|---|---|
| Ort | Datum | Unterschrift |

# Inhaltsverzeichnis

# Abkürzungsverzeichnis

Die wichtigsten Abkürzungen und ihre Bedeutung sind im Folgenden zusammengestellt. Die hier aufgelisteten Abkürzungen sind nach ihrer erstmaligen Verwendung in dieser Arbeit sortiert.

| **Symbol** | **Bedeutung** |
|---|---|
| ISR | **I**nterrupt-**S**ervice-**R**outine: Instruktionen, die als Reaktion auf einen Interrupt ausgeführt werden |
| INTA! | **Int**errupt-**A**cknowledge-Leitung: Wird vom Prozessor zum Quittieren einer Interrupt-Anfrage verwendet. Das "**!**" zeigt, dass die Leitung nullaktiv ist, dass also ein LOW-Pegel einen Interrupt quittiert, während die Leitung im Ruhezustand HIGH ist |
| IP | **I**nstruction **P**ointer: Adresse der nächsten Instruktion, die vom Prozessor bearbeitet wird |
| SS | **S**tack-**S**egment: speichert das RAM-Segment, das den Call-Stack des aktuell ausgeführten Programms enthält |
| SP | **S**tack-**P**ointer: Haupt-Stack-Register des 8086 |
| CS | **C**ode **S**egment: Segment, in dem Instruktionen stehen |
| RAM | **R**andom-**A**ccess-**M**emory: Arbeitsspeicher |
| IVT | **I**nterrupt-**V**ektor-**T**abelle: Teil des RAMs, in dem die Adressen der ISRs stehen |
| JRE | **J**ava **R**untime **E**nvironment: Java-Laufzeitumgebung |

# Abbildungsverzeichnis

# Tabellenverzeichnis



Sämtliche Abbildungen und Tabellen in dieser Arbeit wurden, sofern nicht anders angegeben, selbst erstellt. Das schließt das Titelbild mit ein, welches die weiterentwickelte Simulation der Interruptverarbeitung eines Intel 8086 in dem Schaltungssimulator Digital zeigt.

Die Darstellung von Quellcode Ausschnitten in dieser Arbeit wurde mithilfe der Google Docs Erweiterung “Code Blocks v1.4.2” realisiert. Dabei wurde als Theme “vs” (Visual Studio) gewählt. Für die Einstellung Language wurde “cs” (C Sharp, auch bekannt als C#) gewählt. Eine Referenz zu der Code Blocks Erweiterung findet sich im Literaturverzeichnis

# Vorwort

Diese Studienarbeit entstand im 5. Theoriesemester meines Studiums zum Bachelor of Engineering (B.Eng.) im Studiengang Elektrotechnik mit der Studienfachrichtung Automation an der Dualen Hochschule Baden-Württemberg (DHBW) Mannheim.

Ich persönlich kannte die Funktionsweise des Intel 8086 Chips, seiner Speichersegmentierung, I/O und Interruptverarbeitung vor der Bearbeitung dieser Studienarbeit nicht. Dieses Wissen musste ich mir mit Hilfe der bestehenden Simulation, der dazugehörigen Anleitung sowie der Studienarbeit von Silas Gerhard [Gerhard, 2021] und sonstigen Materialien, wie Intels Handbuch zur Funktionsweise des 8086 [Intel Corporation, October 1979], aneignen. Diese Ausgangssituation ermöglichte mir eine neutrale Betrachtung der Simulation sowie der Schwierigkeiten, die ich an einigen Stellen mit der Simulation hatte. Aus diesen Schwierigkeiten konnten direkt Verbesserungswünsche formuliert werden, die zu den in dieser Arbeit beschriebenen Optimierungen geführt haben.

Bei der Bearbeitung dieser Arbeit half mir mein bereits bestehendes Interesse für die Funktionsweise moderner Mikrochips in die Einarbeitung in ein “Urgestein” wie den Intel 8086. Auch bereits bestehende Erfahrung mit der verwendeten Simulationssoftware Digital, die ich im Rahmen meines Studiums an der DHBW sammeln konnte, erwiesen sich als hilfreich. An dieser Stelle sei Herr Dr. Ing. Bernhard Spitzer hervorgehoben, dessen Unterstützung maßgeblich zur Entwicklung des Clock-Dividers in Kapitel *ISR* beigetragen hat.

Die schnellen Antworten des Digital-Entwickler Helmut Neemann zu spezifischen Fragen der Funktionsweise seiner Schaltungssimulations-Software waren bei der Bearbeitung dieser Arbeit äußerst hilfreich und haben auch dabei geholfen, die Digital-Software selbst zu verbessern [Hermes & Neemann, 2022].

Ich möchte mich an dieser Stelle bei meinem Betreuer, Herrn Dipl.-Ing. Gerhard Lehmann, für die angenehme Betreuung bedanken. Ein besonderes Dankeschön gilt meiner Freundin Svenia Ratheiser und meinen Eltern Anna und Tom Hermes für das Korrekturlesen dieser Arbeit und für die inhaltlichen Anmerkungen.

# Kurzfassung

Diese Studienarbeit befasst sich mit der Optimierung einer bestehenden Simulation der Interruptverarbeitung des Intel 8086 Chips. Die bestehende Simulation ist eine Entwicklung im Rahmen einer Studienarbeit aus dem Jahr 2021 [Gerhard, 2021] mit dem Ziel, die Interrupt-Verarbeitung des Intel 8086 zu Lehrzwecken zu simulieren und zu visualisieren. Die Funktionsweise der bestehenden Simulation ist unabdinglich für das Verständnis der Interruptverarbeitung des Intel 8086 sowie den in dieser Arbeit erarbeiteten Optimierungen der bestehenden Simulation. Daher werden die Schritte der bestehenden Simulation im Detail analysiert, um die Optimierungsmöglichkeiten der bestehenden Simulation nachvollziehen zu können. Es werden ggf. Vor- und Nachteile der Optimierungsmöglichkeiten diskutiert und insgesamt vorteilhafte Optimierungen umgesetzt. Dabei wird sowohl der Start und die Installation für den Benutzer vereinfacht wie auch die Simulation selbst um optische Funktionen und Audioelemente ergänzt, die dem Benutzer die Bedienung der Simulation erleichtern und beim Verständnis der Interruptverarbeitung des Intel 8086 helfen. Bestehende und neu hinzukommende Bausteine, die den Ablauf der Simulation steuern, werden kommentiert und an einigen Stellen vereinfacht. Dadurch entsteht Platz für die im Rahmen dieser Arbeit neu hinzugefügten Funktionen und die Schaltungen werden übersichtlich und nachvollziehbar, was künftige Arbeiten an der Simulation einfacher macht. Schließlich wird ein Fazit gezogen und Ansatzpunkte für weitere Forschung diskutiert. Dabei stehen Möglichkeiten zur weiteren Optimierung der Simulation im Fokus, die im Rahmen einer möglichen zukünftigen Studienarbeit realisiert werden können.

# Abstract

This paper discusses improvements made by the author to an existing simulation of the interrupt processing of an Intel 8086 chip. The existing simulation was developed as part of a student research project in 2021 [Gerhard, 2021] with the aim of simulating and visualizing the interrupt processing of the Intel 8086 for teaching purposes. In order to understand the interrupt processing of the Intel 8086 chip it is essential to consider the original functions of the existing simulation before looking at the subsequent optimizations that were carried out for the purpose of this paper. Therefore, the original functions of the existing simulation are analyzed in detail in order to be able to understand where optimization potential lies. Advantages and disadvantages of the optimization options are discussed and overall advantageous optimizations are implemented. The start and the installation of the simulation are simplified for the end user and the simulation itself is supplemented with optical functions and audio elements that make it easier for the user to operate the simulation and help to understand the interrupt processing of the Intel 8086. Existing and newly added modules that control the simulation process are commented on and simplified in some places. This creates space for new functions that were added as part of the work for this paper and the circuits were constructed to be clearer and more comprehensible, which makes future work on the simulation easier. Finally, a conclusion is drawn and starting points for further research are discussed. The focus is on possibilities for further optimization of the simulation, which can be realized as part of a possible future student research project.

# 1. Motivation

Microcomputer umgeben uns. Sie sind heute in jedem elektronischen Gerät verbaut und erleichtern uns Menschen das Leben. Ohne Mikrocomputer kommt die Weltwirtschaft ins Stocken, wie die aktuelle Chipkrise im Zuge der Covid-19 Pandemie eindrucksvoll zeigt. Laut Goldman Sachs sind 169 Industriezweige von dem Mangel an Halbleitern betroffen, inklusive der Herstellung von Toilettenpapier [Dan Howley, Goldman Sachs & Julie Hyman, Yahoo Finance, 2021]. Angesichts dieser Relevanz für unser Leben wird ersichtlich, warum Studenten der Elektrotechnik die Funktionsweise von Mikrocomputern zumindest grob verstehen müssen.

Mikrocomputer arbeiten normalerweise Befehle im Programmcode nacheinander ab, indem sie die folgenden Schritte durchlaufen:

1. Befehl laden (Englisch: Fetch instruction)
   ⇒ z.B. “ADD A,B”
2. Befehlsoperanden laden, wenn nötig (Englisch: Fetch operands)
   ⇒ Für den beispielhaften Befehl “ADD A,B” muss A und B geladen werden. Angenommen A=1, B=2.
3. Befehl ausführen (Englisch: Execute instruction)
   ⇒ 1 + 2 = 3
4. Ergebnis speichern, wenn nötig (Englisch: Store results)
   ⇒ Da kein Speicherziel angegeben wurde, wird das Ergebnis 3 in A gespeichert: A = 3

Dieses strukturierte Abarbeiten von Befehlen muss jedoch unterbrochen werden können, damit der Mikrocomputer z.B. auf Benutzereingaben über die Tastatur reagieren kann. Das Unterbrechen des regulären Programmablaufs zur Reaktion auf bestimmte Umstände wird durch sogenannte Interrupts realisiert. Dabei handelt es sich um ein Signal an den Mikrocomputer, das nach jedem Befehl überprüft wird. Erhält der Mikrocomputer das Interrupt-Signal, unterbricht er den regulären Programmablauf, springt an eine andere Stelle mit Code zur Reaktion auf die Unterbrechung, führt diesen Code aus, und fährt anschließend

mit dem regulären Programmablauf fort. Der Code, der als Reaktion auf einen Interrupt bearbeitet wird, wird als Interrupt Service Routine (kurz: ISR) bezeichnet.

Damit der Mikrocomputer nach der ISR genau dort fortfahren kann, wo er unterbrochen wurde, muss er den Zustand sichern, in dem er zum Zeitpunkt der Unterbrechung war. Das ist vergleichbar mit dem Speichern eines Spielstandes, bevor man sich etwas anderem zuwendet, um später an genau dieser Stelle im Spiel unter den genau gleichen Bedingungen (Ausrüstung, Level etc.) fortfahren zu können.

Nach dem Speichern des Zustands benötigt der Mikrocomputer Informationen über die Art der Unterbrechung, um auf verschiedene Typen von Unterbrechungen stets angemessen reagieren zu können. Dies ist nötig, damit der Mikrocomputer beispielsweise anders auf eine Tastatureingabe reagieren kann, als auf einen Mausklick. Auch hier kann man das Verhalten von Mikrocomputern mit dem Verhalten von Menschen vergleichen, denn Menschen reagieren auch anders, je nachdem ob sie vom Chef oder von Kollegen unterbrochen werden.

Neben den hier anekdotisch beschriebenen Schritten gehören einige weitere Schritte zu der erfolgreichen Interruptverarbeitung von Mikrocomputern dazu, die von Studenten der Elektrotechnik nicht immer auf Anhieb verstanden werden. Diese Arbeit setzt sich daher zum Ziel, eine bereits bestehende Simulation der Interruptverarbeitung von Intel 8086 Mikrocomputern zu optimieren. Dabei wird die Simulation benutzerfreundlicher gestaltet und die Interruptverarbeitung an einigen Stellen klarer dargestellt als in der bestehenden Simulation. Dazu werden unter anderem auch Audio-Elemente in der Simulation verwendet. Außerdem werden Aufgaben erstellt, die von den Studenten mit der Simulation bearbeitet werden können, um das Verständnis der Interruptverarbeitung des Intel 8086 zu festigen und ein Interesse an der Mikrocomputertechnik zu wecken bzw. zu festigen.

# 2. Projektkonzept

Diese Studienarbeit wird anhand eines Projektkonzeptes bearbeitet, welches im Folgenden vorgestellt wird. Das Projektkonzept ist als eigenständiges Dokument ebenfalls in *Anhang 1 – Projektkonzept* wiederzufinden.

## 2.1. Anforderungsspezifikation

Die Anforderungsspezifikation definiert Kernziele bzw. Festforderung sowie optionale Ziele bzw. Wünsche des Projektes sowie in welchem Umfang und in welcher Form diese Ziele zu erreichen sind.

### 2.1.1. Kernziele

Für das Projekt sind folgende Kernziele bzw. Festforderungen definiert, die zum Abschluss des Projektes unbedingt zu erreichen sind:

- Simulation soll in einer Datei zusammengefasst werden und sich mit einem Mausklick öffnen
- Der aktuelle Schritt der Simulation muss jederzeit klar sein
- Möglichkeit, Schritt(e) zurück zu gehen
- Signale sollen sich verzögert auf Leitungen fortbewegen, sodass man auch deren Ursprung, Ziel und Richtung erkennen und erklären kann
- Logik der Simulation soll vereinfacht und kommentiert werden, sodass ein Einarbeiten in die Simulation und weiteres Optimieren der Simulation in Zukunft einfacher ist

### 2.1.2. Optionale Ziele

Optionale Ziele zeichnen sich dadurch aus, dass sie nicht erreicht werden müssen, sondern abhängig von den gegebenen Möglichkeiten und Umständen umgesetzt werden. Auch wirtschaftliche Aspekte (Zeit, Kosten, etc.) spielen hier eine Rolle. Folgende optionale Ziele sind festgelegt:

- Simulation soll in einer Datei zusammengefasst werden und sich mit einem Mausklick öffnen
- Einbettung von Audio-Elementen in die Simulation
- Steuerung soll auch mit der Tastatur möglich sein (Schrittfolge, Lautstärke)
- Implementierung von Übungsaufgaben

### 2.1.3. Dateien und Dokumente

Dieses Kapitel definiert, welche Ergebnisse des Projekts konkret in welchem Format vorliegen sollen. Folgende Dateien bzw. Dokumente sind zum Abschluss des Projektes in folgendem Umfang notwendig

| Name | Beschreibung | Format |
|---|---|---|
| Simulation Interruptverarbeitung Intel 8086 | Simulation zur Visualisierung der Interruptverarbeitung eines Intel 8086 Prozessors nach den Anforderungsspezifikationen | Anwendung bzw. Simulation in einer Anwendung, inkl. aller nötiger Komponenten Für Windows-Nutzer zusammengefasst in einer .exe-Datei |
| Aufgaben zur Simulation | Aufgaben für Studenten, die mithilfe der Simulation die Interruptverarbeitung des Intel 8086 erlernen | Textdatei |

Tabelle I: Spezifikation der Dateien und Dokumente

### 2.1.4. Risiken

Die geplanten Optimierungen können die Simulationsoberfläche unübersichtlicher machen als die ursprüngliche Simulation. Es ist darauf zu achten, dass hinzukommende neue Elemente ein klar definiertes Ziel verfolgen und den Nutzen der Simulation steigern.

### 2.1.5. Testung

Die Testung der Simulation erfolgt in einzelnen Komponenten sowie als zusammengefügtes Gesamtkonzept. Dabei wird hinsichtlich verschiedener Kriterien wie Funktionalität, Realitätsbezug, Verständlichkeit und Handhabung getestet.

## 2.2. Projektplanung

Die Projektplanung untergliedert sich in zwei Teile. Zum einen in die Ablaufplanung anhand eines Lebenszyklusmodells und zum anderen in die zeitliche Planung. Beide werden im Folgenden erläutert.

## 2.3. Ablaufplanung und Lebenszyklusmodell

Das Projekt soll anhand des Elementarprozessschemas ETVX bearbeitet werden. Das ETVX-Modell dient zur Modellierung der Phasen eines Software-Entwicklungsprozesses und identifiziert dabei vier Aspekte bei der Modellierung einer Entwicklungsphase (Radice et al., 1985):

- **E**ntrance Criteria: Kriterien, die erfüllt sein müssen, damit die Phase gestartet werden kann
- **T**asks: Die zu erledigenden Aufgaben in dieser Phase
- **V**erification/Validation: Tests, die die erfolgreiche Fertigstellung der Aufgabe überprüfen und sicherstellen
- **E**xit: Bedingungen zum Abschluss der Phase

Das ETVX-Modell kann graphisch folgendermaßen dargestellt werden:

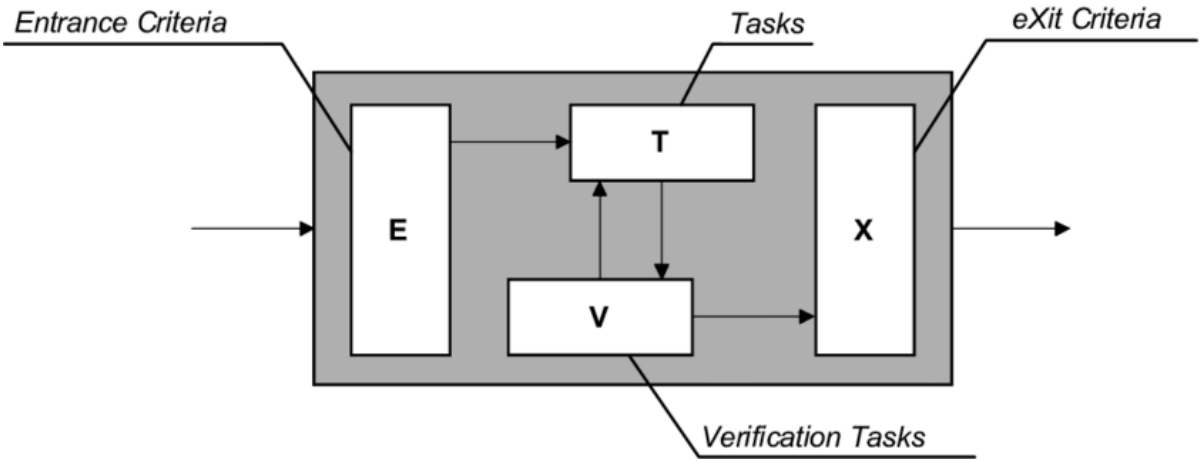



Abbildung 1: ETVX-Modell [Radice et al, 1985]

Eine Projektphase wird gestartet, wenn alle Eingangskriterien (Entry Criteria) erfüllt sind. Die Aufgaben (Tasks) werden bearbeitet und anschließend verifiziert und validiert. Sind Verifizierung und Validierung bestanden, kann die Prozessphase durch das Erfüllen aller Ausgangskriterien abgeschlossen werden. Andernfalls müssen die Aufgaben (Tasks) erneut bearbeitet werden, bis die Verifizierung und Validierung bestanden wird.

