PSALM

Projektseminar asynchron laufender Mikroprozessor

# Eine asynchrone Implementierung eines Microprozessors auf einem FPGA


| | |
|---|---|
| Marco Balke | mbalke@techfak.uni-bielefeld.de |
| Thomas Dettbarn | tdettbar@techfak.uni-bielefeld.de |
| Robert Homann | rhomann@techfak.uni-bielefeld.de |
| Sebastian Jaenicke | sjaenick@techfak.uni-bielefeld.de |
| Tim Köhler | tkoehler@techfak.uni-bielefeld.de |
| Henning Mersch | hmersch@techfak.uni-bielefeld.de |
| Holger Weiss | hweiss@techfak.uni-bielefeld.de |

Betreut von Ingo Glöckner


16. Oktober 2001

## Ein Dankeschön an

**Ingo Glöckner** für sein Engagement für *PsalM* weit über seine Vertragslaufzeit hinaus.

**Jan Krüger und Thomas Schmitt** für die große Hilfe mit der Xilinx Software und der FPGA-Karte.

**Garvin Gripp** für 1.5GB RAM.

# Inhaltsverzeichnis

# Kapitel 1

# Motivation und Ziele

## 1.1 Motivation

Die bisherige Technologie zur Chipentwicklung stößt langsam aber sicher an ihre theoretischen Grenzen. Um weitere Fortschritte zu erzielen und die rasante Entwicklung im Sinne von Moore's Gesetz[1] [MOORE] fortzusetzen, wird derzeit in unterschiedlichen Richtungen geforscht.

Ein möglicher Lösungsweg besteht im Übergang zu asynchronen Prozessoren. Dieser Ansatz, der bereits von von Neumann verfolgt wurde, geht effizienter mit dem wertvollen Gut Zeit um (siehe dazu [HAUCK]). Der Gewinn liegt in dem Grundgedanken der synchronen CPUs. Der für diese notwendige Takt kann nur soweit gesteigert werden, daß trotzdem sichergestellt ist, daß alle Gatter die Möglichkeit haben, durchzuschalten. Also ist die Reihe der längsten Leitungsstrecke der zeitbestimmende Faktor für alle Takte. Bei synchronen CPUs ergeben sich die Geschwindigkeitssteigerungen ausschließlich durch genauere und kleinere Implementierungen. Dadurch wird der Weg, den die Elektronen nehmen müssen, kürzer, Gatter schalten schneller durch, und letztendlich können die CPUs schneller getaktet werden.

Anders bei einem asynchronen Prozessor: Hier geben die Gatter oder Baugruppen von Gattern selbst ein Signal weiter, wenn sie fertig sind. Dadurch kann verhindert werden, daß in einem Durchlauf länger gewartet wird als nötig. Weiterhin wirken sich hier Weiterentwicklungen einzelner Funktionsgruppen direkt auf die Geschwindigkeit aus. In technische Hinsicht können asynchrone Prozessoren durch unterschiedliche Ansätze realisiert werden, von denen wir zwei untersucht haben:

- Dual-Rail
- Micro-Pipelining

Der entscheidene geschwindigkeitsbestimmende Faktor ist somit die Leitungsstrecke und die Schaltgeschwindigkeit der Gatter auf diesen Strecken, die durch neue Implementierungsmethoden immer noch in gewissen Grenzen weiter gesteigert werden kann

[1]Gordon E. Moore sagte 1965 voraus, daß sich die Anzahl der in einem integriertem Schaltkreis verwendeten Gatter alle 18 Monate verdoppeln würde, was bis heute auch tatsächlich der Fall ist.

(aktuell: $18\mu m$-Fertigung in Kupfer). So wirkt sich auch die Temperatur auf heutige Gatter direkt aus. Aufgrund geänderter atomphysikalischer Eigenschaften können die Bauteile bei geringeren Temperaturen Informationen schneller übermitteln. Bei asynchronen CPUs wird dieser eklatante Unterschied unmittelbar ausgenutzt, die Verarbeitungsgeschwindigkeit paßt sich der Umgebungstemperatur automatisch an, niedrige Temperaturen schlagen sich also in höherer Geschwindigkeit nieder. Anders als bei synchronen CPUs, bei denen sich die Taktfrequenz an der Schaltgeschwindigkeit der verwendeten Gatter und den Signallaufzeiten (bis zu einer zu spezifizierenden maximalen Arbeitstemperatur) orientieren muß.

Außer dem hier behandelten Ansatz der asynchronen Prozessoren, gibt es weitere auch schon in der Praxis relevante Techniken zur Leistungssteigerung wie zum Beispiel die sogenannte rekonfigurierbare Logik. Bei diesem Ansatz wird versucht, die Software- mit der Hardwareseite zu verschmelzen. Dabei werden die Grundelemente so konstruiert, daß sie nachträglich programmiert werden können und sich dann wie ein oder mehrere Gatter verhalten. Diese sogenannten CLBs (Configurable Logic Block) werden dann zu Netzen verschaltet und verhalten sich als Gruppe wie ein konventioneller Schaltplan.

## 1.2 Ziele

Asynchrone Prozessoren sind ein vielversprechender Ansatz zur Leistungssteigerung von Microprozessoren. Desweiteren gibt es die Technik der rekonfigurierbaren Logik, z. B. FPGAs, welche die Grenze zwischen Hard- und Software aufhebt. Das Ziel von *PsalM* (**P**rojekt**s**eminar **a**synchron **l**aufender **M**icroprozessor) ist eine Untersuchung der Kombination beider Techniken. Dabei sollen grundlegende Probleme gelöst werden und untersucht werden, ob sich Vorteile bei der Umsetzung von asynchronen Prozessoren auf FPGAs ergeben.

Während der Untersuchung der Einsatzmöglichkeiten asynchroner Technologie auf FPGAs wird versucht, eine realistische Einschätzung dieser Methodik zu erreichen. Im Vordergrund stehen insbesondere die Eignung von FPGAs als Basis für asynchrone CPUs, für die unterschiedliche Ansätze existieren. Aufgrund der Eigenschaften von FPGAs, logische Funktionen zur Verfügung zu stellen, sollen Grundbausteine wie Speicher, logische Gatter, Multiplexer etc. entwickelt werden, die in beliebigen asynchronen Schaltungen zum Einsatz kommen können.

Um das Ziel zu erreichen, wird zunächst eine kleinere CPU mit acht Befehlen und zwei Adressierungsarten implementiert, die sogenannte „DEP-CPU“. Sie soll zur Entwicklung verschiedener Ansätze für die Realisierung asynchroner Schaltungen auf FPGAs, deren Geschwindigkeit und Größenbedarf herangezogen werden. Außerdem soll eine Abschätzung der beiden zur Verfügung stehenden Entwicklungsmöglichkeiten mittels des *Schematic Editors* von *Xilinx, Inc.* und der Hardwarebeschreibungssprache *VHDL* ermöglichen.

Der für das Vorhaben beste Ansatz soll dann exemplarisch für die Implementierung eines binärcodekompatiblen Klons des wohlbekannten 6502-Mikroprozessors getestet werden.

# Kapitel 2

# Entwicklungsumgebung

Bevor auf die Ansätze zur Realisierung asynchroner Prozessoren und deren beispielhafte Implementierung eingegangen wird, soll hier eine Darstellung der für das Vorhaben verfügbaren Entwicklungsumgebungen erfolgen. Dies ist erforderlich, da die für die Implementierung praktisch einsetzbaren Verfahren entscheidend davon abhängen, in welchem Maße die Entwicklungsumgebung die erforderlichen Ausdrucksmittel bietet und eine Spezifikation auf einer geeigneten Abstraktionsebene unterstützt. Desweiteren ist die konkrete Implementierung der Ansätze eng an die verfügbare Hardware gebunden, d.h. hier die verfügbaren FPGA-Bausteine und die Art von deren äußerer Beschaltung in der Hardware-Karte.

## 2.1 Software: *Schematic Editor* von *Xilinx, Inc.*

Der *Schematic Editor* ist Teil der *Xilinx Foundation Series 3.1i* und vollständig integriert in diese Entwicklungsumgebung. Er ermöglicht den Schaltungsentwurf über eine graphische Oberfläche, an der die einzelnen Bauteile per Maus auf einem virtuellen Reißbrett angeordnet und verbunden werden. Beim Zeichnen der Verbindungen übernimmt die Software die Anordnung der Leitungen, was die Arbeit sehr erleichtert.

Die grundlegenden Logikgatter, Flip-Flops, Buffer und Ähnliches sind schon als fertige Bauteile vorhanden und können in eigenen Schaltungen benutzt werden. Die so erzeugten Schaltungen können zu einem Bauteil gekapselt werden und dann als „Blackbox" mit den Ein- und Ausgabeleitungen in anderen (auf einer hhöheren Hierarchieebene) Schaltungen benutzt werden.

## 2.2 Die Alternative: *VHDL*

Die Sprache *VHDL* dient der Beschreibung und Simulation digitaler Systeme und deren Umgebung. Die Unterteilung des Entwurfs in (Teil-)Komponenten wird über mehrere Hierachie- und Abstraktionsebenen unterstützt.

Die Entwürfe werden in sogenannten Bibliotheken organisiert, die folgende vier Teile enthalten (können):

| | |
|---|---|
| package: | globale Deklaration |
| entity: | Design Sicht von außen |
| architecture: | Design Implementation |
| configuration: | Festlegung einer Designversion (Zuordnung entity-architecture) |

### 2.2.1 Hierarchie

#### package

In `packages` werden Deklarationen zusammengefaßt, die in mehreren Designs benutzt werden, z.B. für Komponenten einer Bibliothek, Unterprogramme, Typen und Konstanten.

#### entity

Eine `entity` definiert für eine Komponente des Entwurfs dessen externe Sichtweise. Dabei werden der Name, die Ein- und Ausgänge und zusätzliche Deklarationen festgelegt. Nach der Übernahme in eine Bibliothek kann eine `entity` dann als Teil anderer Entwürfe benutzt werden.

#### architecture

Eine `architecture` beschreibt das Verhalten, bzw. die interne Realisierung einer vorher beschriebenen `entity`. Dabei können mehrere Architekturen für eine `entity` deklariert werden.

#### configuration

Durch eine `configuration` kann zwischen verschiedenen Design-Alternativen und -Versionen ausgewählt werden. Dabei bestimmt eine `configuration`, welche Realisierung von möglicherweise mehreren vorhandenen `architectures` für eine `entity` aus der Bibliothek bei dessen Instantiierung benutzt wird.

### 2.2.2 Zeitliche Kontrolle

Bei der Einführung des Zeitbegriffes innerhalb der Simulation, muß man unterscheiden zwischen der Art und Weise, wie Anweisungen im *VHDL*-Code sequentiell abgearbeitet werden und dem Verstreichen von simulierter Zeit.

#### Verhaltensbeschreibung

Wie in Programmiersprachen, wird das Verhalten durch einen *sequentiellen* Ablauf von Anweisungen beschrieben. Die Abarbeitung dieser Anweisungen beeinflußt die simulierte Zeit nicht.

**Strukturbeschreibung**

Entsprechend realen Schaltungen sind alle Strukturelemente *parallel* aktiv, eine Reihenfolge der entsprechenden Anweisungen im Code ist irrelevant. Die Verzögerung der einzelnen Elemente bestimmt die simulierte Zeit.

## 2.3 *Schematic Editor* vs. *VHDL*

Die beiden Möglichkeiten des Schaltungsentwurfs, *Schematic Editor* und *VHDL* haben beide ihre Vorzüge und Nachteile, von denen hier die wichtigsten aufgelistet sind.

**pro *Schematic Editor***

- übersichtlicher bei kleinen Schaltungen
- weniger Lernaufwand

**pro *VHDL***

- *VHDL*-Code kann man durch andere Programme generieren
- maschinell einfacher zu verarbeiten
- übersichtlicher bei großen Schaltungen
- keine Probleme mit nicht verbundenen Leitungen

Da der 6502-Prozessor nicht aus kleinen Schaltungen konstruierbar ist, und unser Decoder mit Hilfstools generiert werden sollte, fiel unsere Wahl auf die Beschreibungssprache *VHDL*.

## 2.4 Hardware

Hardwareseitig stützt sich unser Projektseminar *PsalM* auf PCI-Karten der Virtual Computer Corporation, Reseda, die mit FPGAs der Firma *Xilinx, Inc.* bestückt sind. Zur Verfügung standen uns zwei verschiedene Karten, von denen die von der Kapazität her kleinere mit einem XC4062XLT, die größere mit einem XC4085XLA FPGA bestückt ist.

Als weitere Bestandteile besitzen die Karten Speicher in Form von 4MB SRAM, 2MB Configuration Flash und 512kB Cache RAM, zwei Erweiterungsstecker sowie einen programmierbaren Taktgenerator, der Frequenzen von 360kHz bis zu 100MHz unterstützt. Abgesehen von der Bestückung mit unterschiedlichen FPGA-Chips sind die Karten baugleich und werden von der Virtual Computer Corporation unter dem Namen „HOT2-XL“ vertrieben. Die Ansteuerung der Karten erfolgt über den PCI-Bus des Host-Computers, dies ermöglicht eine schnelle Rekonfiguration des FPGA-Bausteins ohne zusätzliche externe Kabel. Desweiteren kann auf diesem Wege die Steuerung durch auf dem Computer entwickelte Software erfolgen, vorausgesetzt es wurde ein passendes Interface hierfür auf dem FPGA implementiert. Zu diesem Zweck wird aber von *Xilinx, Inc.* bereits ein prototypisches Interface zur Verfügung gestellt, das relativ leicht an die gegebene Anwendung angepaüt werden kann.

Das Kernstück der PCI-Karten, die *Xilinx, Inc.* FPGA Chips, bestehen im wesentlichem aus zwei verschiedenen Typen von konfigurierbaren Elementen. Dies sind zum einen die konfigurierbaren logischen Blöcke (Configurable logic blocks, CLB), zum anderen Input-/Output-Blöcke (IOB).

Die CLBs stellen die grundlegenden funktionalen Elemente zur Konstruktion der vom Benutzer definierten logischen Schaltungen bereit, die IOBs dienen als Interface zwischen den externen Anschlüssen und den intern genutzten Signalleitungen.

Zusätzlich sind auf dem Chip noch Tristate-Buffer vorhanden, die mit jedem einzelnen CLB verbunden sind, sogenannte „Wide Edge decoder", die sich in der Peripherie des FPGAs befinden, und ein Oszillator.

Während der Rekonfiguration wird das Verhalten jedes einzelnen Elementes auf dem FPGA durch die Programmierung interner statischer Speicherzellen definiert.

### CLB

Die CLBs stellen den Großteil der Logik auf dem FPGA dar. Ein CLB besteht aus zwei Funktionsgeneratoren mit je vier Eingängen, sowie einem dritten nachgeschalteten Funktionsgenerator mit drei Eingängen, die die Implementation der meisten kombinatorischen logischen Funktionen ermöglichen. Hierbei können bis zu zwei Eingänge des dritten Generators mit Ausgängen der ersten beiden Generatoren verknüpft sein, die anderen Eingänge stammen von außerhalb des CLBs. Somit kann ein CLB als Grundeinheit Funktionen mit bis zu neun Eingängen implementieren. Die Funktionsgeneratoren sind durch sogenannte Memory Look-up Tables (LUTs) implementiert, so daß die Durchschaltzeit unabhängig von der Art der implementierten logischen Funktion ist. Jeder CLB enthält desweiteren zwei Speicherelemente, die entweder unabhängig von den anderen Bestandteilen des CLBs verwendet werden können oder aber auch, um die Ergebnisse an den Ausgängen der Funktionsgeneratoren zwischenzuspeichern. Es handelt sich hierbei um zwei pegelgesteuerte D-Flip-Flops. Jede logische Zelle ist mittels acht Leitungen mit den vier benachbarten Zellen verbunden („shortlines"), dazu kommen noch zwei Leitungen je Zelle, die mit allen Zellen auf der X- bzw. Y-Achse verbunden sind („longlines").

### IOB

Die IOBs stellen das Interface zwischen FPGA und Außenwelt dar, jeder IOB kontrolliert einen Pin des Chips in drei verschiedenen Ansteuerungsarten, so daß jeder Pin als Input, Output oder bidirektionales Signal genutzt werden kann. Die Eingänge des Chips sind sowohl TTL- als auch (abhängig von der Art des verwendeten FPGAs) CMOS-kompatibel konfigurierbar, allerdings nur global und nicht unabhängig voneinander. Die Hysterese der Eingänge beträgt in etwa 300mV. Ein IOB besitzt zwei Eingänge, die intern an je einem Register enden, das je nach Konfiguration als flankengesteuertes Flip-Flop oder als pegelsensitives Latch fungiert. Der Dateneingang der Register kann optional um einige Nanosekunden verzögert werden, um unzuverlässige Signale zu vermeiden, die durch Temperaturschwankungen oder auch logikbedingt auftreten können.

Die Ausgangssignale eines IOBs können optional IOB-intern invertiert werden; auch können sie entweder direkt weitergegeben oder in einem flankengesteuerten Flip-Flop zwischengespeichert werden. Mit Hilfe eines Tristate-„Output buffers“, der in einen hochohmigen Zustand versetzt werden kann, erfolgt die Kontrolle darüber, ob der Ausgang als einfacher Ausgang oder bidirektional genutzt werden soll.

Je nach FPGA kann ein IOB auch noch einen Multiplexer enthalten, der auch als 2-bit Funktionsgenerator konfiguriert werden kann. Hinzu kommen noch einige Widerstände zwischen 50 und 100k$\Omega$, die dazu verwendet werden, unbenutzte Pins auf Masse zu legen, um den Stromverbrauch so gering wie möglich zu halten und störende Einflüsse zu minimieren. Diese sind notwendig, da nach der Konfiguration des FPGAs an allen Pins im Sinne der TTL-Logik gültige Signale anliegen müssen.

#### Tristate-Buffer

Von den auf dem FPGA befindlichen Tristate-Buffern sind immer jeweils zwei mit einem CLB verbunden. Sie werden in Verbindung mit den „longlines“ zur Implementation gemultiplexter oder bidirektionaler Bus-Leitungen verwendet, um hierfür keine CLBs verwenden zu müssen.

#### Wide Edge Decoder

Kleine Decoder mit bis zu neun Eingängen können problemlos innerhalb eines CLBs realisiert werden, bei größeren Decodern müßte die Decodierung in mehreren Stufen und auf mehrere CLBs verteilt erfolgen, was einen Geschwindigkeitsnachteil darstellen würde. Daher haben FPGAs der XC4000 Baureihe vier programmierbare Decoder, die sich in den vier Ecken des Chips befinden. Die Eingänge dieser Decoder können sowohl Ausgänge der IOBs in dieser Region des Chips als auch Interconnect-Leitungen der CLBs sein. Die Anzahl der Decodervariiert hier ebenfalls mit den verschiedenen Ausführungen der Chips, z.B. 96 Eingängen beim XC4028X und 132 beim XC4052X.

#### On-Chip Oszillator

Der interne Oszillator läuft mit einer internen Frequenz von 8MHz. Der Oszillator dient dazu, das Configuration Memory zu löschen sowie als Quelle für die `CCLK`. An seinen Ausgängen stehen die Frequenzen 500kHz, 16kHz, 490Hz, 15Hz und 8MHz zur Verfügung, diese Werte schwanken aber in Abhängigkeit von Temperatur und von außen angelegter Spannung sowie logikbedingt.

Von den uns zur Verfügung stehenden PCI-Karten stellt die kleinere der beiden, mit dem XC4062XLT bestückte, 2304 CLBs und 5376 Flip-Flops, die größere, mit dem XC4085XLA bestückte, 3136 CLBs und 7168 Flip-Flops zur Verfügung. Hiervon werden allerdings rund 500 CLBs für die Schnittstelle zum PCI-Bus verbraucht, so daß noch rund 1800 beziehungsweise 2600 CLBs verwendet werden können.

## 2.5 Anbindung der CPUs an den Host-PCI-Bus

Die Anbindung der CPUs an den PCI-Bus gliedert sich in folgende drei Teile:

- PCI-Interface
- H.O.T.-Interface
- PCI-Wrapper

Die folgende Grafik skizziert diese drei Schichten und deren Einbindung auf der PCI-Karte.

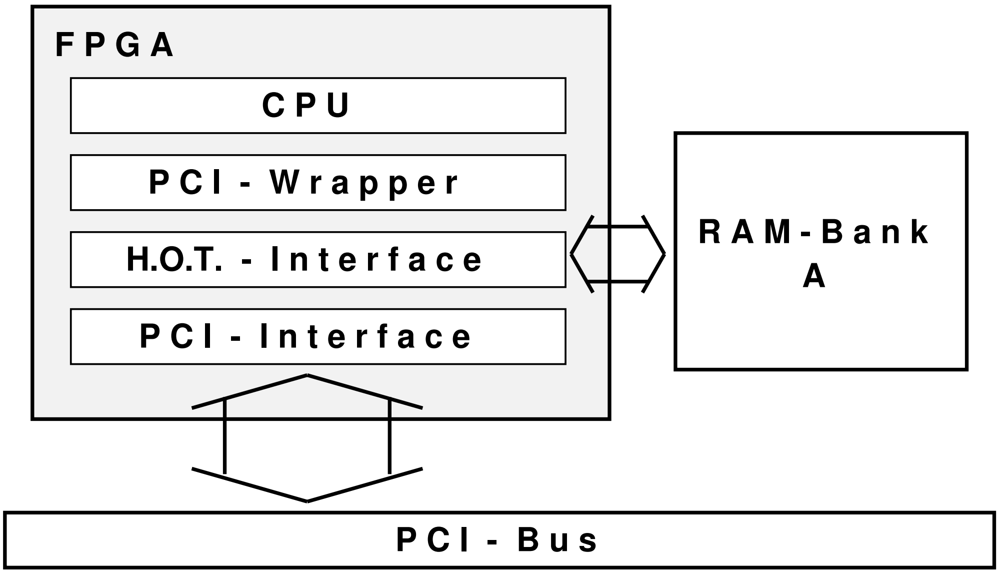



Abbildung 2.1: Die äußersten *VHDL*-Komponenten

Zu beachten ist hierbei, daß der FPGA direkt mit dem PCI-Bus verbunden ist und nach der Konfiguration keine externen Logikbausteine benötigt. Die gesamte CPU- und Buslogik befindet sich auf dem FPGA und ist somit auch Teil des eigenen Projekts. Auch die RAM-Bank A, auf die neben dem PCI-Bus ebenfalls zugegriffen wird, ist direkt mit dem FPGA verbunden. Siehe hierzu auch [VCCHARDAPI].

### 2.5.1 Die unteren Schichten

Da der FPGA direkt am PCI-Bus hängt, befindet sich als unterste Schicht in jeder Konfiguration ein PCI-Interface. Das von *V.C.C.* beziehungsweise *Xilinx, Inc.* bereitgestellte Interface ist 32Bit/33MHz-PCI-konform und erlaubt einen maximalen Datendurchsatz von 132MB/s.

Ebenso unverändert von *V.C.C.* übernommen ist die nächsthöhere Schicht, das H.O.T.-Interface. Hiermit wird zum einen der Zugriff auf die RAM-Bänke (und falls gewünscht die Erweiterungsanschlüsse) ermöglicht und zum anderen der PCI-Datenaustausch erleichtert.

Diese beiden Schichten tauchen im Projekt als *VHDL*-Komponenten „**PCI_CORE**“, „**pcim_top**“ und „**pcim_lc**“ auf. Auf deren Quellcode kann (in der Regel) jedoch nicht zugegriffen werden.

### 2.5.2 Der PCI-Wrapper und die CPUs

Bei dem Design der CPU-Schnittstellen haben wir uns – soweit in dieser Konfiguration sinnvoll – an die jeweiligen Spezifikationen gehalten. Dies betrifft die Daten- und Adreßbusse und die für die Funktionalität entscheidenden Steuerleitungen (RESET, Interrupt, etc.). Für die jeweiligen Abweichungen von den Original-CPUs siehe auch Kapitel 4 und 6.

Da diese Schnittstellen weder mit den *V.C.C.*-Schnittstellen der RAM-Bänke noch mit denen des PCI-Buses übereinstimmen, mußten wir zwischen H.O.T.-Interface und CPU noch eine Schicht, den „PCI-Wrapper“, einfügen.

Die Grundversion des PCI-Wrappers hat folgende Schnittstellen:

**Schnittstelle zum H.O.T.-Interface (PCI_CORE)**

| **Eingänge** | **Beschreibung** |
|---|---|
| D_IN | Daten vom PCI-Bus |
| SLV_DCD | Anforderung und Adresse von Schreibzugriffen des Hosts auf die PCI-Karte |
| MAD_IN | Daten von RAM-Bank A |
| MA_BSY | RAM-Bank A gesperrt (durch Host) |
| RESET_IN | Haupt-RESET, direkt nach Konfiguration gesetzt |
| F_CLK | Der programmierbare „User-Clock“ |
| PCI_CLK | PCI-Bustakt (33MHz) |

| **Ausgänge** | **Beschreibung** |
|---|---|
| D0_OUT | Daten vom PCI-Wrapper zum IO-Port 0 |
| D1_OUT | Daten vom PCI-Wrapper zum IO-Port 1 |
| D2_OUT | Daten vom PCI-Wrapper zum IO-Port 2 |
| D3_OUT | Daten vom PCI-Wrapper zum IO-Port 3 |
| MAD_OUT | Daten an RAM-Bank A |
| MA_ADR | Adresse an RAM-Bank A |
| MA_WE | RAM-Bank A - Schreibzugriff |
| MA_OE | RAM-Bank A - Lesezugriff |
| MA_ON | RAM-Bank A - Zugriff allgemein |
| HOLDOFF | (nicht verwendet) |
| INTONBOOT | (nicht verwendet) |
| URDY | (nicht verwendet) |
| USER_INT | (nicht verwendet) |
| USER_LED | (nicht verwendet) |
| PRT | FPGA-Typ (konstant „0100“) |

### Schnittstelle zur CPU

| Eingänge | Beschreibung |
|---|---|
| `FROM_CPU` | Daten von der CPU zu RAM oder IO-Port |
| `ADDR` | Adresse der zu lesenden/speichernden Daten |
| `PHI2` | Anforderung an den PCI-Wrapper (siehe Text) |
| `RNOTW` | Schreib-/Leseoperation |
| `RAMSEL` | RAM-Zugriff |
| `ROMSEL` | ROM-Zugriff |
| `IOSEL` | Zugriff auf einen IO-Port |
| `DEBUG` | Ausgabeleitungen der CPU zu Testzwecken |
| **Ausgänge** | **Beschreibung** |
| `NOT_RESET` | CPU-Reset |
| `CLK` | Takt für die CPU |
| `TO_CPU` | Daten von RAM oder IO-Port zur CPU |
| `ACK` | Bestätigung vom Wrapper (siehe Text) |
| `NOT_NMI` | Non Maskable Interrupt |
| `NOT_IRQ` | Interrupt Request |

Für einen objektiven Vergleich der verschiedenen CPUs, und um das Ändern und Debuggen des PCI-Wrappers zu erleichtern, sollten alle CPUs ursprünglich denselben PCI-Wrapper benutzen. Zwangsläufig kommt es jedoch zu Unterschieden nach der Optimierung durch den Compiler (`PHI2` und `ACK` werden nicht von den synchronen, `NMI` nicht von DEP-CPUs benutzt). Daher entstanden während der letzten Entwicklungphase verschiedene Varianten der oben beschriebenen Grundversion. Hinsichtlich der Performance und Größe der verschiedenen Gesamtdesigns (PCI-Wrapper mit den *V.C.C.*-Komponenten und der CPU) sind die Unterschiede der PCI-Wrapper jedoch vernachlässigbar.

### Datenaustausch über die IO-Ports

Der wichtigste Datenaustausch zwischen Host-Anwendung und PCI-Wrapper/CPU findet über die beiden ersten IO-Ports (0 und 1) statt. Über die untersten acht der 32 Bit von Port 0 ist das „Special Function Register“ (SFR) zugänglich. Hier kann der Host die zentralen CPU-Steuerfunktionen RESET, SingleStep, etc. kontrollieren.

| 7 (MSB) | 6 | 5 | 4 | 3 | 2 | 1 | 0 (LSB) |
|---|---|---|---|---|---|---|---|
| — | — | SingleStep En | SingleStep Clk | — | IRQ | NMI | RESET |

Special Function Register

Der zweite IO-Port (Adresse 1) ist für den Datenaustausch zwischen Host-Anwendung und *laufender* CPU zuständig.

Die beiden IO-Ports 2 und 3 werden nur für Debug-Zwecke und nur von manchen CPU-Versionen verwendet.

### Datenaustausch vom/zum RAM

Die von der CPU angelegte RAM-Adresse wird direkt der RAM-Bank A übergeben. Die Leitung `RNOTW` bestimmt – in Verbindung mit `RAMSEL`, `PHI2` und `ACK` – ob gelesen oder geschrieben wird. Die Daten der CPU werden gegebenenfalls direkt an die RAM-Bank weitergeleitet. Zur Einhaltung des sogenannten „DEP-Formates“ (siehe unten) kommt noch die jeweilige Adresse und `0xFF` hinzu.

### Besonderheiten der asynchronen CPUs

Der Datenaustausch mit asynchronen CPUs (IO-Port- ebenso wie RAM-Bank-Zugriffe) wird über die „Handshake“-Leitungen `PHI2` und `ACK` gesteuert. Möchte die CPU auf RAM oder IO-Port zugreifen, setzt sie Adreß-, Daten- und Steuerbus (`RNOTW`, `RAMSEL`, `IOSEL`) entsprechend und legt `PHI2` auf `high`. Hat der PCI-Wrapper diese Anforderung bearbeitet und stehen die gewünschten Daten zur Verfügung (beziehungsweise sind die Daten gespeichert), so antwortet dieser mit einem Setzen von `ACK`. Danach nimmt zuerst die CPU und dann der PCI-Wrapper die `high`-Pegel wieder zurück.

Die SingleStep-Betriebsart bei asynchronen CPUs wird bei der Übergabe des `ACK` vom Wrapper zur CPU realisiert. Diese Weiterleitung geschieht erst nach einem Setzen des „SingleStep-En“-Bit im SFR (siehe Absatz „Datenaustausch über die IO-Ports“).

Das Problem, das der PCI-Wrapper nun hat, ist festzustellen, wann das RAM Daten sicher gespeichert beziehungsweise ausgegeben hat. Um dies beim Lesezugriff feststellen zu können, wurde ein spezielles Datenformat, das bereits erwähnte „DEP-Format“, eingeführt (siehe Glossar). Der Wrapper vergleicht die im RAM gespeicherte Adresse mit der angeforderten und schließt aus gültigen Adreßdaten auf ein gültiges Datum. Da der Vergleichsschritt selbst auch Zeit benötigt, ist diese Methode zuverlässig genug. Außerdem sind die RAM-Zellen der PCI-Karte 32bit breit, so daß Daten- und Adreßinformation in einem Schritt gelesen und gespeichert werden.

Einen gültigen, abgeschlossenen Speicherzugriff kann der Wrapper ebenfalls mit ausreichender Zuverlässigkeit durch die getrennten Ein- und Ausgänge der RAM-Bank feststellen.

# Kapitel 3

# DEP-CPU

Die Abkürzung „DEP“ steht für „Digitalelektronisches Praktikum“. Dieses Praktikum wird im Rahmen des Grundstudiums der Naturwissenschaftlichen Informatik angeboten. Am Ende des Praktikums wird die „DEP-CPU“ entwickelt. Zum Praktikum siehe auch [DEPSCRIPT].

## 3.1 Beschreibung

Die sogenannte DEP-CPU ist eine recht unkompliziert aufgebaute 8-Bit CPU, die hier eine gute Möglichkeit bietet, verschiedene Konzepte der Asynchronität und Realisierung zu erforschen. Wie schon in Kapitel 1 erklärt, wurde erst eine konventionelle, synchrone Variante der DEP-CPU entworfen. Mittels dieser konnten zwei Aspekte erörtert werden: Zum einen entsteht so die Möglichkeit, später die asynchrone Umsetzung mit der synchronen zu vergleichen und einen Benchmark durchzuführen. Zum anderen ist Erfahrung mit dem Umgang des „Schematic Editor“s von *Xilinx, Inc.* wichtig. Im Gegensatz dazu gibt es die Realisierung in der Hardwarebeschreibungssprache „VHDL“ umzusetzen. Es stellte sich die Frage, ob und wie groß die Unterschiede des Kompilates sind und welche der beiden Möglichkeiten praktikabler sind. Diese Aspekte werden in dem Abschnitt 2 diskutiert.

In diesem Kapitel soll es deshalb mehr um die DEP-CPU an sich gehen. Es werden sowohl die Befehle als auch der Aufbau vorgestellt. Abgeschlossen wird das Kapitel durch ein paar Worte zu der synchronen Umsetzung.

### 3.1.1 Befehlssatz und Decoder

Herzstück der DEP-CPU ist der Decoder. Er regelt, welches Bauteil zu einem bestimmtem Zeitpunkt seine Daten erhält. Seine Anschlüsse sind wie folgt:

| **Eingänge** | **Beschreibung** |
|---|---|
| `IWR` | 8-Bit Daten vom IWR |
| `CCRIN` | 2-Bit Flags vom CCR |
| `MSROUT` | 2-Bit aktueller MSR-Status |

| **Ausgänge** | **Beschreibung** |
|---|---|
| `MSRIN` | 2-Bit nachfolgender MSR-Status |
| `CEIWR` | Chip Enable für das IWR |
| `CEOPR` | Chip Enable für das OPR |
| `MuxAB` | Select Signal für den MuxAB |
| `MuxPC` | Select Signal für den MuxPC |
| `CEAccR` | Chip-Enable für den Akkumulator |
| `CECCR` | Chip-Enable für das CCR |
| `CEPCR` | Chip-Enable für das PCR |
| `NRAM` | Select Leitung für das RAM mit negativer Logik |
| `NROM` | Select Leitung für das ROM mit negativer Logik |
| `NIO` | Select Leitung für den Input/Output mit negativer Logik |
| `RNW` | Select Leitung, ob gelesen oder geschrieben wird. |
| `AluOP` | ALU-Operationen für die ALU |

Tabelle 3.2: Die Anschlüsse des Decoders der DEP-CPU

Die DEP-CPU besitzt im Gegensatz zum 6502 nur diese beiden Adressierungsarten:

- Absolute: (kurz: ABS) Der Operand eines Befehls stellt eine Adresse im Speicher dar.
- Immediate: (kurz: IMM) Der Operand eines Befehls stellt direkt den Wert dar.

Eine genaue Übersicht der Befehle für beide Adressierungsmodi findet sich in der folgenden Tabelle:

| **Befehl** | **OpCode IMM** | **OpCode ABS** | **Beschreibung** |
|---|---|---|---|
| LDA | 48 | 40 | Laden des AccR mit dem Operanden |
| ADD | 49 | 41 | Addieren des Operanden zum AccR |
| OR | 4A | 42 | Bitweises Verodern von AccR und Operand |
| SBC | 4B | 43 | Subtrahieren des Operanden (und des Carry-Bits) vom AccR |
| XOR | 4C | 44 | Bitweises exklusiv-Verodern von AccR und Operand |
| ADC | 4D | 45 | Addieren den Operanden und des Carry-Bits zum AccR |
| AND | 4E | 46 | Bitweises Verunden von AccR und Operand |
| SETF | 4F | 47 | Setzen des Carry- und Zero- Flags entsprechend Bit 6 und 7 des Operanden |

Tabelle 3.3: Befehle der DEP-CPU, die in beiden Adreßmodi vorkommen.

Zusätzlich existieren noch einige Befehle, die nicht von einem Adressierungsmodus abhängen:

| Befehl | OpCode | Beschreibung |
|---|---|---|
| STA | 80 | Speichern des Inhalts des AccR an Adresse des Operanden |
| BCS | 28 | Springen an die Adresse des Operanden, wenn Carry Flag gesetzt ist. |
| BEQ | 18 | Springen an die Adresse des Operanden, wenn Zero Flag gesetzt ist. |
| IN | 50 | Laden des AccR mit Inhalt des Ports des Operanden |
| OUT | 90 | Speichern des AccR an Port des Operanden |

Tabelle 3.4: Zusätzliche Befehle der DEP-CPU

Die vollständige Decodertabelle findet sich in Abschnitt C. Die sich daraus ergebenden aussagenlogischen Vereinfachungen, die wir auch implementiert haben, finden sich weiter unten in Abschnitt 3.2.

### 3.1.2 ALU

Die ALU verarbeitet sämtliche Befehle, die direkt eine Berechnung aus dem OPR und dem Akkumulator erfordern.

| Eingänge | Beschreibung |
|---|---|
| `A` | 8-Bit Daten vom OPR |
| `B` | 8-Bit Daten vom Akkumulator |
| `AluOP` | 3-Bit ALU-Operation vom Decoder |
| `CIN` | Carry-Flag vom CCR |
| **Ausgänge** | **Beschreibung** |
| `TOCCR` | 2-Bit zum CCR |
| `OUT` | 8-Bit Daten zum Akkumulator |

Tabelle 3.6: Die Anschlüsse der ALU der DEP-CPU

Das Rechenwerk der DEP-CPU kann auf zwei grundsätzlich verschiedene Arten realisiert werden:

- 1-Bit-ALU: Es werden mehrere kleinere Rechenwerke, die jeweils mit einem Bit operieren, hintereinander geschaltet. Zusätzlich zum Ergebnis wird dann noch der Übertrag übermittelt.

- 8-Bit-ALU: Hierbei kann jeder Befehl in einem Bauteil realisiert werden . Entweder rechnen einige Bauteile nur, wenn ihr jeweiliges Ergebnis auch benötigt wird, oder alle Bauteile rechnen das Ergebnis aus und es wird nachher über einen Multiplexer selektiert. Bei diesem Ansatz kann man zusätzlich noch einige Befehle clustern, was die Übersichtlichkeit erhöht.

### 3.1.3 Blockschaltbild

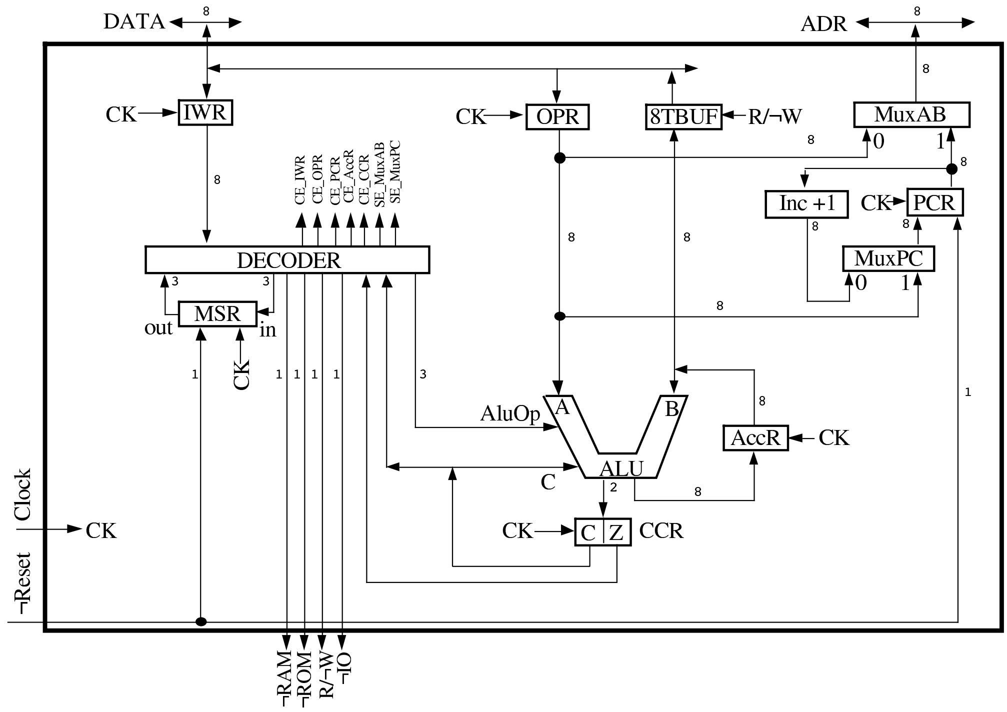



Abbildung 3.1: DEP-CPU

| Eingänge | Beschreibung |
|---|---|
| `CK` | Clock Signal |
| `NReset` | Reset Signal mit negativer Logik |
| **Ausgänge** | **Beschreibung** |
| `ADR` | 8-Bit Adressenbus |
| `NRAM` | RAM-Enable mit negativer Logik |
| `NROM` | ROM-Enable mit negativer Logik |
| `NIO` | I/O-Enable mit negativer Logik |
| `RNW` | Signalisiert den lesenden oder schreibenden Zugriff |
| **Bidirektional** | **Beschreibung** |
| `DATA` | 8-Bit Datenbus |

Tabelle 3.9: Die Anschlüsse für die Peripherie der DEP-CPU

Die Daten gelangen von der Peripherie über den Datenbus in die CPU. Dabei wird, wie üblich, ein Adreßbus zur Referenzierung eingesetzt. Die so in den Prozessor gelangenden Daten werden je nach Zustand des MSR (Maschinenstatusregister), der einfach pro Takt inkrementiert und am Ende eines Befehls auf „00“ gesetzt wird, in das IWR (MSR ist „00“) oder in das OPR (MSR ist „01“) geladen. Von dem IWR aus werden sie nun im nächsten Takt vom Decoder interpretiert. Von dem OPR aus wird nun je nach Befehl entweder eine ALU-Operation auf dem Akkumulator und dem OPR ausgeführt, oder die Daten gelangen in die AGU (Address Generation Unit).

Sollte der Befehl eine ALU-Operation beinhalten, wird diese stets ausgeführt, wenn die CPU sich im dritten Takt des Befehls befindet. Die ALU bekommt von dem Decoder eine Operation und rechnet entsprechend dieser Instruktion ein Ergebnis aus den Daten des OPR und Akkumulators aus, was er zurück in den Akkumulator schreibt. Dabei setzt die ALU entsprechend dem Ergebnis zusätzlich die Flags des CCR.

Die AGU besteht aus dem eigentlichen PCR, zwei Multiplexern und einem Inkrementer. Im „Normalfall“ bekommt der MuxPC vom Decoder eine `0` und der MuxAB eine `1`. Dabei wird nun das PCR über den Inkrementer um eins erhöht. Der MuxPC bietet nun die Möglichkeit eine neue Adresse aus dem OPR in das PCR zu laden - also einen Jump auszuführen. Hingegen ist der MuxAB zur Bearbeitung der Adressen, die sich im Adressierungsmodus „Absolute“ nach dem zweiten Befehlstakt im OPR befinden. Wenn an dem MuxAB eine `1` anliegt wird direkt der Inhalt des OPR auf den Adreßbus gelegt, was zur Folge hat, daß der Prozessor den Inhalt am Datenbus auslesen kann bzw. dorthin schreiben kann. Zum Schreiben des Inhalts des Akkumulators gibt es noch einen achtfach Tristatebuffer zwischen Akkumulator und Datenbus.

## 3.2 Synchrone Umsetzung

Der Implementierung der DEP-CPU nach gegebenem Vorbild verläuft recht rasch, und aufgrund der didaktischen Herkunft ist die geringe Komplexität für geübte Chipdesigner innerhalb von wenigen Stunden zu implementieren.

Betrachtet man das Blockschaltbild genauer, so fallen einem sofort drei Sektionen auf, die **AGU**, die **ALU**, der **Decoder**, welche am geeignetesten in dieser Reihenfolge aufzubauen sind.

### 3.2.1 AGU

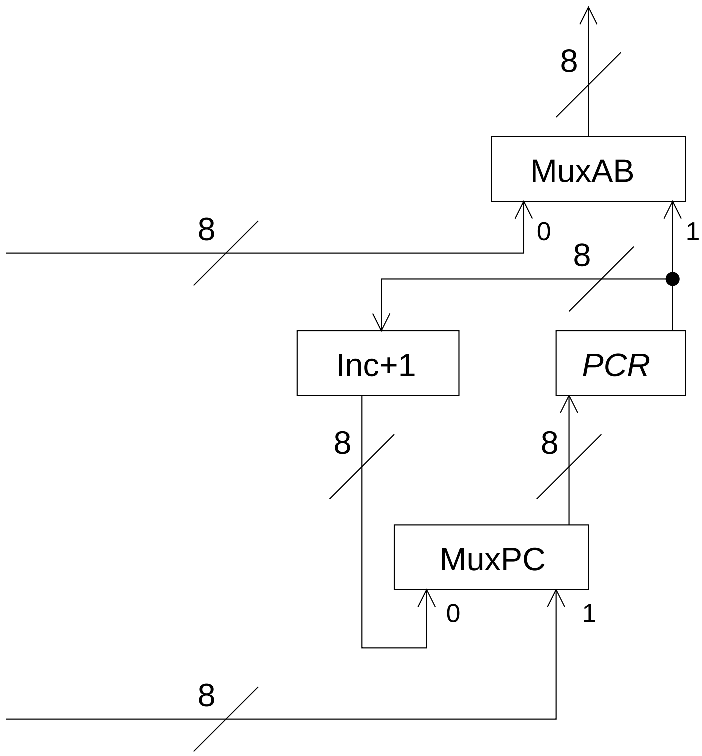



Abbildung 3.2: Die Address-Generation-Unit der DEP-CPU

Die **AGU** besteht aus insgesamt vier Komponenten: **INC+1**, **MUXAB**, **MUXPC** und **PCR**.

**INC+1** ist ein einfacher 8-Bit Halbaddierer, bestehend aus acht einzelnen 1-Bit Halbaddierern. Seine Aufgabe ist es, das Eingangssignal um 1 zu inkrementieren.

**MUXAB** und **MUXPC** werden aus acht 2-fach 1-Bit Multiplexern implementiert. Ihre Aufgabe ist es, eine Speicheradressierung direkt aus dem **OPR** zu ermöglichen (**MUXAB**) bzw. für Sprungbefehle das **PCR** zu überschreiben. Der **Decoder** steuert dabei den Datenfluß durch die `SEMUXAB` und `SEMUXPC` Leitungen.

**PCR** ist ein einfaches 8-Bit-Register. “PCR“ steht dabei für “Programm Counter Register“ und stellt innerhalb der CPU den Befehlszähler dar.

### 3.2.2 ALU

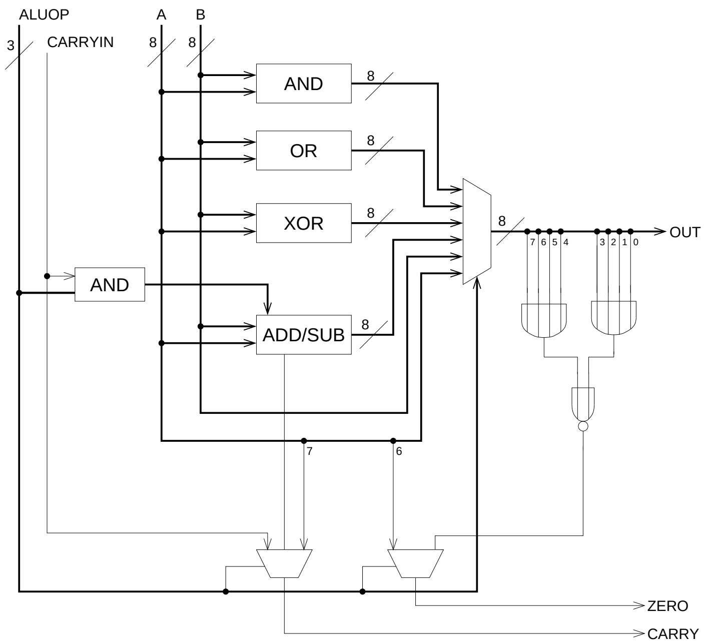



Abbildung 3.3: Schematischer Aufbau der Arithmetisch Logischen Einheit der DEP-CPU

Die Aufgabe der Arithmetisch-Logischen Einheit **ALU** ist es, die angegebenen Rechenwerksbefehle zu implementieren. Bei ihr handelt es sich um die wahrscheinlich komplexeste Einheit der gesamten DEP-CPU. Unsere Implementierung sieht vor, die sechs der acht möglichen ALU-Befehle, bei denen sie aktiv eines ihrer Eingabesignale verändert, in vier Rechenblöcke aufzuteilen. Jeweils einen für die logischen Funktionen AND, OR und XOR, und einen für die Additions- und Subtraktionsbefehle ADD, ADC sowie SBC.

Die drei Blöcke für die Befehle AND, OR und XOR bestehen aus jeweils acht AND-, OR-, bzw. XOR-Gattern. Die Gatter sind so geschaltet, daß ihre beiden Eingänge sich von beiden Eingabebussen in aufsteigender Reihenfolge jeweils ein Bit holen und ihre Ausgabe wieder in aufsteigender Reihenfolge als ein Bit auf den Ausgabebus legen.

Der Block für die Befehle ADD, ADC und SBC enthält in unserer Implementierung acht einzelne kombinierte Additions-/Subtraktionsbausteine für jeweils ein Bit. Durch gesteuerte Invertierung eines der beiden Eingabebusse wird eine Subtraktion nach Zweierkomplementdarstellung erreicht. Ebenso wird mit der `CARRYIN`-Leitung verfahren, die zusätzlich noch durch ein AND-Gatter beim ADD-Befehl auf `low` gezogen wird.

Da alle vier Blöcke gleichzeitig rechnen, ist in unserer Implementierung ein sechsfach 8-Bit-Multiplexer eingebaut. Durch das vom Decoder kommende ALU-OP wird ausgewählt, von welchem der vier Blöcke das Gesamtergebnis der ALU ausgeht. Die restlichen zwei Möglichkeiten sind direkt die beiden Eingänge der ALU, die einmal vom **OPR** kommend für den Befehl LDA, andererseits vom **AccR** für den Befehl SETF wichtig sind.

### 3.2.3 Decoder

Der Decoder wird intern mit Hilfe von logischen Gattern in unserer Implementierung so aufgebaut, daß seine Wertetabelle direkt der Decodertabelle entspricht. Eine Implementierung mit Hilfe eines Microcode-Speichers wäre ebenso möglich gewesen, aufgrund der ansonsten geringen Komplexität der CPU allerdings nicht praktikabel.

Die Formeln, die der Decoder in unserer Implementierung enthält, sind:

$$
\begin{aligned}
\mathtt{MSRIN1} &= \mathtt{MSROUT1} \dot{\vee} (\mathtt{MSROUT0} \vee (\mathtt{MSROUT1} \wedge (\mathtt{IMM} \vee \mathtt{ST}))) \\
\mathtt{MSRIN0} &= \neg(\mathtt{MSROUT0} \vee \mathtt{MSROUT1}) \vee (\mathtt{MSROUT1} \wedge \neg(\mathtt{IMM} \vee \mathtt{ST} \vee \mathtt{MSROUT0})) \\
\mathtt{CE\_IWR} &= \neg(\mathtt{MSROUT0} \vee \mathtt{MSROUT1}) \\
\mathtt{CE\_OPR} &= (\neg\mathtt{MSROUT1} \wedge \mathtt{MSROUT0}) \vee (\mathtt{LD} \wedge \mathtt{MSROUT1} \wedge \neg(\mathtt{MSROUT0} \vee \mathtt{IMM})) \\
\mathtt{SE\_MUXAB} &= \neg\mathtt{MSROUT1} \\
\mathtt{SE\_MUXPC} &= \mathtt{IMM} \wedge \mathtt{MSROUT1} \wedge ((\mathtt{BCS} \wedge \mathtt{CARRY}) \vee (\mathtt{BEQ} \wedge \mathtt{ZERO})) \\
\mathtt{CE\_ACCR} &= \mathtt{LD} \wedge \mathtt{MSROUT1} \wedge (\mathtt{MSROUT0} \vee \mathtt{IMM}) \\
\mathtt{CE\_PCR} &= \neg\mathtt{MSROUT1} \vee (\mathtt{BEQ} \wedge \mathtt{ZERO} \wedge \mathtt{IMM}) \vee (\mathtt{BCS} \wedge \mathtt{CARRY} \wedge \mathtt{IMM}) \\
\mathtt{NOTRAM} &= \neg(\mathtt{MSROUT1} \wedge (\mathtt{ST} \vee \mathtt{LD}) \wedge \neg(\mathtt{MSROUT0} \vee \mathtt{BEQ} \vee \mathtt{IMM})) \\
\mathtt{NOTROM} &= \mathtt{MSROUT1} \\
\mathtt{NOTIO} &= \neg(\mathtt{BEQ} \wedge \mathtt{MSROUT1} \wedge \neg\mathtt{MSROUT0} \wedge (\mathtt{ST} \vee \mathtt{LD})) \\
\mathtt{RNOTW} &= \neg(\mathtt{MSROUT1} \wedge \mathtt{ST})
\end{aligned}
$$

### 3.2.4 Umsetzung der Gesamtarchitektur

Nachdem die drei Teile der CPU fertig geworden sind, ist es nun wichtig, das „drumherum“ zu implementieren. **AccR**, **OPR** und **IWR** bestehen aus 8-Bit Registern. **MSR** und **CCR** bestehen aus 2-Bit-Registern. Ihre `Chip Enable`-Signale kommen aus dem Decoder.

Der `DATENBUS` wurde bidirektional implementiert. Dies macht es erforderlich, daß der **AccR** als einziges schreibendes Bauteil einen Tristatebuffer nachgeschaltet bekommt, die Register **IWR** und **OPR** als lesende Bauteile insgesamt genau einen Inputbuffer.

Das `Phi2`-Signal, welches bei aufsteigender Taktflanke einen Speicherzugriff auslösen soll, wird der Einfachheit halber aus dem negierten Taktsignal gebildet.

Verschaltet werden alle Bauteile wie im Blockschaltbild angegeben.

# Kapitel 4

# Asynchrone Ansätze und Anwendungen

## 4.1 Micropipelining

### 4.1.1 Die Theorie

Der Ansatz des Micro-Pipeliningbasiert darauf, den Prozessor in kleine Teilstücke aufzuteilen, die jeweils dieselbe Laufzeit zugewiesen bekommen. Der Vorteil hierbei ist, daß die Zeitverluste, die bei der Worst-Case-Abschätzung der Zeit entstehen, sich insgesamt in kleinerem Maßstab bewegen, als wenn man diese Abschätzung für die komplette CPU als Ganzes macht. Genau dieses ist bei der Konstruktion einer synchronen CPU zwingend erforderlich: Der Takt kann nur so schnell sein, wie die CPU im ungünstigsten Fall für die Abarbeitung eines Prozessorzyklus braucht.

Beim Micro-Pipelining-Ansatz werde die einzelnen Abschnitte der CPU werden durch sogenannte Micropipelines gekapselt, die nacheinander abgearbeitet werden [AM2E]. Dabei meldet das vorherige Bauteil dem Nachfolger, wann es mit der Bearbeitung der Daten fertig ist. Dies wird durch ein Signal realisiert, das, sobald es den Wert ändert, die erfolgreiche Abarbeitung der Daten signalisiert. Das Ende der Bearbeitung wird durch ein zu der Logik parallelgeschaltetes Delay-Element festgestellt, das das eingehende Signal so lange verzögert, bis die Logik ein festes Ergebnis liefert.

Die Kommunikation unter den einzelnen Micropipelines geschieht über zwei Leitungen: auf der ersten Leitung schickt eine Micropipeline ein Signal an seinen Nachfolger, wenn sie mit der Abarbeitung ihres logischen Blockes fertig ist und der Nachfolger seine Arbeit beginnen kann. Am Ende der gesamten Folge von einzelnen Micropipelines wird dieses Signal auf die zweite Leitung gegeben und tritt seinen Weg zurück durch alle Micropipeline-Elemente an (jetzt natürlich in umgekehrter Reihenfolge). Hierdurch wird die Abarbeitung eines kompletten Befehls beendet. Sobald das Signal beim ersten Element ankommt, kann mit dem folgenden Befehl fortgefahren werden.

### 4.1.2 Die einzelnen Bestandteile eines Micropipeline-Elementes

Ein Micropipeline-Element ist grundsätzlich in drei verschiedene Teile untergliedert: den Logikblock, das Delay-Element und das eigentliche Pipeline-Element.