Die wesentliche Idee hinter diesem Modell ist, dass jeder Arbeitsschritt unmittelbar mit einer Qualitätssicherungs-Teilphase verbunden wird. Fehler können auf diese Weise früh gefunden und relativ kostengünstig behoben werden. Tabelle 1, Tabelle 2 und Tabelle 3 definieren die Prozessphasen für dieses Projekt nach dem ETVX Schema.

| Phase | I | II |
|---|---|---|
| Name | Recherche & Einarbeitung | Projektvorbereitung & Planung |
| Eingangskriterium/en | Mit Beginn des Projekts | Abschluss von Phase I |
| Aufgabe(n) | Intel 8086 recherchieren<br><br>Dokumentation der bestehenden Simulation lesen<br><br>In bestehende Simulation einarbeiten | Projektkonzept, Anforderungsspezifikationen und Zeitplan erstellen<br><br>Projekttitel auswählen |
| Verifikation und Validierung | Ist die Interruptverarbeitung des Intel 8086 verstanden?<br><br>Kann mit der bestehenden Simulation die Interruptverarbeitung des Intel 8086 nachvollzogen werden? | Sind Anforderungen und Zeitplan realistisch umsetzbar?<br><br>Ist der Arbeitsaufwand realistisch und absehbar?<br><br>Spiegelt der Projekttitel die Ziele wieder? |
| Ausgangskriterium/en | Keine offenen Fragen bezüglich der Interruptverarbeitung des Intel 8086 und der Funktion der bestehenden Simulation. | Der zeitliche Ablauf des Projekts ist vollständig festgelegt.<br><br>Es besteht eine genaue Vorstellung über den Arbeitsaufwand. |

Tabelle II: Projektphasen I und II nach ETVX

| Phase | III | IV |
|---|---|---|
| Name | Vereinfachen & Verbessern | Umsetzung & Tests |
| Eingangskriterium/en | Abschluss von Phase II | Abschluss von Phase III |
| Aufgabe(n) | Vereinfachen & Kommentieren der existierenden Simulation<br><br>Identifizierung von Verbesserungsmöglichkeiten | Besprechung & Umsetzung der Verbesserungsmöglichkeiten<br><br>Erstellen von Aufgaben,<br><br>Tests & Optimierungen |
| Verifikation und Validierung | Sind die Hauptteile der Schaltungssimulation vereinfacht & kommentiert, sodass deren Logik von nachfolgenden Studenten nachvollzogen werden kann?<br><br>Ist überflüssige Schaltungslogik eliminiert?<br><br>Sind Verbesserungsmöglichkeiten klar? | Ist die Priorität der Verbesserungsvorschläge klar?<br><br>Sind die Aufgaben nützlich zum Verständnis der Interruptverarbeitung des Intel 8086 und sind die Aufgaben lösbar?<br><br>Funktionieren die Kernfunktionen der ursprünglichen Simulation sowie die hinzugekommenen Verbesserungen? |
| Ausgangskriterium/en | Die Schaltung ist soweit vereinfacht & kommentiert, dass nachfolgende Studenten die Logik nachvollziehen können.<br><br>Überflüssige Schaltungslogik ist eliminiert.<br><br>Es ist hinreichend Platz zur Implementierung weiterer Funktionen und Verbesserungen vorhanden. | Die Simulation ist in den besprochenen Punkten verbessert und erweitert worden.<br><br>Die Aufgaben erfüllen ihren Zweck.<br><br>Die Simulation funktioniert und verhält sich in jeder Situation wie erwartet. |

Tabelle III: Projektphasen III und IV nach ETVX

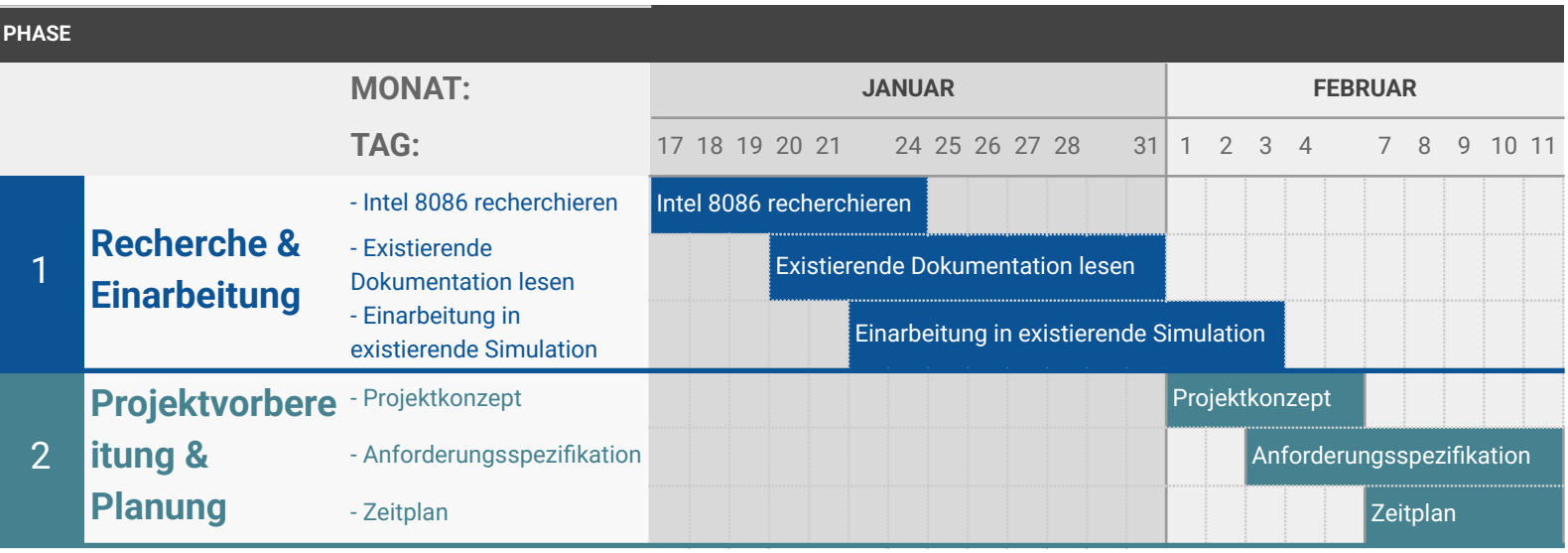
| Phase | V |
|---|---|
| Name | Abschluss |
| Eingangskriterium/en | Abschluss von Phase V |
| Aufgabe(n) | Evaluation der weiterentwickelten Simulation<br><br>Überprüfung aller Ziele<br><br>T3_3100 Studienarbeit schreiben |
| Verifikation und Validierung | Ist die weiterentwickelte Simulation ein wesentlicher Fortschritt gegenüber Version 1?<br><br>Funktioniert die Schaltung insgesamt?<br><br>Sind die Projektziele erreicht?<br><br>Ist die T3_3100 Studienarbeit zur Abgabe bereit? |
| Ausgangskriterium/en | Die weiterentwickelte Schaltung ist wesentlich leichter zu bedienen.<br><br>Selbst Nutzern, die die Simulation zum ersten Mal sehen ist immer sofort klar, in welchem Schritt sich die Simulation befindet.<br><br>Die weiterentwickelte hilft Studenten beim Verständnis der Interruptverarbeitung des Intel 8086 mehr als Version 1.<br><br>Die T3_3100 Studienarbeit ist sowohl digital als auch in Papierform abgegeben. |

Tabelle IV: Projektphase V nach ETVX

1. **Zeitplanung**

# PROJEKT ZEITPLAN

| | |
|---|---|
| **PROJEKT TITEL** | Simulation Interruptverarbeitung Intel 8086 |
| **PROJEKT MANAGER** | Daniel Hermes |
| **AUFTRAGGEBER** | DHBW Mannheim |
| **SUPERVISOR** | Dipl. Ing. Gerhard Lehmann |



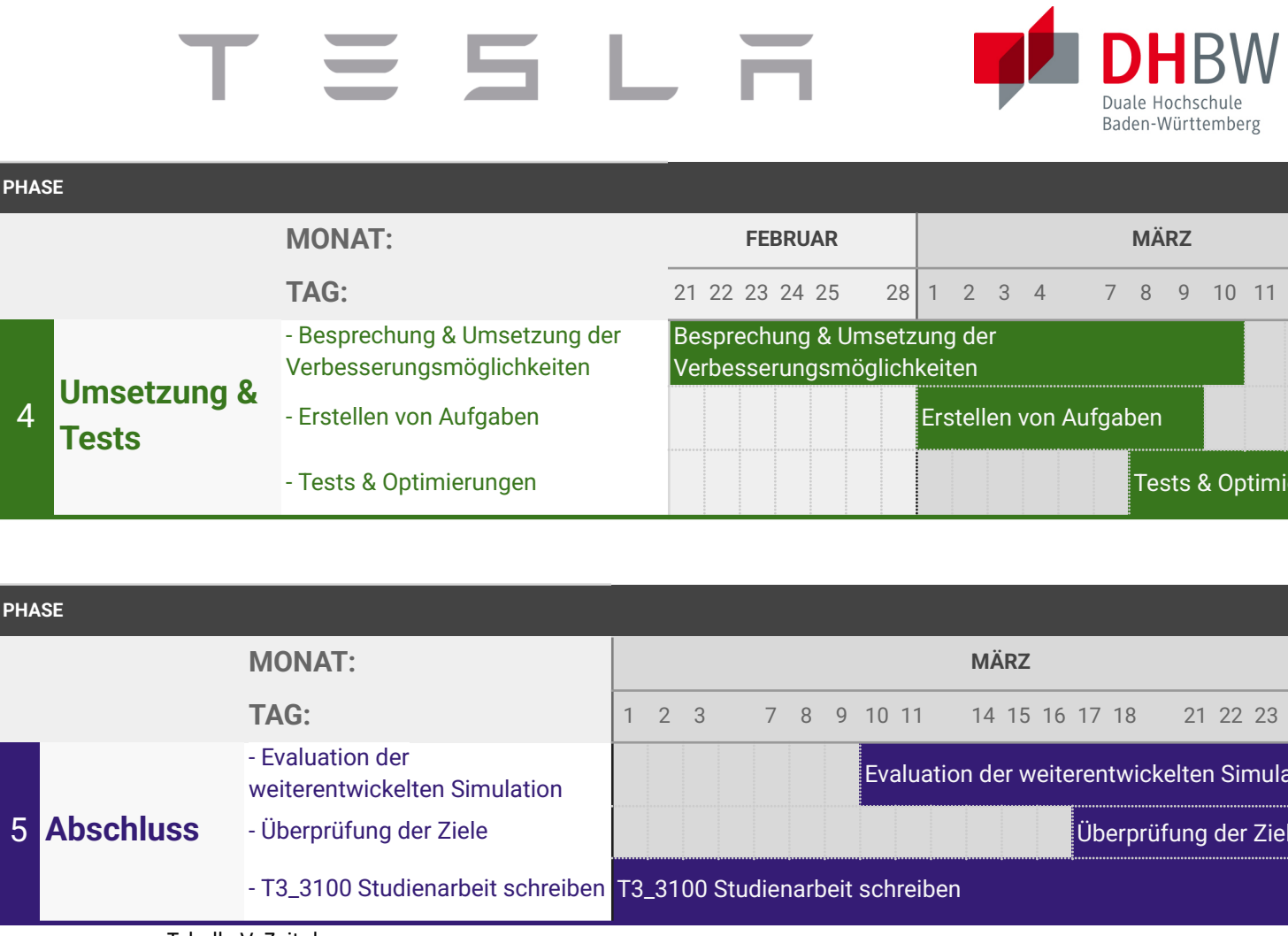




Tabelle V: Zeitplan

**2. KPIs (Key Performance Indicators)**

- Der Zeitplan wird eingehalten
- Jedem Benutzer ist jederzeit der aktuelle Schritt klar (auch Benutzern, welche die Simulation zum ersten Mal sehen)
- Man kann in der Simulation beliebig viele Schritte vor und zurück gehen und die Simulation funktioniert weiterhin wie erwartet
- Signale auf Bussen “wandern” in eine eindeutige Richtung
- Die Logik der Schaltung ist vereinfacht und kommentiert

# 3. Stand der Technik

In diesem Kapitel wird die Funktionsweise der bestehenden Simulation vorgestellt. Aus der Funktionsweise werden Optimierungsmöglichkeiten der bestehenden Simulation abgeleitet und beschrieben. Es werden dabei Ideen formuliert, wie diese Schwächen behoben werden können, was im darauffolgenden Kapitel beschrieben wird.

In einer Studienarbeit aus dem Jahr 2021 hat Silas Gerhard, zu dem Zeitpunkt Student der Elektrischen Energietechnik an der DHBW Mannheim, die Entwicklung einer Simulation der Interruptverarbeitung des Intel 8086 beschrieben [Gerhard, 2021]. Die Simulation wurde auf Basis von Digital aufgebaut, einem frei verfügbaren Simulator für digitale Schaltkreise [Neemann, 2022]. Anhand der Simulation soll der Ablauf eines Interrupts vom Triggern der Interrupt-Bedingung (simuliert durch Druck auf einen von sieben Knöpfen in der Simulation) bis zum Ausführen der ISR simuliert und Studenten erklärt werden. Die Funktionsweise der bestehenden Simulation ist fundamental für die im Rahmen dieser Arbeit entwickelten Optimierungen und wird daher im Folgenden analysiert.

## 3.1. Funktionsweise der bestehende Simulation

Die Simulation wird durch Anklicken des “Play-Symbols” im Kopfmenü gestartet. Alternativ kann auch die Leertaste verwendet werden, welche gleichzeitig die Simulation beenden kann. Durch den Start der Simulation werden Standardeinstellungen der Registerinhalte sichtbar. Außerdem leuchtet die Interrupt-Acknowledge-Leitung (kurz: INTA!) und zeigt damit eine logische 1 (High-Pegel) an, da noch kein Interrupt quittiert wurde. Weder Lesen- noch Schreiben-Leitung sind aktiv, genauso wird noch nichts auf den Adress- oder Datenbus übertragen, da noch kein Interrupt ausgewählt wurde, weshalb über diesen Bussen ein “Z” steht. Der Prozessor wartet in diesem Initialzustand auf das Triggern eines Interrupts.

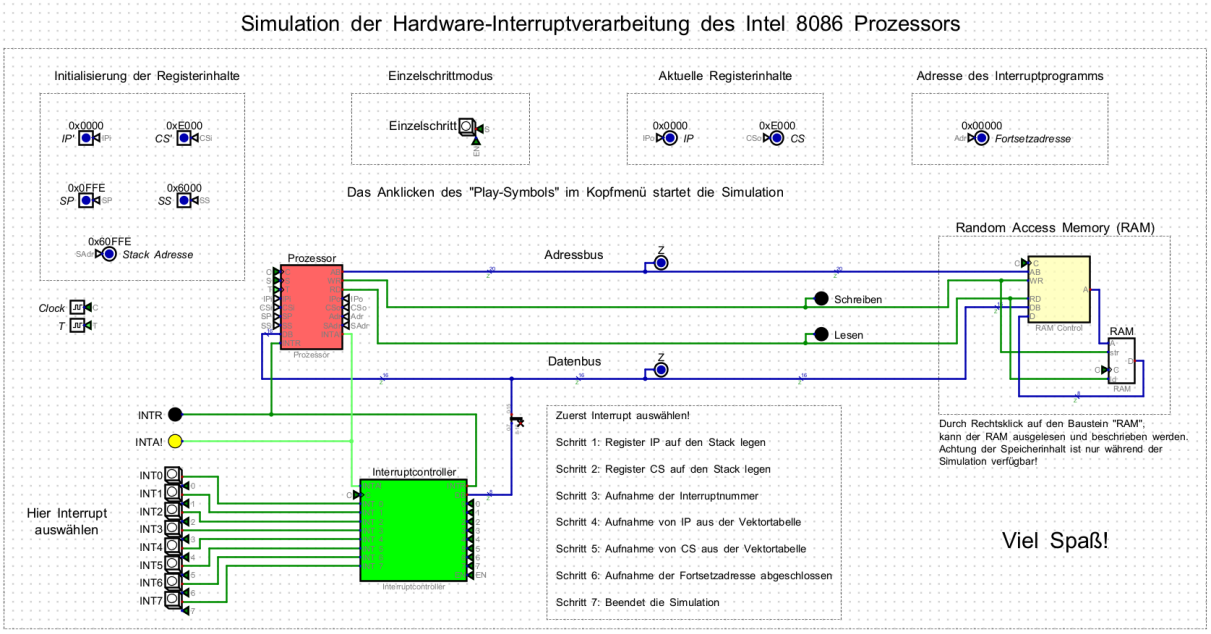



Abbildung 2: Übersicht bestehende Simulation [Gerhard, 2021, S. 28]

Wie die Schrittfolge unten zeigt, muss nun zunächst ein Interrupt ausgewählt werden. Die Auswahl eines Interrupts geschieht durch Anklicken von einem der acht Knöpfe links unten. Der ausgewählte Knopf leuchtet daraufhin hellgrün. Der Interruptcontroller aktiviert die INTR-Leitung, um dem Prozessor die Unterbrechung zu signalisieren, also leuchtet die INTR-Leitung ebenfalls hellgrün. Im folgenden Screenshot wurde Interrupt Nummer eins gewählt:

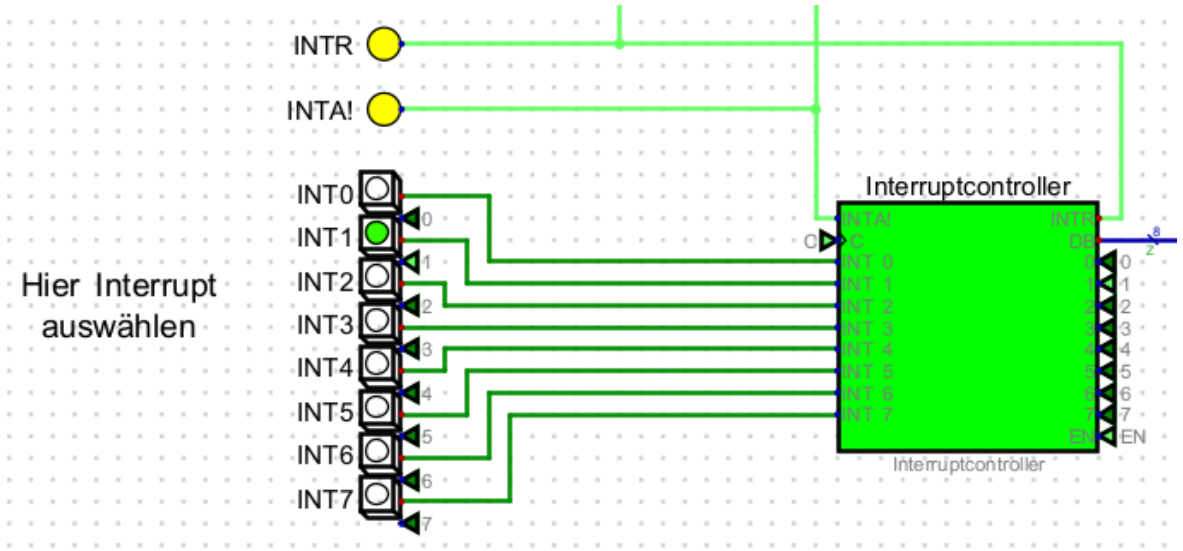



Abbildung 3: Interrupt 1 ausgewählt

### 3.1.1. Schritt 1

Nun muss der Benutzer durch Anklicken des Einzelschritt-Buttons oben zu Schritt 1 wechseln. Dabei wird der Instruction-Pointer (kurz: IP, Deutsch: Instruktions-Zeiger) auf den Stack gesichert. Durch Sichern des IPs auf den Stack kann der Prozessor diesen nach dem Interrupt vom Stack zurückholen und sich in den Ausgangszustand, vor Triggern des Interrupts, versetzen. Das ist wichtig, da für das Bearbeiten des Interrupts ein anderer IP geladen wird.

Da der Prozessor den Stack beschreibt, wird die Schreiben-Leitung aktiv und leuchtet hellgrün. Eine extra LED zeigt ebenfalls an, dass die Schreiben-Leitung aktiv ist. Die Stack-Adresse errechnet der 8086 durch Linksschieben des Stack-Segments (kurz: SS) um vier Stellen und anschließendes Addieren des Stack-Pointers (kurz: SP). Im folgenden Screenshot wird SS = 0x6000 um 4 Stellen nach links geschoben, was 0x60000 ergibt (einfach eine Null von rechts reingezogen). Die anschließende Addition von SP = 0x0FFE ergibt die Stack-Adresse von 0x60FFE, also wird 0x60FFE über den Adressbus übertragen (in dem folgendem Screenshot hellblau markiert). Standardmäßig wird der IP mit 0x0000 initialisiert, also wird 0x0000 über den Datenbus übertragen (in dem folgenden Screenshot gelb markiert). Damit ist die erste Hälfte der Zustandssicherung abgeschlossen.