#### Der Logikblock

Der Logikblock ist der Teil der Micropipeline, der die eigentlichen, für den Prozessor wichtigen Logikfunktionen enthält. Teilweise ist der Logikblock 1-zu-1 aus der synchronen Variante der CPU übernommen, d.h. die Logik auf Gatterebene, die für die Funktionalität des Prozessors zuständig ist, wurde beibehalten und lediglich in Micropipelines gekapselt. Er bekommt seine Daten vom Logikblock der vorherigen Pipeline und gibt sie an den nachfolgenden Logikblock weiter.

#### Das Delay-Element

Das Delay-Element verzögert das Signal, das der nachfolgenden Micropipeline das Vorliegen eines Ergebnisses anzeigt, so lange, wie der Logikblock im Worst Case braucht, um die Daten zu verarbeiten.

#### Das Pipeline-Element

Das Pipeline-Element ist für die Kommunikation der einzelnen Micropipelines untereinander verantwortlich. Es übernimmt das Request-Signal vom Vorgänger, leitet es an das Delay-Element weiter und gibt es schließlich an die nachfolgende Micropipeline weiter.

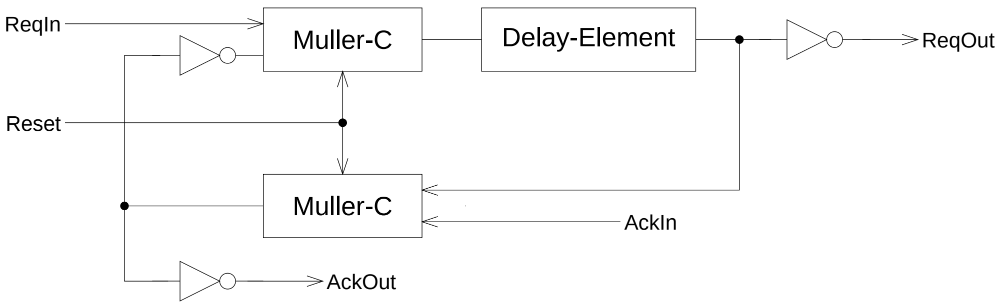



Abbildung 4.1: Micropipeline

Eine Micropipeline hat jeweils zwei Eingänge und zwei Ausgänge: `ReqIn`, `ReqOut`, `AckIn` und `AckOut`. Liegt an `ReqIn` ein `high`-Signal, ist das das Signal für das Element, mit der Arbeit zu beginnen. Wenn es fertig ist, also das Delayelement durchgeschaltet hat, gibt es das Signal über `ReqOut` an das nachfolgende Bauteil weiter. Das letzte Bauteil in der Reihe übernimmt das eigene Signal `ReqOut` am Eingang `AckIn` und leitet es über `AckOut` an das vorhergehende Bauteil weiter, bis es wieder am Anfang der Pipeline angekommen ist und der Prozessor den nächsten Befehl ausführen kann.

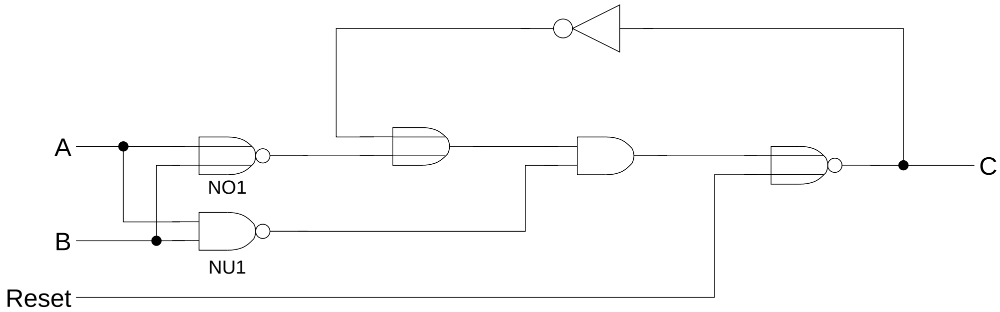



Abbildung 4.2: Muller-C Element

Ein vom Vorgänger kommendes Signal wird allerdings erst übernommen, wenn das Acknowledge des Nachfolgers eingetroffen ist. So wird sichergestellt, daß die Daten nicht zu schnell weitergegeben werden und das Timeing durcheinander bringen; realisiert wird dieses Verhalten durch den Einsatz von **`Muller-C-Bausteinen`**.

### Muller-C Element

Das Muller-C Element (siehe Abbildung 4.1.2) arbeitet wie ein digitalisiertes Hysterese-Glied. Der Ausgang C liefert solange `low` wie mindestens ein Eingang, A oder B, nicht auf `high` liegt. Sobald alle Eingänge auf `high` liegen, liefert der Ausgang `high` und das solange, wie mindestens ein Eingang nicht auf `low` liegt. Durch den Eingang Reset kann das Element in den Ausgangszustand versetzt werden, in dem der Ausgang `low` liefert.

Das Element kann durch Austausch der Gatter NU1 und NO1 durch andere Gatter mit mehr Eingängen beliebig erweitert werden.

## 4.1.3 Implementierungsversuch der DEP-CPUmit Micro-Pipelining

Die Micro-Pipelining-CPU teilt sich ähnlich wie die synchrone Variante in den **`Decoder`** , die **`ALU`** , eine **`Adressierungseinheit`** und **`Dateneinheit`**. Die ALU beinhaltet allerdings zum eigentlichen Rechenwerk auch noch den **`Akkumulator`**. Diese Integration erschien sinnvoll, da auf den **`Akkumulator`** in dieser CPU ausschließlich von der ALU aus zugegriffen wird. Die Kontrolle des eigentlichen Ablaufs liegt beim Decoder der sich in insgesamt drei Decoderstufen aufteilt. Von diesen Stufen werden, wenn der jeweilige Befehl dies erfordert, die **`Adresspipe`** und die **`Datenpipe`** in den Ablauf eingebunden. Ein Maschinenstatusregister ist nicht erforderlich, da die Ablaufsteuerung von Decoderstufe zu Decoderstufe weitergereicht wird und damit der aktuelle Maschinenstatus durch die Aktivität der Decoderstufe gegeben ist.

Den Anfang eines jeden Prozessorzyklus bildet die erste Decoderstufe. Sie schaltet nacheinander die **`Datenpipe`** [DP] und die **`Adresspipe`** [AP], um das IWR zu laden. Die zweite Decoderstufe läuft identisch ab, um den Operand zu laden. Parallel zum Laden des Operanden wird der Befehl im eigentlichen Decoderdecodiert und abhängig vom Befehl wird die dritte Decoderstufe oder die ALU aktiviert. Die dritte

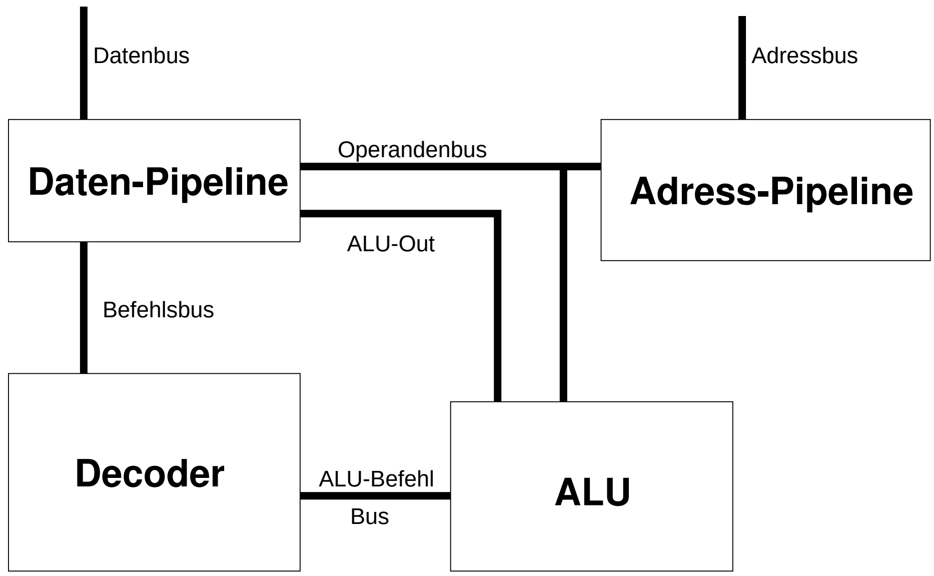



Abbildung 4.3: Schematischer Aufbau der DEP-CPU mit dem Micro-Pipelining-Ansatz

Decoderstufe aktiviert die **`AP`** und **`DP`** in umgekehrter Reihenfolge, da meistens die benötigte Adresse erst generiert werden muß. Handelt es sich um einen nichtarithmetischen Befehl, endet die Abarbeitung nach dieser dritten Decoderstufe. Andernfalls wird noch die ALU als „quasi vierte Decoderstufe“ aktiviert.

## 4.2 Dual-Rail

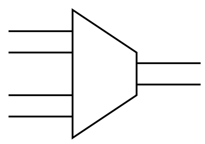


Abbildung 4.4: Flankengesteuertes Dual-Rail-AND-Gatter

Ein weiterer Ansatz, um eine asynchrone CPU zu realisieren, ist das „Dual-Rail“-Konzept, z. B. [CT99]. Hierbei wird ein Signal auf zwei Leitungen codiert. Ein Flankenwechsel auf einer der Leitungen symbolisiert eine (gültige) `0`, auf der anderen eine `1`. Dies bedeutet nun zunächst eine Verschwendung gegenüber der konventionellen Logik, die ja mit einer Leitung für `1` und `0` auskommt. Wie jedoch schon im Kapitel 1 gesagt wird, gibt es bei der asynchronen Technik das Problem, die Zeit für einen Rechen- bzw. Logikschritt feststellen zu müssen (oder wie im Falle der Micropipelines festzulegen). Es müssen von den (Teil-)Baugruppen, beziehungsweise der gesamten CPU nicht nur `0` und `1` der Eingänge ausgelesen und `0` und `1` an die Ausgänge weitergegeben werden, sondern es muß zusätzlich noch festgestellt werden, ob die vorhergehende Logik mit ihrer Berechnung schon fertig und das gelesene Eingangsbit damit gültig ist. Oder zumindest muß der nachfolgenden Logik (Baugruppe als auch CPU-externe Logik) mitgeteilt werden, ob die eigene Berechnung schon abgeschlossen und das Ausgabebit gültig ist. Diese Mitteilung über die Gültigkeit eines Wertes geschieht bei dem Micro-Pipelining-Ansatz über eine parallel zur eigentlichen Logik ablaufende Verzögerung: Gibt ein Delay-Element ein verzögertes „Ready" aus, so weiß die nachfolgende Stufe, daß der anliegende Wert korrekt ist. Beim Dual-Rail-Ansatz soll nun auf diese begleitende Logik verzichtet werden. Statt dessen sollen die Leitungen selbst über ihre Gültigkeit berichten. Und genau dafür wird die scheinbare Verschwendung benötigt.

So sieht man in Abbildung 4.4 ein AND-Gatter. Es besitzt für jeden Eingang eines herkömmlichen Gatters zwei Eingänge und genauso zwei statt einem Ausgang. Für uns stellten sich nun zwei mögliche Implementierungen zur Auswahl, deren spezifische Implementierung wir selbst erarbeitet haben:

- Flankengesteuertes Dual-Rail: Hier wird auf jede Flanke, gleich ob von `high` auf `low` oder `low` auf `high`, reagiert. Dies erfordert sehr großen Hardwareaufwand.

- Pegelgesteuertes Dual-Rail: Hier zeigt nur der Zustand `high` die Gültigkeit der betreffenden Leitung an (Leitung `1` auf `low` und Leitung `0` auf `high` bedeutet `0`). Es wird dabei meist auf eine Flanke von `low` auf `high` reagiert. Zu Problemen führt dabei, daß nach einem Signal erst wieder eine sogenannte „Nullwelle" durch die CPU geleitet werden muß, damit sich alle Leitungen wieder im Ursprungszustand (`low`) befinden. Dies geht natürlich zu Lasten der Performance, dafür ist der Hardwareaufwand geringer.

### 4.2.1 Flankengesteuertes Dual-Rail

Wie oben schon beschrieben wird bei diesem Ansatz nicht mit Zuständen sondern mit Flanken gearbeitet. Jedes logische Signal wird über zwei Leitungen codiert, wobei jede Leitung für einen logischen Zustand steht. Ein Flankenwechsel auf einer der Leitungen bedeutet somit `1` und auf der anderen `0`, dabei ist es irrelevant, ob der Flankenwechsel von `low` nach `high` oder von `high` nach `low` stattfindet. Somit wird nicht nur ein Bit übertragen, sondern gleichzeitig die Information, daß ein Bauteil komplett durchgeschaltet hat. Daraus ergibt sich, daß ein Signal nur gültig ist, wenn genau eine Leitung einen Flankenwechsel anzeigt.

Ein gleichzeitiger Flankenwechsel auf mehr als einer Leitung würde einen Implementierungsfehler bedeuten und zu nicht vorhersehbaren Auswirkungen auf die nachfolgenden Komponenten führen.

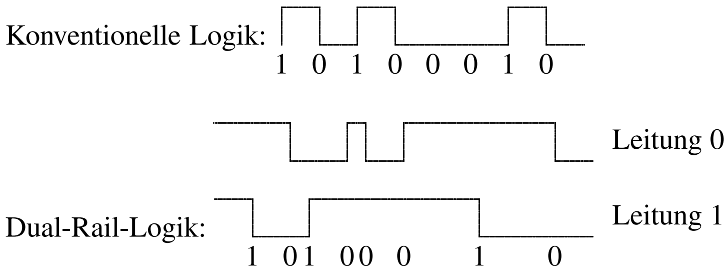



Abbildung 4.5: Dual-Rail-Signal vs. Konventionelles Signal

### Implementierung von Grundbausteinen mit flankengesteuertem Dual-Rail

#### Edge-Detector

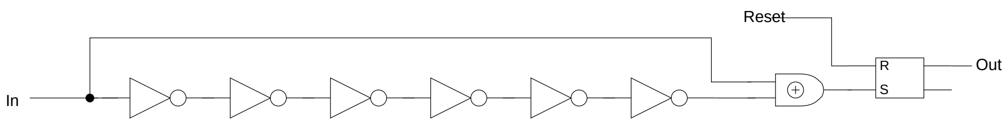



Abbildung 4.6: Edge-Detector für flankengesteuertes Dual-Rail

| **Eingänge** | **Beschreibung** |
|---|---|
| `IN` | Ein flankengesteuerter Dual-Rail Eingang |
| `RESET` | Signal vom Edge-Generator zum Zurücksetzen des Flip-Flops |

| **Ausgänge** | **Beschreibung** |
|---|---|
| `OUT` | Ein statisches Signal, mit dem konventionell gerechnet werden kann |

Wenn eine Flanke an dem Eingang `IN` anliegt, wird durch das XOR-Gatter der Flankenwechsel detektiert, da durch die Laufzeitverzögerung der NOT-Gatter an beiden Eingängen kurzzeitig unterschiedliche Signale anliegen. Das nachgeschaltete RS-Flip-Flop wird auf `1` gesetzt und erst durch das `Reset` auf `0` gesetzt, das vom Edge-Generator generiert wird, nachdem dieser den gültigen Wert übernommen hat.

**Edge-Generator**

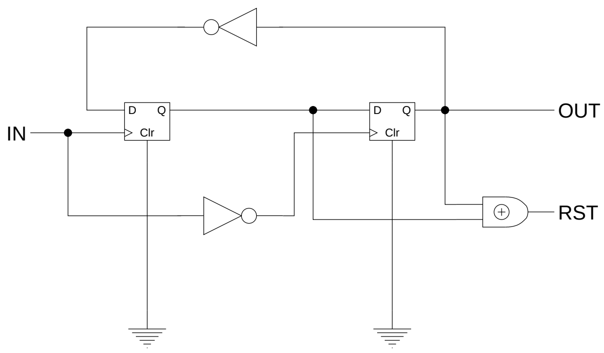



Abbildung 4.7: Edge-Generator für flankengesteuertes Dual-Rail

| **Eingänge** | **Beschreibung** |
|---|---|
| `IN` | Ein Eingang in konventioneller Logik |
| **Ausgänge** | **Beschreibung** |
| `OUT` | Ein flankengesteuerter Dual-Rail Ausgang |
| `RESET` | Signal zum Edge-Detector zum Zurücksetzen von dem Flip-Flop |

Der Edge-Generator bekommt eines der Signale, die von dem Logikblock des Gatters erzeugt werden. Seine Aufgabe ist es, am Ausgang `OUT` eine Flanke anzulegen, und durch den Ausgang `RST` die Edge-Detectoren zurückgesetzt (siehe Abschnitt 4.2.1). Zentraler Bestandteil des Edge-Generator sind zwei D-Flip-Flops. Das Eingangssignal `IN` wird an den Takteingang `C` des ersten Flip-Flops gelegt und invertiert an den des zweiten Flip-Flops. Der Dateneingang des ersten Flip-Flops wird mit dem invertierten Datenausgang des anderen Flip-Flops verbunden und der Datenausgang des zweiten Flip-Flops mit dem Dateneingang des ersten Flip-Flops. Dadurch wird gewährleistet, daß bei jedem Eintreffen einer Flanke von `0` nach `1` auf `IN` eine Flanke generiert wird. Der Ausgang `RST` wird dann auf `1` gelegt, solange das `IN`-Signal auf `1` liegt. Dadurch werden die Edge-Detectoren zurückgesetzt und das `IN` wechselt auf `0`, wodurch das zweite Flip-Flop den Wert des ersten übernimmt. Die daraus resultierende Flanke am Ausgang des zweiten Flip-Flops wird auf den Ausgang `OUT` gelegt.

**Gatter am Beispiel AND**

| **Eingänge** | **Beschreibung** |
|---|---|
| `A` | Ein flankengesteuerter Dual-Rail Eingang |
| `B` | Ein flankengesteuerter Dual-Rail Eingang |
| **Ausgänge** | **Beschreibung** |
| `OUT` | Ein flankengesteuerter Dual-Rail Ausgang |

Haben beide Dual-Rail-Eingänge geflankt, sind zwei der vier Edge-Detectoren aktiv

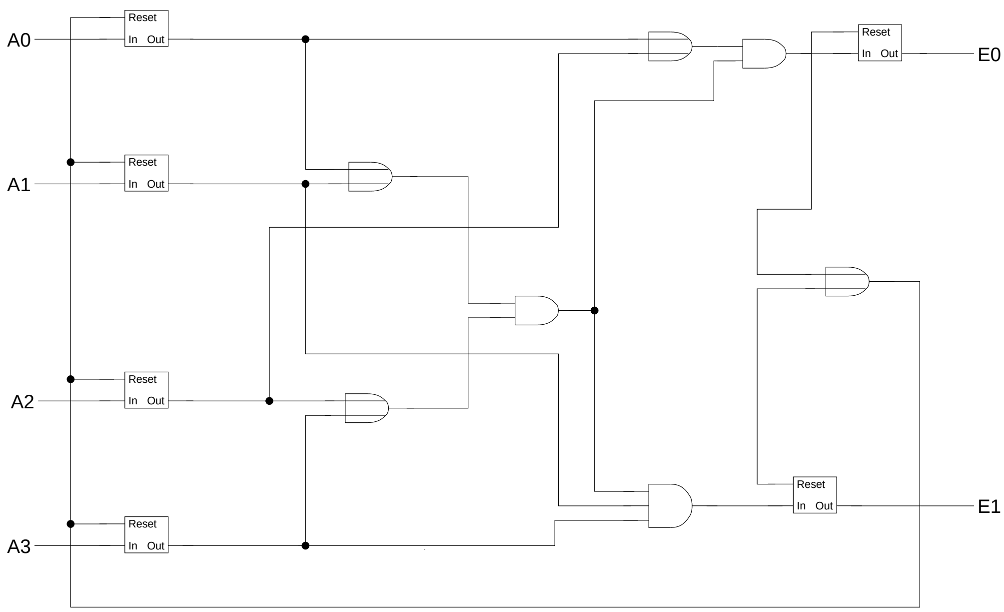



Abbildung 4.8: Flankengesteuertes Dual-Rail-AND-Gatter

und damit zwei logische Signale verfügbar. Durch eine Disjunktion von Konjunktionen von positiven und negativen Literalen läßt sich jede beliebige logische Verknüpfung – und damit auch jedes Gatter – realisieren. Im einfachsten Fall kann eine DNF verwenden. In Abbildung 4.8 sieht man, daß der Ausgang für den Flankenwechsel auf der Leitung `1` durch ein AND-Gatter aus den beiden Signalen, die eine `1` symbolisieren, besteht. Der Ausgang für die `0` ist durch ein OR-Gatter der drei restlichen Kombinationen realisiert.

Die logischen Funktionen dazu sind:

- $\mathtt{E1} = \mathtt{A}_1 \wedge \mathtt{B}_1$
- $\mathtt{E0} = (\mathtt{A}_0 \wedge \mathtt{B}_0) \vee (\mathtt{A}_0 \wedge \mathtt{B}_1) \vee (\mathtt{A}_1 \wedge \mathtt{B}_0)$

So wird genau einer der beiden Ausgänge `high` ergeben. Das so erzeugte Ausgangssignal wird nun durch die oben beschriebenen Edge-Generatoren in Flanken auf den entsprechenden Leitungen konvertiert. Gleichzeitig erzeugen die Edge-Generatoren `RST`, damit die Edge-Detectoren für den nächsten Flankenwechsel, und damit für das nächste Signal, bereit sind.

**Speicher**

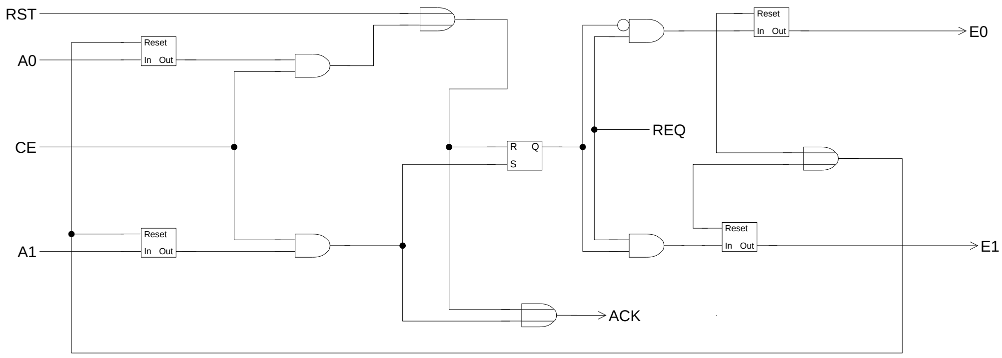



Abbildung 4.9: Flankengesteuerter Dual-Rail-Speicher

| **Eingänge** | **Beschreibung** |
|---|---|
| `A` | Ein flankengesteuerter Dual-Rail Eingang |
| `CE` | Chip enable |
| `RST` | Reset |
| `REQ` | Requestor |
| **Ausgänge** | **Beschreibung** |
| `E` | Ein flankengesteuerter Dual-Rail Ausgang |
| `ACK` | Acknowledge-Signal, wenn der Wert gespeichert wurde. |

Der eigentliche Speicher eines flankengesteuerten Dual-Rail-Speichers ist ein RS-Flip-Flop. Da Flanken nicht gespeichert werden können, müssen Edge-Detectoren vor- und Edge-Generatoren nachgeschaltet werden. Nachdem Flanken auf einem der beiden Eingänge $A_0$ oder $A_1$ detektiert wurden, wird das RS-Flip-Flop (abhängig vom detektierten Eingang) gesetzt oder gelöscht unter der Voraussetzung, daß der Eingang `CE` (Chip Enable) auf `1` liegt. Desweiteren kann das RS-Flip-Flop über den Eingang `RST` zurückgesetzt werden. Wurde das RS-Flip-Flop gesetzt oder gelöscht, wird der Ausgang `ACK` auf `1` gesetzt. Dieses zeigt an, daß die Speicherung des Wertes beendet ist. Der Ausgang des Flip-Flops wird an den jeweiligen Edge-Generatorfür die Ausgänge $E_0$ und $E_1$ angelegt. Durch das von den Edge-Generatoren erzeugt `Reset` werden die Edge-Detectoren zurückgesetzt.

Für mehrbittige Register wird eine Erweiterung benötigt, die die Edge-Detectoren erst dann zurücksetzt, wenn ein Flankenwechsel für jedes Bit detektiert wurde.

**Konverter von konventioneller Logik in flankengesteuertes Dual-Rail**

| **Eingänge** | **Beschreibung** |
|---|---|
| `IN` | Ein Eingang in konventioneller Logik |
| `CE` | Chip enable |

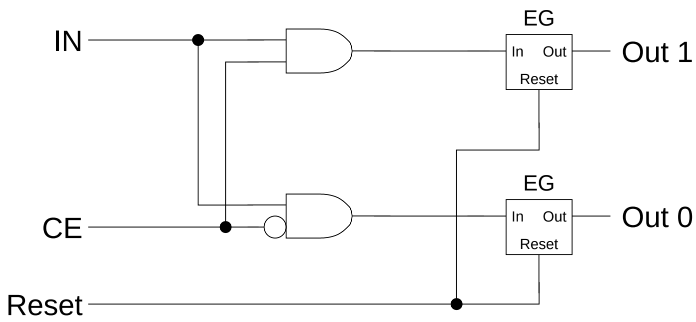



Abbildung 4.10: Konverter von konventioneller Logik nach flankengesteuertem Dual-Rail

| Ausgänge | Beschreibung |
|---|---|
| `OUT` | Ein flankengesteuerter Dual-Rail Ausgang |

Pro Bit sorgen zwei Edge-Generatoren abhängig vom Pegel des Eingang `IN`, wenn `CE` auf `1` liegt, dafür, daß am jeweiligen Ausgang $OUT_0$ oder $OUT_1$ eine Flanke generiert wird.

**Konverter von flankengesteuertem Dual-Rail nach konventioneller Logik**

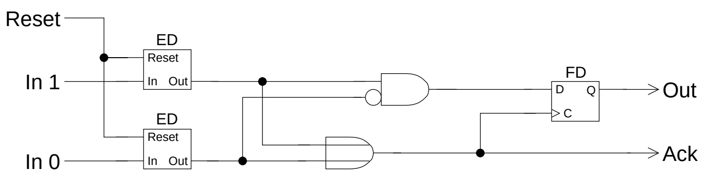



Abbildung 4.11: Konverter von flankengesteuertem Dual-Rail in konventionelle Logik

| Eingänge | Beschreibung |
|---|---|
| `IN` | Ein flankengesteuerter Dual-Rail Eingang |
| **Ausgänge** | **Beschreibung** |
| `OUT`<br>`ACK` | Ein Ausgang in konventioneller Logik<br>Acknowledge-Signal, wenn der Wert am Ausgang anliegt. |

Diese Konverter warten auf einen Flankenwechsel an einem der beiden Eingänge. Je nachdem, welcher von ihnen flankt, liefert das interne Flip-Flop am Ausgang `OUT 0` bzw. `1`. Nachdem das „richtige“ Signal am Ausgang `OUT` anliegt, wird der Ausgang `ACK` auf `1` gelegt. Durch einen Flankenwechsel an `Reset` werden die internen Edge-Detectoren wieder zurückgesetzt. Dies geschieht, wenn für jedes Bit das `ACK` erzeugt wurde.

### Implementierungsversuch der DEP-CPU mit flankengesteuerte Dual-Rail

Hier wird beschrieben, wie die DEP-CPU (siehe Kapitel 3) in dem flankengesteuerten Dual-Rail implementiert wurde. Dabei werden zunächst sämtliche elementaren Bauteile der asynchronen CPU beschrieben, aus denen dann die komplexeren Funktionsgruppen erstellt wurden.

Dabei wurde die Kommunikation der größeren Baugruppen untereinander in Dual-Railimplementiert, die `Select`- und `ChipEnable`-Leitungen jedoch aus Platzersparnisgründen in konventioneller Logik realisiert.

Die CPU wurde mit dem *Schematic Editor* der *Xilinx, Inc.* erstellt.

#### Der DEP-Decoder

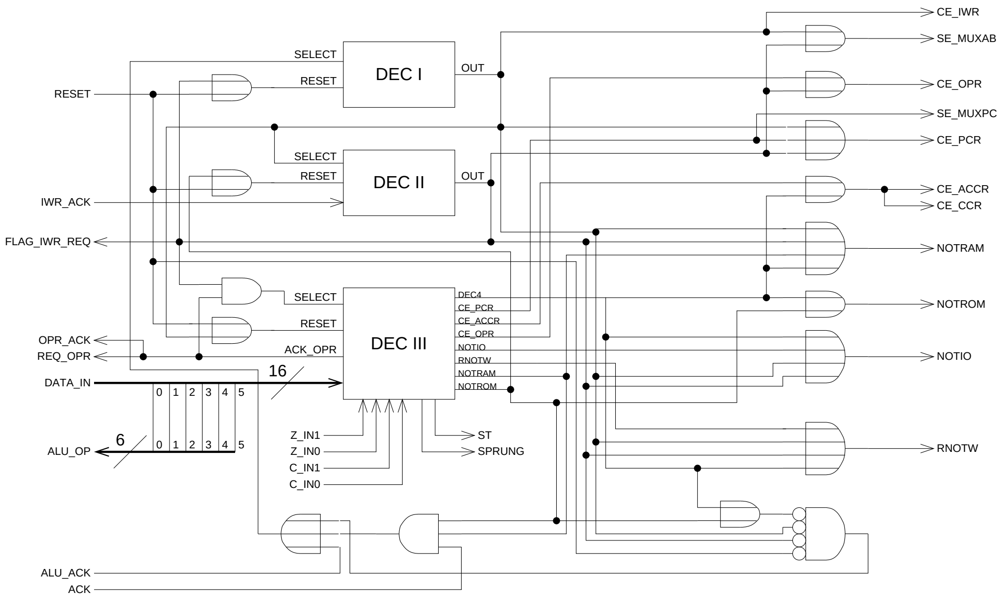



Abbildung 4.12: Asynchrone Implementierung des DEP-Decoders in flankengesteuerte Dual-Rail

Wie auch der Decoder des Micro-Pipelining-Ansatzes (siehe Abschnitt 4.1) wurde dieser Decoder dreigeteilt, um so eine einfachere Ablaufsteuerung zu erhalten. Das Layout des Decoder ist in Abbildung 4.2.1 zu sehen. Die drei Teile rufen sich gegenseitig auf und sind immer nur abwechselnd aktiv. Dabei wird auf die jeweilige Benachrichtigung der Komponenten gewartet, die fertig sein müssen, um den ordnungsgemäßen Ablauf eines Programmes sicherzustellen. Dadurch kann auch das **`MSR`** eingespart werden, da der aktuelle Zyklus des Decoder durch die aktivierte Decoderstufe determiniert wird.

Die **`Decoder I`** und **`II`** bestehen ausschließlich bzw. hauptsächlich aus Flip-Flops, die den Ausgang so lange auf `high` legen, bis die aktivierten Register (**`IWR`** bzw. **`OPR`**) die Werte gespeichert haben. Danach wird die dritte Decoderstufe aktiviert, in der abhängig vom geladenen Instruktionswort die Steuerleitungen geschaltet werden. Die Ein- und Ausgänge des Decoders entsprechen denen der DEP-CPU aus Kapitel 3, lediglich die Breite der Busse mußte angepaßt werden.

Da die Steuerleitungen der CPU keine flankengesteuerten Dual-Rail-Signale verwenden, sondern Standard-Logik, mußten nach der Decodierung der Befehle Edge-Detectoren plaziert werden, um die Signale dann an die weiteren Bauteile zu führen.

Die Ausgaben der einzelnen Decoderstufen wurden der Decodertabelle folgend verodert, so daß das gleiche Schaltverhalten wie in der synchronen Variante erreicht wurde.

Zusätzlichen Aufwand erforderten die Implementierung der beiden unterschiedlichen Adressierungsarten, da in dem Fall der absoluten Adressierung zwei mal ein `ACK` vom **`OPR`** gesendet wird. Durch eine Fallunterscheidung mit Zwischenspeicherung der Signale konnte aber eine relativ einfache Lösung gefunden werden.

Bei arithmetischen Befehlen fungiert die ALU als eine Art zwischengeschaltete vierte Stufe, die mit dem Beenden ihrer Rechnung und der erfolgreichen Speicherung bzw. Ausgabe des Ergebnisses die erste Decoderstufe aktiviert.

**Die DEP-AGU**

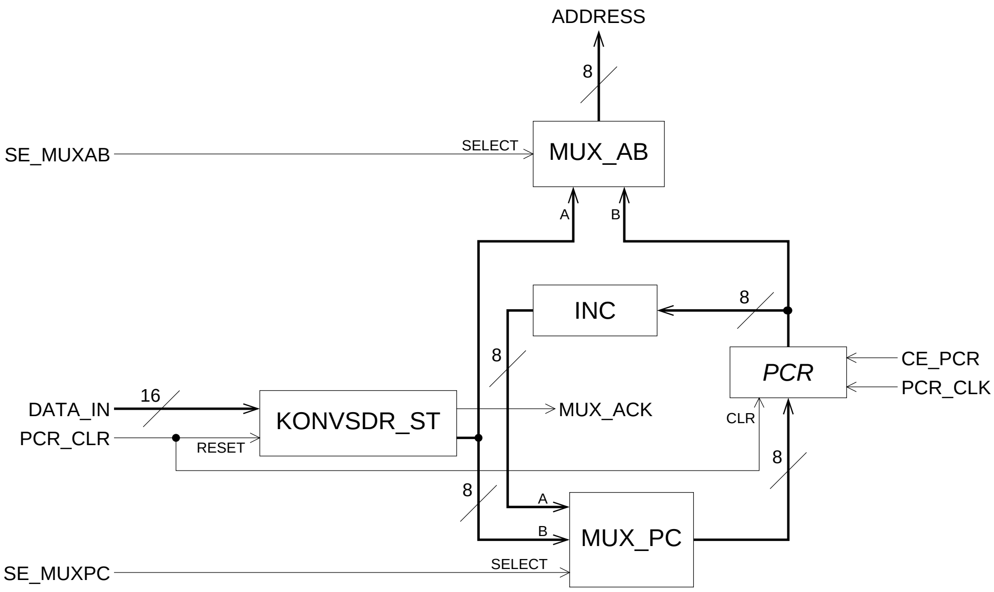



Abbildung 4.13: DEP-AGU in flankengesteuertem Dual-Rail

Die **`AGU`** generiert die Adressen, auf die zugegriffen wird. Dabei mußte dieser Teil der CPU in konventioneller Logik implementiert werden, da Dual-Rail-Signale sich nicht effektiv multiplexen lassen. Deswegen wurde der vom **`OPR`** kommende Datenbus in konventionelle Logik konvertiert und dann mit diesem weitergearbeitet. Die eigentliche **`AGU`** (ohne den Konverter) ist genau so aufgebaut wie ihr synchrones Pendant, die eigentliche Arbeit steckt hierbei in der Ansteuerung des Konverters und des **`PCR`**s. Auch hierfür wurde eine einfache Lösung gefunden: Das Signal `Phi2_ACK`, das von der Peripherie gesendet wird, um die erfolgreiche Übernahme der Werte mitzuteilen, wird zur *Taktung* des **`PCR`** verwendet. Dadurch liegen zwar oft unsinnige Werte am Adreßbus an, da das Signal `Phi2_ACK` öfter als benötigt anliegt, aber immer nur dann, wenn die Adresse ohnehin von der angeschlossenen Peripherie nicht ausgelesen wird.

**Die DEP-ALU**

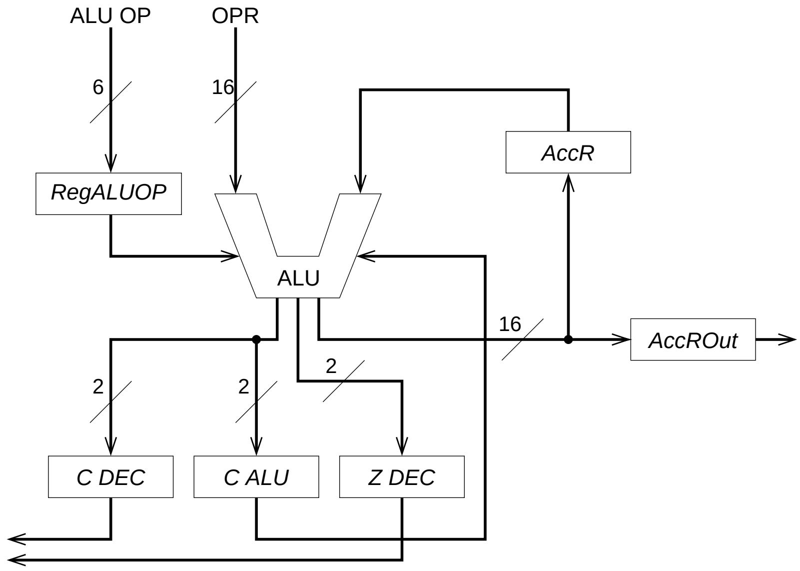



Abbildung 4.14: DEP-ALU in flankengesteuertem Dual-Rail – schematisch

Die eigentliche **ALU** ist eine genaue Kopie der ursprünglichen DEP-CPU (siehe Kapitel 3). Probleme macht die Ansteuerung. Der grundsätzliche Unterschied zwischen der synchronen und der Dual-Rail-CPU besteht darin, daß der Zeitfaktor mit durch die Berechnungen geschliffen wird. So ist genau bekannt, wann ein Rechenschritt zu Ende ist und der nächste beginnen kann. Dieses Verfahren zieht nach sich, daß jeder Wert nur einmal anliegt und nicht mehr bekannt ist, wenn er einmal verarbeitet wurde. Die **ALU** erhält den ALU-OP Befehl sobald das **IWR** geladen wurde. Wenn nun kein IMM-Befehl anliegt, rechnet die **ALU** zwar, ohne daß nachher das Ergebnis einen Nutzen hat, aber mit der Folge, daß der ALU-OP Befehl bei dem nächsten Schritt, in dem die Rechenoperation ausgeführt wird, nicht mehr bekannt ist. Hierfür haben wir einen Zwischenspeicher eingefügt, der den ALU-OP Befehl auf Abruf (`Req`-Signal) bereithält.

Ähnliche Probleme gibt es auch beim zweiten Eingang der **ALU**, an dem der Inhalt des **AccR** anliegen soll und dem **CCR**. Auch hier muß dafür gesorgt werden, daß das Signal wiederholt übertragen wird. Damit die **ALU** jedesmal, wenn sie rechnet, auch die Signale bekommt, wird an der **ALU** am ersten Eingang ein gültiges Dual-Rail-Signal erkannt und daraufhin ein `Req` an alle benötigten Speicher (**AccR**, **CCR** und ALU-OP) geschickt.

Hieraus resultieren zwei weitere Probleme: Zum einen kann das Ergebnis der **ALU** nicht wie bisher nach dem **AccR** auf den Datenbus geschrieben werden, da es hier bei den `Req`-Signalen zu Konflikten kommen würde. Also wird der Datenbus direkt an den Ausgang der **ALU** vor dem Eingang des **AccR** angeschlossen. Nun muß aber noch ein zweites **AccR** hinzugenommen werden um für den OUT-Befehl die Daten auf Abruf bereitzuhalten. Andererseits brauchen der **Decoder** und die **ALU** das Carry-Flag, so daß dies ebenfalls auf ein `Req`-Signal hin an das jeweilige Bauelement geschickt werden kann. Also wird auch hierfür ein zusätzlicher Speicher benötigt. Da dem Decoder auch insgesamt bekannt ist, wann die ALU fertig ist, werden die Carry- und Zero-Flag-Signale ausgewertet. Diese werden in der ALU als letztes ermittelt und stellen so ein sicheres Kriterium für den Abschluß der Rechenoperation dar.

**Resumee**

Da uns keine Informationen über die Interna der Soft- und Hardware der *Xilinx, Inc.* zur Verfügung standen, mußten wir bei auftauchenden Problemen experimentell ermitteln, welche Umgebungen oder Lösungen möglich sind. Dabei stellte sich heraus, daß mit Hilfe von *VHDL* oder dem *Schematic Editor* in der uns zur Verfgung stehenden Entwicklungsumgebung eine Implementierung von zeitkritischen Elementen nicht möglich ist. Recherchen in [ASYMT] ergaben, daß ein manueller Entwurf des Bitfiles, welches auf den FPGA übertragen wird, die einzige Möglichkeit darstellen würde, diese Idee umzusetzen. Hierdurch könnte man das Verhalten des FPGAs in der Simulation determinieren. Insgesamt ergab sich, daß dieser Ansatz bei der Implementierung der DEP-CPU 22 mal so groß wird, wie die synchrone Variante.

### 4.2.2 Pegelgesteuertes Dual-Rail

Wie schon oben angedeutet, wird beim pegelgesteuerten Dual-Rail nicht auf jede, sondern nur auf die steigende Flanke reagiert. Oder anders ausgedrückt: Eine `1`-Leitung mit Pegel `high` steht für eine gültige (siehe oben) `1`, ein `high` auf der `0`-Leitung für eine gültige `0`. Damit gibt es jetzt im Unterschied zum flankengesteuerten Dual-Rail „verbotene“ Zustände, wenn entweder beide Leitungen `low` oder beide Leitungen `high` sind. Diese beiden Zustände drücken nun gerade die Ungültigkeit des Ergebnisses aus und sagen der nachgeschalteten Logik, daß die vorhergehende Schaltung ihre Berechnung noch nicht abgeschlossen hat.

Per Definition werden beim Reset alle Dual-Rail-Leitungen auf `low` gelegt. Nach Rücknahme des Reset warten alle Baugruppen auf definierte, „erlaubte“ Eingangswerte. Liegen diese an, so führen sie ihre Berechnung durch und geben danach ihr Ergebnis als gültigen Dual-Rail-Wert an die nachfolgende Logik aus. Die „ersten“ Dual-Rail-Baugruppen, die nur übliche „Ein-Leiter“-Eingangssignale besitzen, können nach einem Reset direkt ihre Berechnung beginnen. Hat nun die letzte Dual-Rail-Baugruppe ihre Berechnung durchgeführt, so kann das Ergebnis ausgegeben und die nächste Berechnung durchgeführt werden.

Doch gerade da ist der Nachteil des pegelgesteuerten Dual-Rail: Angefangen bei der letzten Logik, die nun soeben das gültige Ergebnis ausgegeben und der ersten Baugruppe ein Signal zum Beginn der nächsten Berechnung gegeben hat, warten alle Dual-Rail-Baugruppen auf das Ende der Berechnung der vorhergehenden Logik. Nun müßte die letzte Logik eigentlich wegen der (alten) gültigen noch an ihrem Eingang anliegenden Signale erneut (dasselbe) Ergebnis ausgeben und der ersten Schaltung wieder ein Startsignal geben – noch bevor selbst die erste Logik ihre Berechnung abgeschlossen hat. Um das zu vermeiden sind die schon in Kapitel 4.2 genannten „Nullwellen“ nötig.

Erhält die erste Baugruppe nämlich das Startsignal der letzten Logik, so setzt sie zunächst ihre Ausgänge auf `low`. Dies ist für die zweite Logik das Zeichen, daß es noch kein gültiges Ergebnis gibt und setzt ihre Ausgänge ebenfalls auf `low`. Die letzte Logik schließlich setzt sich bei anliegenden undefinierten Eingängen – falls nötig – selbst zurück und gibt – ebenso nur wenn nötig – ein entsprechendes Signal „nicht gültig/noch am Rechnen“ an die Ausgänge. Erst wenn ihre Eingänge wieder gültig sind, werden auch die Ausgänge entsprechend geschaltet und wiederum der ersten Stufe das Startsignal gegeben.

**Ein AND-Gatter in pegelgesteuertem Dual-Rail**

Dem Nachteil der erforderlichen Nullwellen steht ja nun, wie schon oben angedeutet, ein großer Vorteil hinsichtlich des Bauteilaufwandes entgegen. So besteht ein pegelgesteuertes Dual-Rail-AND-Gatter aus gerade einmal zwei traditionellen Gattern, siehe Abbildung 4.15.

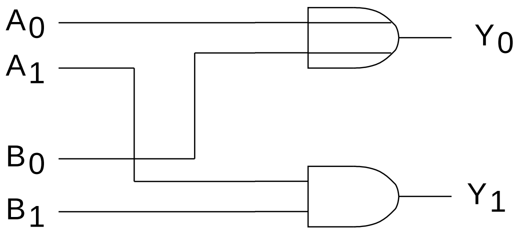



Abbildung 4.15: Pegelgesteuertes Dual-Rail-AND-Gatter

Der `1`-Ausgang wird durch eine AND-Verknüpfung der beiden `1`-Eingänge und der `0`-Ausgang durch eine OR-Verknüpfung der beiden `0`-Eingänge gebildet. Wie die Logiktabelle zeigt, bildet sich genau das gewünschte Verhalten. Daß die Ausgänge bei ungültigen Eingängen ebenfalls ungültig werden können, ist ja ebenfalls wünschenswert.

| $A_0$ | $A_1$ | $B_0$ | $B_1$ | $Y_0$ | $Y_1$ | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | |
| 0 | 0 | 0 | 1 | 0 | 0 | |
| 0 | 0 | 1 | 0 | 1 | 0 | |
| 0 | 0 | 1 | 1 | 1 | 0 | |
| 0 | 1 | 0 | 0 | 0 | 0 | |
| 0 | 1 | 0 | 1 | 0 | 1 | * |
| 0 | 1 | 1 | 0 | 1 | 0 | * |
| 0 | 1 | 1 | 1 | 1 | 1 | |
| 1 | 0 | 0 | 0 | 1 | 0 | |
| 1 | 0 | 0 | 1 | 1 | 0 | * |
| 1 | 0 | 1 | 0 | 1 | 0 | * |
| 1 | 0 | 1 | 1 | 1 | 0 | |
| 1 | 1 | 0 | 0 | 1 | 0 | |
| 1 | 1 | 0 | 1 | 1 | 1 | |
| 1 | 1 | 1 | 0 | 1 | 0 | |
| 1 | 1 | 1 | 1 | 1 | 1 | |

Tabelle 4.13: `0` = `low`, `1` = `high`

Markiert sind die Zustände mit gültigen Eingängen, also gerade die, die es auch bei einem „einfachen“ AND-Gatter gibt.

Zu beachten ist noch, daß es neben den markierten noch sechs weitere gültige Ausgangszustände gibt. Vier davon treten dann auf, wenn ein Eingang ein gültiges Signal führt, das alleine schon das Ergebnis bestimmt (beim AND-Gatter wäre dies `0`). Der jeweils andere Eingang führt noch kein gültiges Signal, die vorgeschaltete Logik ist also noch „am Rechnen“.

Gerade hier liegt jetzt der Vorteil der Dual-Rail-Technik, da die nachfolgende Logik schon „losrechnen“ kann, obwohl Teile der Vorgänger Ihre Berechnung noch nicht abgeschlossen haben. In der synchronen Technik wäre bei ungültigem Eingang auch der Ausgang ungültig – zumindest nicht deterministisch gültig – obwohl das Verknüpfungsergebnis durch andere Eingänge schon feststeht.

Die verbleibenden zwei Fälle der Logiktabelle scheinen auf den ersten Blick kritisch, da hier beide Eingänge ungültig, der Ausgang jedoch gültig ist. Genauer betrachtet, ist in diesen Fällen aber immer ein Eingang „`high-high`“. Nun ist aber auch in den Fällen, in denen diese (ungültige) Konstellation ausgegeben wird mindestens ein Eingang „`high-high`“ – diese Ausgabe ist mithin nicht möglich.

Die obigen Ausführungen gelten für alle Gatter, die nach diesem Prinzip aufgebaut sind, z. B. OR-Gatter oder NOT-Gatter.

### Implementierung von Grundbausteinen mit pegelgesteuertem Dual-Rail

Bei komplexeren Schaltungen wie der DEP-CPU bietet sich (wie schon in den Abschnitten über flankengesteuertes Dual-Rail beschrieben wurde) eine Aufteilung in (Teil-)Logikgruppen an, beziehungsweise ist unumgänglich. Ebenso gilt beim pegelgesteuerten Dual-Rail wie beim flankengesteuerten Dual-Rail, daß nicht unbedingt alle Logikfunktionen einer traditionellen Schaltung (z. B. einer CPU) in ihre Dual-Rail-Pendants umgewandelt werden müssen, respektive können. Letzteres ist beim pegelgesteuerten Dual-Rail nun allerdings nicht mehr das Problem, da selbst Flip-Flops und somit auch Register ohne *prinzipielle* Änderungen möglich sind.

Alle verwendeten Bausteine wurden in der Hardwarebeschreibungssprache *VHDL* entwickelt.

#### Gatter

Die Gatter beim pegelgesteuertem Dual-Rail-Ansatz wurden nicht als eigenständige Bauteile (`entity`, `architecture`, `component`), sondern als reine *VHDL*-Funktionen auf neu definierten Typen implementiert, werden also direkt in der Sprache verwendet. Abbildung 4.16 zeigt die Funktionsdefinitionen des AND-Gatters und des OR-Gatters als Schaltungen.

Die Anschlüsse sind jeweils vom Typ `dr_bit` und können innerhalb von *VHDL* abstrakt als eine Leitung benutzt werden. Desweiteren wurden die Typen `dr_byte` für Busse aus acht `dr_bits` (also insgesamt 16 Leitungen) und `dr_vec` für Busse beliebiger Breite und die dazugehörigen Funktionen auf diesen Typen definiert. So existieren z. B. AND-Verknüpfungen zwischen `dr_bit` und `dr_bit` und zwischen `dr_vec`

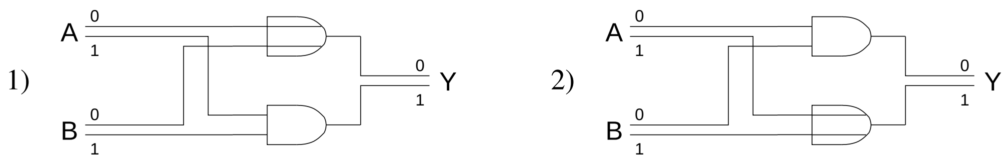



Abbildung 4.16: 1) Pegelgesteuertes Dual-Rail-AND-Gatter und 2) pegelgesteuertes Dual-Rail-OR-Gatter

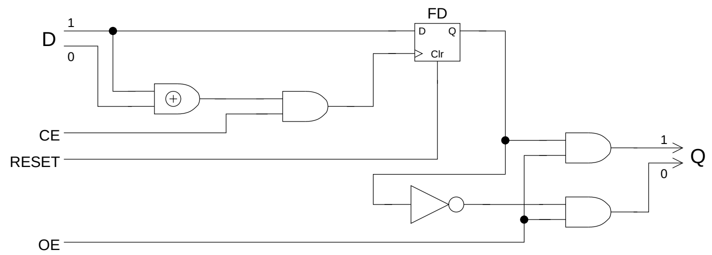



Abbildung 4.17: D-Flip-Flop in pegelgesteuertem Dual-Rail

und `dr_vec`, aber auch zwischen `bit` und den `dr_` Typen zum expliziten „Ein- und Ausschalten“ dieser Leitungstypen.

Allgemein hat jede Funktion bis auf eine paar Ausnahmen folgende „Anschlußbelegung“:

| **Eingänge** | **Beschreibung** |
|---|---|
| `A` | Dual-Rail-Eingang |
| `B` | Dual-Rail-Eingang |
| **Ausgänge** | **Beschreibung** |
| `Y` | Dual-Rail-codierte Verknüpfung von `A` und `B` |

Ein Ausnahme wäre z. B. das NOT-Gatter, welches natürlich nur einen Eingang hat.

**Speicher**

Speichernde Elemente sind im pegelgesteuerten Dual-Rail-Ansatz kein größeres Problem, solange der Ausgang eines solchen nicht über ein Schaltnetz in seinen Eingang zurückgeführt wird. Diese Problematik ergab sich bei der Umstellung der AGU, die den Ausgang von **`PCR`** über einen Inkrementer und einen Multiplexer wieder in den Eingang von **`PCR`** führt.

Ein einfaches pegelgesteuertes Dual-Rail D-Flip-Flop vom Typ **`DR_DFF`** (siehe Abbildung 4.17) besitzt folgende Anschlußbelegung:

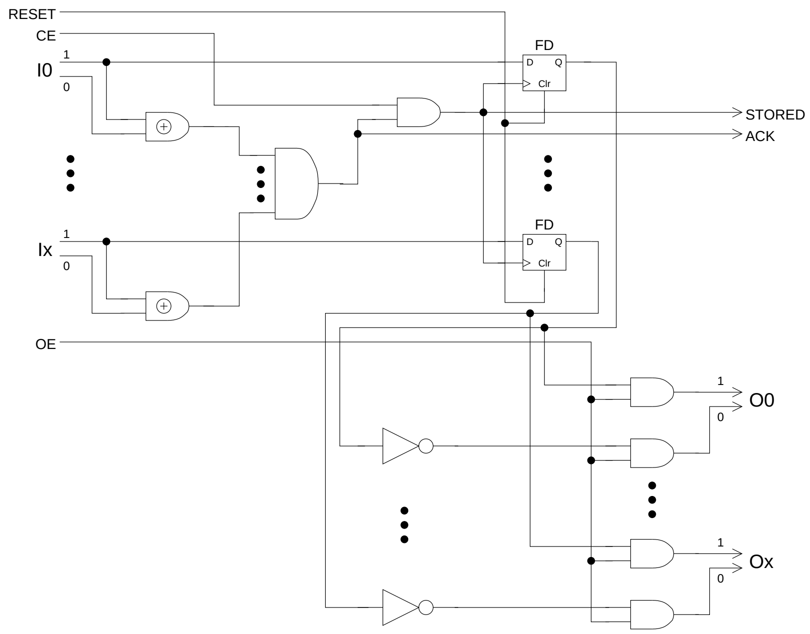



Abbildung 4.18: Register in pegelgesteuertem Dual-Rail

| **Eingänge** | **Beschreibung** |
|---|---|
| `RESET` | Asynchrones Reset |
| `CE` | Chip-Enable |
| `OE` | Output-Enable |
| `D` | Dual-Rail-Eingang |
| **Ausgänge** | **Beschreibung** |
| `Q` | Dual-Rail-Ausgang |

Wenn `CE` auf `high` gezogen ist, wird der an `D` anliegende Wert gespeichert, sobald er definiert ist. Wird `OE` auf `low` gezogen, gibt `Q` einen undefinierten Wert aus (beide Leitungen auf `low`), sonst den gespeicherten Wert in Dual-Rail-Codierung.

Normale Register vom Typ **`DR_REGx`** (siehe Abbildung 4.18), deren Ausgänge nicht wieder in ihre Eingänge zurückgeführt werden müssen, besitzen folgende Anschlüsse:

| **Eingänge** | **Beschreibung** |
|---|---|
| `RESET` | Asynchrones Reset |
| `CE` | Chip-Enable |
| `OE` | Output-Enable |
| `I` | Dual-Rail-Eingang |
| **Ausgänge** | **Beschreibung** |
| `ACK` | ist `high`, wenn `i` einen definierten Wert führt |
| `STORED` | ist `high`, wenn `i` einen definierten Wert führt und `CE` auf `high` gezogen wurde |
| `O` | Dual-Rail-Ausgang |

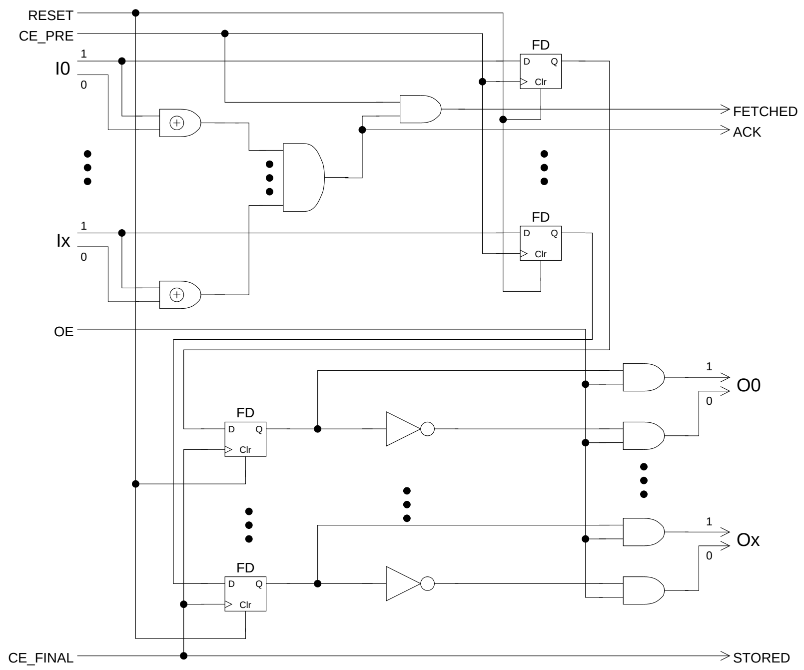



Abbildung 4.19: Mehrschrittiges Register in pegelgesteuertem Dual-Rail

Ein solches Register kann aus den oben genannten **DR_DFF** aufgebaut werden und zeigt über zwei Leitungen den Zustand seiner Eingänge an. Ist `CE` auf `high` gezogen ist, wird der an `I` anliegende Wert gespeichert, wenn er vollständig (also auf allen dazugehörigen Leitungen gleichzeitig, denn `I` ist normalerweise ein Dual-Rail-Bus) definiert ist. Die Anschlüsse `ACK` und `STORED` ermöglichen der umgebenden Logik, das mit dem Register verbundene Timing einzuhalten.