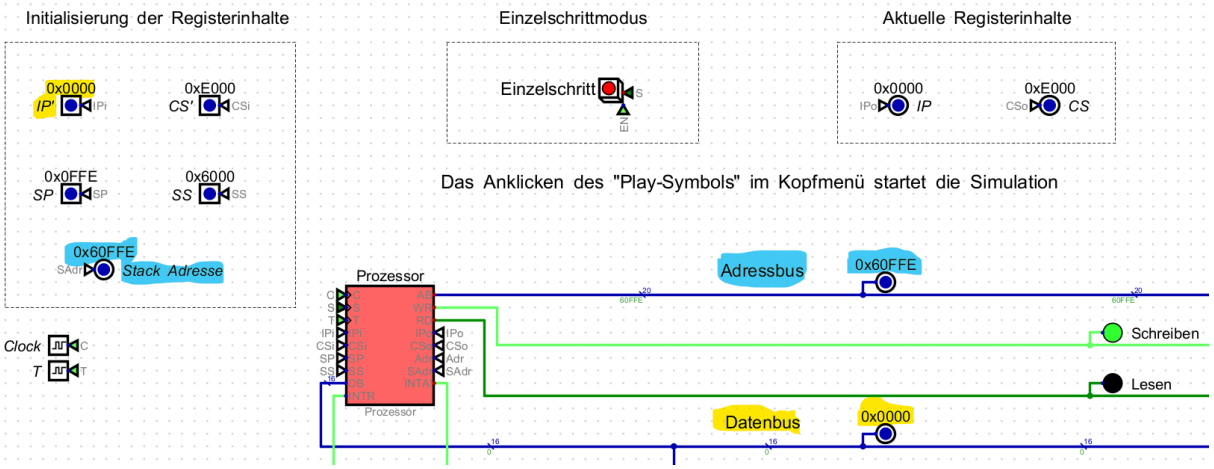



Abbildung 4: Schritt 1

### 3.1.2. Schritt 2

Als nächstes muss der Benutzer erneut den Einzelschritt-Knopf drücken, um zu Schritt 2 fortzufahren. In Schritt 2 wird das Code-Segment (kur: CS) auf dem Stack abgelegt. Dies dient dem gleichen Zweck wie das Sichern von IP in *Schritt 1* und ist nötig, da auch CS für das Bearbeiten des Interrupts überschrieben wird.

Auch in Schritt 2 ist die Schreiben-Leitung aktiv, da der Prozessor auf den Stack schreibt. Der Adressbus zeigt eine um 2 Hexadezimalstellen verminderte Adresse an. Das liegt daran, dass (A) der Stack nach unten hin wächst und (B) das RAM und damit der Stack pro Adresse nur 2-Hexadezimalstellen speichern kann, wodurch jeder 4-Hexadezimalstellen breite Registereintrag im Stack zwei Adressen benötigt.

Die standardmäßig eingestellte Stack-Adresse 0x60FFE, an der in *Schritt 1* der IP gespeichert wurde, wird also um 0x2 vermindert zu 0x60FFC. Über den Datenbus wird das voreingestellte CS übertragen: 0xE000. Nun ist der Zustand des Prozessors vor Triggern des Interrupts auf dem Stack gespeichert.

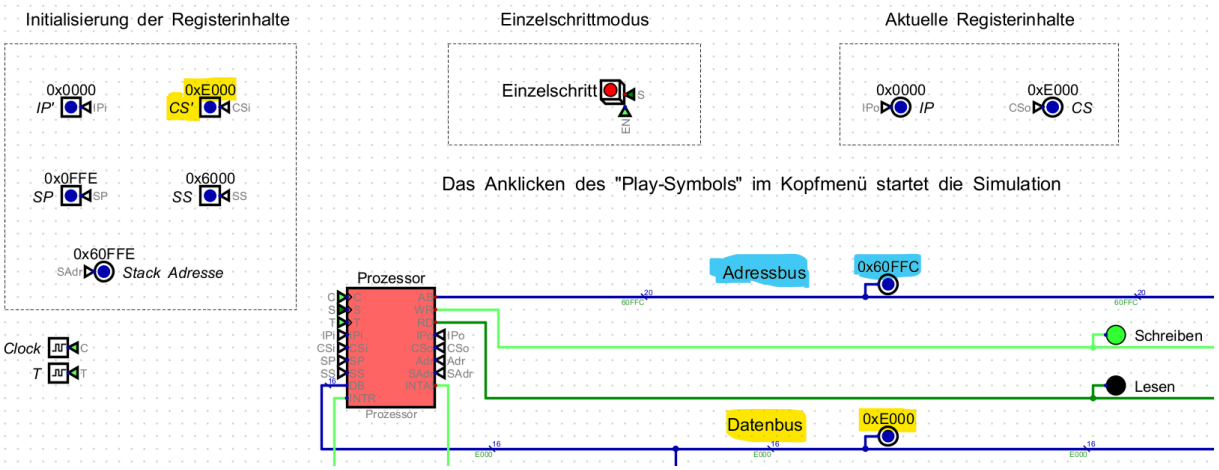



Abbildung 5: Schritt 2

### 3.1.3. Schritt 3

Im nächsten Schritt (Schritt 3) bestätigt der Prozessor den Interrupt dadurch, dass er INTA! auf LOW zieht. Das INTA!-Signal ist nullaktiv, d.h. im Ruhezustand HIGH und wird zur Bestätigung eines Interrupts auf LOW gezogen. Es verhält sich also genau umgekehrt wie man es erwarten würde (LOW im Ruhezustand, HIGH bei Bestätigung), weshalb es mit einem "!" versehen ist, das diese Negation ausdrückt.

Das "INTA! = LOW"-Signal nutzt der Interrupt-Controller, um die Interrupt-Nummer (0 bis 7) auf den Datenbus zu legen (in folgendem Screenshot gelb markiert). Da der Interrupt nun vom Prozessor bestätigt wurde, deaktiviert der Interrupt-Controller die INTR-Leitung. Der Adressbus (in folgendem Screenshot hellblau markiert) wird in diesem Schritt nicht verwendet, bleibt also hochohmig ("Z"). Weder die Lesen- noch die Schreiben-Leitung sind aktiv, da diese Leitungen zur Steuerung des Random-Access-Memorys (kurz: RAMs) dienen und der Prozessor in diesem Schritt nicht mit dem RAM interagiert. Der Prozessor nimmt die Interrupt-Nummer vom Interrupt-Controller über den Datenbus auf.

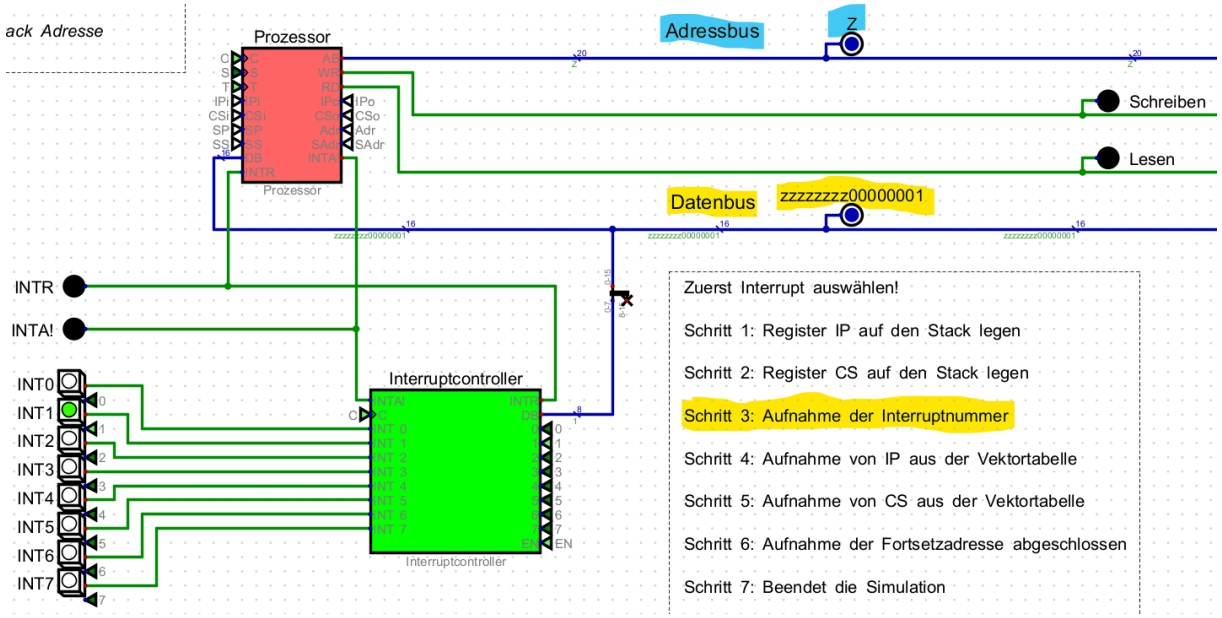



Abbildung 6: Schritt 3

### 3.1.4. Schritt 4

In Schritt 4 liest der Prozessor mithilfe der aufgenommenen Interrupt-Nummer den IP für die ISR aus der Interrupt-Vektortabelle (kurz: IVT) im RAM. Dazu multipliziert der Prozessor die Interrupt-Nummer mit einer dezimalen vier, wodurch sich bei gewähltem Interrupt 1 die Adresse 0x00004 ergibt (in dem folgenden Screenshot hellblau markiert). Würde man Interrupt 2 wählen, ergäbe sich entsprechend die Adresse 0x00008, bei Interrupt 3 die Adresse 0x00012 usw.

Über den Datenbus wird der Inhalt des RAMs an der über die Interrupt-Nummer bestimmten Adresse (hier: 0x00004) übertragen. Der Datenbus ist in folgendem Screenshot beispielhaft 0x1234. Diese Daten stellen den IP der ISR dar und damit die Stelle, an der der Prozessor mit der ISR beginnen wird. Daher ändert sich der IP unter “Aktuelle Registerinhalte” auf 0x1234 (in dem folgenden Screenshot gelb markiert).

Da der Prozessor vom RAM liest, aktiviert er die Lesen-Leitung. Die INTA!-Leitung ist wieder aktiv, da der Prozessor den Interrupt im letzten Schritt (Schritt 3) bestätigt hat.

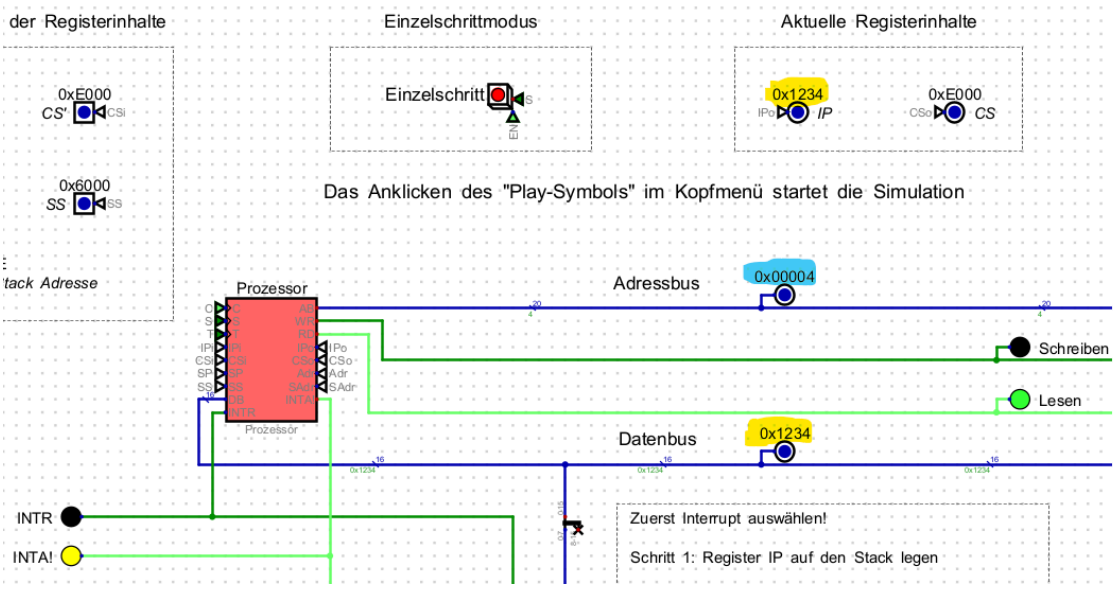



Abbildung 7: Schritt 4

### 3.1.5. Schritt 5

In Schritt 5 muss der Prozessor das Code-Segment (kurz: CS) der ISR aus der IVT lesen. Da der IP 2 Bytes in der IVT einnimmt, steht das CS 2 Bytes weiter. Für Interrupt 1 (Adresse IP im vorigen Schritt: 0x00004) steht CS also 2 Bytes weiter bei der Adresse 0x00006 (im folgendem Screenshot hellblau markiert). Für Interrupt 2 (Adresse IP in Schritt 4: 0x00008) ergäbe sich CS 2 Bytes weiter bei der Adresse 0x00010 usw.

Der Datenbus überträgt den Inhalt von CS, im folgenden Screenshot beispielhaft 0xCAFE, analog zum Screenshot in Schritt 4 gelb markiert. Die Lesen- sowie die INTA!-Leitung bleiben so wie bei Schritt 4 auf HIGH.

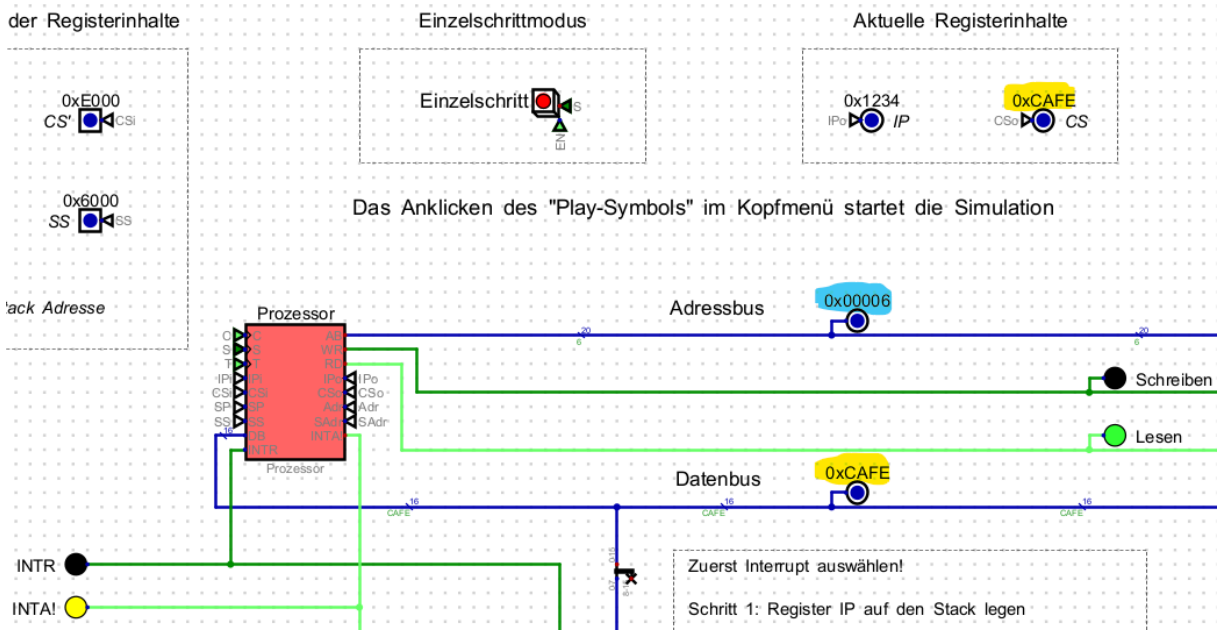



Abbildung 8: Schritt 5

Mit dem IP und CS aus der IVT errechnet der 8086 nun die Adresse der ISR, an der der Programmablauf fortgesetzt wird (daher auch “Fortsetzadresse”). Dies analog zur Berechnung der Stack-Adresse, die in Schritt 1 beschrieben wurde: Schieben von CS = 0xCAFE um 4 Stellen nach rechts, was 0xCAFE0 entspricht und anschließende Addition von IP = 0x1234:

0xCAFE0

+ 0x1234

= 0xCC214

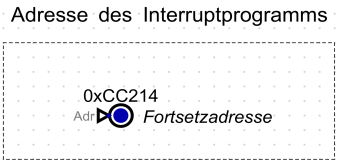



Abbildung 9: Fortsetzadresse

Anmerkung zu Schritt 4 und Schritt 5: Welche Daten für IP und CS der Prozessor aus der IVT im RAM liest hängt davon ab, was an der zuvor bestimmten Adresse im RAM steht. Startet man die Simulation und trägt dann nichts im RAM ein, wird das RAM voller Nullen initialisiert. Entsprechend würde in dem Fall bei Schritt 4 und 5 0x0000 übertragen werden und der Prozessor würde die ISR bei der Adresse 0x00000 starten, was einer Wahl des Interrupts 0 entspricht.

### 3.1.6. Schritt 6 und 7

In Schritt 6 werden Adress- und Datenbus sowie Lese- und Schreib-Leitung deaktiviert, um zu zeigen, dass die Aufnahme der Fortsetzadresse nun abgeschlossen ist. Schritt 7 beendet die Simulation.

# 4. Optimierungen

Aus dem detaillierten Verständnis der Funktionsweise der bestehenden Simulation heraus können einige Optimierungsmöglichkeiten festgehalten werden. Dabei wird von innen nach außen vorgegangen. Zunächst werden Optimierungen hinsichtlich *Installation und Start* diskutiert, weil Benutzer der Simulation damit als erstes in Kontakt kommen. Anschließend wird auf optische Optimierungsmöglichkeiten sowie Änderungen der Funktion und Bedienung eingegangen, wie beispielsweise die Anzeige des *aktuellen Schritts* oder die Möglichkeit, einen *Schritt zurück* zu gehen. Zuletzt wird die Implementierung von *Audio*-Elementen in die Simulation angesprochen, die Benutzern ein besseres Feedback ermöglichen sollen.

## 4.1. Installation und Start

In der Bedienungsanleitung der bestehenden Simulation ist beschrieben, dass zunächst Digital von Github heruntergeladen und entpackt werden muss [Gerhard, 2021, 1]. Eine Installation von Digital selbst ist nicht nötig, aber Java Runtime Environment (kurz: JRE) muss installiert sein. Weiterhin beschreibt die Bedienungsanleitung, wie die Simulation plattformunabhängig aus Digital heraus geöffnet wird, was allerdings einige Schritte erfordert, die man sich merken muss, und jedes mal Zeit kostet.

Die Plattformunabhängigkeit ist praktisch, aber in der Realität wird die Simulation von den meisten Nutzern auf Windows ausgeführt. Da Digital keine Installation benötigt, kann die Simulation in eine mit einem Doppelklick startfähige .exe-Datei gepackt werden. Dazu wird mithilfe der Entwicklungsumgebung Microsoft Visual Studio ein Projekt erstellt, in das alle nötigen Simulationsdateien einschließlich der Simulationsumgebung Digital eingebettet werden. Zusätzlich wird für das Projekt ein Programm in C# geschrieben, das die in dem Projekt verpackten Simulationsdateien einschließlich der Simulationsumgebung Digital entpackt und die Simulationsoberfläche startet. Das Programm erstellt zunächst einen Ordner, in den die eingebetteten Dateien extrahiert werden:

```
Directory.CreateDirectory("Embedded_Circuits");
```

Anschließend werden die Namen aller eingebetteter Dateien (“Assemblies”) in einen Array von Strings gelesen und durch jeden dieser Strings iteriert:

```
string[] assemblyNames =
    Assembly.GetExecutingAssembly().GetManifestResourceNames();
foreach (string assemblyName in assemblyNames)
{
```

Aus jedem dieser Strings wird eine Variable “assembly” gewonnen sowie ein FileStream-Objekt angelegt, das die entpackte Datei beschreibt. Die Assembly wird byte-weise in den FileStream überführt, wodurch die Assembly in den soeben erstellten Ordner kopiert bzw. entpackt wird:

```
    var assembly = Assembly.GetExecutingAssembly().
        GetManifestResourceStream(assemblyName);
    var file = new FileStream("Embedded_Circuits\\" + assemblyName,
        FileMode.Create, FileAccess.Write);
    assembly.CopyTo(file);
```

Nach dem Entpacken wird die Simulation mithilfe eines Prozesses gestartet. Der Prozess soll keine eigene Kommandozeile starten, sondern in dem Skript starten. Auch ein eigenes Fenster für die Kommandozeile ist nicht nötig, da der Prozess nur zum Starten der Simulation dient. Die Simulationssoftware Digital basiert auf Java, daher muss Java gestartet werden:

```
Process p = new Process();
p.StartInfo.UseShellExecute = false;
p.StartInfo.CreateNoWindow = true;
p.StartInfo.FileName = "java";
```

Als nächstes wird der Zielpfad gebraucht, damit die Simulationsdateien geöffnet werden können. Dieser besteht aus dem aktuellen Pfad, aus dem das Skript ausgeführt wird, und dem soeben erstellten Ordner “Embedded_Circuits”. Als Parameter werden der Pfad der Simulationssoftware Digital sowie der Pfad der Simulationsoberfläche übergeben. Mit diesen Einstellungen kann der Prozess gestartet werden. Zum Schluss wird abgewartet, damit die

Simulation sich öffnen kann. Andernfalls terminiert das Skript, bevor die Simulation sich öffnen konnte.

```
string targetDir = Directory.GetCurrentDirectory() +
    "\\Embedded_Circuits\\";
p.StartInfo.Arguments = "-jar " +
    targetDir + "Digital.jar " +
    targetDir + "Main_Interruptverarbeitung_8086.dig";
p.Start();
Thread.Sleep(50);
```

Das Skript überprüft zudem, ob eine Java-Laufzeitumgebung (JRE) vorhanden ist, da andernfalls Digital nicht startet. Es informiert den Benutzer entsprechend, wenn keine JRE festgestellt wird.

Nachteilig an dem Archiv ist die verlängerte Entwicklungszeit, wenn die Simulation weiterentwickelt wird. Anstatt die veränderten Simulationsdateien einfach abzuspeichern, müssen diese nun nach dem Abspeichern noch mit allen anderen Dateien zu dem Archiv gepackt werden, was mehr Zeit kostet und Kenntnis von Microsoft Visual Studio erfordert.

Das gleiche Problem besteht bei der Simulationssoftware Digital, die aktiv weiterentwickelt wird. Selbst während diese Arbeit entstand, erschien eine neue Version von Digital, die Probleme unter Windows im Vollbildmodus mit dem Tutorial behoben hat [Hermes & Neemann, 2022]. Jetzt wo Digital in ein Archiv gepackt wird, muss das Archiv neu erstellt werden, sobald eine neue Digital-Version zur Simulation verwendet werden soll. Dieses Problem ist allerdings vernachlässigbar, da die Simulation unter Digital v0.29 entwickelt wurde und nicht zwangsläufig auf Digital-Updates angewiesen ist, um ihren Zweck auch in Zukunft zu erfüllen.

Insgesamt überwiegen die Vorteile der einfachen und schnellen Handhabung einer einzigen Datei den Nachteilen in Form eines komplizierten Update-Prozesses. Die Simulation wird sowohl von Dozenten wie auch von Studenten viel eher benutzt werden, wenn der Start so einfach und reibungslos wie möglich ist.

## 4.2. Aktueller Schritt

Eine weitere Optimierungsmöglichkeit der bestehenden Simulation besteht in der Anzeige des aktuellen Schrittes. Das Fehlen einer Schrittanzeige ist vor allem am Anfang ein Problem, wenn man sich noch in die Simulation einarbeitet. Obwohl die Simulation nur sieben Schritte hat, vergisst man gelegentlich, in welchem Schritt man sich befindet. Das liegt daran, dass man oft einige Minuten bei einem Schritt stehen bleibt, um sich alle Signale anzuschauen und zu verdeutlichen, was in dem Schritt alles passiert. Dabei kann man vergessen, in welchem Schritt der Simulation man sich gerade befindet, zumal die Schritte 1 und 2 sowie die Schritte 4 und 5 sich stark ähneln.

Um dieses Problem zu lösen, muss der eingebettete Prozessor.dig-Baustein, der die Schritte zählt und verarbeitet, den aktuellen Schritt ausgeben. Diese 3 Bit lange Schrittzahl wird auf der Oberfläche von einem 3-Bit-Decoder auf acht Tunnel ausgegeben, von denen immer nur ein Tunnel aktiv ist, je nach aktuellem Schritt:

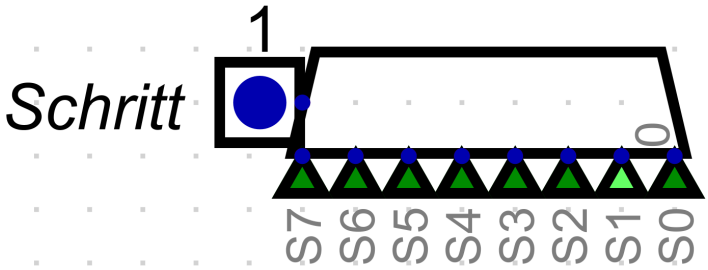



Abbildung 10: Schritt-Decoder

Diese Signale S0, S1 usw. können nun auf der Oberfläche der Simulation verwendet werden, um mit LEDs den aktuellen Schritt anzuzeigen:

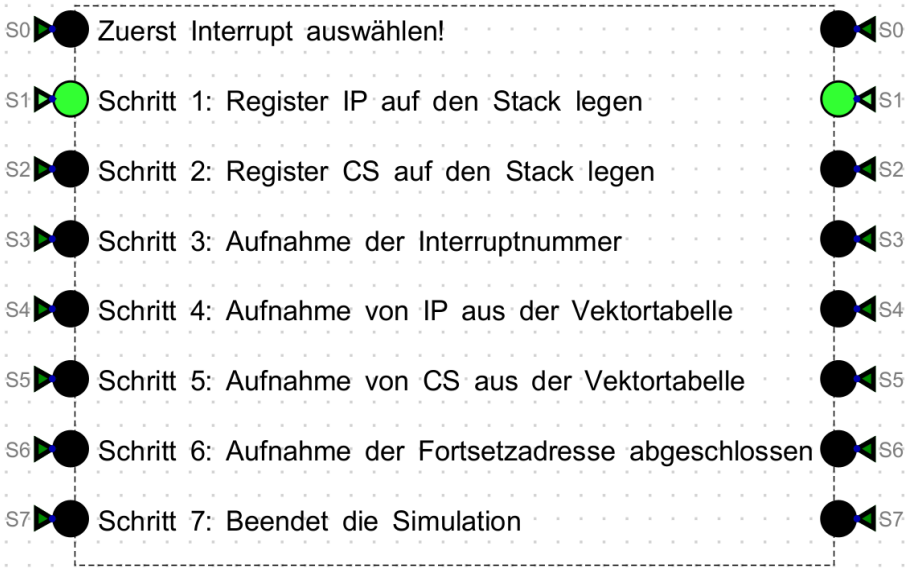



Abbildung 11: Schritt-Anzeige

Auf diese Weise ist jedem Benutzer sofort und zu jeder Zeit klar, in welchem Schritt sich die Simulation befindet.

### 4.3. Schritt zurück

Eine weitere Optimierungsmöglichkeit besteht in der Option, einen Schritt zurück gehen zu können. Manchmal wird man zu schnell über einen Schritt gesprungen sein oder hat etwas doch nicht so gut verstanden wie gedacht und möchte daher nochmal einen oder mehrere Schritt(e) zurück gehen. In der bestehenden Simulation ist das nicht möglich, man muss stattdessen die Simulation von vorne beginnen, die Ausgangsbedingungen herstellen (Interrupt entsprechend wählen, Register und RAM entsprechend einstellen) und bis zu dem gewünschten Schritt fortfahren. Dieser Umweg kostet Zeit und sollte daher vermieden werden, indem man in der Simulation einen Schritt zurück gehen kann.

Gegen diese Idee spricht die Idee einer hardwarenahen Abbildung der Interruptverarbeitung eines 8086. Schritte zurückgehen ist aus dieser Sicht unrealistisch, da der 8086 Prozessor während der Interruptverarbeitung auch nicht einen Schritt rückwärts geht, sondern alle Schritte nacheinander abarbeitet.

Da der Lernerfolg der Studenten, die mit dieser Simulation die Interruptverarbeitung des 8086 erlernen, im Vordergrund steht, wird dennoch die Möglichkeit geschaffen, einen oder

mehrere Schritt(e) zurück zu gehen. Die Vorteile für den Lernerfolg überwiegen hier die Nachteile der abnehmenden Hardware-Nähe.

Zur Realisierung von Rückwärts-Schritten muss im eingebetteten Prozessor die Logik des Schrittzählers erweitert werden. In der bestehenden Simulation ist der Schrittzähler ein einfacher Zähler, dessen enable-Eingang (en) konstant 1 und dessen clear-Eingang (clr) konstant 0 ist. Das Schritt-Signal geht auf den Clock-Eingang (C), wodurch jede Betätigung des Schritt-Signals zu einem Hochzählen des Zählers führt:

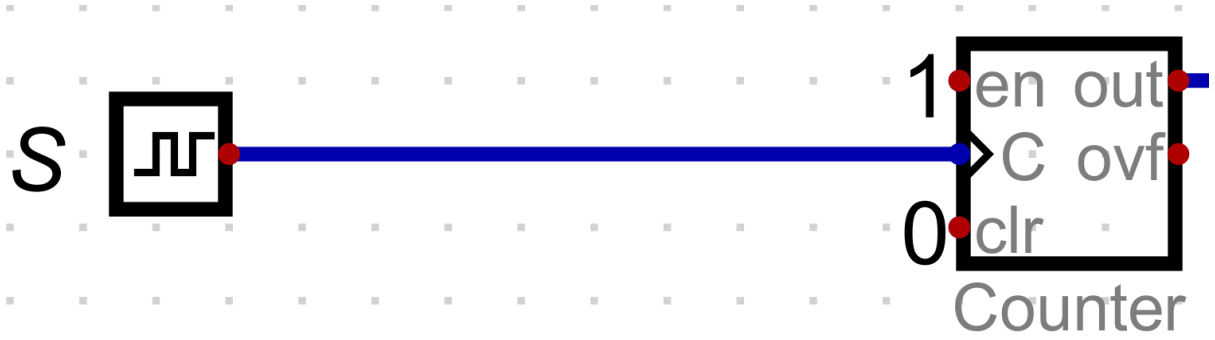



Abbildung 12: Bestehender Schrittzähler

Digital bietet neben dem einfachen Zähler-Baustein einen komplexeren Zähler, dessen Wert man setzen und dessen Zählrichtung man bestimmen kann. Das Setzen des Zählwertes wird im Rahmen dieser Arbeit nicht verwendet, aber das Einstellen der Zählrichtung wird eingesetzt. Zunächst wird die Zähl-Logik hergestellt, erstmal ohne Richtung. Gezählt werden soll beim Signal vorwärts oder beim Signal rückwärts. Die beiden Eingänge “Prev” und “Next” werden also über einen einfachen Oder-Baustein auf den Clock-Eingang des Schrittzählers geführt:

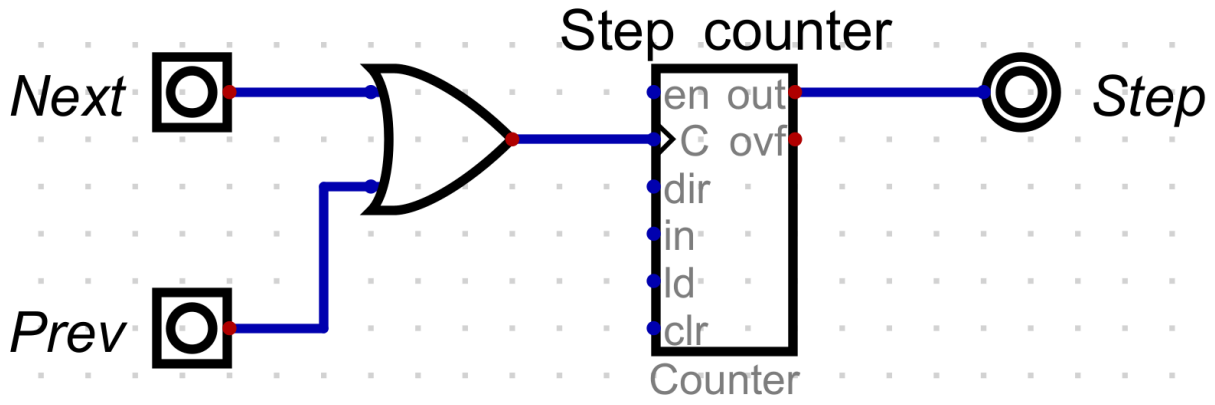


Abbildung 13: Schrittzähler auf- und abwärts

Nun muss der Richtungs-Eingang (dir) beschaltet werden. Die Beschreibung des dir-Eingangs verrät, dass eine 0 am dir-Eingang aufwärts zählen bedeutet, während eine 1 am dir-Eingang abwärts zählen bedeutet [Neemann, 2022, S. 84/110]. Um die 1 am dir-Eingang zu erzeugen, wenn abwärts gezählt werden soll, muss Prev direkt mit dem dir-Eingang verbunden werden. Der Enable-Eingang (en) verbleibt konstant auf 1, die restlichen Eingänge konstant auf 0, da sie nicht benötigt werden:

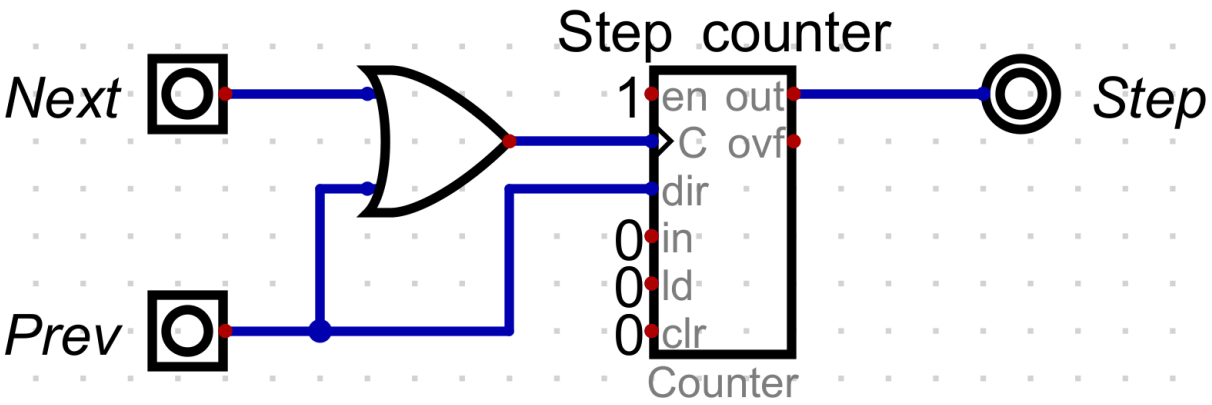



Abbildung 14: Schrittzähler komplett

Auf der Simulationsoberfläche muss der bisherige Einzelschritt-Push-Button als “Next” definiert werden. Eine Kopie dieses Push-Buttons wird als “Prev” definiert. Damit die Schrittfolge zusätzlich mit den Pfeiltasten der Tastatur gesteuert werden kann, wird der Prev-Push-Button als “UP” und der Next-Push-Button als “DOWN” definiert. Auf diese Weise können diese Push-Buttons mit den Pfeil-hoch- bzw. Pfeil-runter-Tasten der Tastatur gesteuert werden, wobei eine Steuerung über Mausklick auf die Push-Buttons wie bisher weiterhin möglich ist. Auf der folgenden Abbildung 15 ist die Einzelschritt-Steuerung in der bisherigen Simulation links zu sehen, während rechts die optimierte Variante mit Schritt vor und Schritt zurück gezeigt ist:

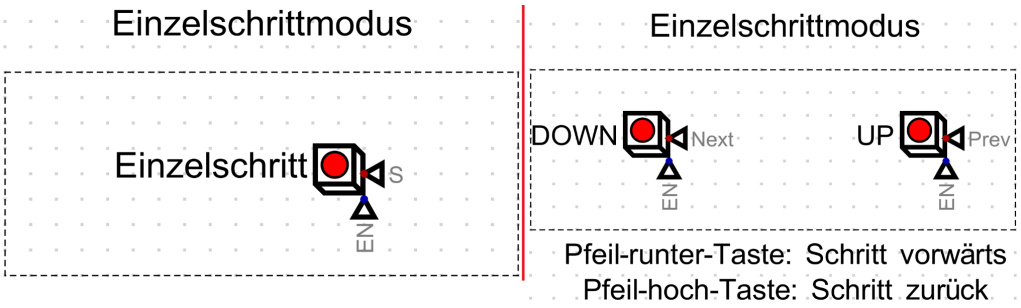



Abbildung 15: Vergleich Einzelschrittmodus bisher | Einzelschrittmodus optimiert

### 4.4. Start der Simulation

Die bestehende Simulation kann gestartet werden, auch ohne einen Interrupt auszuwählen. Die Simulation verhält sich in diesem Fall so, als wäre Interrupt-Nummer 0 gewählt worden. Das ergibt keinen Sinn, da der 8086 die simulierten Schritte der Interruptverarbeitung nicht durchläuft, ohne dass ein Interrupt anliegt.

Behoben wird dies durch eine Erweiterung des Schrittzählers sowie der eingebetteten Schaltung Prozessor.dig. Zunächst muss der in *Schritt zurück* beschriebene Schrittzähler derart erweitert werden, dass sein enable-Eingang (en) statt konstant 1 nun steuerbar wird. Da der Schrittzähler selbst nicht bestimmen kann, wann er zählen soll und wann nicht, wird der enable-Eingang (en) neben Next und Prev als dritter Eingang definiert und somit von außerhalb des Schrittzählers steuerbar gemacht. Die Änderung (rechts) gegenüber dem Entwurf in *Schritt zurück* (links) ist in der folgenden Abbildung 16 mit einem roten Kreis markiert:

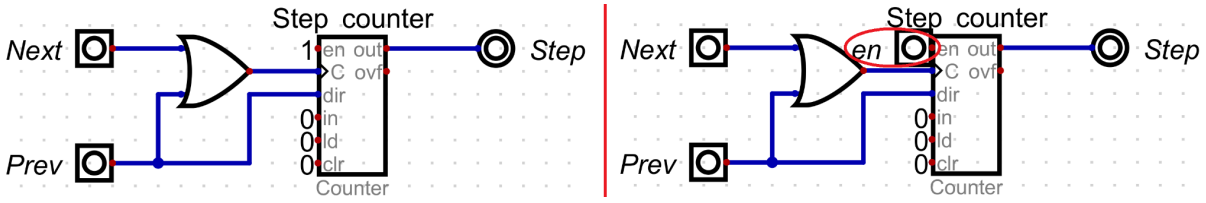



Abbildung 16: Vergleich Schrittzähler Erstentwurf lt. Abb. 15 | Steuerbarer Schrittzähler

Diese in sich abgeschlossene Schrittzähler-Schaltung kann nun mit ihren drei Eingängen Next, Prev und en sowie ihrem Ausgang Step in den Prozessor eingebettet werden. Next und Prev werden von der Oberfläche gesteuert, müssen also genau wie im Schrittzähler von außen

steuerbar sein. Zum Steuern des enable-Eingangs (en) auf Prozessor-Ebene wird das INTR-Signal benutzt. Das INTR-Signal wird in der bestehenden Simulation ohnehin zum Prozessor geführt, aber im Prozessor nicht verwendet. Damit das Zählen auch dann noch funktioniert, wenn das INTR-Signal in *Schritt 3* deaktiviert wird, wird ein RS-Flip-Flop verwendet, das mit dem INTR-Signal gesetzt (S-Eingang) und nie zurückgesetzt wird, weshalb der Reset-Eingang (R) auf Masse ist. Wenn einmal ein Interrupt gewählt wurde, bleibt der Schrittzähler also aktiv, bis die Simulation beendet wird.

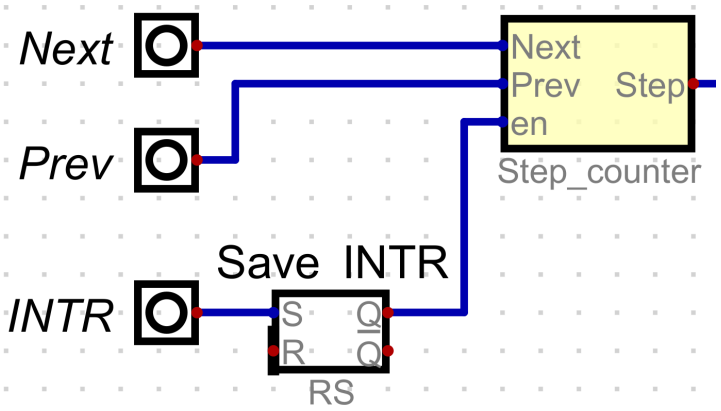



Abbildung 17: Schrittzähler eingebettet

### 4.5. Fortsetzadresse

In der bestehenden Simulation wird die Fortsetzadresse von Beginn an mit 0x00000 angezeigt. Das ist verwirrend, da die Fortsetzadresse erst in *Schritt 5* errechnet wird. Falls man nichts im RAM einträgt, bleibt die Fortsetzadresse über die gesamte Simulation bei 0x00000, was den Anschein erwecken kann, als würde gar nichts passieren.

Um dies zu optimieren, wird zwischen der Berechnung der Fortsetzadresse (Berechnung erfolgt durch die eingebettete Schaltung Offsetadd) und deren Ausgabe an “Adr” ein Transistor eingefügt, der erst ab Schritt 5 durchschaltet. Das führt dazu, dass die Fortsetzadresse vorher als High-Z bzw. unbestimmt angezeigt wird, was der Realität entspricht, da die Fortsetzadresse vorher noch nicht bekannt ist. Das von unten kommende Steuersignal des Transistors wird durch die “ver-Oder-ung” der Schritte 5 und aufwärts erreicht.