Die oben beschriebene Version des pegelgesteuerten Dual-Rail-Registers speichert den Wert an seinem Eingang, wenn dieser gültig und `CE` auf `high` gezogen ist. Wie schon erwähnt, ist ein solches Register für rückgeführte Ausgänge ungeeignet, ein weiterer Entwurf beinhaltet eine Reihe herkömmlicher D-Flip-Flop als Puffer für den Eingang, deren Inhalte erst in einem weiteren Schritt im „eigentlichen“ Speicher gesichert werden, um die Ausgabe nicht schon beim Speichern zu verändern.

**Beispiel für den Fall eines rückgeführten Ausgangs:** Der Ausgang des Inkrementers in der DEP-AGU ist undefiniert, solange `OE` von **PCR** auf `low` gezogen und damit auch der Eingang des Inkrementers undefiniert ist.
Werden `OE` und `CE` aber beide wie in der synchronen Variante der DEP-CPU auf `high` gezogen, sollte man zunächst vermuten, daß der Ausgang von **PCR** durch den Inkrementer um eins erhöht und bei Beendigung der Berechnung „**PCR**+1“ das Ergebnis in **PCR** gespeichert wird. Dem ist auch so, jedoch wird der gerade neu berechnete und gespeicherte Wert von **PCR** wieder ausgegeben, weil `OE` immer noch `high` ist, der Inkrementer berechnet wieder einen neuen Wert und legt während der Berechnung an seinem Ausgang kurzzeitig einen undefinierten Wert an, der aber schnell zu einem Definiertem und abermals von **PCR** Gespeichertem wird. Dieser Kreislauf setzt sich solange fort, wie an beiden Anschlüssen `CE` und `OE` `high` anliegt.

Die daraufhin entwickelte Lösung vom Typ **DR_REG8_PREFETCH** (Abbildung 4.19) speichert den anliegenden Wert in zwei Stufen, besitzt also noch ein zusätzliches Puffer-Flip-Flop pro gespeichertes Bit.

| **Eingänge** | **Beschreibung** |
|---|---|
| `RESET` | Asynchrones Reset |
| `CE_PRE` | erstes Chip-Enable für den Puffer |
| `CE_FINAL` | zweites Chip-Enable für die Übernahme |
| `OE` | Output-Enable |
| `I` | Dual-Rail-Eingang |
| **Ausgänge** | **Beschreibung** |
| `ACK` | ist `high`, wenn `i` einen definierten Wert führt |
| `FETCHED` | ist `high`, wenn `i` einen definierten Wert führt und `CE_PRE` auf `high` gezogen wurde (entspricht dem `STORED` des normalen pegelgesteuerten Dual-Rail Registers) |
| `STORED` | ist `high`, wenn `CE_FINAL` auf `high` gezogen wurde |
| `O` | Dual-Rail-Ausgang |

Die Funktionsweise ist ähnlich dem oben beschriebenen Register, es besitzt jedoch zwei `CE`-Leitungen – eine für den dem Nutzspeicher vorgeschalteten Puffer und eine für den Speicher selbst. Liegt an `I` ein vollständig definierter Wert an, kann er mit `CE_PRE` im Puffer gespeichert werden. Dann, unabhängig von `I`, kann später der Pufferinhalt in den Speicher übernommen werden, der dann auch Dual-Rail-codiert am Ausgang `O` anliegt. Die Signale `ACK`, `FETCHED` und `STORED` dienen wiederum der Synchronisation.

#### Multiplexer

Anstelle von Multiplexern werden im pegelgesteuertem Dual-Rail einfache OR-Gatter benutzt, in *VHDL* als Funktion `connect()` implementiert. Zur Auswahl der Leitung, die durchgelassen werden soll, werden alle anderen in den Multiplexer gehenden Leitungen auf `low`, also undefiniert, gezogen. Der Multiplexer gibt damit nur die eine Leitung aus, die einen definierten Wert führt.

Beliebige Leitungen können zu diesem Zwecke durch mit einer Steuerleitung verknüpftem AND-Gatter „ausgeschaltet“ werden, die Register haben hierfür den `OE`-Anschluß. Die Steuerleitungen der ursprünglichen Multiplexer werden also an die `OE`-Anschlüsse der Register und an die mit den zu steuernden Bussen verbundenen AND-Gatter geführt.

Durch diese einfache Maßnahme werden im Verhältnis zu einer direkten Umsetzung der Schaltung eines Multiplexers in pegelgesteuertes Dual-Rail viele Gatter eingespart.

### Implementierung der DEP-CPU in pegelgesteuertem Dual-Rail

Zur Entwicklung des pegelgesteuerten Dual-Rail wurde die DEP-CPU herangezogen, die *VHDL*-Quellen der DEP-CPU aus Kapitel 3 dienten als Grundlage. Alle Schaltnetze aus den Originalquellen konnten einfach übernommen werden, indem ein *VHDL*-Package entworfen wurde, das unter anderem den Datentypen `dr_bit` und dazugehörige Funktionen definierte und in der DEP-CPU der Datentyp `bit` durch `dr_bit` ersetzt wurde. Weitere Änderungen waren für die Schaltnetze nicht nötig.

Diesem einfachen Schritt gingen natürlich die oben erläuterten theoretischen Überlegungen voraus, die auch eine allgemeine Lösung für Schaltwerke in Form verschiedener Registervarianten hervorbrachten. Bei der Umsetzung der Theorie sollte der Decoder ohne **MSR** auskommen, da der Zustand der CPU auch durch die „Position" der Daten innerhalb der Schaltung festgestellt werden kann.

#### Der DEP-Decoder

Für die Realisierung der DEP-CPU in pegelgesteuertem Dual-Rail wurde eine Aufteilung des Decoders in mehrere Teildecoder gewählt. Jeder Teildecoder steht für einen bestimmten Arbeitsschritt während der Abarbeitung eines Befehls und läuft „exklusiv" einer nach dem anderen. Die in der synchronen CPU verwendeten MSR-Zustände spiegeln sich so in den Teildecodern wieder. Vorteil ist ein sehr leichtes und zuverlässiges „Hand-in-Hand-gehen" der einzelnen Teildecoder. Jeder Teildecoder hat einen oder mehrere vorhergehende Teildecoder, auf den oder die er wartet und einen oder mehrere Nachfolgedecoder, der oder die ihrerseits auf die abgeschlossene Berechnung warten. Hat ein Decoder das „Ok" von seinem Vorgänger, so startet er die ihm zugewiesene eigentliche Rechenlogik. Ist diese fertig (zu erkennen an gültigen Dual-Rail-Zuständen), so startet er wiederum den Nachfolgedecoder.

##### DR_GROUP_START

Zwar unterscheiden sich die Teildecoder in ihrer Funktionsweise, jedoch ist allen das Warten auf den Vorgänger, das Starten der jeweiligen Logik, das Warten auf diese Logik und das Starten des Nachfolgers gemein. Für diese gemeinsame Funktion wurde ein eigenes Bauteil **DR_GROUP_START** entwickelt.

| **Eingänge** | **Beschreibung** |
|---|---|
| `START` | Signal vom Vorgänger |
| `DONE` | Soll-Status erreicht |
| `NEXT` | Signal an den Nachfolger |
| `RESET` | Initialisierung |
| **Ausgänge** | **Beschreibung** |
| `CALC` | Berechnung im Gange |

Wenn die mit dem Bauteil assoziierte Logik mit seiner Berechnung anfangen soll, wird der Eingang `START` auf `high` gesetzt. Der Ausgang `CALC` gibt `high` aus, nachdem `START` auf `high` gesetzt wurde, ansonsten `low`, er muß also in die Logik eingebunden werden, die von dem jeweiligen **DR_GROUP_START** gesteuert werden soll.

Um anzuzeigen, daß die Berechnung beendet ist, muß die Leitung `DONE` auf `high` gesetzt werden, als Reaktion wird `CALC` von **DR_GROUP_START** wieder auf `low` gesetzt. Mit einer fallenden Flanke auf `DONE` wird auf `NEXT` ein kurzer Impuls erzeugt, der dem nachfolgenden Teildecoder anzeigt, daß er seinen Logikblock starten soll.

`DONE` reagiert auf die steigende *und* die fallende Flanke, um sicherzustellen, daß es keine Timingprobleme gibt. Denn würde `NEXT` den Impuls geben, sobald die steigende Flanke auf `DONE` kommt, wäre es möglich, daß der `CALC`-Ausgang einer nachfolgenden **DR_GROUP_START** auf `high` gesetzt wird, noch während `CALC` der vorherigen **DR_GROUP_START** `high` ausgibt. Zwei Logikblöcke wären gleichzeitig aktiv, deren Wechselwirkungen jedoch in den meisten Fällen zu einem totalen Chaos führen würden.

### CMS

Um die „Ok“/„Ready“-Leitungen der einzelnen Bauteile (Speicherbestätigungen der Register, Rechnungsendebestätigung der ALU, etc.) mit dem jeweiligen Teildecoder zu verbinden, steht dem Decoder noch eine „CMS“ – „Check-Machine-State“ genannte Komponente zur Seite. Diese besteht aus reiner Kombinatorik und erlaubt ein einfacheres Design des Decoders.

### Die Teildecoder

Für alle Befehle gleich ist der Durchlauf der ersten sechs Teildecoder:

- **IWR0**, **IWR1**, **IWR2** legen die nächste Adresse an den Adreßbus, lesen und speichern das Instruction Word.
- **OPR0**, **OPR1**, **OPR2** erledigen das gleiche für den Operanden

Danach folgen – abhängig vom **IWR** – ein oder mehrere der folgenden Teildecoder:

- **BRTAKE0**, **BRTAKE1** führen einen Sprung durch
- **FETCHIND0**, **FETCHIND1** holen ein indirekt adressiertes Datum
- **ALUOP0**, **ALUOP1** führen eine ALU-Operation durch
- **LDA0**, **STA0** laden, beziehungsweise speichern den Akkumulatorinhalt

### Die ALU

Da es sich beim pegelgesteuerten Dual-Rail um eine statische Technik handelt und die ALU ein Schaltnetz ist, konnte sie durch einfaches Ersetzen der Datentypen `bit` und `bit_vector` in der Bauteildefinition durch `dr_bit`, `dr_vec` und `dr_byte` auf pegelgesteuertes Dual-Rail umgestellt werden. So konnte die synchrone ALU fast eins zu eins übernommen werden. Die pegelgesteuertee Dual-Rail-ALU hatte gegenüber der synchronen nur noch einen zusätzlichen Ausgang, der die Gültigkeit der Ausgabe anzeigt. Die restliche Umstellung ergab sich aus der „**CMS**“-Komponente und dem jeweiligen Teildecoder.

Die mit den neuen Typen notwendig gewordenen Verknüpfungen wurden im Zuge der Umstellung im oben beschriebenen *VHDL*-Package definiert. Ein Test der ALU im Simulator von *Xilinx Foundation Series 3.1i* brachte die Erkenntnis, daß die Umstellung die richtigen Ergebnisse lieferte und auf alle weiteren Schaltnetze anwendbar sein würde.

**Resumee**

Die asynchrone Realisierungsform des pegelgesteuerten Dual-Rail ist sehr einfach umzusetzen und benötigt nur rund das zweieinhalb- bis vierfache an CLBs der synchronen Variante. Dafür kosten die Nullwellen Zeit, weshalb sich diese Technik – ebenso wie die anderen asynchronen – nur bei größeren Schaltungen mit möglichst wenig Kommunikation nach außen lohnt.

Da wir aufgrund eines fehlerhaften Simulators der *Xilinx Foundation Series 3.1i* zu lange für die Entwicklung des **`DR_GROUP_START`** benötigten, konnten wir diese Technik nicht mehr an dem (relativ) großen Design eines 6502 ausprobieren. Bei ersten Tests im Simulator konnten mit dem DEP-CPU-Design keine größeren Leistungsgewinne festgestellt werden. Da der Simulator aber fehlerhafte Ausgaben produzieren kann und Simulatormessungen sowieso nicht viel über die tatsächlichen Rechenzeiten aussagen, sollte dieses Ergebnis nicht überbewertet werden.

> Der Simulator bietet nach dem ersten Implementierungsschritt („Synthese") neben einer rein funktionalen auch eine Timing-Simulation an. Beide Ergebnisse unterscheiden sich auch, jedoch sind selbst die (zuverlässigeren) Ergebnisse der Timing-Simulation nicht mit dem zugehörigen *VHDL*-Design zu erklären. Erst die Simulation nach der kompletten Implementierung liefert ein (anscheinend) korrektes Timing. Da die Unterschiede zwischen synchronem und asynchronem Design gerade bei diesem Bauteil zum Tragen kommen, führten die Timing-Unterschiede anscheinend zu einem sogar funktional falschem Ergebnis.
> Jedoch kann auch ein fehlerfreier Simulator nur eine Abschätzung des Timings einer logischen Schaltung liefern. „Gute" Simulatoren geben für einzelne Logikpfade 3 Zeiten an (minimale, durchschnittliche, maximale Laufzeit); üblich ist nur eine Laufzeit je Pfad. So wird schnell klar, daß die Simulatorergebnisse bei größeren Designs merklich von den realen Werten abweichen – keine zwei logischen Gatter besitzen die gleichen Schaltzeiten, die Einzelabweichungen addieren sich.

Ein genauere Untersuchung in weiteren Projekten ist diese Form des Dual-Rail sicherlich wert.

## 4.3 *PsalM*-Dual-Rail

### 4.3.1 Schaltnetze

Die bisherigen Ansätze erwiesen sich für die Realisierung des 6502 als nicht praktikabel. Entweder waren sie zu groß, es traten Timingprobleme auf o.ä. Aus diesem Grunde entwickelten wir einen weiteren Ansatz, der auf einer einfachen Idee basierte: Die Edge-Detectoren und Edge-Generatoren[1] sollten sich selbst um ihr Timing kümmern. Jedes Bauteil, auch die größeren, wird deswegen prinzipiell nach dem Schema aus Abb. 4.20 aufgebaut.

[1] Genauere Schaltpläne ihres inneren Aufbaus, der von dem bisher erwähnten sehr stark abweicht, folgen in Abschnitt 6.1.

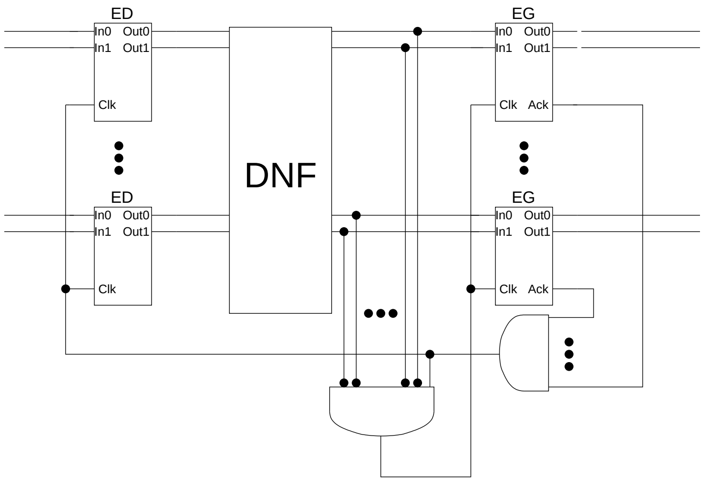



Abbildung 4.20: Schema der gewählten Dual-Rail-Variante

Begriffserklärung:

**DNF** Disjunktive Normalform; Disjunktion von Konjunktionen. Eine DNF hat somit z.B. die Form

$$X = ((A_{0_0} \wedge A_{1_0} \wedge \ldots \wedge A_{n_0}) \vee \ldots \vee (A_{0_k} \wedge \ldots \wedge A_{n_k}))$$

**KNF** Konjunktive Normalform; Konjunktion von Disjunktionen. Eine KNF hat somit z.B. die Form

$$X = ((A_{0_0} \vee A_{1_0} \vee \ldots \vee A_{n_0}) \wedge \ldots \wedge (A_{0_k} \vee \ldots \vee A_{n_k}))$$

Im Normalzustand liegen alle Ausgänge der Edge-Detectoren auf `low`. Ein ankommender Flankenwechsel an ihren Eingängen wird von den Edge-Detectoren nun in ein pegelgesteuertes Dual-Rail Signal umgewandelt.

Deswegen ist es möglich, die DNF nach pegelgesteuerter Dual-Rail Logik aufzubauen. Bekommt diese DNF nun ein pegelgesteuerte Dual-Rail Signal an ihren Eingängen, so gibt es nach einiger Gatterlaufzeit ein pegelgesteuerte Dual-Rail Signal an ihren Ausgängen wieder aus. Aus technischen Gründen ist es erforderlich, die DNF in *VHDL* 1:1 zu übernehmen, was durch eine geschickte Wahl der Datentypen unter *VHDL* auch erreicht werden kann. Siehe dazu Abschnitt 4.3.1 Dadurch kann sichergestellt werden, daß keine Hazards auftreten. Von Seiten der DNF ist dies gewährleistet, weil keine negierten Literale erforderlich sind. Stattdessen kann jeweils die `0`-Leitung des pegelgesteuerten Dual-Rail verwendet werden.

Die Edge-Generatoren auf der rechten Seite nehmen das pegelgesteuerte Dual-Rail Signal von der DNF, und bereiten in ihrer Master-Stufe (s.u.) einen Flankenwechsel an ihren Ausgängen vor. Haben alle Edge-Generatoren einen Flankenwechsel vorbereitet, so wird dies als der Zeitpunkt interpretiert, an dem das Ergebnis der DNF fertig berechnet ist. Die `ACK` Leitung wird auf `high` gelegt, und die Edge-Detectoren am Eingang somit zurückgesetzt.

Durch das Zurücksetzen der Edge-Detectoren gehen ihre Ausgänge wieder zurück auf `low`. Dadurch wird die Nullwelle in der DNF ausgelöst. Aufgrund der erwähnten Gatterlaufzeiten muß gewartet werden, bis sich die Nullwelle bis zum Ausgang der DNF fortgepflanzt hat. Dies erkennt das untere AND-Gatter. Sobald die Eingänge der Edge-

Generatoren nur noch `low`-Pegel aufweisen, legt es seinen Ausgang auf `high`. Dies steuert die Slavestufe der Edge-Generatoren an, welche nun endgültig den Flankenwechsel auslösen.

Insgesamt ist durch dieses System gewährleistet, daß auch wirklich *alle* Ausgänge eines Schaltnetzes zum selben Zeitpunkt einen Flankenwechsel aufweisen. Dadurch werden zeitkritische Überlegungen an anderer Stelle überflüssig.

#### Innerer Aufbau der DNF

Die DNF sollte sich in konventioneller Logik idealerweise auf die Gatter AND, OR, XOR und NOT beschränken, da hiermit die Übersetzung in pegelgesteuerte Dual-Rail-Gatter am einfachsten von der Hand geht. In unserem Ansatz haben wir für sie einfach einen neuen Datentyp eingeführt, und für die Gatter selbst die folgende Funktionen festgelegt[2].

**AND-Gatter** $X_0 = (A_0 \wedge B_0) \vee (A_0 \wedge B_1) \vee (A_1 \wedge B_0)$
$X_1 = A_1 \wedge B_1$

**OR-Gatter** $X_0 = A_0 \wedge B_0$
$X_1 = (A_1 \wedge B_0) \vee (A_0 \wedge B_1) \vee (A_1 \wedge B_1)$

**XOR-Gatter** $X_0 = (A_0 \wedge B_0) \vee (A_1 \wedge B_1)$
$X_1 = A_0 \wedge B_1) \vee (A_1 \wedge B_0)$

**NOT-Gatter** $X_0 = A_1$
$X_1 = A_0$

Füllt man die Wahrheitstabelle für diese Funktionen aus, und geht davon aus, daß eine `0` durch einen `high`-Pegel auf der `0`-Leitung, eine `1` hingegen durch einen `high`-Pegel auf der `1`-Leitung repräsentiert wird, so erkennt man schnell die ursprünglichen Funktionen wieder.

Bei den zweistelligen Funktionen fällt auf, daß wenn nur einer der beiden Eingänge einen `high`-Pegel auf einer Leitung aufweist, der Ausgang trotzdem noch einen `low`-Pegel trägt, solange, bis auch der zweite Eingang `high` ist. Dies ist ein Unterschied zum bisherigen pegelgesteuerten Dual-Rail: Dort genügte es, einen der Eingänge auf `high` zu legen, was im Endeffekt allerdings zu Timingproblemen geführt hatte.

Weiterhin fällt auf, daß durch einen `low`-Pegel auf beiden Leitungen das Ergebnis der Funktion selbst ebenfalls `low` ist. Die Grundvorraussetzung für eine Nullwelle.

### 4.3.2 Register

In diesem Ansatz werden die Register zweigeteilt. Ein Teil des Registers ist zum Speichern eingehender Werte gedacht, der andere für das Auslösen eines Flankenwechsels, wenn dies durch den Decoder gefordert wird. Dadurch kann ein einmaliger Flankenwechsel beliebig oft wiederholt werden.

[2] $X_0$ und $X_1$ sind dabei die `0`- bzw. `1`-Leitungen der Ausgänge, $A_0$, $A_1$, $B_0$ und $B_1$ die Eingänge. Alle Funktionen sind in konventioneller Logik beschrieben.

Ein weiterer Trick ist nun, daß an ein Register mehrere solche Ausgabebausteine angeschlossen werden können:

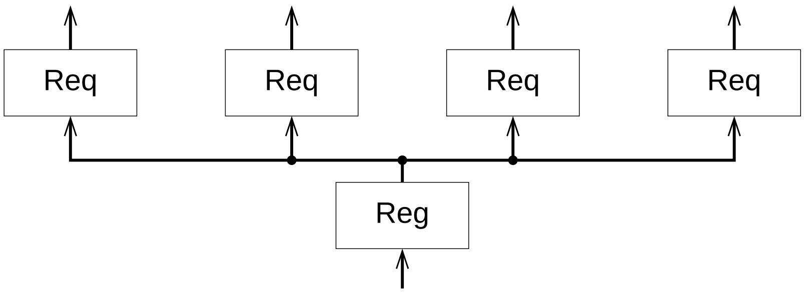



Abbildung 4.21: Schema der Register mit mehreren Ausgängen.

Die Kommunikation zwischen den beiden Teilen verläuft über einen pegelgesteuerten Dual-Rail Bus. Als Folge dieses Aufbaus ist es möglich, die Multiplexer sehr viel simpler (und platzsparender) aufzubauen. Allerdings wird dies durch eine höhere Komplexität des Decoders erkauft (siehe dazu Kapitel 6.3).

### 4.3.3 Multiplexer

Die Multiplexer wurden durch **`DDRFASTOR`** (Siehe hierzu Kapitel 6.1 Bausteine ersetzt. Diese lösen bei einem Flankenwechsel an einem ihrer beiden Eingänge einen Flankenwechsel an ihrem Ausgang aus.

### 4.3.4 Asynchrone Implementierung der DEP-CPU

Die asynchrone Implementierung der DEP-CPU beginnt mit der Implementierung des Decoders und der ALU. Sie werden mit dem in Abschnitt 4.3.1 erwähnten Schema übersetzt, ihre Schaltfunktionen ändern sich nicht.

Die AGU wird wie in Abb. 4.22 implementiert:

Die Leitungen, welche zuvor die `Select`-Leitungen für **`MUXAB`** und **`MUXPC`** darstellten, steuern nun die Requestbausteine für die Register an. Dadurch wird je nach Decodervorgabe der Wert eines der beiden durch Flankenwechsel entweder auf den Adreßbus gelegt oder an das **`PCR`** gelegt. **`INC+1`** ist wieder ein einfaches Schaltnetz, welches nach dem bekannten Schema übersetzt werden kann.

Nachdem sämtliche Komponenten übersetzt wurden, sollte man sich klar machen, wann welches Bauteil genau rechnen soll. Das einzige wirklich kritische Bauteil in unserem Ansatz stellt hierbei die ALU dar, die immer genau dann rechnen soll, wenn `CEACCR1` flankt, und ansonsten vor Flankenwechseln geschützt werden sollte.

In einem Befehlszyklus wird entweder auf den Speicher zugegriffen oder aber die ALU rechnet. Das Ende einer ALU-Operation wird durch einen Flankenwechsel auf einer der beiden `CARRY`-Leitungen, die aus der ALU kommen, angezeigt. Das Ende eines Speicherzugriffes sollte durch einen zusätzlichen CPU-Eingang gekennzeichnet sein, der immer dann einen Flankenwechsel aufweist, wenn der Zugriff abgeschlossen ist. Alle diese drei Leitungen werden an ein XOR-Gatter übergeben, an dessen Ausgang

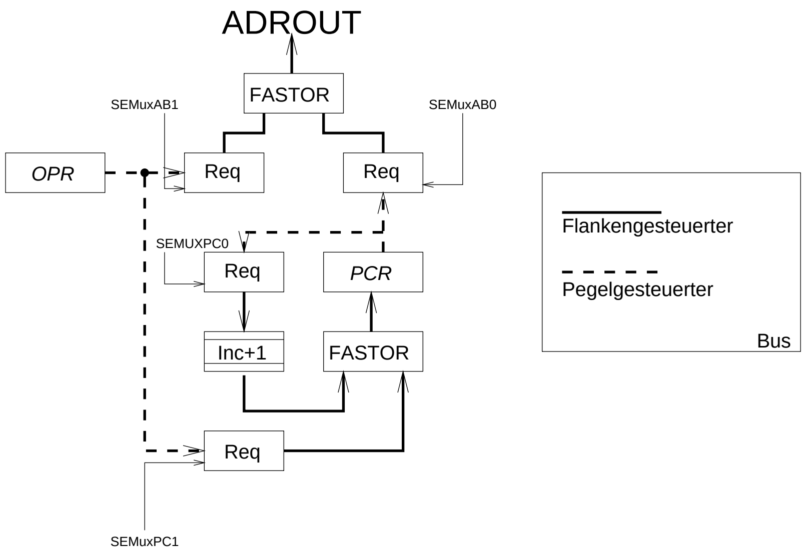



Abbildung 4.22: Schema der asynchronen AGU

das `nextStep` ausgegeben wird. Dieses zeigt dann mit einem Flankenwechsel das Ende eines Zyklus an.

Dann werden die Register im Topleveldesign gesetzt. Die einzelnen Register werden nun wie folgt mit ihren **REQ_DDR_REG** verbunden:

**OPR** Bekommt zwei **REQ_DDR_REG**, einen, der über `nextStep` requested wird, und dessen Ausgang wird zum Decoder geleitet wird. Einen weiteren, der bei einem `CEACCR1` requested wird, und dessen Ausgang als `ALUOP` in die ALU geht.

**ACCR** Bekommen zwei **REQ_DDR_REG**, der eine wird bei `CEACCR1` requested und zur ALU geleitet, der andere wird bei einem `RNOTW0` requested, und geht als `DATAOUTBUS` nach außen.

**OPR** Wird in drei Richtungen requested, wobei zwei davon in der AGU verwendet werden. Der dritte wird bei `CEACCR1` requested, und wird in die ALU geleitet.