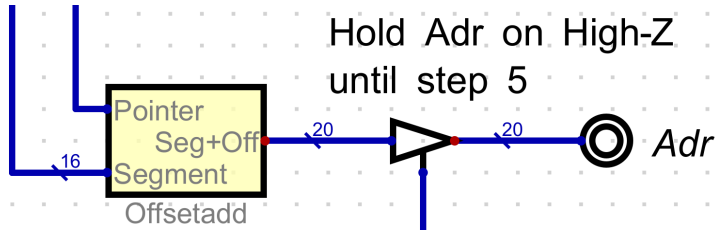



Abbildung 18: Rückhaltung Fortsetzadresse

### 4.6. Signalrichtung

Eine weitere Optimierungsmöglichkeit besteht in der Darstellung der Signalrichtung. Damit ist gemeint, dass dem Betrachter der Simulation gezeigt wird, wo der Ursprung eines Signals und wo sein Ziel ist. Bei den Lesen- bzw. Schreiben-Signalen ist die Darstellung der Signalrichtung nicht nötig, denn diese Signale haben ihren Ursprung stets im Prozessor und ihr Ziel ist stets das RAM. Wichtig ist diese Darstellung jedoch für Signale, die nicht immer denselben Ursprung oder dasselbe Ziel haben, also Signale des Adress- und Datenbusses. Adressen und Daten auf diesen Bussen “bewegen” sich hauptsächlich zwischen Prozessor und RAM hin und her, wobei die Richtung sich ändert. In *Schritt 1* und *Schritt 2* schreibt der Prozessor ins RAM, während sich Adressen und Daten in *Schritt 4* und *Schritt 5* vom RAM zum Prozessor “bewegen”. *Schritt 3* ist nochmal eine Ausnahme, hier schickt der Interrupt-Controller Daten, nämlich die Interrupt-Nummer, an den Prozessor.

In der bestehenden Simulation wird die Signalrichtung nicht angezeigt, alle Signale stehen praktisch sofort an, sobald man den Schritt wechselt. Dadurch muss man sich entweder überlegen, woher ein Signal kommt und was sein Ziel ist, oder es nachschlagen bzw. erklärt bekommen.

Damit die Signalrichtung in der Simulation verfolgt werden kann, werden an einigen Stellen auf oder nah an den Adress- und Datenbussen Ausgabe-Komponenten platziert, die mit einer Verzögerung und in der für den jeweiligen Schritt entsprechenden Reihenfolge das Signal anzeigen. Durch diese ort- und zeitversetzte Anzeigen entsteht das Gefühl einer gerichteten Bewegung des Signals beim Benutzer der Simulation. In der folgenden Abbildung 19 ist ein solcher gerichteter Signalfluss dargestellt. Es handelt sich dabei um *Schritt 3*, in dem die Interrupt-Nummer vom Interrupt-Controller zum Prozessor wandert. Die fünf dargestellten

Situationen folgen zeitlich im Abstand von etwa einer Sekunde aufeinander, sodass es so erscheint, als würde die gewählte Interrupt-Nummer 0x0007 vom Interrupt-Controller zum Prozessor fließen.

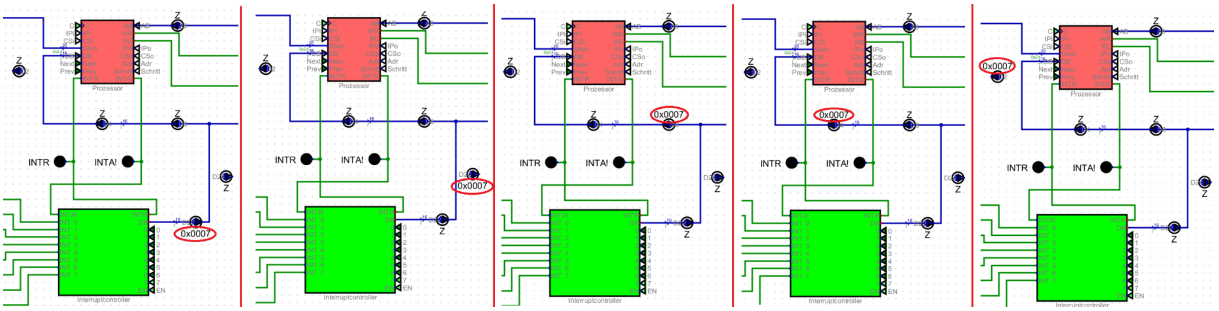



Abbildung 19: Gerichteter Signalfluss

Die wandernden Signale zwischen Adress- und Datenbus verhalten sich analog. Erzeugt werden diese Signale in der eingebetteten Schaltung “Misc” auf der Simulationsoberfläche. Auf diese Weise überflutet die Logik zum Erzeugen der gerichteten Signale nicht die Oberfläche.

Die “Misc”-Schaltung aktiviert je nach aktuellem Schritt einen Clock-Counter mit nachgelagertem Decoder, der wiederum die Bussignale auf die entsprechenden Anzeigen durchschaltet. Die nachfolgende Abbildung zeigt beispielhaft die Schaltung, die die gerichtete Interrupt-Nummer in *Schritt 3* erzeugt, so wie in Abbildung 19 oben gezeigt. Man sieht beispielhaft, wie beim ersten Clock-Tick der Datenbus auf das “D22”-Signal durchgeschaltet wird, welches die vom Interrupt-Controller gesehen erste Anzeige ist.

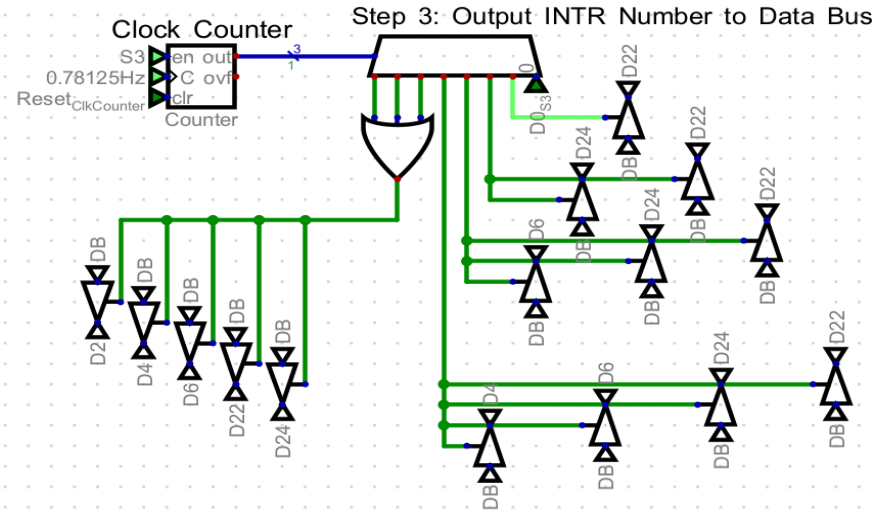



Abbildung 20: Schaltung der gerichteten Bewegung

Jeder nachfolgende Clock-Tick schaltet weitere Anzeigen in Richtung des Prozessors hinzu, sodass es so erscheint, als würde sich das Signal auf dem Datenbus vom Interrupt-Controller zum Prozessor fortbewegen, wie in Abbildung 19 gezeigt. Diese Logik wurde analog für die Signale zwischen Prozessor und RAM implementiert.

## 4.7. Audio

Die bestehende Simulation arbeitet ausschließlich visuell, d.h. es gibt kein zusätzliches Feedback an den Benutzer der Simulation außer das, was man auf dem Bildschirm sehen kann. Dabei wäre Audio-Feedback an den Benutzer an vielen Stellen der Simulation hilfreich, um z.B. bei Start der Simulation einen “normalen Programmablauf” vor dem Triggern eines Interrupts zu simulieren oder in Schritt 6, um hörbar zu machen, dass die ISR läuft.

Die Simulationssoftware Digital verfügt über einen integrierten Baustein, der Audio abspielen kann. Dabei handelt es sich um den MIDI-Baustein, wobei MIDI für **M**usical **I**nstrument **D**igital **I**nterface steht (engl. für “digitale Schnittstelle für Musikinstrumente“). Dabei handelt es sich um einen Industriestandard zum Austausch musikalischer Steuerinformationen zwischen elektronischen Instrumenten [The MIDI Association, 2021]. Im Folgenden wird beschrieben, wie dieser Baustein benutzt wird, um die Simulation mit Audio-Feedback zu ergänzen.

### 4.7.1. Normaler Programmablauf

Bevor ein Interrupt ausgewählt wird, arbeitet ein Prozessor wie der Intel 8086 den Programmcode ab. In der bestehenden Simulation kann dieser “normale Programmablauf” nicht wahrgenommen werden, er muss explizit von einem Vortragenden der Simulation erklärt werden. Das ist insbesondere in dem Fall problematisch, in dem Nutzer der Simulation sich die Interruptverarbeitung des Intel 8086 selbstständig anhand der Simulation aneignen.

Aus diesem Grund wird ein normaler Programmablauf vor der Auswahl eines Interrupts mit Hilfe der Beispielschaltung “BillieJean.dig” realisiert, die gemeinsam mit Digital ausgeliefert wird (Digital/examples/misc/BillieJean.dig). Die Schaltung spielt bei Start der Simulation das Lied “Billie Jean” von Michael Jackson in der Instrumentalversion ab. Realisiert wird dies über einen ROM-Baustein, aus dem der Quelltext des Liedes in MIDI-Format ausgelesen und an acht verschiedene MIDI-Bausteine geleitet wird, die den Ton erzeugen:

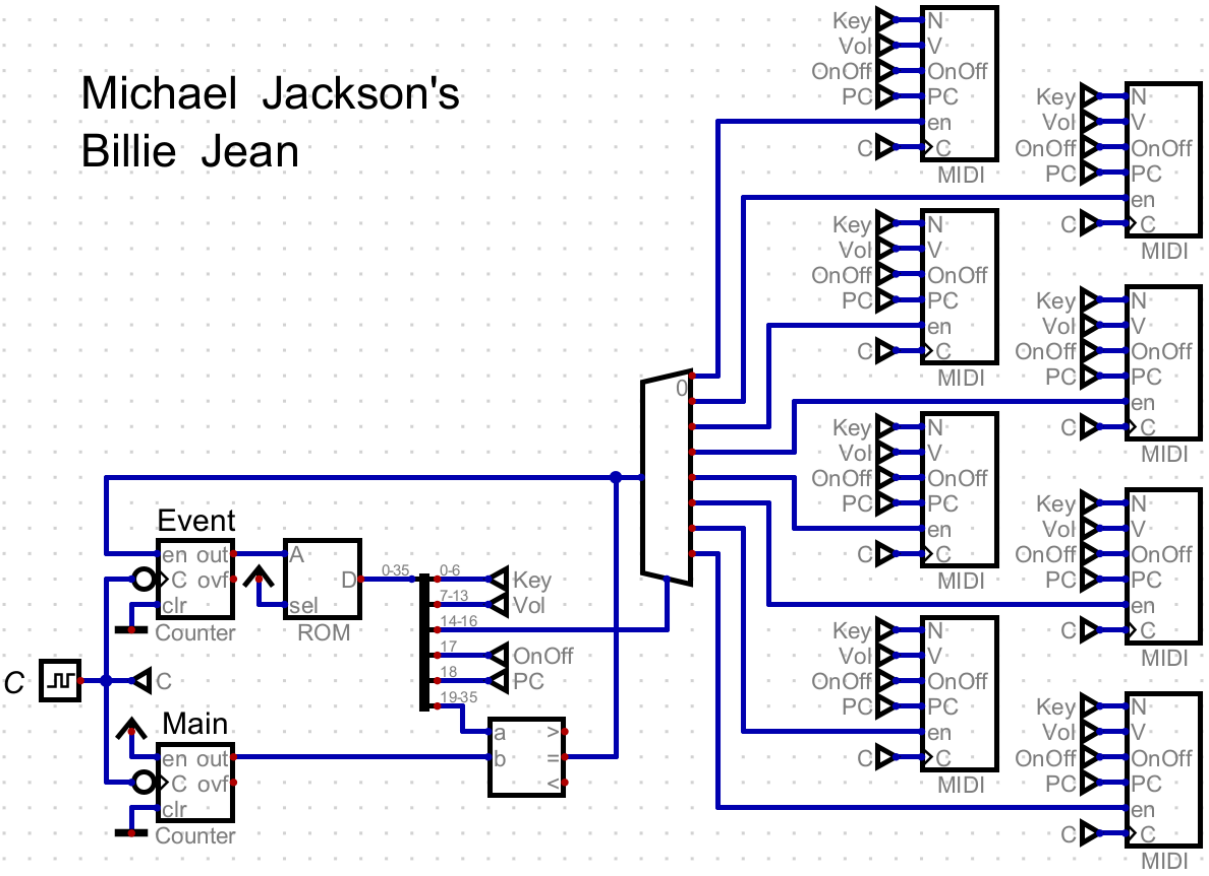


Abbildung 21: Billie Jean im Original

Eine genauere Betrachtung der Funktion dieser Schaltung würde den Rahmen dieser Arbeit überschreiten. Um die Schaltung für den angedachten Zweck nutzen zu können ist ein grundlegendes Verständnis der Schaltung ohnehin nicht erforderlich. Es reicht aus, an bestimmten Stellen in die Schaltung einzugreifen, um mit ihr einen normalen Programmablauf der Auswahl eines Interrupt realisieren zu können.

Damit das Lied angehalten werden kann sobald ein Interrupt ausgewählt wird, darf der Enable-Eingang (en) des “Main” genannten Zählers nicht konstant an Masse angeschlossen sein, wie in Abbildung 21. Stattdessen wird der en-Eingang als Eingang der BillieJean-Schaltung definiert, sodass über ein LOW-Signal an diesen Eingang das Lied angehalten werden kann. Die Änderung gegenüber Abbildung 21 ist in der folgenden Abbildung 22 rot markiert:

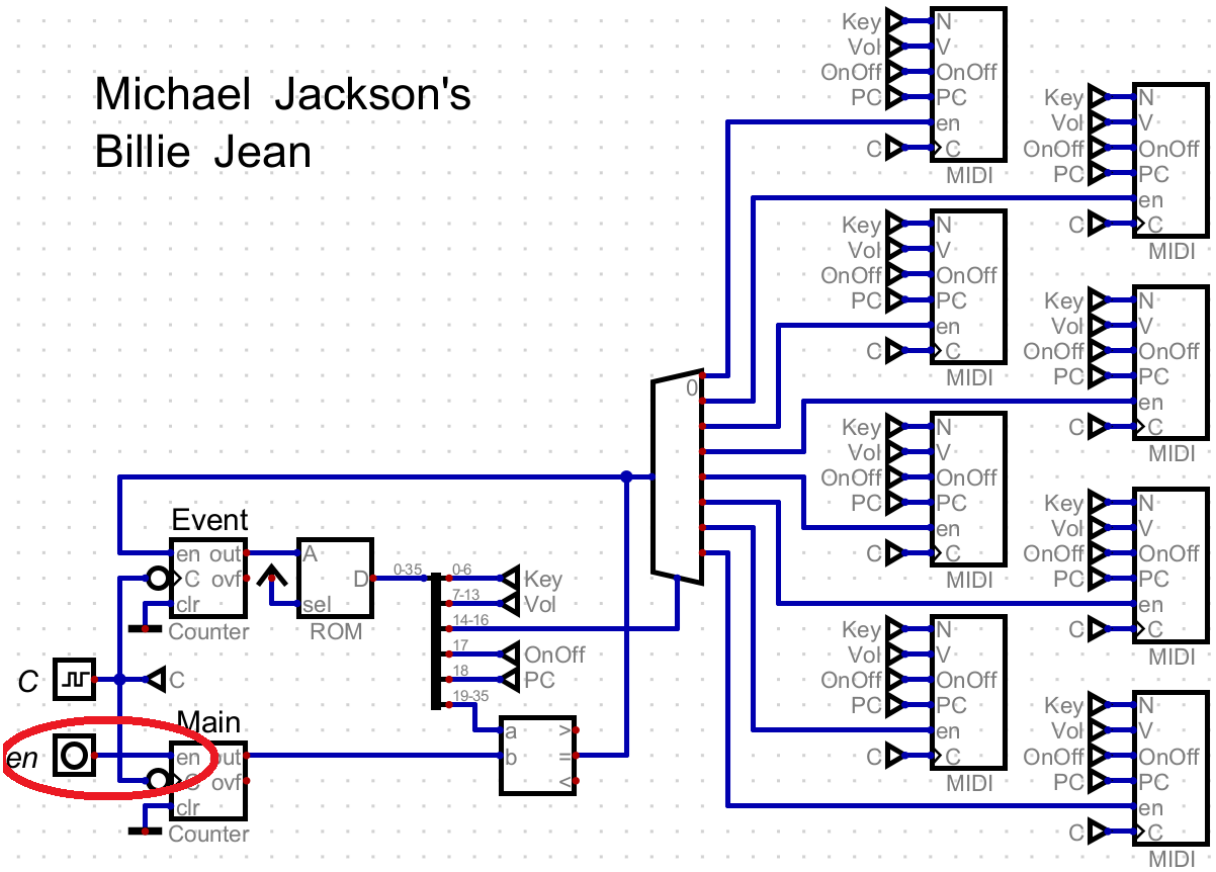



Abbildung 22: Billie Jean mit Enable-Eingang

Diese leicht veränderte BillieJean-Schaltung wird in die Simulationsoberfläche eingebettet, wodurch sie als übersichtlicher Baustein mit nur zwei Eingängen und keinem Ausgang erscheint.

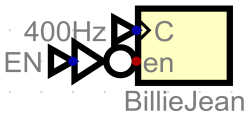



Abbildung 23: Billie Jean eingebettet

Versorgt wird die Schaltung mit 400Hz an ihrem Clock-Eingang (C) sowie dem invertierten Enable-Signal (EN), das der Interrupt-Controller als Ausgang bereitstellt und mit dem auch die LED der Einzelschritt-Steuerung versorgt wird. Dieses Signal wird aktiv und bleibt aktiv, sobald ein Interrupt gewählt wurde, d.h. es ist aktiv, solange kein Interrupt ausgewählt wurde und bleibt danach aus. Prinzipiell verhält sich dieses Signal damit wie das gespeicherte INTR-Signal, das dem steuerbaren Schrittzähler in Abbildung 16 (rechts) als Enable-Eingang dient. Da der Schrittzähler allerdings im Prozessor untergebracht ist und dieses Signal vom Prozessor nicht ausgegeben wird, wird das EN-Signal des Interrupt-Controllers verwendet, da dieses Signal vom Interrupt-Controller bereits für die LEDs der Einzelschritt-Steuerung ausgegeben wird.

### 4.7.2. Interrupt Auswahl

Neben dem Stopp von Billie Jean soll der Benutzer Feedback durch einen Ton erhalten, welcher Interrupt gewählt wurde. Dazu werden fast eins zu eins die Ausgänge des Encoders im Interruptcontroller auf einen MIDI-Baustein gegeben. Der Encoder hat alle acht Interrupts als Eingänge und gibt die Interrupt-Nummer in 3-Bit über seinen “num”-Ausgang aus. Hieraus wird die Note (N-Eingang des MIDI-Bausteins) bestimmt, die der MIDI-Baustein abhängig vom gewählten Interrupt spielt. Der MIDI-Baustein erwartet eine 7-Bit Note an seinem N-Eingang, daher wird diese 7-Bit Note mithilfe eines Splitters hergestellt. Der Splitter setzt die 3-Bit Interrupt-Nummer als höchstwertige Bits fest und füllt die niederwertigsten vier Bits mit Nullen auf.

Der Clock-Eingang (C) des MIDI-Bausteins wird direkt mit dem “any”-Ausgang des Encoders verbunden. Der any-Ausgang des Encoders wird aktiv, wenn irgendein Interrupt am Eingang aktiv ist, also wird immer dann der jeweilige Ton gespielt, wenn ein Interrupt gewählt wird.

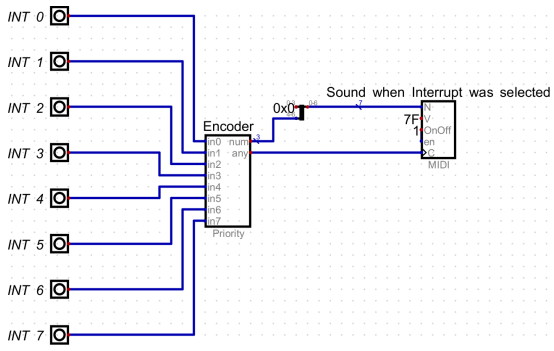



Abbildung 24: Interrupt Auswahl

Die Ausgänge des Encoders werden vom Interruptcontroller für weitere Zwecke verwendet, die in Abbildung 24 aus Gründen der Übersichtlichkeit nicht dargestellt sind.

### 4.7.3. ISR

In Schritt 6 ist die Aufnahme der Fortsetzadresse abgeschlossen und der 8086 führt die ISR aus. Genauso wie im Falle des *normalen Programmablaufs* merkt der Benutzer der Simulation davon allerdings nichts. Es passiert schlichtweg nichts in der Simulation in diesem Schritt. Damit Benutzer der Simulation merken, dass in diesem Schritt eine ISR ausgeführt wird, soll ähnlich zu der im Kapitel *Normaler Programmablauf* beschriebenen Optimierung auch in Schritt 6 ein Audio-Feedback erfolgen, welches das Ausführen einer ISR simuliert. Dazu wird analog zu dem Vorgehen im Kapitel *Normaler Programmablauf* eine bestehende Beispielschaltung eingebettet, die von außen steuerbar gemacht wird, um nur in Schritt 6 aktiviert zu werden. Verwendet wird für die ISR die Beispielschaltung “MIDIexample.dig”, die im gleichen Ordner wie “BillieJean.dig” standardmäßig mit Digital ausgeliefert wird:

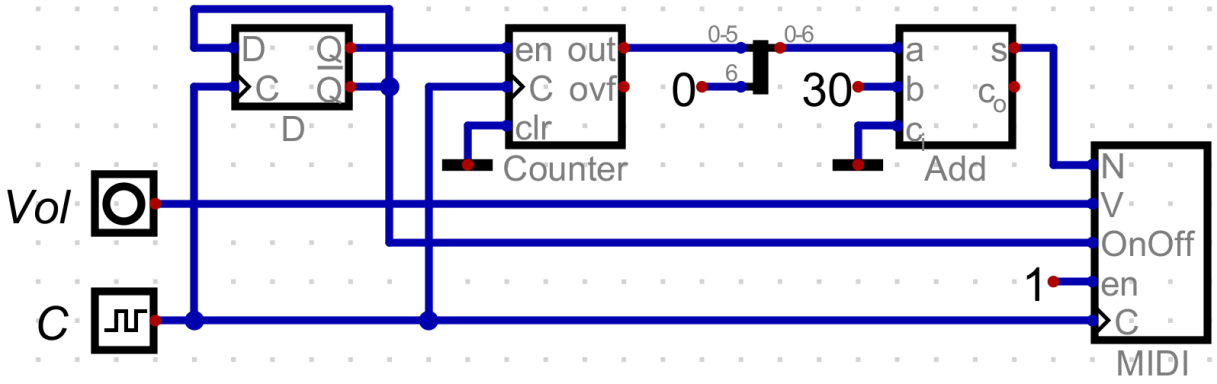



Abbildung 25: MIDIexample im Original

Folgende Änderungen werden vorgenommen: Volume (Vol) wird nicht als Eingang, sondern als Konstante “0x7F” definiert, was der maximalen Lautstärke entspricht. In der folgenden Abbildung XYZ ist diese Änderung auf der rechten Seite rot umkreist. Stattdessen wird die Konstante 1 am Enable-Eingang (en) des MIDI-Bausteins als variabler, von außen steuerbarer Eingang festgelegt. In der folgenden Abbildung XYZ ist diese Änderung auf der rechten Seite grün umkreist. Die linke Seite zeigt die ursprüngliche Schaltung gemäß Abbildung XYZ im Original zum Vergleich.

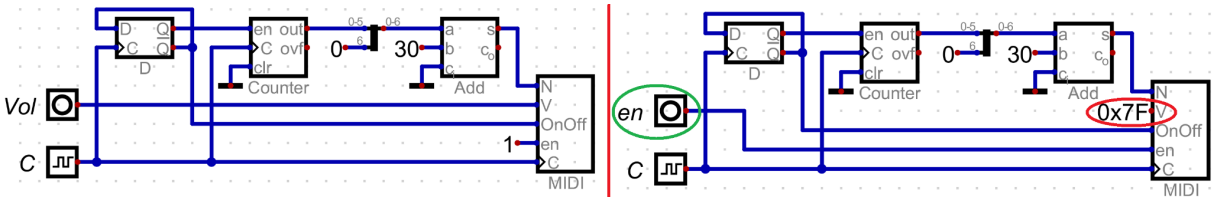



Abbildung 26: Vergleich MIDIexample Original || MIDIexample mit Enable-Eingang

Diese leicht veränderte MIDIexample-Schaltung wird in die Simulationsoberfläche eingebettet, wodurch sie analog zur BillieJean-Schaltung in Abbildung 23 als übersichtlicher Baustein mit nur zwei Eingängen und keinem Ausgang erscheint.

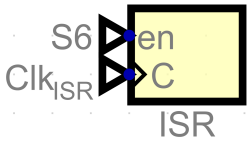



Abbildung 27: MIDIexample eingebettet

Versorgt wird die Schaltung mit dem Signal für Schritt 6 (S6) am Enable-Eingang (en) sowie der Clock der ISR ($CLK_{i\square r}$) am Clock-Eingang (C).

Man könnte nun $CLK_{i\square r}$ fixieren und hätte so immer die gleiche Tonspur in Schritt 6. Stattdessen bietet es sich an, die $CLK_{i\square r}$ von dem gewählten Interrupt abhängig zu machen, denn so ist es in der Realität auch. Jeder Interrupt hat eine unterschiedliche ISR, die auf diesen Interrupt reagiert. Variationen der $CLK_{i\square r}$ zeigen, dass die MIDIexample-Schaltung anfällig für Änderungen ihres Clock-Signals ist, d.h. Änderungen des Clock-Signals sorgen dafür, dass sich das Audiosignal der MIDIexample-Schaltung ändert. Das ist anders als bei der BillieJean-Schaltung, bei der ein verändertes Clock-Signal nur die Abspielgeschwindigkeit des Liedes verändert, nicht aber wesentlich seinen Klang, und der Hauptgrund für die Wahl der MIDIexample-Schaltung zur Simulation der ISR.

Durch Variation des Clock-Signal kann also auf einfache Weise der Klang der eingebetteten ISR-Schaltung verändert werden. Eine andere Methode, den Klang der Schaltung zu verändern, ist die Verwendung des optionalen Program-Change-Eingangs (PC) von MIDI-Bausteinen. Wird diese optionale Funktion aktiviert, erhält der MIDI-Baustein einen weiteren PC-Eingang. ”Wird dieser PC-Eingang auf High gesetzt, wird mit dem Wert am Eingang N das Programm (Instrument) gewechselt.” [Neemann, 2022, S. 53/110]. Auf diese Weise kann der Klang der ISR-Schaltung stärker verändert werden, da jeder Interrupt sein eigenes Instrument haben könnte. Schlussendlich wurde diese Variante nicht gewählt und bei der Variation von $CLK_{i\square r}$ geblieben, da die Variation von $CLK_{i\square r}$ für den Zweck ausreichend ist und der Weg über den PC-Eingang einen eigenen PC-Eingang an der ISR-Schaltung nötig machen würde. Mit der Variation von $CLK_{i\square r}$ bleibt die ISR-Schaltung kompakt mit zwei Eingängen und ohne Ausgang (s. Abbildung 27), wie die BillieJean-Schaltung (s. Abbildung 23) auch.

$CLK_{i\square r}$ muss also acht verschiedene Frequenzen, eine für jeden Interrupt, annehmen können. Um diese acht verschiedenen Frequenzen zu erzeugen, könnte jede Frequenz einzeln mit einem eigenen Clock-Signal erzeugt werden. Vorteil hierbei wäre, dass man die einzelnen Frequenzen genau einstellen könnte. Nachteil wäre, dass man acht verschiedene Clock-Signale erzeugt, deren Bausteine auf der Simulation Platz aufnehmen und diese unübersichtlich machen würden. Da die Feineinstellung der Frequenzen für den Zweck der

Unterscheidung von Audio-Signalen nicht so wichtig ist und die Übersichtlichkeit der Simulation im Fokus steht, werden die acht verschiedenen Frequenzen für CLK$_{i\square r}$ mit Hilfe eines Clock-Dividers aus einem einzigen Clock-Signal erzeugt. Dabei zählt ein Clock-Counter die Takte des einen Haupt-Taktes, in diesem Fall 400Hz, und gibt diese in 9-Bit aus. Hier würden 7-Bit reichen, da man aus 7-Bit plus Haupt-Takt die gewünschten acht verschiedenen Frequenzen erzeugen könnte, aber weder der Haupt-Takt noch der langsamste Takt (0,78125Hz) werden für CLK$_{i\square r}$ verwendet, da diese Frequenzen sich unangenehm schnell bzw. langsam anhören. Daher werden insgesamt zehn Frequenzen erzeugt, 400Hz Haupt-Takt sowie die neun Frequenzen des Clock-Dividers. Dabei hat das höchstwertigste Bit des Clock-Counter-Ausgangs (out) die halbe Frequenz des Haupt-Taktes, also 200Hz, und jedes weitere Bit wieder die halbe Frequenz des darüberliegenden Bits, also 100Hz, 50Hz, 25Hz usw., sodass das niederwertigste Bit die Frequenz 0.78125Hz hat. Diese Frequenz wird zwar nicht für CLK$_{i\square r}$ verwendet, aber für die bereits beschriebene Darstellung der *Signalrichtung*.

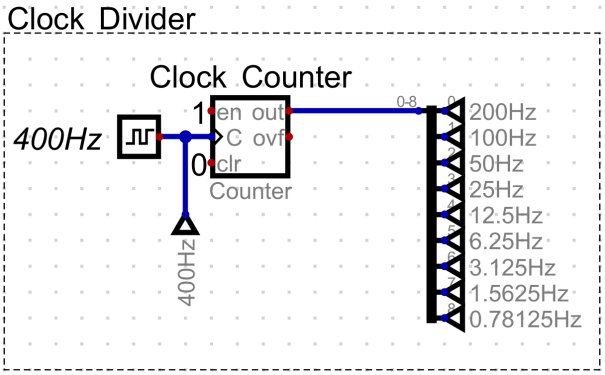



Abbildung 28: Clock Divider

Nun werden die acht “mittleren” Frequenzen des Clock-Dividers je nach gewähltem Interrupt auf CLK$_{i\square r}$ durchgeschaltet. Diese Aufgabe übernimmt ein Decoder, der die Interrupt-Nummer in 3-Bit als Eingang erhält und abhängig davon eine einzige Leitung aktiviert. Diese Steuerleitung aktiviert einen Transistor, der eine entsprechende Takt-Rate des Clock-Dividers auf CLK$_{i\square r}$ durchschaltet. Die Interrupt-Nummer wird dabei in Schritt 3 vom Datenbus gelesen und gespeichert, daher das D-Flip-Flop mit S3 als Clock-Eingang.

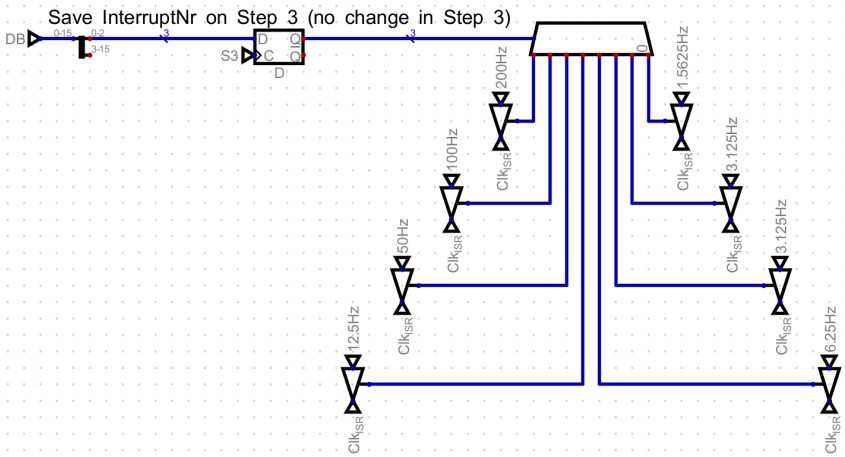



Abbildung 29: CLKi□r Schaltung

Der Interrupt-Controller führt die Interrupt-Nummern zwar auch als Tunnel auf die Oberflächen-Ebene heraus, um die entsprechende LED des gewählten Interrupt-Buttons leuchten zu lassen. Daher könnten auch diese Tunnel direkt zum Steuern der acht Transistoren verwendet werden, anstatt des Umwegs über den Datenbus und Decoder. Dieser Umweg wurde dennoch gewählt, da nahezu alle Audio-Funktionen mit Ausnahme der *Interrupt Auswahl* (diese findet direkt im Interrupt-Controller statt), in eine eigene Schaltung "Misc" zusammengefasst werden, um die Übersichtlichkeit der Oberfläche zu wahren. Auf diese Weise kann die Interrupt-Nummer innerhalb des Bausteins "Misc" vom Datenbus gelesen werden, wodurch dem Baustein "Misc" nicht acht einzelne Signale nur für diesen Zweck zugeführt werden müssen.

### 4.7.4. Lautstärke

Während der Präsentation der Simulation fiel auf, dass das Audio-Feedback in seiner bisher beschriebenen Form störend sein kann. An den entsprechenden Stellen können die Audio-Elemente einen selbst oder jene, denen man die Simulation zeigt, übertönen. Die globale Systemlautstärke kann in diesen Situationen nicht zur Lautstärkeregelung eingesetzt werden, da damit auch die Lautstärke jener beeinflusst wird, denen man die Simulation zeigt. Stellt man z.B. die globale Systemlautstärke leiser, da man das Audio-Feedback der Simulation zu laut findet, hört man gleichzeitig jene leiser, denen man die Simulation zeigt.

Windows bietet zwar die Möglichkeit, die Lautstärke eines jeden Programms individuell einzustellen [Glenn, 2016], aber nicht jeder Benutzer wird diese Möglichkeit kennen. Außerdem ist dieser Umweg mit einigen Klicks verbunden und daher unbefriedigend.

Aus diesem Grund wird eine Lautstärke-Steuerung in die Simulation selbst eingebaut. Auf der Oberfläche ähnelt diese dem optimierten Einzelschrittmodus wie in Abbildung XYZ dargestellt. Damit auch die Lautstärke über die Tastatur steuerbar ist, wird der Leiser-Knopf “LEFT” benannt, während der Lauter-Knopf “RIGHT” benannt wird. So kann die Lautstärke mit Pfeil-links leiser und mit Pfeil-rechts lauter gemacht werden.

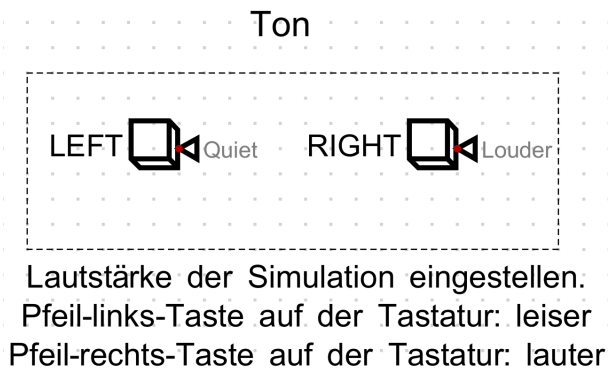



Abbildung 30: Lautstärkeeinstellung

Zur Einstellung der Lautstärke wird sowohl bei der BillieJean-Schaltung als auch bei der ISR-Schaltung ein weiterer Eingang “Vol” definiert. Versorgt werden diese Vol-Eingänge durch die bereits in Kapitel *Schritt zurück* vorgestellten Schrittzähler-Schaltung. Der Schrittzähler ist dauerhaft an, daher liegt eine konstante 1 an seinem Enable-Eingang (en). Das Lauter-Signal soll dabei einen Schritt hochzählen, geht also auf den Next-Eingang des Schrittzählers, während das Leiser-Signal einen Schritt runterzählen soll, weshalb es auf den Prev-Eingang des Schrittzählers geht. Ein Splitter erzeugt aus dem 3-Bit Schritt eine 7-Bit Lautstärkeeinstellung für die MIDI-Bausteine in der BillieJean- und ISR-Schaltung.

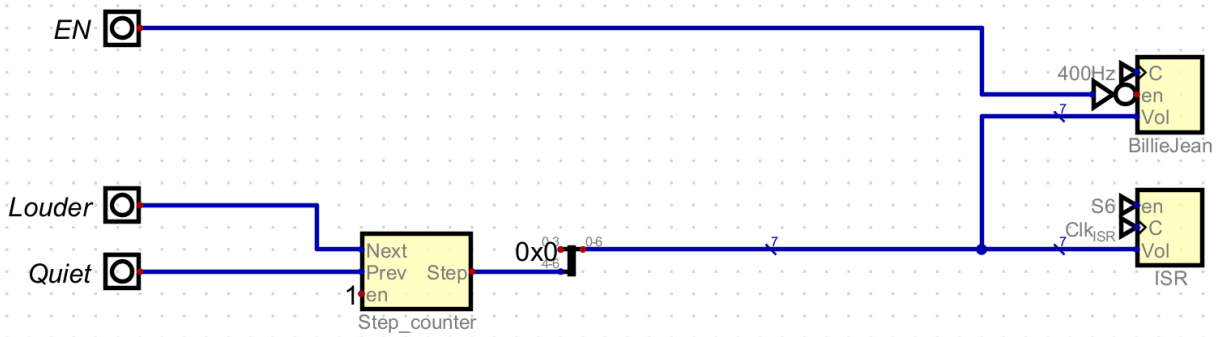



Abbildung 31: Erster Entwurf Lautstärke-Schaltung

Dieser Entwurf hat den Nachteil, dass die Lautstärke beim Start der Simulation immer null ist, man also nichts hören würde, wenn man nicht lauter stellt. Dieses Problem kann durch eine mittige Standardeinstellung für die Lautstärke behoben werden. In der folgenden Abbildung erhöht ein Addierer den Ausgang des Schrittzählers konstant um 0x30, wodurch die null beim Start der Simulation zu einer 0x30 wird und die Simulation somit eine mittige Lautstärke als Standardeinstellung hat.

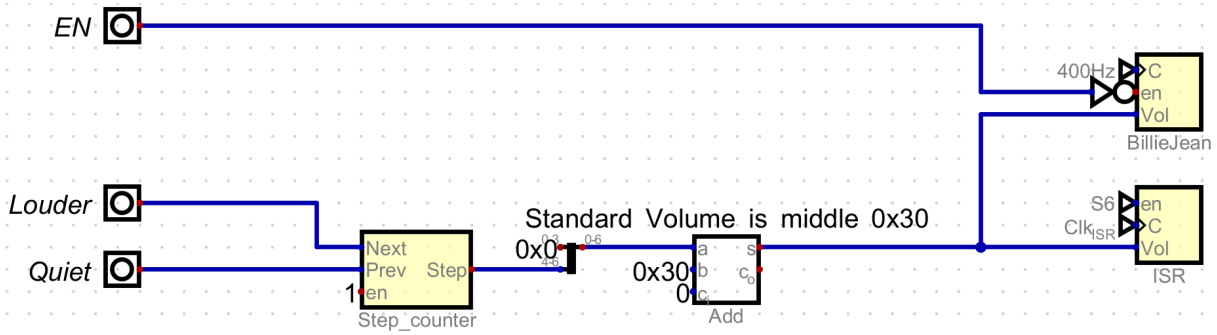



Abbildung 32: Lautstärke-Schaltung mit Standardeinstellung

Ein immer noch bestehendes Problem dieses Entwurfs ist das Verhalten an den Lautstärkegrenzen. Stellt man bei maximaler Lautstärkeeinstellung lauter, dann geht der Ton aus, da die 7-Bit Lautstärkeeinstellung überläuft und wieder bei null beginnt. Stellt man umgekehrt bei Ton (0x50 vom Schrittzähler + 0x30 vom Addierer = 0, Ton aus ist also praktisch ein Überlauf) aus eine Stufe leiser, dann steht die Simulation auf maximaler Lautstärke (0x40 vom Schrittzähler + 0x30 vom Addierer = 0x70, maximale Lautstärke).