**CCR** Wird in zwei Richtungen requested, einmal bei `CEACCR1` für die ALU, einmal bei `nextStep` für den decoder.

**MSR** Wird in eine Richtung requested, und zwar bei `nextStep` für den decoder.

Damit kommt dem zusätzlichen Eingang für die CPU die Rolle des Initiators zu: Bevor die DEP-CPU sinnvoll rechnen kann, muß erst hier ein Flankenwechsel stattfinden, der in der CPU selber alle weiteren Flankenwechsel auslöst.

Insgesamt sieht das Blockschaltbild der asynchronen DEP-CPU als *PsalM*-Dual-Rail somit aus, wie in Abb. 4.23

Zum Vergleich nochmal das Blockschaltbild der synchronen DEP-CPU in Abb. 4.24

Von all unseren Ansätzen war dies der einzige, welcher eine auch auf dem FPGA lauffähige DEP-CPU zur Folge hatte, weswegen wir ihn auch als Ansatz für den 6502

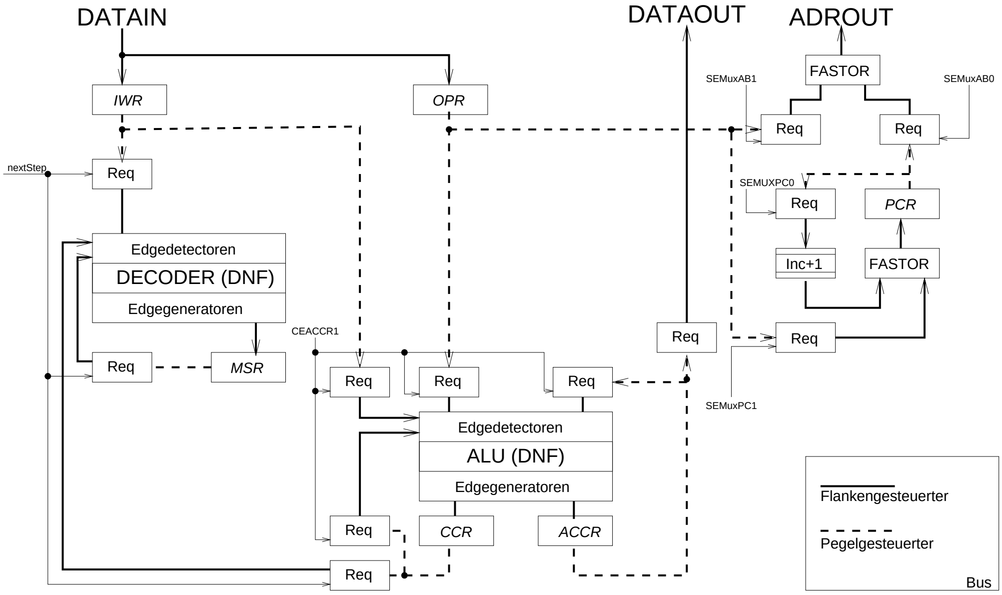



Abbildung 4.23: DEP-CPU (asynchron)

genommen haben. Mit 1521 CLBs war er etwa um den Faktor 30 größer als die synchrone Variante, allerdings sind hierbei noch einige CLBs für den **`PCICORE`** sowie einige Debugbausteine abzuziehen, die mit insgesamt etwa 500 CLBs zu Buche schlagen. Damit liegt der Faktor bei etwa 20, wovon die meisten CLBs für die Edge-Detectoren und Edge-Generatoren verbraucht werden.

### 4.3.5 Abschließende Bemerkungen zum Blockschaltbild

Sofern nicht anders angegeben, werden wir im Folgenden auf die Flankengesteuerten Busse und die Requestbausteine in Blockschaltbildern verzichten, und diese stattdessen optisch in die Register integrieren, was der Übersicht mehr als dienlich ist, wie Abb. 4.25 zeigt.

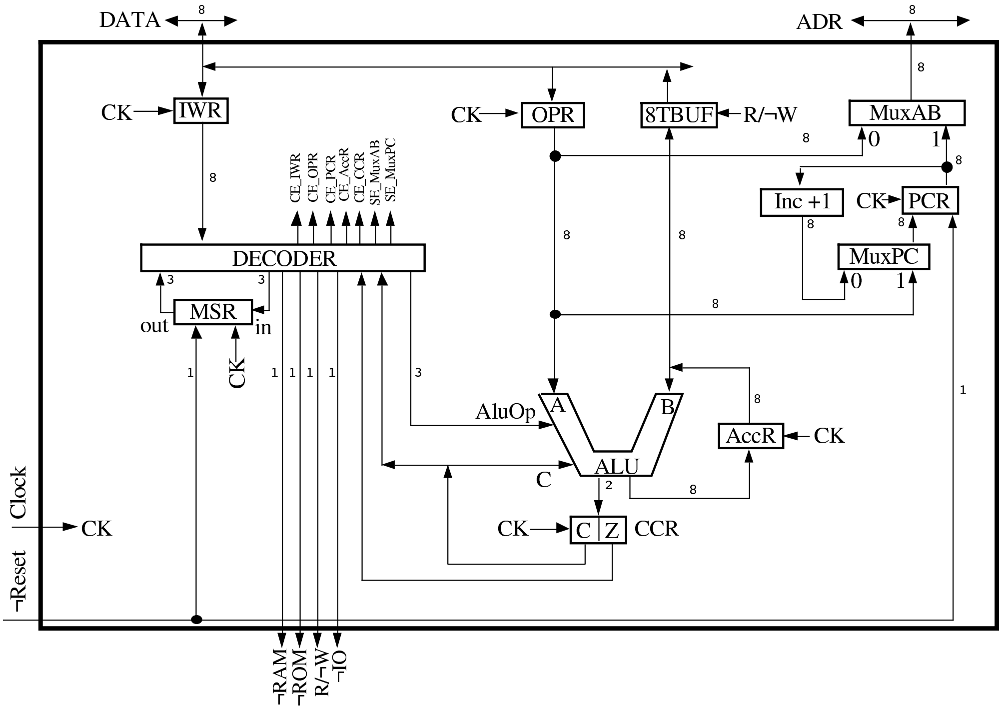



Abbildung 4.24: DEP-CPU (synchron)

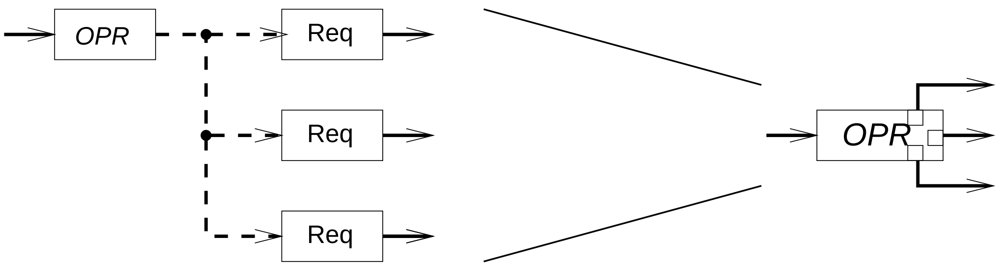



Abbildung 4.25: Optische Vereinfachung

# Kapitel 5

# Der 6502

## 5.1 Einführung

Der MOS6502 ist ein 8-Bit-Prozessor der 65xx-Familie von *MOS Technologies*, einer von ehemaligen *Motorola*-Mitarbeitern gegründeten Firma. Die 65xx-Familie wurde basierend auf *Motorola*s 6800 entwickelt und im Juni 1975 – zunächst als 6501, pinkompatibel zum 6800 und wegen rechtlichen Einspruchs *Motorola*s gleich wieder vom Markt genommen, danach als 6502 (nur noch „buskompatibel“ zum 6800) mit seinen Varianten. Durch die Buskompatibilität war es möglich, den kostengünstigeren Prozessor mehr oder weniger direkt auf Platinen zu verwenden, die eigentlich für den 6800 entwickelt wurden, allein die Software mußte noch angepaßt werden. Die Firma *MOS Technologies* wurde später ein Teil von *Commodore* und damit in *Commodore Semiconductor Group* (*CSG*) umbenannt. Unter diesem Namen wurden unter anderem weitere Prozessoren für *Commodore* auf 6502-Basis entwickelt.

Alle Prozessoren der 65xx-Familie besitzen einen 8-Bit-Datenbus und vier dem Programmierer zugängliche 8-Bit-Register (wenn man mal vom Prozessorstatusregister **`P`** und 16-Bit-Programmzähler **`PC`** absieht) und können mit maximal ein, zwei, drei oder vier MHz getaktet werden[1]. Der Adreßbus ist nur beim 6502 und 6512 16 Bit breit, der adressierbare Speicherbereich ist also nur bei diesen Modellen 64kB groß, alle anderen Modelle der originalen 65xx-Familie haben zur Kostenersparnis eingeschränkte Adreßbusbreiten. Tabelle 5.4 zeigt eine Auflistung aller Mitglieder der MOS65xx-Familie.

| **Prozessor** | **Beschreibung** |
|---|---|
| 6501 | erster 8-Bit-Prozessor der MOS65xx-Familie, 16-Bit-Adreßbus, pinkompatibel zum M6800; wird nur in frühen Papers erwähnt und wurde eigentlich nie produziert |
| 6502 | Basismodell der 65xx-Familie mit 16-Bit-Adreßbus; verwendet in vielen kommerziellen Systemen wie dem *Apple* II, dem *Atari* 800 und *Commodore*s 5 1/4”-Diskettenlaufwerken |
| 6503 | 12-Bit-Adreßbus, keine $\Phi_1$-, `RDY`- und `SYNC`-Leitungen |
| 6504 | wie 6503, aber 13-Bit-Adreßbus und kein $\neg$`NMI` |

[1] zu erkennen am nachgestellten Buchstaben: kein Buchstabe=1MHz, A=2MHz, B=3MHz, C=4Mhz

| Prozessor | Beschreibung |
|---|---|
| 6505 | wie 6503, aber 12-Bit-Adreßbus und statt ¬`NMI` mit `RDY` |
| 6506 | wie 6503, aber 12-Bit-Adreßbus und statt ¬`NMI` mit $\Phi_1$ |
| 6507 | wie 6504, aber 13-Bit-Adreßbus und statt ¬`IRQ` mit `RDY`; verwendet in *Atari*s Spielekonsole 2600 VCS |
| 6512 | wie der 6502 mit 16-Bit-Adreßbus, aber ohne Taktgenerator und mit einer zusätzlichen `DBE`-Leitung (Data Bus Enable), außerdem sind $\Phi_1$ und $\Phi_2$ nicht als Aus-, sondern als Eingänge ausgelegt, $\Phi_0$ heißt hier $\Phi_2$ `(OUT)` und ist ein Ausgang |
| 6513 | wie 6512, aber 12-Bit-Adreßbus und keine $\Phi_2$ `(OUT)`-, `RDY`-, `SYNC`- und `DBE`-Leitungen |
| 6514 | wie 6513, aber 13-Bit-Adreßbus und kein ¬`NMI` |
| 6515 | wie 6513, aber 12-Bit-Adreßbus und statt ¬`NMI` mit `RDY` |

Tabelle 5.4: Übersicht über die 65xx-Familie

Anmerkungen zu Tabelle 5.4: $\Phi_0$, $\Phi_1$ und $\Phi_2$ sind Taktsignale; beim 650x (also mit Taktgenerator) sind $\Phi_2$ und $\Phi_1$ (invertiertes $\Phi_2$) Taktsignale für externe Chips, mit der auch die CPUselbst synchronisiert wird (das Referenztaktsignal ist immer $\Phi_2$), $\Phi_0$ ist der Eingang des Systemtakts zum integriertem Taktgenerator. Bei 651x-CPUs sind $\Phi_1$ und $\Phi_2$ Eingänge (zur exakten Synchronisierung mehrerer parallel betriebener CPUs), die von einem extern generierten Takt belegt werden müssen.
Das `SYNC`-Signal zeigt mit einem `high`-Pegel an, ob der Prozessor einen Befehl aus dem Speicher lädt, `RDY` (memory ready) ist zur Verwendung der CPUin Kombination mit langsamem Speicher gedacht: solange `RDY low` ist, bleibt die CPUstehen, bis es wieder auf `high` gezogen wird. Mit `SYNC` und `RDY` ist es somit einfach möglich, den Prozessor von Hand in Einzelschritten arbeiten zu lassen.
Siehe auch Abbildung 5.1 zur Pinbelegung.

## Die Nachfolger

Das einfache Design und der niedrige Einführungspreis der 650x-Familie haben ihr zu großer Popularität verholfen, und durch ihre weite Verbreitung wurden auf ihr basierend weitere Prozessoren entworfen, mit erweiterten Befehlssätzen und neuen Adressierungsmodi, aber im Großen und Ganzen immer kompatibel zum Ur-6502. Sogar heute noch werden diese Prozessoren hergestellt, z. B. von *California Micro Devices*[2]. Tabelle 5.5 enthält eine kleine Auflistung der uns bekannten Prozessoren, die Weiterentwicklungen der 65xx-Familie sind.

| Prozessor | Beschreibung |
|---|---|
| CSG6509 | ein 6502 mit 1MB Adreßraum (20-Bit-Adreßbus), der über ein zusätzliches, vier Bit breites Register angesprochen werden kann |
| CSG6510 | ein 6502 mit 6-Bit-I/O-Port und zusätzlicher `AEC`-Leitung (address enable control) zum Ausschalten des Adreßbusses; nur verwendet im *Commodore* C64 (später als 8500 bezeichnet) |
| CSG8502 | ein 6510 mit 2MHz, 100% kompatibel zum normalen 6510; verwendet im *Commodore* C128 |

[2] http://www.calmicro.com/prod/rockwell_promo.htm

| Prozessor | Beschreibung |
|---|---|
| WDC65C02 | verbesserter Nachbau des 6502 von *Western Digital Center, Inc.* in CMOS-Technologie mit ein paar neuen Befehlen und zwei neuen Adressierungsmodi, der alle anderen undefinierten Opcodes als `NOP` behandelt |
| R65C02 | erweiterter 65C02 von *Rockwell* mit 32 zusätzlichen Bitmanipulationsbefehlen in Zero Page-Adressierung |
| WDC65816 | erweiterte Version des 6502 mit 16-Bit-Registern, 16-Bit-ALU, verschiebbarer Stack- und Zero Page (hier „Direct Page“ genannt), 24 Bit Adreßraum, 11 neuen Adressierungsmodi und vielen neuen Befehlen, umschaltbar auf 6502-Kompatibilitätsmodus; verwendet in *Nintendo*s Spielekonsole Super NES und im *Apple* IIgs |
| WDC65802 | ein 65816, der pinkompatibel zum 6502 und damit auch nur mit einem 16-Bit-Adreßbus ausgestattet ist |
| HuC6280 | modifizierte Version eines 65C02, mit neuen Befehlen, neuen Adressierungsmodi, integrierter MMU zur Adressierung von 2MB Speicher, Interrupt Controller, einem Timer und einem 8-Bit-I/O-Port; verwendet in *NEC*s Spielekonsole „PC Engine“ und ihren Nachfolgern. |

Tabelle 5.5: Weiterentwicklungen der 65xx-Familie Prozessor

## 5.2 Interna

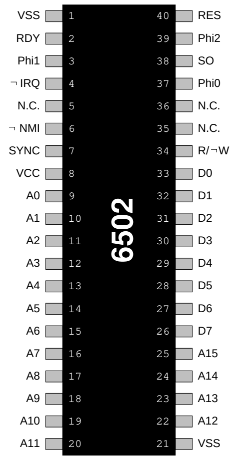
Im folgenden Text wird die 65xx-Familie vereinfachend nur noch „6502“ genannt, da der 6502 als Repräsentant der gesamten Prozessorfamilie gesehen werden kann.

### 5.2.1 Register

Jeder Prozessor braucht zum Funktionieren etwas Speicher, auf den er direkt ohne Umwege über externe Busse zugreifen kann, um Statusinformationen zu speichern oder um etwas berechnen zu können. Dieser Speicher ist normalerweise direkt im Prozessor integriert, um die in ihm gespeicherten Informationen direkt und auf dem schnellsten Wege verarbeiten zu können. Jedem Teil dieses Speichers, den sogenannten Registern, wird im 6502 eine spezielle Funktion zugeordnet, und jedes Register kann auch nur für den ihm zugeschriebenen Zweck verwendet werden, beziehungsweise nur sehr eingeschränkt für andere Zwecke.

Der 6502 besitzt insgesamt fünf nach „außen“ hin sichtbare (also vom Programmierer mehr oder weniger direkt beeinflußbare und auslesbare) 8-Bit-Register und ein 16-Bit-Register:

- Den Programmzähler **`PC`**, der auf die Adresse zeigt, von der der Prozessor das nächste Befehlswort lädt, um es auszuführen. Dieses Register ist 16 Bit breit, Programmcode kann also ohne weitere externe Speicherverwaltung maximal 64kB lang werden. Mit jeder aus dem Speicher geladenen Instruktion wird **`PC`**




| **Eingänge** | **Beschreibung** |
|---|---|
| VCC | Versorgungsspannung (z. B. +5V) |
| VSS | Masse |
| RES | Reset |
| $\Phi_0$ | Takteingang |
| RDY | Memory Ready |
| SO | Set Overflow Flag |
| ¬IRQ | Hardwareinterrupt |
| ¬NMI | Hardwareinterrupt |

| **Ausgänge** | **Beschreibung** |
|---|---|
| $\Phi_1$ | generierter Takt (in Phase mit $\Phi_0$) |
| $\Phi_2$ | generierter Takt (phasenverschoben zu $\Phi_0$) |
| SYNC | Sync Opcode Fetch |
| R/¬W | Lese-/Schreibzugriff |
| A$nn$ | Adreßbus |

| **Bidirektional** | **Beschreibung** |
|---|---|
| D$n$ | Datenbus |

Abbildung 5.1: 6502 Pinbelegung

erhöht, ändert also ständig seinen Wert, explizit kann er nur durch Sprungbefehle verändert werden.

- Den Akkumulator **A**, mit dem alle Berechnungen wie Addition und Subtraktion oder logische Operationen vorgenommen werden.
- Die beiden Indexregister **X** und **Y**, die vorwiegend beim Zugriff auf den Speicher benutzt werden, die aber auch als schneller Zwischenspeicher benutzt werden können, einfachere Operationen sind mit ihnen ebenfalls durchführbar (z. B. Inkrement/Dekrement).
- Das Stackregister **S**, das innerhalb des nach unten wachsenden Stackbereiches auf das nächste freie Byte zeigt (siehe hierzu Abschnitt 5.2.2).
- Das Prozessorstatusregister **P**, das insgesamt sieben Flags enthält, die den Zustand des Prozessors speichern. Die einzelnen Flags sind

  **N** (**N**egative) Wird von arithmetischen Operationen beeinflußt. Wenn das vorzeichenbehaftete Ergebnis negativ ist, wird das Flag gesetzt, ansonsten gelöscht, entspricht also dem obersten Bit des Ergebnisses.

  **V** (O**V**erflow) Wird von arithmetischen Operationen, `BIT` und dem Hardwaresignal `¬SO`[3] beeinflußt. Bei einem Vorzeichenwechsel nach einer arithmetischen Operation wird **V** gesetzt, sonst gelöscht, um beim Rechnen mit vorzeichenbehafteten Zahlen Bereichsüberschreitungen feststellen zu können.

  **B** (**B**reak) Zur Unterscheidung von Soft- und Hardwareinterrupts. **B** wird gelöscht, wenn ein Hardwareinterrupt (`¬IRQ`, `¬NMI`) auftritt und ist sonst immer gesetzt, insbesondere bei einem Softwareinterrupt durch `BRK`.

  **D** (**D**ecimal mode) Um den Prozessor in den Dezimalmodus für arithmetische Befehle zu versetzen, muß **D** gesetzt, für den binären Modus gelöscht sein.

[3] In der *PsalM*-Implementierung des 6502 wurde auf `¬SO` verzichtet, welches, wenn auf `low` gezogen, das **V**-Flag setzt (siehe auch Abschnitt 5.3 zur Implementierung).

- **I** (**I**nterrupt disable) Ist dieses Flag gesetzt, wird kein Interrupt mehr ausgelöst, wenn `¬IRQ` auf `low` gezogen wird. Tritt ein Interrupt bei gelöschtem **I** auf, wird das Flag automatisch gesetzt und nach Abarbeitung der Interruptroutine wieder gelöscht. Das Flag betrifft nicht `¬NMI` und den `BRK`-Befehl.
- **Z** (**Z**ero) Wird von binären und arithmetischen Operationen und von Ladebefehlen beeinflußt. Wenn in ein Register der Wert `0x00` gespeichert wird oder eine ALU-Operation `0x00` ergibt, wird **Z** gesetzt, sonst gelöscht.
- **C** (**C**arry) Wird von arithmetischen und binären Rotationsoperationen beeinflußt. Nach einer Addition wird **C** bei einem Überlauf gesetzt, sonst gelöscht. Nach einer Subtraktion wird **C** bei einem Unterlauf gelöscht, sonst gesetzt. Bei einer Rotation wird **C** auf den Wert gesetzt, der aus dem Byte „herausgeschoben“ wurde.

Das Register ist eigentlich acht Bit breit, eins davon ist allerdings immer gesetzt. **P** kann ausgelesen werden, indem es auf den Stack geschoben und danach in den Akkumulator zurückgeschrieben wird. Um das Resultat auswerten zu können, muß die Position jedes Flags innerhalb von **P** bekannt sein:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| N | V | 1 | B | D | I | Z | C |

Wie auf einem *Commodore* C64 mit 6510 getestet wurde, kann **P** nicht beliebig durch Überschreiben mit `PLP` (pull stack into **P**) geändert werden:

```
LDA  #$00        ; load $00 into A
PHA              ; push A onto stack
PLP              ; pull stack into P
```

**P** hat nach diesem direkten Löschen aller Flags den Wert `0x30`, **B** (und natürlich das **1**-Flag) sind demnach immer noch gesetzt.

### 5.2.2 Speicher

Der 6502 organisiert den Speicherzugriff intern in 256 Speicherseiten („Pages“) mit je 256 Bytes, was insgesamt 64kB zusammenhängenden Speicher ergibt. Eine Speicheradresse wird aus der Page-Nummer und einem Offset innerhalb der Page gebildet und wird durch ein 16-Bit-Wort repräsentiert, mit der Page im höherwertigen und dem Offset im niederwertigen Byte – der Programmierer arbeitet also mit einem linearen 16-Bit-Adreßraum und braucht sich nicht um die konkrete Realisierung der Speicherverwaltung in Pages zu kümmern[4], wenn ihn Laufzeiten nicht interessieren.

16-Bit-Daten werden vom 6502 grundsätzlich „little endian“ gespeichert, das niederwertige Byte eines 16-Bit-Wortes wird also immer vor dem höherwertigen Byte gespeichert (oder: das niederwertige Byte befindet sich an einer niedrigeren Adresse als

[4] Es gibt einen Bug betreffend Page-Grenzen in Verbindung mit dem `JMP`-Befehl bei indirekter Adressierung, siehe Abschnitt „6510 features“ in [OP6502], der auch auf den 6502 zutrifft.

das höherwertige). Die einzigen 16-Bit-Worte, die der 6502 direkt verarbeiten kann, sind Speicheradressen, wenn sie als solche verwendet werden, ansonsten können nur 8-Bit-Worte benutzt werden. Für die Verarbeitung größerer Zahlen muß die dafür notwendige Arithmetik explizit programmiert werden.

Ein Teil des Adreßraums ist für spezielle Zwecke reserviert, wie in Tabelle 5.6 aufgeführt.

| Bereich | Beschreibung |
|---|---|
| `0x0000-0x00FF` | Zero Page |
| `0x0100-0x01FF` | Stack |
| `0x0200-0xFFF9` | frei verfügbarer Speicherbereich |
| `0xFFFA-0xFFFB` | Pointer auf NMI-Behandlungsroutine |
| `0xFFFC-0xFFFD` | Pointer auf RESET-Behandlungsroutine |
| `0xFFFE-0xFFFF` | Pointer auf IRQ-Behandlungsroutine |

Tabelle 5.6: Speicherbelegung

Die Zero Page, wie der Name schon sagt, bezeichnet den Speicher zwischen `0x0000` und `0x00FF`, also die erste Page, oder Seite null. Für diesen Bereich existieren spezielle Adressierungsmodi (siehe 5.2.5), die die Länge des Opcodes um ein Byte gegenüber normaler Adressierung verringern und dadurch den Zugriff auf den Speicher beschleunigen und die Codegröße reduzieren.

Der Stack liegt fest in der zweiten Page zwischen `0x0100` und `0x01FF` und umfaßt damit 256 Bytes, er kann nicht in einen anderen Bereich verschoben oder vergrößert werden. Der Stackpointer **S** zeigt auf einen Offset innerhalb der Stackpage, der bei Stackoperationen zusammen mit der Pagenummer `0x01` eine komplette 16-Bit-Adresse ergibt. Hat z. B. **S** den Wert `0x42`, dann ergibt dies die Adresse `0x0142`, und bei einem push-Befehl würde das auf den Stack geschobene Byte an Adresse `0x0142` gespeichert und **S** auf `0x41` dekrementiert werden.

Die letzten sechs Bytes im Adreßraum enthalten drei Adressen, die der 6502 bei bestimmten Ereignissen auswertet. An `0xFFFC` ist der Resetvektor gespeichert, die Adresse des Programmcodes, der bei einer steigenden Flanke auf ¬`RES` ausgeführt werden soll[5]. Dies geschieht typischerweise nach dem Einschalten des Prozessors, er lädt dann den Pointer, der an `0xFFFC` und `0xFFFD` gespeichert ist, in das **PC**-Register und beginnt dort mit der Programmausführung. Adressen `0xFFFA` und `0xFFFB` beinhalten einen Pointer auf den Programmcode, der bei einer fallenden Flanke auf ¬`NMI` ausgeführt werden soll, `0xFFFE` und `0xFFFF` einen Pointer auf auszuführenden Programmcode bei fallender Flanke auf ¬`IRQ`. Die Anschlüsse ¬`NMI` und ¬`IRQ` lösen Hardwareinterrupts aus, von denen letzterer auch softwareseitig unterdrückbar ist. Sie funktionieren ähnlich wie ¬`RES`, speichern jedoch **PC** und **P** auf dem Stack, bevor sie mit der Programmabarbeitung beginnen, um später an der unterbrochenen Stelle fortfahren zu können.

[5]Während ¬`RES` `low` ist, „steht" der Prozessor, lesende und schreibende Operationen werden unterdrückt.

### 5.2.3 Befehlssatz

Tabelle 5.7 führt alle 56 im 6502 implementierten Befehle mit Kurzbeschreibung auf. Viele sind in mehreren Adressierungsmodi verwendbar, siehe hierzu Tabelle 5.8 und Abschnitt 5.2.5.

| Instr. | Beschreibung | Instr. | Beschreibung |
|---|---|---|---|
| `ADC` | Addition mit Übertrag | `JSR` | Sprung in Subroutine |
| `AND` | logisches AND | `LDA` | Operand in **A** laden |
| `ASL` | Operand nach links schieben | `LDX` | Operand in **X** laden |
| `BCC` | Sprung, wenn **C** nicht gesetzt | `LDY` | Operand in **Y** laden |
| `BCS` | Sprung, wenn **C** gesetzt | `LSR` | Operand nach rechts schieben |
| `BEQ` | Sprung, wenn **Z** gesetzt | `NOP` | keine Operation |
| `BIT` | Bit-Test | `ORA` | logisches OR |
| `BMI` | Sprung, wenn **N** gesetzt | `PHA` | **A** auf den Stack schieben |
| `BNE` | Sprung, wenn **Z** nicht gesetzt | `PHP` | **P** auf den Stack schieben |
| `BPL` | Sprung, wenn **N** nicht gesetzt | `PLA` | **A** vom Stack holen |
| `BRK` | Softwareinterrupt | `PLP` | **P** vom Stack holen |
| `BVC` | Sprung, wenn **V** nicht gesetzt | `ROL` | Operand nach links rotierten |
| `BVS` | Sprung, wenn **V** gesetzt | `ROR` | Operand nach rechts rotierten |
| `CLC` | **C**-Flag löschen | `RTI` | aus Interruptroutine zurückkehren |
| `CLD` | **D**-Flag löschen | `RTS` | aus Subroutine zurückkehren |
| `CLI` | **I**-Flag löschen | `SBC` | Subtraktion mit Übertrag |
| `CLV` | **V**-Flag löschen | `SEC` | **C**-Flag setzen |
| `CMP` | vergleiche Operand mit **A** | `SED` | **D**-Flag setzen |
| `CPX` | vergleiche Operand mit **X** | `SEI` | **I**-Flag setzen |
| `CPY` | vergleiche Operand mit **Y** | `STA` | **A** speichern |
| `DEC` | Operand dekrementieren | `STX` | **X** speichern |
| `DEX` | **X** dekrementieren | `STY` | **Y** speichern |
| `DEY` | **Y** dekrementieren | `TAX` | Inhalt von **A** in **X** speichern |
| `EOR` | logisches XOR | `TAY` | Inhalt von **A** in **Y** speichern |
| `INC` | Operand inkrementieren | `TSX` | Inhalt von **S** in **X** speichern |
| `INX` | **X** inkrementieren | `TXA` | Inhalt von **X** in **A** speichern |
| `INY` | **Y** inkrementieren | `TXS` | Inhalt von **X** in **S** speichern |
| `JMP` | unbedingter Sprung | `TYA` | Inhalt von **Y** in **A** speichern |

Tabelle 5.7: Befehlssatz des 6502

### 5.2.4 Opcodes

Jeder Opcode im 6502 ist ein Byte lang, gefolgt von null, einem oder zwei Argumenten. Wieviele Argumente ein Opcode benötigt, hängt alleine von seinem Adressierungsmodus ab (siehe Abschnitt 5.2.5).