Aus diesem Grund werden zwei Rückführungen der Lautstärke mit integrierten Komparatoren implementiert. Die obere Rückführung deaktiviert die Lauter-Taste, wenn die Lautstärke größer oder gleich 0x70 ist. Wenn die Lautstärke maximal ist, kann man also nicht noch lauter machen. Die untere Rückführung deaktiviert die Leiser-Taste, wenn die Lautstärke kleiner oder gleich null ist. Wenn der Ton aus ist, kann man also nicht noch leiser machen.

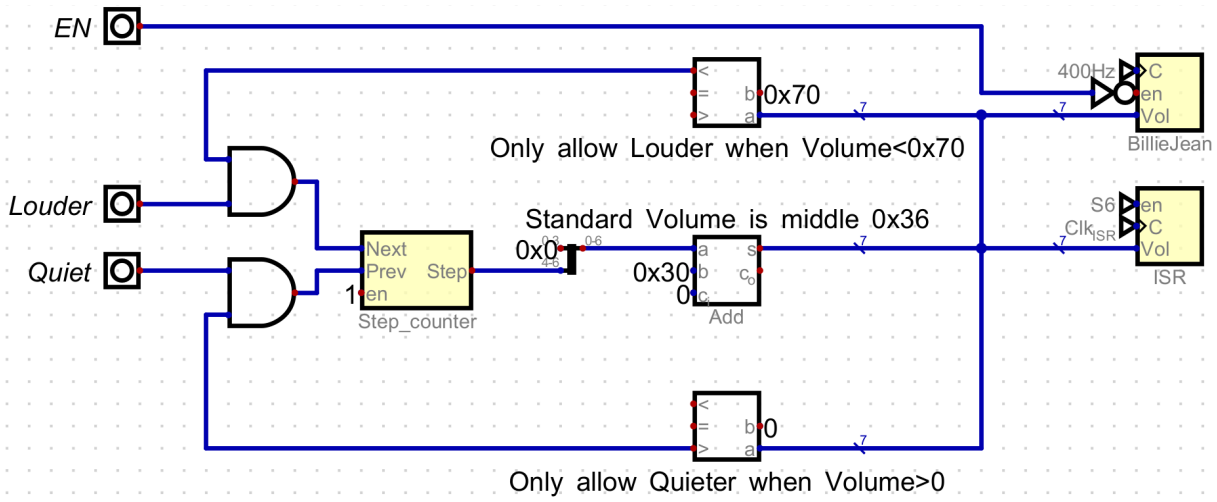



Abbildung 33: Lautstärke-Schaltung mit Rückführungen

## 4.8. Stack-Adresse

Die bestehende Simulation zeigt die initial aus SS und SP errechnete Stack-Adresse an. Da eine Anzeige der aktuellen Stack-Adresse abhängig vom Schritt fehlt, wird dem Benutzer der Simulation nicht bewusst, dass und wie sich die Stack-Adresse während des Durchlaufens der einzelnen Simulationsschritte verändert. Die aktuelle Stack-Adresse wird vom Prozessor ohnehin errechnet, um diese Adresse auf den Adressbus zu legen. Zur Anzeige der aktuellen Stack-Adresse muss der Prozessor diese lediglich ausgeben. Dazu wird die aktuelle Stack-Adresse, im Prozessor als $SAdr_{ou}$ bezeichnet, abhängig vom aktuellen Schritt aus der initialen Stack-Adresse, im Prozessor als $SAdr_{i}$ bezeichnet, errechnet.

In Schritt 0, bevor ein Interrupt getriggert wurde, oder in Schritt 8, nachdem der 8086 den Interrupt verarbeitet hat, entspricht die aktuelle Stack-Adresse der initialen Stack-Adresse.

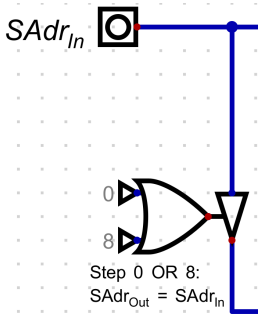



Abbildung 34: Stack-Adresse in Schritt 0 und Schritt 8

In Schritt 1 wird Register IP auf den Stack gesichert. Dadurch verringert sich die aktuelle Stack-Adresse um 0x2 gegenüber der initialen Stack-Adresse, da der Stack nach unten wächst. Auch in *Schritt 7: POP CS* steht diese Stack-Adresse an. Mehr zu diesem neu hinzugekommenen Schritt im entsprechenden Kapitel.

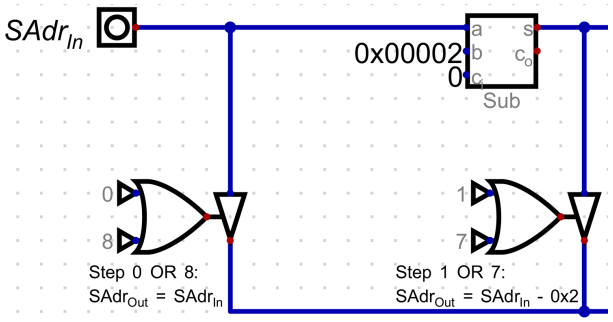



Abbildung 35: Stack-Adresse in Schritt 1 und Schritt 7

In Schritt 2 wurden sowohl IP als auch CS auf den Stack gesichert. Die aktuelle Stack-Adresse entspricht daher der initialen Stack-Adresse um 0x4 verringert. Die durch diese Bedingungen bestimmte aktuelle Stack-Adresse wird in den entsprechenden Schritten, in denen sich die Stack-Adresse ändern kann (Schritt 0, 1, 2, 7 oder 8) in einem Register gespeichert, um auch in den anderen Schritten 3, 4, 5, 6 auf der Oberfläche angezeigt werden zu können.

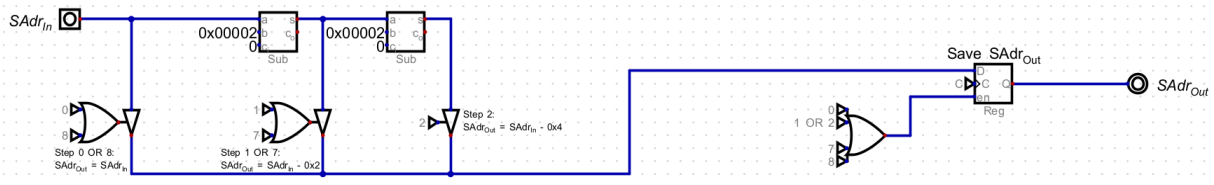



Abbildung 36: Vollständige Schaltung zur Ausgabe der aktuellen Stack-Adresse

## 4.9. Return from Interrupt

Bisher wird die Interruptverarbeitung nur bis zur Aufnahme der Fortsetzadresse simuliert, also bis zum Start der ISR. Zu einer vollständigen Simulation der Interruptverarbeitung gehört jedoch auch die Rückkehr in den Ursprungszustand, bevor der Mikrocomputer unterbrochen wurde. Dazu müssen nach der ISR die Register IP und CS vom Stack gelesen werden und zwar in umgekehrter Reihenfolge zu wie sie auf dem Stack gesichert wurden. Die Simulation wird zu diesem Zweck um drei neue Schritte ergänzt, die im Folgenden näher erläutert werden.

### 4.9.1. Schritt 7: POP CS

Nach der ISR liegt das CS ganz oben auf dem Stack, da CS in *Schritt 2* zuletzt auf dem Stack abgelegt wurde. Damit der 8086 nach dem Ausführen der ISR wieder dort weitermachen kann, wo er unterbrochen wurde, muss CS vom Stack zurückgeholt werden. Das Vorgehen ist analog zum Lesen aus der IVT in *Schritt 4* und *Schritt 5*. Die Read-Leitung wird aktiviert und ein Transistor legt die Adresse, von der gelesen werden soll, auf den Adressbus. Dies sorgt dafür, dass der Prozessor das RAM an der Adresse auf dem Adressbus liest, wo das vor dem Interrupt gesicherte CS liegt. CS liegt 0x4 Stellen im RAM tiefer als die initiale Stack-Adresse, daher wird diese um 0x4 verringerte initiale Stack-Adresse abgegriffen und in Schritt 7 auf den Adressbus gelegt.

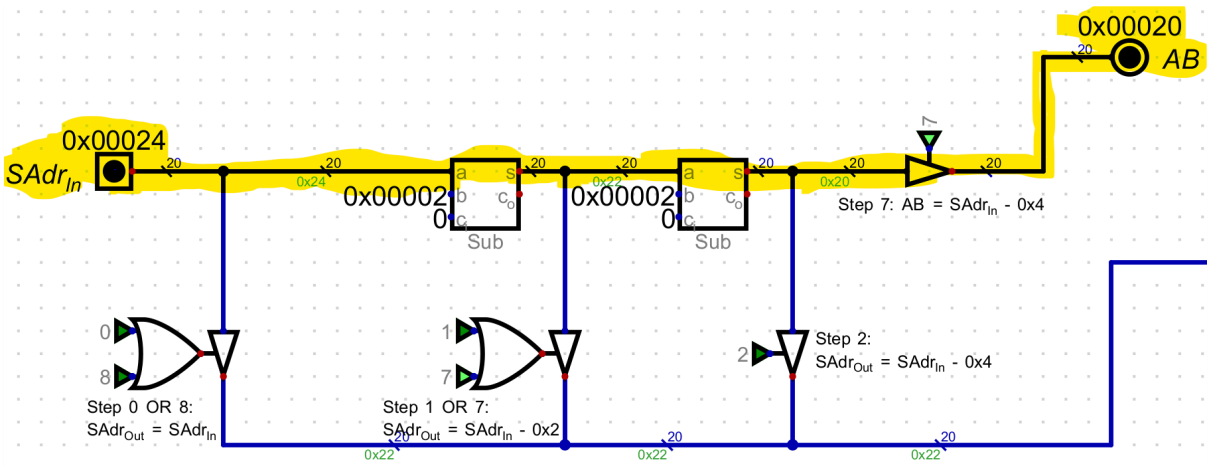



Abbildung 37: Schritt 7

### 4.9.2. Schritt 8: POP IP

In Schritt 8 wird IP vom Stack zurückgeholt. Dazu wird analog zu *Schritt 7: POP CS* vorgegangen. Die Read-Leitung wird aktiviert und die entsprechende Adresse wird auf den Adressbus gelegt. CS liegt 0x2 Stellen im RAM unter der initialen Stack-Adresse, da der PUSH-Befehl in *Schritt 2* die Stack-Adresse um 0x2 verringert hat.

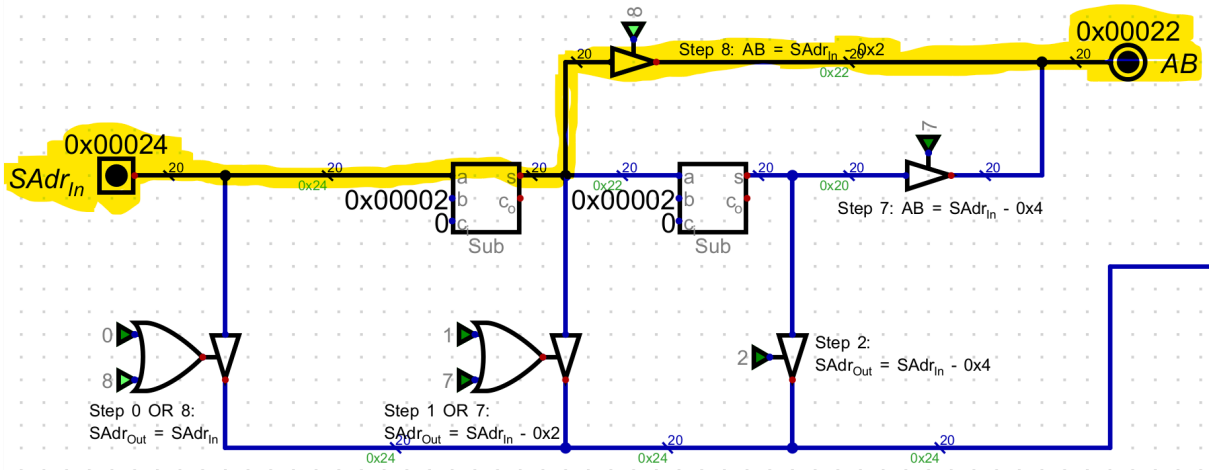



Abbildung 38: Schritt 8

### 4.9.3. Schritt 9: Ausgangszustand

Schritt 9 stellt nun den Zustand dar, in dem der 8086 sich vor dem Interrupt befand. Um das zu verdeutlichen, wird das anfangs pausierte Lied “Billie Jean” von Michael Jackson weitergespielt. Dazu wird der Enable-Eingang (en) der eingebetteten “BillieJean”-Schaltung entweder über das EN-Signal oder über Schritt 9, kurz S9, gesteuert. In der folgenden Abbildung 39 sieht man, wie “BillieJean” in Schritt 9 gespielt wird, obwohl Enable aktiv ist, wodurch “BillieJean” in dem bisherigen Entwurf von Abbildung 23 ausgeschaltet wäre.


Abbildung 39: Schritt 9

# 5. Rückblick auf die Projektplanung

Nach Fertigstellung des Projektes erfolgt ein Rückblick auf die Projektplanung. Dabei wird einerseits die Erfüllung der Anforderungsspezifikationen überprüft und andererseits die Einhaltung des Zeitplans.

## 5.1. Erfüllung der Anforderungsspezifikationen

In den folgenden Tabellen wird der Erfüllungsgrad der Kernziele (Festforderungen) sowie der Erfüllungsgrad der optionalen Ziele (Wünsche) aus der Anforderungsspezifikation bewertet.

| Forderung | Bewertung |
|---|---|
| **Kernziele (Festforderungen)** | |
| Der aktuelle Schritt der Simulation muss jederzeit klar sein | Die Forderung ist durch zwei LEDs, die den aktuellen Schritt der Simulation jederzeit anzeigen, vollständig erfüllt |
| Möglichkeit, Schritt(e) zurück zu gehen | Die Forderung ist durch die Implementierung eines Schrittzählers, der sowohl vor- als auch rückwärts zählen kann, vollständig erfüllt. |
| Signale sollen sich gerichtet auf Leitungen fortbewegen, sodass man auch deren Ursprung, Ziel und Richtung in einem Schritt erkennen und erklären kann | Diese Forderung ist durch die Signale auf oder nahe an den Busleitungen erfüllt. Hier besteht Optimierungspotenzial, da die Anzeigen immer da sind, auch dann, wenn sie nicht zur Anzeige benötigt werden, was die Simulationsoberfläche unnötig füllt. |
| Logik der Simulation soll vereinfacht und kommentiert werden, sodass ein Einarbeiten in die Simulation und weiteres Optimieren der Simulation in Zukunft einfacher ist | Diese Forderung ist teilweise erfüllt. Prozessor sowie Interrupt-Controller wurden im Rahmen dieser Arbeit maßgeblich weiterentwickelt und daher auch vereinfacht und kommentiert. Da die RAM Control-Schaltung im Rahmen dieser Arbeit nicht optimiert wurde, besteht hier noch Vereinfachungs- und Optimierungsbedarf |

Tabelle VI: Erfüllungsgrad der Kernziele (Festforderungen)

| Forderung | Bewertung |
|---|---|
| **Optionale Ziele (Wünsche)** | |
| Einbettung von Audio-Elementen in die Simulation | Diese Forderung wurde in mehrfacher Hinsicht erfüllt, wie im Kapitel *Audio* näher beschrieben. Diese Forderung könnte durch die Einbettung weiterer Audio-Elemente in die Simulation weiterverfolgt werden |
| Steuerung soll auch mit der Tastatur möglich sein (Schrittfolge, Lautstärke) | Schrittfolge und Lautstärke sind durch die Pfeiltasten der Tastatur steuerbar, wodurch die Bedienung der Simulation vereinfacht wird. Weitere Möglichkeiten zur Steuerung per Tastatur wären denkbar, wie z.B. das Triggern eines Interrupts durch die entsprechenden Nummer auf der Tastatur, bedürfen aber der Implementierung dieser Möglichkeiten in der Simulationssoftware Digital |
| Implementierung von Übungsaufgaben | Diese Forderung ist vollständig erfüllt. |

Tabelle VI: Erfüllungsgrad der optionalen Ziele (Wünsche)

### 5.2. Einhaltung des Zeitplans

Rückblickend auf den Zeitplan lässt sich sagen, dass das Projekt sowie diese Arbeit etwa zwei Wochen vor dem geplanten Projektende 25.03.22 abgeschlossen werden konnte. Dies war gewünscht, um genug Puffer für eventuell auftretende Probleme zu lassen sowie für die Klausurvorbereitung in diesen Puffer-Wochen. Der Zeitplan wurde während des gesamten Projekt als worst-case-Szenario betrachtet und sollte daher geschlagen werden. Mit der Fertigstellung des Projekts am 14.03.22 wurde dieses Ziel erreicht. Insbesondere für die *Phase 3: Vereinfachen und Verbessern* wurde rückblickend zu viel Zeit eingeplant. Es zeigte sich, dass das Vereinfachen und Verbessern der bestehenden Simulation bereits während der Einarbeitung in die bestehende Simulation stattfand. Auch die Erstellung von Aufgaben benötigte statt wie ursprünglich geplant rund ein Woche tatsächlich weniger als einen Tag und verkürzte somit den Zeitplan.

# 6. Fazit und Ausblick

Bei einem Blick auf den *Stand der Technik* wurde eine bestehende Simulation der Interruptverarbeitung des Intel 8086 als Basis für dieser Arbeit vorgestellt. Dabei wurde die *Funktionsweise der bestehende Simulation* Schritt für Schritt genauestens analysiert.

Aus dieser fundierten Analyse heraus erfolgte die Feststellung und Beschreibung von einigen *Optimierungsmöglichkeiten der bestehenden Simulation* in verschiedenen Kategorien. Dabei wurden nur beispielhaft einige der wichtigsten umgesetzten Optimierungen angesprochen, da eine vollständige Beschreibung und Diskussion der Optimierungen im Rahmen dieser Arbeit nicht möglich ist. An einigen Stellen wurden Argumente für und gegen die vorgeschlagenen Optimierungsideen abgewogen und die Entscheidung begründet. Als sinnvoll erachtete Optimierungen wurden umgesetzt und die Implementierung beschrieben, sodass diese Arbeit bei einer eventuellen weiteren Überarbeitung der Simulation als Referenz herangezogen werden kann.

Abschließend erfolgte ein *Rückblick auf die Projektplanung*, in dem die *Erfüllung der Anforderungsspezifikationen* sowie die *Einhaltung des Zeitplans* überprüft und reflektiert wurde.

Die ursprüngliche Simulation wurde an vielen weiteren Punkten optimiert, die im Rahmen dieser Arbeit nicht näher angesprochen werden konnten. Zu diesen Änderungen zählen z.B. die Vereinfachungen und Kommentare der Schaltung. So wurden besonders viele Verzögerungs-Bausteine, deren Zweck nicht ersichtlich oder nicht länger gegeben war, entfernt oder durch andere Logiken ersetzt. Der Interrupt-Controller konnte derart vereinfacht werden, dass er vollständig auf einen Clock-Eingang verzichten kann. Eine weitere Änderung ist die Einstellung der Oberfläche als schreibgeschützt, damit Benutzer der Simulation diese nicht versehentlich zerstören können. In der Anleitung der ursprünglichen Simulation musste dieser Hinweis noch als Warnung an Benutzer aufgenommen werden, da die Oberfläche nicht schreibgeschützt eingestellt war [Gerhard, 2021, S. 3].

Eine andere Änderung, die im Rahmen dieser Arbeit nicht thematisiert wurde, ist das standardmäßige Anlegen einer IVT im RAM bei Start der Simulation mit einigen Hex-Codes.

Normalerweise initialisiert der Schaltungssimulator Digital alle Einträge eines RAM-Bausteins bei Start der Simulation mit Nullen [Neemann, 2022, S. 29/110]. Damit dennoch eine Standard-IVT im RAM steht, ohne diese von Hand anlegen zu müssen, werden einige Hex-Codes bei Start der Simulation aus der mitgelieferten IVT.hex-Datei gelesen und an die Stellen der IVT im RAM geschrieben. Auf diese Weise werden in *Schritt 4* und *Schritt 5* für IP und CS Standardwerte aus der IVT.hex Datei geladen.

Als Ansatzpunkte für weitere Forschung lassen sich einige Punkte festhalten, die im Rahmen dieser Arbeit identifiziert, aber nicht umgesetzt wurden. So kann die RAM Control Schaltung vereinfacht werden, um diese Schaltung lesbarer und verständlicher zu machen und somit künftige Optimierungen dieser Schaltung erst zu ermöglichen. Andere Möglichkeiten zur weiteren Forschung sind aus der Reflektion über die *Erfüllung der Anforderungsspezifikationen* ersichtlich. So lassen sich Richtung, Ursprung und Ziel von Signalen auf Busleitungen durch die implementierten Anzeigen auf oder nahe an den Busleitungen verfolgen. Hier besteht allerdings Optimierungspotenzial, da die Anzeigen immer da sind, auch dann, wenn sie nicht zur Anzeige benötigt werden, was die Simulationsoberfläche unnötig füllt. Außerdem springen die Signale von Anzeige zu Anzeige, statt auf den Leitungen zu fließen. Zur Lösung dieser Probleme könnte die Implementierung eigener Digital-Bausteine in Java zielführend sein [Neemann, 2020].

Diese Arbeit hat die Schaltungen des Prozessors sowie des Interrupt-Controllers weiterentwickelt und dazu auch vereinfacht und kommentiert. Da die RAM-Control-Schaltung im Rahmen dieser Arbeit nicht optimiert wurde, könnte hier im Rahmen weiterer Forschung angesetzt werden. Denkbar wäre auch eine vollständige Eliminierung der RAM-Control-Schaltung und die vollständige Verlagerung der RAM-Kontrolle in den Prozessor hinein. Auch der in der Realität übliche Einsatz von zwei RAM-Blöcken, einer für gerade und einer für ungerade Adressen, könnte simuliert werden [Plusquellic, 2002].

# Literaturverzeichnis


Dan Howley, Goldman Sachs & Julie Hyman, Yahoo Finance. (2021, April 26). *Global chip shortage hit these 169 industries: GS*. Yahoo Finance. Retrieved March 8, 2022, from https://finance.yahoo.com/video/global-chip-shortage-hit-169-133627095.html

Forsythe, A. (2022). *Code Blocks - Syntax highlighting for Google Docs*. https://www.alexwforsythe.com/code-blocks/

Gerhard, S. (2021). *Bedienungsanleitung für die Simulation der Hardware-Interruptverarbeitung des Intel 8086 Prozessors*.

Gerhard, S. (2021). *Visualisierung der Hardware-Interruptverarbeitung eines Mikrocomputers zu Lehrzwecken mittels einer Software zur Simulation von digitalen Schaltungen*.

Glenn, W. (2016, March 10). *How to Adjust the Volume for Individual Apps in Windows*. How-To Geek. Retrieved March 5, 2022, from https://www.howtogeek.com/244963/how-to-adjust-the-volume-for-individual-apps-in-windows/

Hermes, D., & Neemann, H. (2022, 02 08). *Opening a .dig file with doubleclick or command line · Discussion #916 · hneemann/Digital*. https://github.com/hneemann/Digital/discussions/916

Intel Corporation. (October 1979). *The 8086 Family Users Manual*. http://bitsavers.informatik.uni-stuttgart.de/components/intel/8086/9800722-03_The_8086_Family_Users_Manual_Oct79.pdf

The MIDI Association. (2021). *Official MIDI Specifications*. https://www.midi.org/specifications

Neemann, H. (2020, 04 18). *hneemann/digitalCustomComponents: An example for the implementation of components in java which can be used in Digital.* GitHub. Retrieved March 12, 2022, from https://github.com/hneemann/digitalCustomComponents

Neemann, H. (2022). *Digital Dokumentation v0.29*. https://github.com/hneemann/Digital/releases/download/v0.29/Doc_Deutsch.pdf

Neemann, H. (2022). *hneemann/Digital: A digital logic designer and circuit simulator.* GitHub. https://github.com/hneemann/Digital

Plusquellic, J. (2002). *8086 - 80386SX 16-bit Memory Interface*. CMPE 310: Systems Design and Programming. Retrieved 03 12, 2022, from https://ece-research.unm.edu/jimp/310/slides/8086_memory3.html

Radice et al. (1985). *The ETVX modelling technique*. ResearchGate. Retrieved March 10, 2022, from https://www.researchgate.net/figure/7-The-ETVX-modelling-technique-Radice-et-al-1985_fig11_243765439

# Anhang 1:

# Projektkonzept

# Projektkonzept

## Simulation Interruptverarbeitung

## Intel 8086

1. **Anforderungsspezifikation**
    a. Definition der Kernziele (Festforderungen): Optimierung der bestehenden Simulation auf Lehrzwecke

- Der aktuelle Schritt der Simulation muss jederzeit klar sein
- Möglichkeit, Schritt(e) zurück zu gehen
- Signale sollen sich gerichtet auf Leitungen fortbewegen, sodass man auch deren Ursprung, Ziel und Richtung in einem Schritt erkennen und erklären kann
- Logik der Simulation soll vereinfacht und kommentiert werden, sodass ein Einarbeiten in die Simulation und weiteres Optimieren der Simulation in Zukunft einfacher ist

    b. Optionale Ziele (Wünsche)

- Einbettung von Audio-Elementen in die Simulation
- Steuerung soll auch mit der Tastatur möglich sein (Schrittfolge, Lautstärke)
- Implementierung von Übungsaufgaben

c. Dateien und Dokumente

| Name | Beschreibung | Format |
|---|---|---|
| Simulation Interruptverarbeitung Intel 8086 | Simulation zur Visualisierung der Interruptverarbeitung eines Intel 8086 Prozessors nach den Anforderungsspezifikationen | Anwendung bzw. Simulation in einer Anwendung, inkl. aller nötiger Komponenten Für Windows-Nutzer zusammengefasst in einer .exe-Datei |
| Aufgaben zur Simulation | Aufgaben für Studenten, die mithilfe der Simulation die Interruptverarbeitung des Intel 8086 erlernen | Textdatei |

## 2. ETVX-Modell

Das ETVX-Modell dient zur Modellierung der Phasen eines Software-Entwicklungsprozesses und identifiziert dabei vier Aspekte bei der Modellierung einer Entwicklugsphase (Radice et al., 1985):

- **E**ntrance Criteria: Kriterien, die erfüllt sein müssen, damit die Phase gestartet werden kann
- **T**asks: Die zu erledigenden Aufgaben in dieser Phase
- **V**erification/Validation: Tests, die die erfolgreiche Fertigstellung der Aufgabe überprüfen und sicherstellen
- **E**xit: Bedingungen zum Abschluss der Phase

Das ETVX-Modell kann graphisch folgendermaßen dargestellt werden:

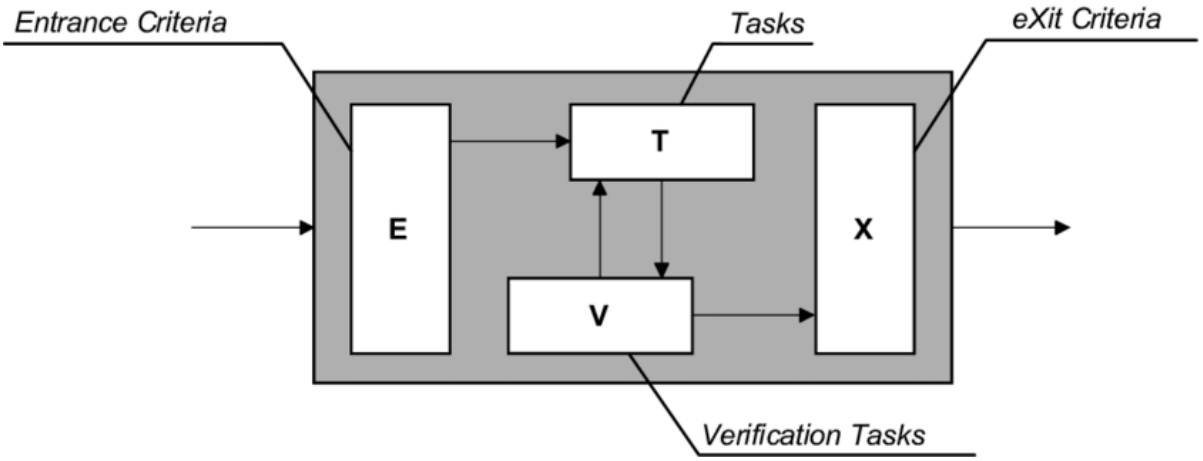



Abbildung 1: ETVX-Modell (Radice et al, 1985) (https://www.researchgate.net/figure/7-The-ETVX-modelling-technique-Radice-et-al-1985_fig11_243765439)

Eine Projektphase wird gestartet, wenn alle Eingangskriterien (Entry Criteria) erfüllt sind. Die Aufgaben (Tasks) werden bearbeitet und anschließend verifiziert und validiert. Sind Verifizierung und Validierung bestanden, kann die Prozessphase durch das Erfüllen aller Ausgangskriterien abgeschlossen werden. Andernfalls müssen die Aufgaben (Tasks) erneut bearbeitet werden, bis die Verifizierung und Validierung bestanden wird.

Folgende Prozessphasen werden definiert:

| Phase | I | II |
|---|---|---|
| Name | Recherche & Einarbeitung | Projektvorbereitung & Planung |
| Eingangskriterium/en | Mit Beginn des Projekts | Abschluss von Phase I |
| Aufgabe(n) | Intel 8086 recherchieren<br><br>Dokumentation der bestehenden Simulation lesen<br><br>In bestehende Simulation einarbeiten | Projektkonzept, Anforderungsspezifikationen und Zeitplan erstellen<br><br>Projekttitel auswählen |
| Verifikation und Validierung | Ist die Interruptverarbeitung des Intel 8086 verstanden?<br><br>Kann mit der bestehenden Simulation die Interruptverarbeitung des Intel 8086 nachvollzogen werden? | Sind Anforderungen und Zeitplan realistisch umsetzbar?<br><br>Ist der Arbeitsaufwand realistisch und absehbar?<br><br>Spiegelt der Projekttitel die Ziele wieder? |
| Ausgangskriterium/en | Keine offenen Fragen bezüglich der Interruptverarbeitung des Intel 8086 und der Funktion der bestehenden Simulation. | Der zeitliche Ablauf des Projekts ist vollständig festgelegt.<br><br>Es besteht eine genaue Vorstellung über den Arbeitsaufwand. |

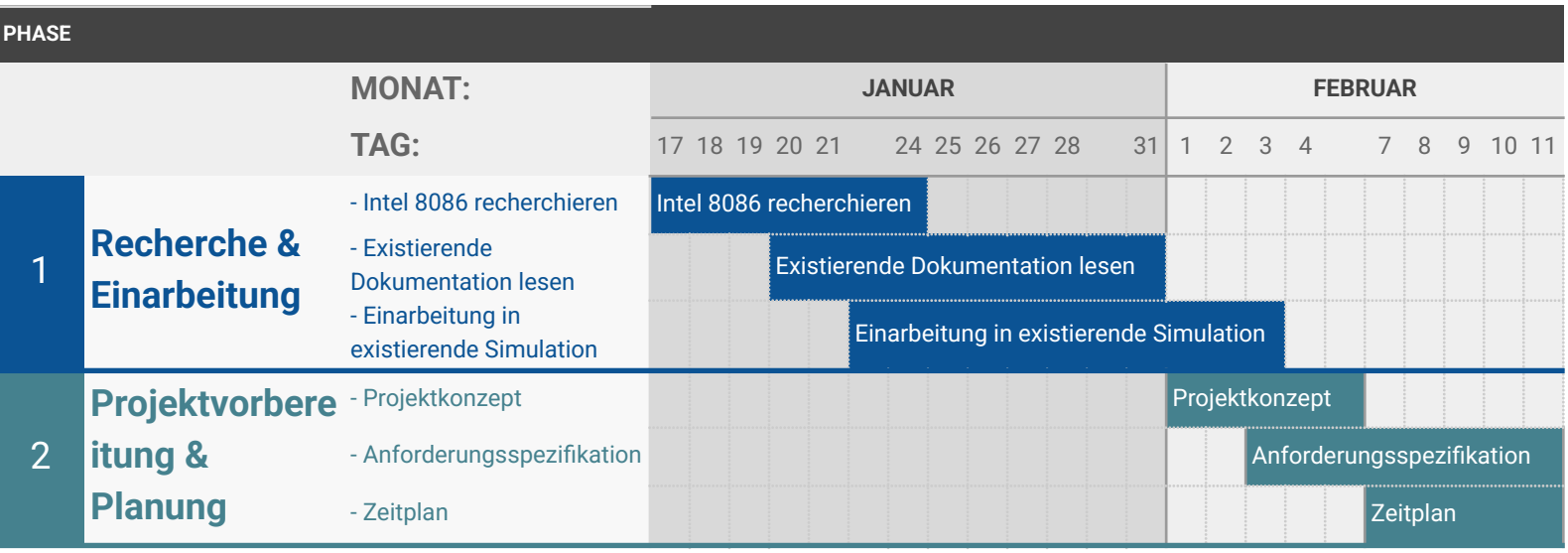
| Phase | III | IV |
|---|---|---|
| Name | Vereinfachen & Verbessern | Umsetzung & Tests |
| Eingangskriterium/en | Abschluss von Phase II | Abschluss von Phase III |
| Aufgabe(n) | Vereinfachen & Kommentieren der existierenden Simulation<br><br>Identifizierung von Verbesserungsmöglichkeiten | Besprechung & Umsetzung der Verbesserungsmöglichkeiten<br><br>Erstellen von Aufgaben,<br><br>Tests & Optimierungen |
| Verifikation und Validierung | Sind die Hauptteile der Schaltungssimulation vereinfacht & kommentiert, sodass deren Logik von nachfolgenden Studenten nachvollzogen werden kann?<br><br>Ist überflüssige Schaltungslogik eliminiert?<br><br>Sind Verbesserungsmöglichkeiten klar? | Ist die Priorität der Verbesserungsvorschläge klar?<br><br>Sind die Aufgaben nützlich zum Verständnis der Interruptverarbeitung des Intel 8086 und sind die Aufgaben lösbar?<br><br>Funktionieren die Kernfunktionen der ursprünglichen Simulation sowie die hinzugekommenen Verbesserungen? |
| Ausgangskriterium/en | Die Schaltung ist soweit vereinfacht & kommentiert, dass nachfolgende Studenten die Logik nachvollziehen können.<br><br>Überflüssige Schaltungslogik ist eliminiert.<br><br>Es ist hinreichend Platz zur Implementierung weiterer Funktionen und Verbesserungen vorhanden. | Die Simulation ist in den besprochenen Punkten verbessert und erweitert worden.<br><br>Die Aufgaben erfüllen ihren Zweck.<br><br>Die Simulation funktioniert und verhält sich in jeder Situation wie erwartet. |

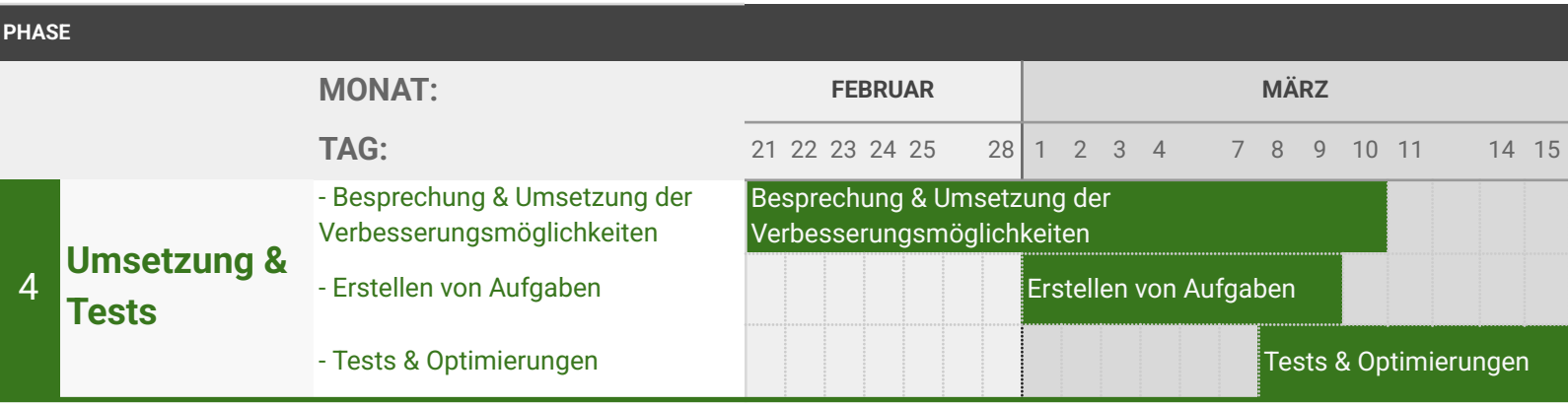
| Phase | V |
|---|---|
| Name | Abschluss |
| Eingangskriterium/en | Abschluss von Phase V |
| Aufgabe(n) | Evaluation der weiterentwickelten Simulation<br><br>Überprüfung aller Ziele<br><br>T3_3100 Studienarbeit schreiben |
| Verifikation und Validierung | Ist die weiterentwickelte Simulation ein wesentlicher Fortschritt gegenüber Version 1?<br><br>Funktioniert die Schaltung insgesamt?<br><br>Sind die Projektziele erreicht?<br><br>Ist die T3_3100 Studienarbeit zur Abgabe bereit? |
| Ausgangskriterium/en | Die weiterentwickelte Schaltung ist wesentlich leichter zu bedienen.<br><br>Selbst Nutzern, die die Simulation zum ersten Mal sehen ist immer sofort klar, in welchem Schritt sich die Simulation befindet.<br><br>Die weiterentwickelte hilft Studenten beim Verständnis der Interruptverarbeitung des Intel 8086 mehr als Version 1.<br><br>Die T3_3100 Studienarbeit ist sowohl digital als auch in Papierform abgegeben. |

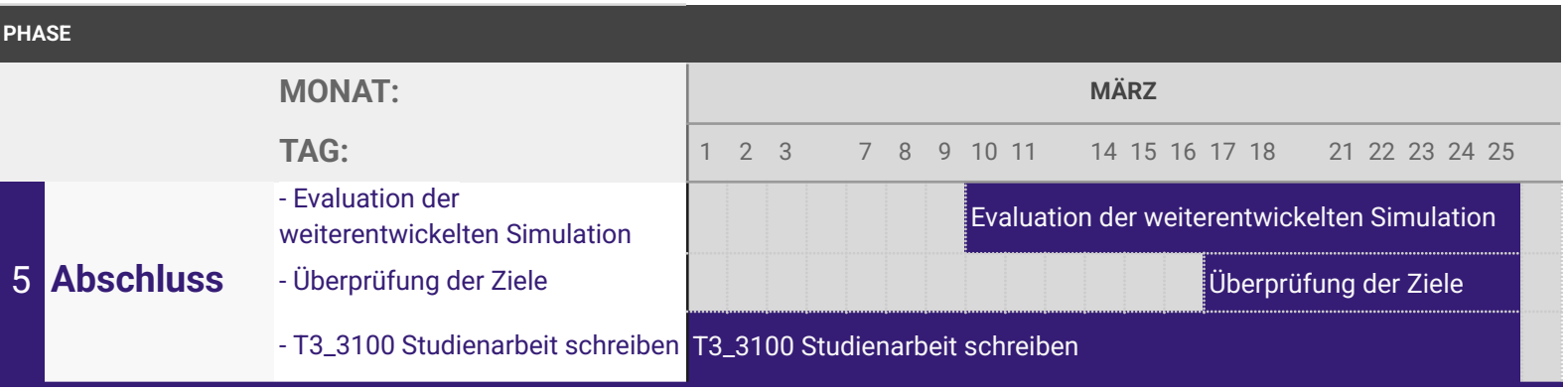
### 3. Zeitplanung

# PROJEKT ZEITPLAN

| | |
|---|---|
| **PROJEKT TITEL** | Simulation Interruptverarbeitung Intel 8086 |
| **PROJEKT MANAGER** | Daniel Hermes |
| **AUFTRAGGEBER** | DHBW Mannheim |
| **SUPERVISOR** | Dipl. Ing. Gerhard Lehmann |




**4. KPIs (Key Performance Indicators)**

- Der Zeitplan wird eingehalten
- Jedem Benutzer ist jederzeit der aktuelle Schritt klar (auch Benutzern, die die Simulation zum ersten Mal sehen)
- Man kann in der Simulation beliebig viele Schritte vor und zurück gehen und die Simulation funktioniert weiterhin wie erwartet
- Signale auf Bussen “wandern” in eine eindeutig Richtung
- Die Logik der Schaltung ist vereinfacht und kommentiert