Für den 6502 sind 151 Opcodes definiert, die Funktionen der restlichen 105 möglichen Codes sind nicht offiziell dokumentiert. Da der Decoder des 6502 nicht vollständig spezifiziert ist, erkennt er auch keine undefinierten Opcodes als solche und versucht sie einfach wie einen normalen Befehl auszuführen. Es existieren einige Dokumente, die das Verhalten eines MOS6502 bei den undokumentierten Opcodes beschreiben, z. B. [OP6502], und zufällig sind unter ihnen auch welche, die deterministische, sinnvolle Funktionen haben[6]. Die *PsalM*-Implementierung des 6502 berücksichtigt jedoch nur die offiziell dokumentierten Befehle, besitzt aber ebenfalls einen nicht vollständig spezifizierten Decoder. Tabelle 5.8 zeigt alle offiziellen 6502-Opcodes auf, wie sie in ähnlicher Form auch in [OP6502] zu finden sind.

| + | 00 | 20 | 40 | 60 | 80 | a0 | c0 | e0 | Adressierung |
|---|---|---|---|---|---|---|---|---|---|
| 00 | BRK | JSR[7] | RTI | RTS | --- | LDY | CPY | CPX | Implied/Imm. |
| 01 | ORA | AND | EOR | ADC | STA | LDA | CMP | SBC | Indexed Indir. |
| 02 | --- | --- | --- | --- | --- | LDX | --- | --- | Immediate |
| 03 | --- | --- | --- | --- | --- | --- | --- | --- | |
| 04 | --- | BIT | --- | --- | STY | LDY | CPY | CPX | Zero Page |
| 05 | ORA | AND | EOR | ADC | STA | LDA | CMP | SBC | Zero Page |
| 06 | ASL | ROL | LSR | ROR | STX | LDX | DEC | INC | Zero Page |
| 07 | --- | --- | --- | --- | --- | --- | --- | --- | |
| 08 | PHP | PLP | PHA | PLA | DEY | TAY | INY | INX | Implied |
| 09 | ORA | AND | EOR | ADC | --- | LDA | CMP | SBC | Immediate |
| 0a | ASL | ROL | LSR | ROR | TXA | TAX | DEX | NOP | Accu/Implied |
| 0b | --- | --- | --- | --- | --- | --- | --- | --- | |
| 0c | --- | BIT | JMP | JMP[8] | STY | LDY | CPY | CPX | Absolute |
| 0d | ORA | AND | EOR | ADC | STA | LDA | CMP | SBC | Absolute |
| 0e | ASL | ROL | LSR | ROR | STX | LDX | DEC | INC | Absolute |
| 0f | --- | --- | --- | --- | --- | --- | --- | --- | |
| 10 | BPL | BMI | BVC | BVS | BCC | BCS | BNE | BEQ | Relative |
| 11 | ORA | AND | EOR | ADC | STA | LDA | CMP | SBC | Indir. Indexed |
| 12 | --- | --- | --- | --- | --- | --- | --- | --- | |
| 13 | --- | --- | --- | --- | --- | --- | --- | --- | |
| 14 | --- | --- | --- | --- | STY | LDY | --- | --- | Zero Page, X |
| 15 | ORA | AND | EOR | ADC | STA | LDA | CMP | SBC | Zero Page, X |
| 16 | ASL | ROL | LSR | ROR | STX[9] | LDX[9] | DEC | INC | Zero Page, X |
| 17 | --- | --- | --- | --- | --- | --- | --- | --- | |
| 18 | CLC | SEC | CLI | SEI | TYA | CLV | CLD | SED | Implied |
| 19 | ORA | AND | EOR | ADC | STA | LDA | CMP | SBC | Absolute, Y |
| 1a | --- | --- | --- | --- | TXS | TSX | --- | --- | Implied |
| 1b | --- | --- | --- | --- | --- | --- | --- | --- | |
| 1c | --- | --- | --- | --- | --- | LDY | --- | --- | Absolute, X |
| 1d | ORA | AND | EOR | ADC | STA | LDA | CMP | SBC | Absolute, X |
| 1e | ASL | ROL | LSR | ROR | --- | LDX[9] | DEC | INC | Absolute, X |
| 1f | --- | --- | --- | --- | --- | --- | --- | --- | |

Tabelle 5.8: Opcodes des 6502

[6]Die Nachfolger der MOS650x-Familie, z. B. der 65C02, haben vollständig spezifizierte Decoder.
[7]Absolute
[8]Absolute Indirect
[9]Befehl benutzt **Y** zur Adressierung

### 5.2.5 Adressierungsmodi

Wie Tabelle 5.8 zu entnehmen ist, können zu einem Mnemonic mehrere Adressierungsmodi existieren. Der Adressierungsmodus bestimmt, wie der zu verarbeitende Operand aus dem Speicher zu lesen oder in den Speicher zu schreiben ist, oder ob ein Befehl überhaupt einen Operanden erwartet. Insgesamt kennt der 6502 dreizehn verschiedene Adressierungsmodi:

**Implied** Die Funktion und die involvierten Operanden sind implizit durch den Befehl vorgegeben, z. B. Setzen eines Flags oder Kopieren eines Registerinhaltes in ein anderes Register. Befehle in diesem Adressierungsmodus sind ein Byte lang.

**Accumulator** Der Befehl impliziert eine Operation mit dem Akkumulator. Befehle in diesem Adressierungsmodus sind ein Byte lang.

**Immediate** Der Operand steht direkt als Konstante hinter dem Opcode. Befehle in diesem Adressierungsmodus sind zwei Bytes lang.

**Absolute** Dem Opcode folgt eine 16-Bit-Adresse, an der der eigentliche Operand gespeichert ist – in diesem Modus kann also auf jedes Byte im Adreßraum direkt zugegriffen werden. Befehle in diesem Adressierungsmodus sind drei Bytes lang.

**Zero Page** Ähnlich wie *Absolute*, jedoch ist das höherwertige Byte der Adresse implizit auf `0x00` gesetzt, dem Opcode folgt nur noch das niederwertige Byte der endgültigen Adresse. In diesem Modus können nur die 256 Bytes der ersten Page angesprochen werden, aber weil die Befehle um ein Byte kürzer sind als bei absoluter Adressierung, wird ein schnellerer Zugriff auf die Daten und eine Reduktion der Programmcodegröße ermöglicht. Befehle in diesem Adressierungsmodus sind zwei Bytes lang.

**Absolute Indexed, X** Wie bei *Absolute* folgt dem Opcode eine 16-Bit-Adresse, die effektive Adresse wird aber durch zusätzliche Addition des Inhalts von **`X`** auf die angegebene Adresse berechnet, das Register wird also als Index benutzt. Befehle in diesem Adressierungsmodus sind drei Bytes lang.

**Absolute Indexed, Y** Anstatt des **`X`**-Registers wird **`Y`** als Index benutzt, sonst genau wie *Absolute Indexed, X*, allerdings können nicht alle Befehle, die **`X`** als Index verwenden können, auch mit **`Y`** indizieren. Für eine genaue Auflistung der möglichen Kombinationen siehe Tabelle 5.8. Befehle in diesem Adressierungsmodus sind drei Bytes lang.

**Zero Page Indexed, X** Ähnlich wie *Absolute Indexed, X*, jedoch ist der Adressierungsbereich auf die Zero Page beschränkt. Die Zero Page kann auch dann nicht „verlassen“ werden, wenn die Summe von der angegebenen Adresse und **`X`** größer als `0x00FF` ist, das höherwertige Byte wird immer implizit auf `0x00` gesetzt. Befehle in diesem Adressierungsmodus sind zwei Bytes lang.

**Zero Page Indexed, Y** Anstatt des **`X`**-Registers wird **`Y`** als Index benutzt, sonst genau wie *Zero Page Indexed, X*. Nur die Befehle `LDX` und `STX` können in diesem Modus benutzt werden und sind jeweils zwei Bytes lang.

**Indexed Indirect** Auf den Wert des **X**-Registers wird das dem Opcode folgende Byte addiert und das Carry verworfen. Das Resultat wird als Adresse in der Zero Page benutzt, von der ein 16-Bit-Wort geladen wird, das wiederum die effektive Adresse des Operanden bildet. Abbildung 5.2 verdeutlicht anhand eines Beispiels die Funktionsweise dieses relativ komplizierten Adressierungsmodus, der z. B. verwendet werden kann, um ein Array von Pointern abzuarbeiten. Befehle in diesem Adressierungsmodus sind zwei Bytes lang.

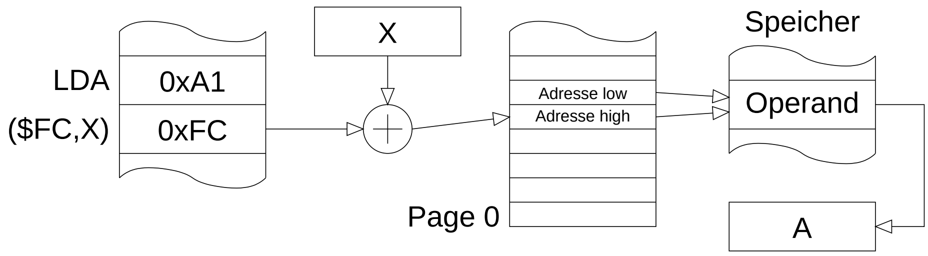



Abbildung 5.2: `LDA` mit Indexed Indirect, **X**=`0x08`

**Indirect Indexed** Das dem Opcode folgende Byte wird als Adresse in der Zero Page benutzt, von der ein 16-Bit-Wort geladen wird. Die Summe des geladenen Wortes und des Inhaltes des **Y**-Registers bilden die effektive Adresse des Operanden. Diese Adressierung kann z. B. verwendet werden, um ein Array abzuarbeiten, das durch einen Pointer referenziert wird – siehe auch hier zur Verdeutlichung der Funktionsweise das Beispiel in Abbildung 5.3. Befehle in diesem Adressierungsmodus sind zwei Bytes lang.

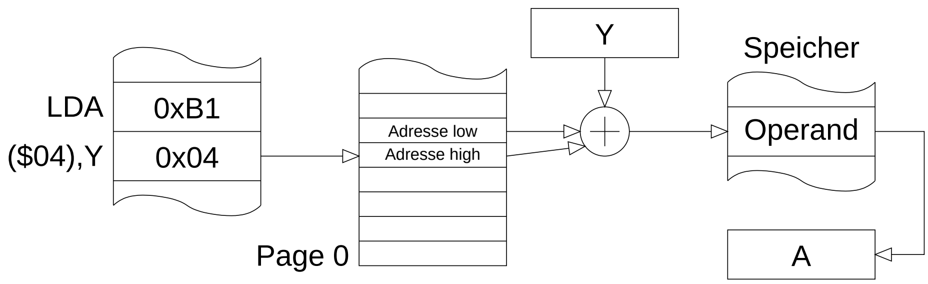



Abbildung 5.3: `LDA` mit Indirect Indexed

**Absolute Indirect** Dem Opcode folgt eine 16-Bit-Adresse, die wiederum auf eine weitere Adresse zeigt, die direkt in das **PC**-Register geladen wird. Daher ist `JMP` die einzige Instruktion, die diesen Modus verwendet; sie ist drei Bytes lang.

**Relative** Auf den Wert von **PC** plus zwei (also die Adresse direkt hinter der Anweisung) wird das dem Opcode folgende Byte addiert oder subtrahiert und wieder in **PC** gespeichert. Dieser Modus wird von den Branchbefehlen, also den konditionellen Sprüngen, verwendet und läßt auch negative Sprünge zu, indem der Operand vorzeichenbehaftet verwendet wird. Befehle in diesem Adressierungsmodus sind zwei Bytes lang.

Darstellung von positiven und negativen Zahlen: Positiv sind die Zahlen, die das höchstwertige Bit auf 0 gesetzt haben, die unteren sieben Bits bilden dann eine Zahl zwischen 0 und 127. Negative Zahlen haben das höchstwertige Bit gesetzt, die unteren Bits bilden dann jedoch nicht den Betrag der dargestellten Zahl, sondern deren um 1 erhöhtes Komplement[10] . Um den Betrag zu erhalten, muß dementsprechend von der Zahl eins abgezogen und das Ergebnis bitweise negiert werden. Beispiel:

```
      0xF0=11110000
 0xF0-0x01=11101111
¬11101111=00010000
  00010000=0x10
```

Die dargestellte Zahl ist also -16. Man kann alternativ die Darstellung der Zahl auch von `0x100` abziehen, um den Betrag zu erhalten. Alle Branchbefehle mit `0x00` im Operanden können übrigens als `NOP` angesehen werden, da **`PC`** im Falle eines Sprunges um `0x00` erhöht und damit auf der Adresse hinter dem Branchbefehl gesetzt, und andernfalls **`PC`** regulär weitergezählt wird. Alle Register (außer **`PC`**) behalten nach einem Branch ihren ursprünglichen Wert, damit kann jeder Branchbefehl mit Sprungweite `0x00` als echter `NOP`-Ersatz mit einer Instruktionslänge von zwei Bytes gelten.

Wie die Adressierungsmodi vom Programmierer benutzt werden, ist in Tabelle 5.9 zu sehen. Die Entscheidung, ob eine Zero Page- oder eine Absolute Adressierung benutzt wird, nimmt demnach der Assembler dem Programmierer ab, abhängig davon, wo die jeweiligen Daten abgelegt werden.

| Adressierungsmodus | Assembler | Speicherinhalt |
|---|---|---|
| Implied | `DEY` | `0x88` |
| Accumulator | `ROL` | `0x2A` |
| Immediate | `LDA #$23` | `0xA9 0x23` |
| Absolute | `LDA $7812` | `0xAD 0x12 0x78` |
| Zero Page | `LDA $42` | `0xA5 0x42` |
| Absolute Indexed, X | `LDA $7812,X` | `0xBD 0x12 0x78` |
| Absolute Indexed, Y | `LDA $7812,Y` | `0xB9 0x12 0x78` |
| Zero Page Indexed, X | `LDA $42,X` | `0xB5 0x42` |
| Zero Page Indexed, Y | `LDX $23,Y` | `0xB6 0x23` |
| Indexed Indirect | `LDA ($FC,X)` | `0xA1 0xFC` |
| Indirect Indexed | `LDA ($04),Y` | `0xB1 0x04` |
| Absolute Indirect | `JMP ($9A7E)` | `0x6C 0x7E 0x9A` |
| Relative | `BEQ $66` | `0xF0 0x66` |

Tabelle 5.9: Assemblerbeispiele für die Adressierungsmodi

[10] auch als Zweierkomplementdarstellung bekannt

## 5.3 *PsalM*-Implementierung des 6502

Die *PsalM*-Implementierung des 6502 entspricht nicht exakt seinem Vorbild, teils weil einige Eigenschaften des Originals nicht nachbildbar sind, teils wegen widersprüchlicher Dokumentation aus verschiedenen Quellen.

Da der resultierende Prozessor für sich alleine auf der FPGA-Karte funktionieren sollte, wurden die Ausgänge $\Phi_1$ und $\Phi_2$ nicht mit implementiert – somit handelt es sich beim *PsalM*-6502 eigentlich um einen 6512, allerdings mit nur einem Takteingang und ohne dem `DBE`-Anschluß, mit dem der bidirektionale Datenbus in den hochohmigen Zustand (Tristate) geschaltet werden kann.

Die Anschlüsse `SYNC` und `RDY` wurden ebenfalls nicht ins Design einbezogen. Mit `RDY` kann die Programmausführung der CPU von außen gestoppt und damit ein Warten erzwungen werden. Jeder einzelne Zyklus (außer Schreibzyklen) kann damit angehalten werden, um z. B. die Fehlersuche zu vereinfachen oder um den Zugriff auf langsame Peripherie zu ermöglichen. Über den Anschluß `SYNC` zeigt der 6502 an, wann er einen Opcode aus dem Speicher lädt. Es ist also sehr leicht, mit Hilfe von `SYNC` und `RDY` den Programmablauf in Einzelschritten abarbeiten zu lassen, für *PsalM* waren diese Anschlüsse jedoch weder nötig noch sinnvoll, da ohnehin mit dem Platz gespart werden mußte.

Der originale 6502 besitzt weiterhin den Anschluß `SO`, mit dem das V-Flag des **`P-Registers`** direkt gesetzt werden kann. Dieses Signal kann z. B. bei der Implementierung von Hardware-Locks[11] oder anderen Synchronisationsaufgaben mit Sekundärbausteinen eingesetzt werden, ein Programm könnte so mit folgender Programmsequenz auf ein extern gesteuertes Signal warten:

```
  clv
wait:
  bvc  wait        ; wait until V is set
```

Dieser Intention im Systemdesign folgt die Tatsache, das es im 6502-Befehlssatz auch keine Instruktion gibt, mit der das V-Flag direkt gesetzt werden könnte.

Uns war es nicht möglich, Zugriff auf eine originale Decodertabelle des MOS6502 zu erhalten, das Verhalten bei den undokumentierten Instruktionen konnte also keinesfalls nachgebildet werden. Da der *PsalM*-6502-Decoder allerdings ebenso wie der des MOS6502 nicht vollständig spezifiziert ist, führt die *PsalM*-Implementierung ebenfalls die nicht dokumentierten Opcodes aus, allerdings sicherlich mit anderen Ergebnissen als ein MOS6502.

Uns war es auch nicht möglich, Zugriff auf ein detailliertes Blockschaltbild des 6502 zu erlangen, somit waren wir gezwungen ein solches selbst zu entwickeln. Damit ist es zweifelhaft, daß unsere Implementierung auch nur ansatzweise der des originalen MOS6502 entspricht.

[11](engl. Lock: Sperre, Verschluß) In Multiprozessorsystemen muß sichergestellt sein, daß nicht mehrere Prozessoren gleichzeitig auf eine Ressource zugreifen, die nur einen einzelnen Zugriff gleichzeitig verarbeiten kann, z. B. ein Speicherbaustein

Abgesehen von diesen Details und wenigen Spezifikationslücken in der uns zugänglichen Dokumentation des 6502 resultierenden Abweichungen funktioniert die *PsalM*-Implementierung im „regulärem Betrieb“ wie ein normaler 6502, bzw. gemäß den vorhandenen Beschreibungen dieses Prozessors.

### 5.3.1 Blockschaltbild

Ein Blockschaltbild der synchronen Implementierung des 6502 ist in Anhang C zu finden (Abbildung C.1). Die unten gegebene Beschreibung bezieht sich auf diese Abbildung.

**IWR** Das Instruktionswortregister. Es liest seine Daten direkt vom eingehenden Datenbus und gibt seinen Inhalt an den Decoder weiter.

**MSR** Das Maschinenstatusregister. Mit seiner Hilfe zählt der Decoder, in welchem Zyklus einer Instruktion sich die CPU gerade befindet.

**Decoder** Dieses Bauteil steuert den Datenfluß innerhalb der CPU. Eine genauere Beschreibung des Bauteiles folgt später.

**IRQTEIL** Dieses Bauteil speichert die kurzen `low`-aktiven Flanken der eingehenden Interrupts und gibt diese an den Decoder weiter.

Die anderen Bauteil sind im Einzelnen:

**MUXDATAIN** Dieses Bauteil ist wichtig für relative Sprungbefehle. Seine Standardausgabe ist `0x00`. Tritt nun ein relativer Sprung auf, so wird aber das Datum vom **DATAINBUS** ans **ADDPCLPCH** übergeben.

**ADDPCLPCH** Dieses Bauteil verändert den Inhalt des Programcounterregisters. Im Normalfall erhöht es dessen Inhalt einfach um eins, bei relativen Sprungbefehlen wird das Sprungziel berechnet.

**MUXPCL/MUXPCH** Diese Multiplexer regeln, woher das Programcounterregister seine Daten beziehen soll.

**MUXDATA** regelt, welches Register seinen Inhalt auf den `DATAOUTBUS` legen darf.

**PCL/PCH** steht für „Programm-Counter-Low“ bzw. „Programm-Counter-High“. Sie bilden gemeinsam das Programcounterregister, das anzeigt, an welcher Stelle im ROM sich der Programmablauf gerade befindet. Es ist deswegen zweigeteilt, da für einige Befehle der Inhalt der Teilregister auf den 8Bit breiten `DATAOUTBUS` gelegt werden muß, die Register gemeinsam aber 16Bit umfassen können.

**MUXALU** Dieses Bauteil wählt aus, mit dem Inhalt welchen Registers die ALU rechnen soll.

**PSR** steht für „Prozessor-Status-Register“, und beinhaltet die Flags der CPU, als da wären das Negative-, Overflow-, Break-, Dezimal-, Interrupt-, Zero- und Carryflag.

**ALU** ist die Arithmetisch-Logische-Einheit, der Teil der CPU, der rechnet. Eine genauere Beschreibung des Bauteiles folgt später.

**A** ist das Akkumulatorregister. Ein Großteil der möglichen Rechenbefehle des 6502 speichert hierhin sein Ergebnis.

**X** ist das X-Register. Zusammen mit

**Y** dem Y-Register, ist es für indizierte Adressierungen des 6502 gedacht.

**SP** ist das Register, welches den Stackpointer enthält.

**temp** Da die CPU in einem Zyklus nur entweder den Speicher lesen oder ihn beschreiben kann, ist für Rechenbefehle, die den Speicher verändern ein zusätzliches Register nötig. In dieses (temporär verwendete) Register wird nach dem Lesen und Rechnen das Ergebnis der ALU gespeichert. Im folgenden Takt wird der Inhalt dieses Registers in den Speicher zurückgeschrieben.

**ABL/ABH** steht für „Address-Buffer-Low“ bzw. „Address-Buffer-High“. Die beiden Register sind für alle Adressierungsmodi gedacht, die nicht *Immediate*, *Accumulator* oder *Implied* sind.

**tempZ** Bei den komplexeren Adressierungsmodi *Indexed Indirect* und *Indirect Indexed* reichen **ABL** und **ABH** alleine nicht aus, da sie teilweise schon mit Werten überschrieben werden, obwohl ihre Inhalte eigentlich noch als Offset für einen Speicherzugriff verfügbar sein müßten. Dieses Register wurde für eben jene Adressierungsmodi eingeführt.

**MUXZP** wählt aus, ob für die *Zero Page*-Adressierung das Lowbyte vom **ABL** oder **tempZ** kommen soll. Dies ist wichtig bei *Indexed Indirect* und *Indirect Indexed* Adressierung.

**MUXINDEX** indizierte Adressierung kann entweder mit dem Index aus **X** oder aus **Y** funktionieren. Soll eine nicht-indizierte Adressierung durchgeführt werden, so gibt dieses Bauteil eine `0x00` aus.

**ADDABLABH** Dieses Bauteil addiert bei der indizierten Adressierung den Offset, der von **ABL** bzw. **tempZ** und **ABH** kommt, mit dem Index, den der **MUXINDEX** ausgewählt hat.

**MUXADDRH** Mit diesem Multiplexer kann man entscheiden, woher das Highbyte der Adresse kommen soll, die auf den Adreßbus gelegt wird. Für *Zero Page*-Adressierung ist es wichtig, daß das Highbyte `0x00` ist, für den Stackpointer `0x01`, und für Interruptvektoren `0xFF`.

**MUXADDRL** Mit diesem Multiplexer kann man auswählen, woher das Lowbyte der Adresse kommt, die auf den Adreßbus gelegt wird.

**INTVEC_LSB** Für die Interruptbehandlung ist es wichtig, daß festgelegt wird, wo sich die Sprungvektoren für die Interruptroutinen befinden. Dieses Bauteil erzeugt, je nach Vorgabe des Decoders, eine Konstante im Bereich von `0xFA` und `0xFF`. Gemeinsam mit der Konstante `0xFF` vom **MUXADDRH** wird daraus dann z.B. `0xFFFA`, der Vektor für nicht maskierbare Interrupts.

### 5.3.2 Synchrone Implementierung des 6502-Decoders

Der Decoder des 6502 regelt (wie auch der Decoder der DEP-CPU) den Datenfluß innerhalb der CPU und erledigt alle Steuerungsaufgaben.

Die Hauptarbeit bei der Decoderimplementierung lag im Entwurf der Decodertabelle. Dabei sollte gewährleistet sein, daß mit dem bestehenden Blockschaltbild eine Binärkompatibilität zum originalen 6502 erhalten blieb. Aufgrund der Tatsache, daß wir keinen Zugriff auf ein Blockschaltbild von (für unsere Zwecke) optimaler Abstraktion hatten, mußte es teilweise mit der Decodertabelle wachsen.

Eine genaue Beschreibung der Vorgehensweise bei der Erstellung der Decodertabelle würde jedoch bei weitem den Rahmen dieser Ausarbeitung sprengen[12].

Nachdem man sicher sein kann, daß die Decodertabelle fehlerfrei ist, wird sie mit Hilfe der verwendeten Tabellenkalkulation in ein ASCII-Format überführt. Einzelne Spalten sollten dabei klar durch einen Tabulator oder ein Leerzeichen getrennt sein. Die Überschriften der Tabelle sollten in ein Format überführt werden, welches von Espresso (siehe dazu Kapitel A.3) verstanden wird. Espresso selbst wird anschließend mit den Parameter `-estrong -oeqntott` gestartet, und die Standardausgabe in eine Datei umgelenkt.

Die Ausgabedatei von Espresso enthält nun mehrere aussagenlogische Formeln, wobei ein `&` einem AND, ein `|` einem OR, ein `!` einem NOT entspricht. Diese Ausdrücke müssen durch ein weiteres Skript durch die Schlüsselworte AND, OR und NOT ersetzt werden, damit man die Formeln als *VHDL*-Quellcode verwenden kann. Wir haben dazu das Perlskript `espresso2vhdl.pl` benutzt.

Nachdem die *VHDL*-`architecture` des Decoders automatisch erzeugt wurde, werden die *VHDL*-typischen Entity-Deklarationen von Hand hinzugefügt. Das Ergebnis ist nun ein vollständiger Decoder, der die angegebene Decodertabelle implementiert. Alternativ dazu wäre es auch möglich gewesen, den Decoder als Microcode zu implementieren, was allerdings aufgrund der geringen Chipfläche mehr Nach- als Vorteile mit sich bringen würde.

### 5.3.3 Synchrone Implementierung der 6502-ALU

Die ALU des 6502 hier für die Verarbeitung sämtlicher arithmetischer und logischer Befehle zuständig. Die Implementierungsstrategie wurde von der DEP-CPU übernommen, jedoch ist die ALU bei weitem komplizierter und verfügt über wesentlich mehr Operationen.

Wie auch die DEP-ALU verfügt diese über zwei Operandeneingänge, den Eingang für den Opcode, den Flag-Eingang, sowie über Ausgänge für das Ergebnis und die Flags.

| **Eingänge** | **Beschreibung** |
|---|---|
| `A` | 8-Bit Daten vom Datenbus |
| `B` | 8-Bit Daten von den Registern |

[12]Es sei nur noch erwähnt, daß es sich als vorteilhaft erwiesen hatte, die Tabellenkalkulation von StarOffice zu Hilfe zu nehmen. Ein Ausdruck zur Fehlersuche der Decodertabelle schlägt allerdings insgesamt mit etwa $3m^2$ Papier zu Buche.

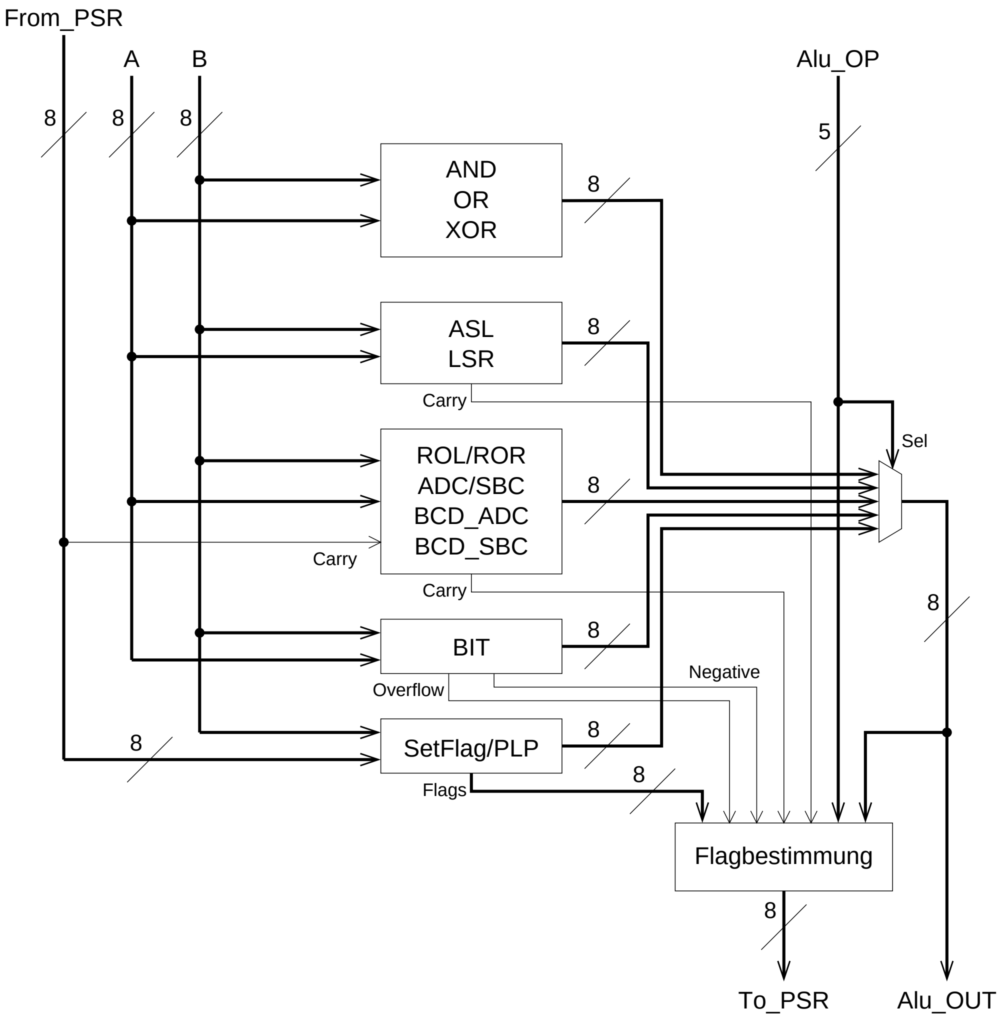



Abbildung 5.4: Blockschaltbild der 6502-ALU

| **Eingänge** | **Beschreibung** |
|---|---|
| `AluOP` | 5-Bit ALU-Operation vom Decoder |
| `From_PSR` | 8-Bit Flag vom PSR |
| **Ausgänge** | **Beschreibung** |
| `To_PSR` | 8-Bit zum PSR |
| `ALU_Out` | 8-Bit Daten zu den Registern /Ausgang |

Prinzipiell läßt sich die ALU in vier Teile zerlegen (siehe Abbildung 5.3.3):

- Logische Operationen: Dieses sind Operationen wie `AND`, `OR` oder `EOR`, die die beiden Operandeneingänge bitweise miteinander verknüpfen.

- Arithmetische Operationen: In dieser Gruppe finden sich Befehle wie `ADC`, `SBC` oder auch `CMP`. Hierbei werden die beiden Busse als Repräsentation einer Dualzahl bzw. BCD-Zahl aufgefaßt und so bitübergreifend verarbeitet.

- Flag-Operationen: Diese Gruppe von Befehlen dient lediglich der Manipulation aller oder einzelner Flags und haben keine relevante Ausgabe auf dem Datenbus. Zur dieser Gruppe gehören alle `SEx` und `CLx`-Befehle, wie aber auch der `BIT`-Befehl.

- Durchschleif-Operationen: Diese kleinste Gruppe besteht nur aus den Befehlen `LD` und `PLP`, die lediglich dazu dienen, Daten ohne Veränderung durch die ALU durchzuschleifen.

Jeder Befehl wird in einem eigenen Bauteil implementiert, lediglich die Befehle zum Flags setzen und löschen, sowie der `PLP`-Befehl bilden hier eine Ausnahme, sie werden zusammen verarbeitet. Alle diese Bauteile arbeiten simultan und durch den Opcode wird zum Schluß bestimmt, welches Ergebnis ausgegeben wird. Analog dazu erfolgt die Bestimmung der Flags. Dabei werden bestimmte Flags nur von bestimmten Befehlen beeinflußt, alle unveränderten Flags werden zurück zum `PSR` geschliffen.

Nachfolgend werden alle Operationen, die die ALU beherrscht, besprochen, dabei ist die Angabe der Flags auf die Einführung in Abschnitt 5.2.1 bezogen.

| **AluOP** | **Operation** | **beeinflußte Flags** | **Beschreibung** |
|---|---|---|---|
| `00000` | `AND` | `N Z` | bitweise Verundung von `A` und `B`. |
| `00001` | `OR` | `N Z` | bitweise Veroderung von `A` und `B` |
| `00010` | `EOR` | `N Z` | bitweises Exklusivoder von `A` und `B`. |
| `00011` | `ROL` | `N Z C` | Rotiert die Werte des an `B` anliegenden Wertes und des C-Flags links herum, d.h. Bit 0 von `B` wird auf den Wert des C-Flags gesetzt, etc. |
| `00100` | `ROR` | `N Z C` | Rotiert die Werte des an `B` anliegenden Wertes und des C-Flags rechts herum, d.h. Bit 7 von `B` wird auf den Wert des C-Flags gesetzt, etc. |
| `00101` | `ASL` | `N Z C` | Arithmetic Shift Left: Setzt das C-Flag auf den Wert des Bit 7 des an `B` anliegenden Wertes, verschiebt alle anderen Bits um 1 nach links und setzt das Bit 0 auf `0`. |
| `00110` | `LSR` | `N Z C` | Logic Shift Right: Setzt das Bit 7 des an `B` anliegenden Wertes auf den Wert des C-Flags, verschiebt alle anderen Bits um 1 nach rechts und setzt das C-Flag auf `0`. |
| `00111` | `LD` | `N Z` | Schleift den an `B` anliegenden Wert durch. |

| AluOP | Operation | beeinflußte Flags | Beschreibung |
|---|---|---|---|
| 01000 | BIT | N Z V | Setzt das N-Flag auf den Wert des Bit 7 des an B anliegenden Wortes, entsprechend wird das V-Flag auf den Wert von Bit 6 gesetzt. Beide Operanden werden verundet und dementsprechend das Z-Flag gesetzt, wobei die Ausgabe von ALU_Out nicht gespeichert wird. |
| 01001 | ADC | N Z C V | Addiert die beiden Operanden unter Berücksichtigung des C-Flags. |
| 01010 | SBC | N Z C V | Subtrahiert die beiden Operanden unter Berücksichtigung des C-Flags. |
| 01001 | BCD_ADC | N Z C | Addiert die beiden Operanden unter Berücksichtigung des C-Flags im BCD-Modus. |
| 01010 | BCD_SBC | N Z C | Subtrahiert die beiden Operanden unter Berücksichtigung des C-Flags im BCD-Modus. |
| 01011 | CMP | N Z V | Subtrahiert B von A, das Ergebnis wird nicht gespeichert, lediglich die Flags werden gesetzt. |
| 01100 | INC | N Z | Inkrementiert den an B anliegenden Wert. |
| 01101 | DEC | N Z | Dekrementiert den an B anliegenden Wert. |
| 01110 | CLB | B | Setzt das B-Flag auf 0. |
| 01111 | SEB | B | Setzt das B-Flag auf 1. |
| 10000 | CLC | C | Setzt das C-Flag auf 0. |
| 10001 | SEC | C | Setzt das C-Flag auf 1. |
| 10010 | CLD | D | Setzt das D-Flag auf 0. |
| 10011 | SED | D | Setzt das D-Flag auf 1. |
| 10100 | CLI | I | Setzt das I-Flag auf 0. |
| 10101 | SEI | I | Setzt das I-Flag auf 1. |
| 10110 | CLV | V | Setzt das V-Flag auf 0. |
| 10111 | PLP | alle | Pull PSR from Stack: Lädt das PSR, das vorher mit dem Befehl PHP[13] auf den Stack geschrieben wurde. |

[13]PHP ist kein nativer ALU-Befehl, sondern ist als einfaches LD implementiert.

# Kapitel 6

# Asynchrone Implementierung des 6502

## 6.1 Umstellung der logischen Gatter

Die meisten dieser Bausteine erhalten einen Eingang GSR, welcher dafür sorgt, daß sämtliche ihrer Ausgänge wieder in den Ursprungszustand zurückfallen.

Im Folgenden stehen Bauteile mit der Bezeichnung **DDR**$x$ für flankengesteuerte Dual-Rail Funktionalität, mit der Bezeichnung **DR**$x$ für pegelgesteuertes Dual-Rail. Dies hat in den gewachsenen Strukturen unseres Dual-Rail-Ansatzes seine Gründe.

**REG1_CE_R**

Abb. 6.1 zeig ein taktflankengesteuertes 1-Bit Register.

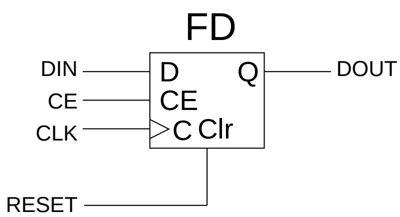



Abbildung 6.1: Einfaches, taktflankengesteuertes 1-Bit Register

| Eingänge | Beschreibung |
|---|---|
| DIN | Data In |
| CLK | Clock |
| CE | Chip Enable |
| RESET | Asynchronous Reset |
| **Ausgänge** | **Beschreibung** |
| DOUT | Data Out |

Liegt `CE` auf `high`, so wird bei einer aufsteigenden Taktflanke der Wert von `DIN` an `DOUT` gelegt und dort gespeichert.

**ED**

Abb. 6.2 zeigt einen Edge-Detector, mit dem aus einer eingehenden, kurzen Flanke ein stabiler `high`-Pegel wird.

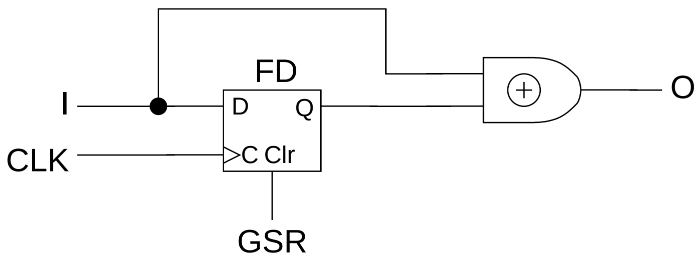



Abbildung 6.2: Schaltplan eines Edge-Detector

| **Eingänge** | **Beschreibung** |
|---|---|
| `I` | Input (flankengesteuerte Ansteuerung) |
| `CLK` | Clock |
| `GSR` | Global State Reset |
| **Ausgänge** | **Beschreibung** |
| `O` | Output (pegelgesteuerte Ansteuerung) |

Im Normalfall liegen an `D` und `Q` identische Pegel an, wodurch das XOR-Gatter an seinem Ausgang einen `low`-Pegel aufweist. Findet nun ein Flankenwechsel am Eingang `I` statt, ist dieses Gleichgewicht nicht mehr gegeben, das XOR-Gatter liefert ein `high`. Zu dem Zeitpunkt, an dem `CLK` eine positive Flanke aufweist, speichert **`REG1_CE_R`** das Eingangssignal; an dessen Eingang `D` und Ausgang `Q` liegen damit wieder die selben Pegel an, das XOR-Gatter liefert an seinem Ausgang ein `low`.

**DOPPELED_DR**

In Abb. 6.3 ist ein Edge-Detector für ein vollständiges flankengesteuertes Dual-Rail Bit dargestellt.

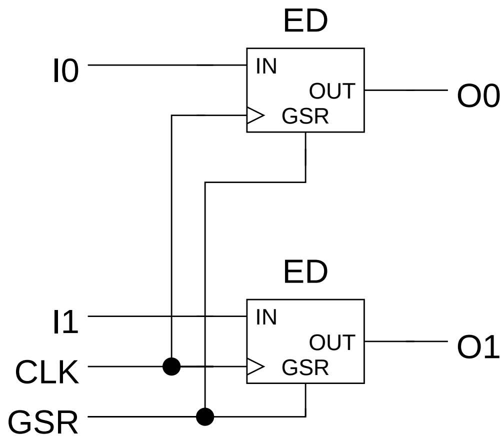



Abbildung 6.3: Schaltplan für einen Dual-Rail Edge-Detector

| Eingänge | Beschreibung |
|---|---|
| `I` | Input (Flankengesteuertes Dual-Rail Bit) |
| `CLK` | Clock |
| `GSR` | Global State Reset |
| **Ausgänge** | **Beschreibung** |
| `O` | Output (Pegelgesteuertes Dual-Rail Bit) |

Dieser Baustein ist dazu gedacht, ein flankengesteuerte Bit am Eingang in ein pegelgesteuertes umzuwandeln. Seine Funktionsweise ist im Prinzip die gleiche wie beim **ED**, mit dem Unterschied, daß er für zwei Leitungen statt nur für eine ausgelegt ist.

**EG**

Abb. 6.4 zeigt einen Edge-Generator für eine einzige Leitung.

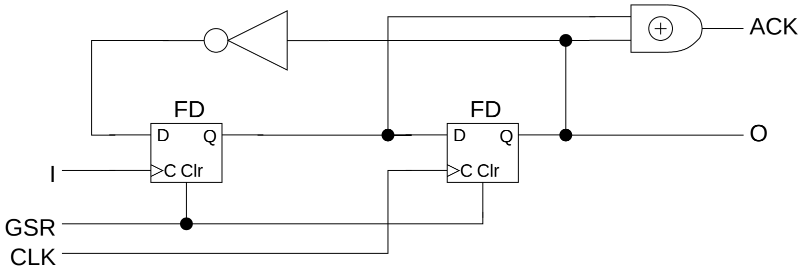



Abbildung 6.4: Schaltplan eines Edge-Generator

| Eingänge | Beschreibung |
|---|---|
| `I` | Input (pegelgesteuerte Ansteuerung) |
| `CLK` | Clock |
| `GSR` | Global State Reset |

| Ausgänge | Beschreibung |
|---|---|
| `ACK` | Acknowledge |
| `O` | Output (flankengesteuerte Ansteuerung) |

Wie schon **ED**, ist auch dieser Baustein nicht für ein vollständiges flankengesteuertes Dual-Rail Bit konzipiert, sondern nur für eine einzelne Leitung.

Prinzipiell besteht er aus einem Master-Slave-Flip-Flop, wobei die Masterstufe in dem Moment reagiert, sobald `I` ein `high`-Signal aufweist. Dadurch erkennt die zweite Stufe, daß sie in naher Zukunft einen Flankenwechsel ausgeben muß, und schaltet `ACK` auf `high`, wodurch meistens die Edge-Detectoren resettet werden.

Die zweite Stufe schaltet in dem Moment, wenn `CLK` auf `high` geht. Dadurch fällt auch `ACK` wieder auf `low`.

**DOPPELEG_DR**

In Abb. 6.5 ist ein Edge-Generator für ein vollständiges flankengesteuertes Dual-Rail Bit dargestellt.

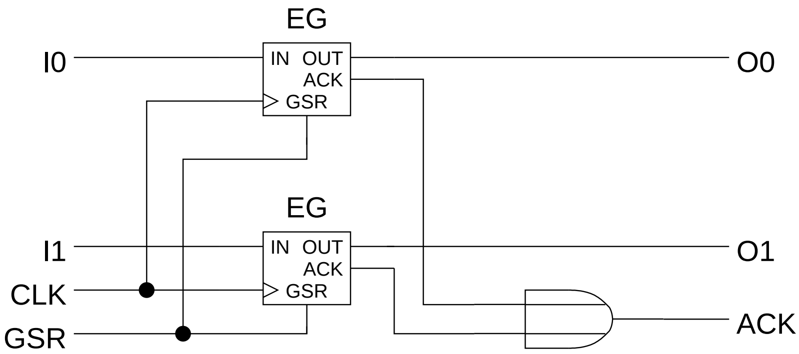



Abbildung 6.5: Schaltplan für einen Dual-Rail Edge-Generator

| **Eingänge** | **Beschreibung** |
|---|---|
| `I` | Input (Pegelgesteuertes Dual-Rail Bit) |
| `CLK` | CLK |
| `GSR` | Global State Reset |

| **Ausgänge** | **Beschreibung** |
|---|---|
| `ACK` | Acknowledge |
| `O` | Output (Flankengesteuertes Dual-Rail Bit) |

Dieser Baustein besteht aus zwei Edge-Generatoren, die ein pegelgesteuertes Dual-Rail Bit am Eingang in ein flankengesteuertes Dual-Rail Bit am Ausgang umwandeln. Sobald einer der beiden **EG** sein `ACK` auf `high` legt, passiert dies durch das OR-Gatter auch beim **DOPPELEG_DR**.

**DRREG**

Der in Abb. 6.6 gezeigte **DRREG**-Baustein bildet das Herzstück jedes flankengesteuerten Dual-Rail-Registers.

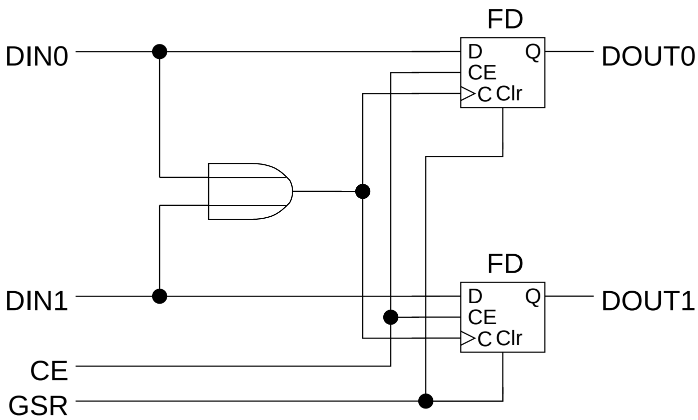



Abbildung 6.6: Schaltplan eines **DRREG**

| Eingänge | Beschreibung |
|---|---|
| `DIN0` | Input |
| `DIN1` | Input |
| `CE` | Chip Enable |
| `GSR` | Global State Reset |
| **Ausgänge** | **Beschreibung** |
| `DOUT0` | Output |
| `DOUT1` | Output |

Speichern kann dieses Register nur dann, wenn der Eingang `CE` auf `high` liegt.

Wird anschließend auf einem der beiden Eingänge `DIN0` oder `DIN1` ein `high` angelegt, so erzeugt das OR-Gatter die für die beiden FD-Register notwendige Taktflanke am `CLK` Eingang. Nach dem Speichern liefert der dazugehörige Ausgang `DOUT0` bzw. `DOUT1` ein `high`, der andere ein `low`.

Ein `GSR` liefert am Ausgang `DOUT0` ein `high`, an `DOUT1` ein `low`.

**DDR_REG_BODY**

Der Baustein **DDR_REG_BODY** nach Abb. 6.7 benutzt den **DRREG**, allerdings für Eingänge nach flankengesteuerter Dual-Rail Logik.

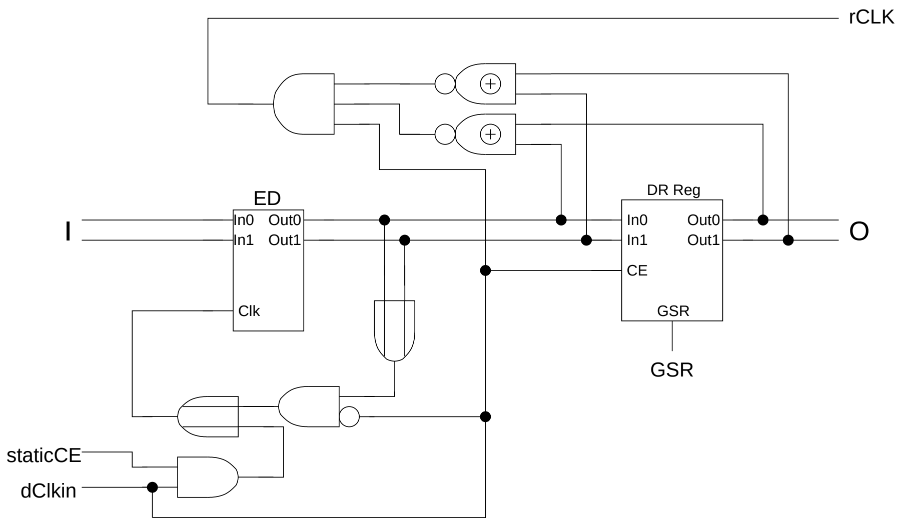



Abbildung 6.7: Schaltplan des **DDR_REG_BODY**

| Eingänge | Beschreibung |
|---|---|
| `GSR` | Global State Reset |
| `I` | Input (Flankengesteuertes Dual-Rail Bit) |
| `staticCE` | Chip Enable (konventionelle Logik) |

| Ausgänge | Beschreibung |
|---|---|
| `O` | Output (Flankengesteuertes Dual-Rail Bit) |
| `dCLKin` | Clockleitung für die Edge-Generator |
| `rCLK` | Acknowledge |

Realisieren die Edge-Detectoren am Eingang `I` einen Flankenwechsel, während `staticCE` auf `high` liegt, so wird dies im **DRREG** gespeichert. Um zu signalisieren, daß der Speichervorgang erfolgreich war, wird `rCLK` auf `high` gelegt.

Sollte ein Flankenwechsel an `I` auftreten, während `staticCE` auf `low` liegt, so werden die Edge-Detectoren durch das Bauteil selbstständig wieder in den Ursprungszustand versetzt. Ansonsten wartet es auf ein Signal an `dCLKin` von außen.

**DDR_REG**$x$

Diese Bauteile nach Abb. 6.8 speichern die Signale für einen Bus von $x$ flankengesteuerten Dual-Rail Bits.

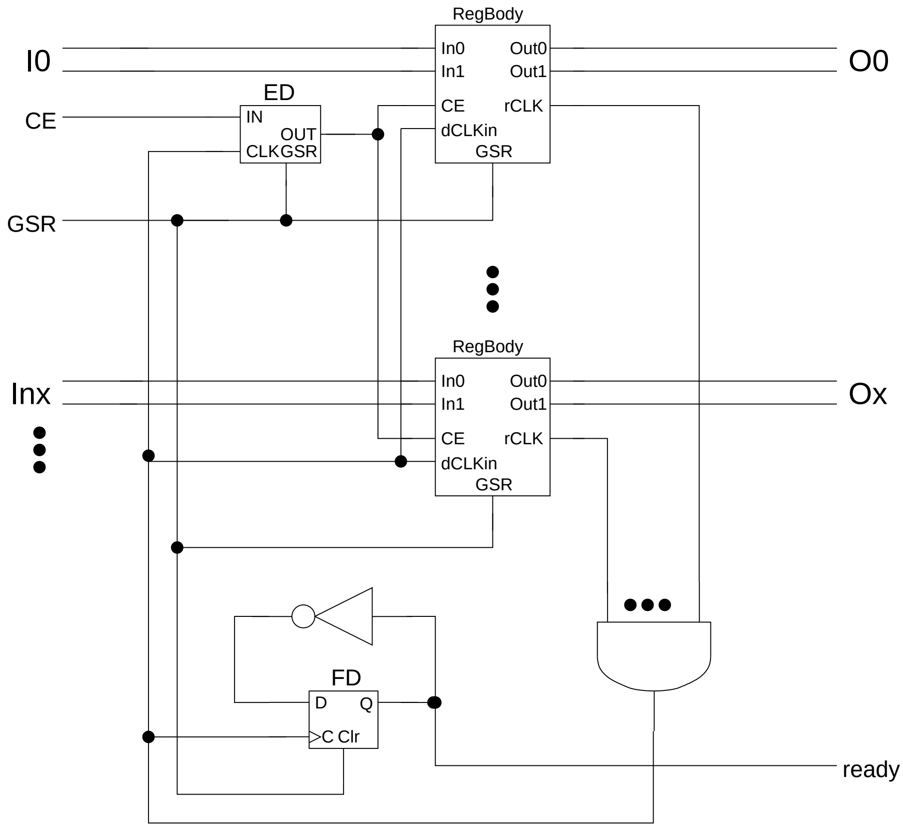



Abbildung 6.8: Schema eines fertigen flankengesteuerten Dual-Rail-Registers

| Eingänge | Beschreibung |
|---|---|
| `GSR` | Global State Reset |
| `I` | Input ($x$ flankengesteuerte Dual-Rail Bits) |
| `CE` | Chip Enable (flankengesteuert) |

| Ausgänge | Beschreibung |
|---|---|
| `O` | Output ($x$ pegelgesteuerte Dual-Rail Bits) |
| `ready` | Acknowledge (flankengesteuert) |

Im Inneren befinden sich $x$ **DDR_REG_BODY**. Nach einem Flankenwechsel auf `CE` werden die eingehen Flankenwechsel an `I` als pegelgesteuerte Dual-Rail Bits am Ausgang gespeichert.

Haben alle **DDR_REG_BODY**s gespeichert, werden sämtliche Edge-Detectoren zurückgesetzt, danach wird auf dem `ready` Ausgang ein Flankenwechsel ausgelöst, der für das `CMS` (siehe Kapitel Abb. 6.5) wichtig ist.

**REQ_BODY**

Dieses Bauteil nach Abb. 6.9 wandelt den gespeicherten pegelgesteuerten Dual-Rail Zustand eines **DDR_REG** auf Anfrage wieder in ein flankengesteuertes Dual-Rail Signal um.

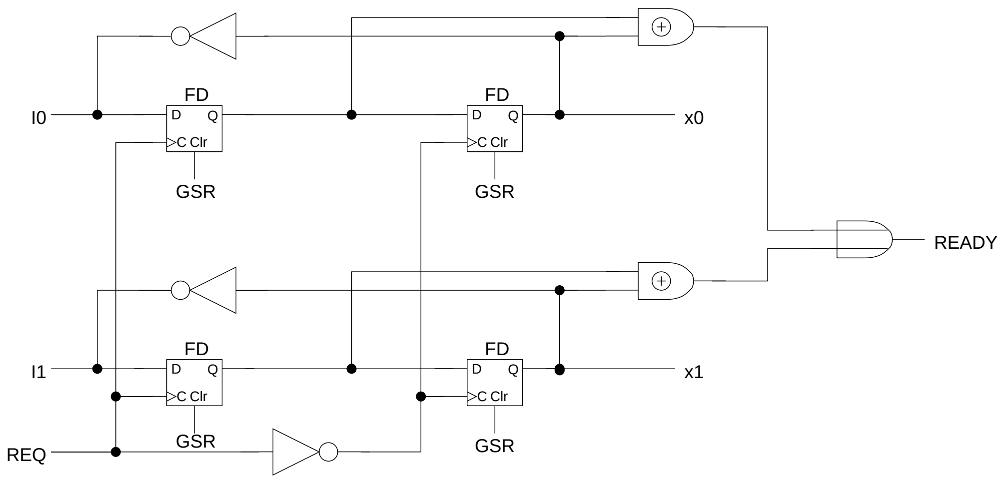



Abbildung 6.9: Das Herzstück eines Requestbausteinens

| Eingänge | Beschreibung |
|---|---|
| GSR | Global State Reset |
| I | Input (pegelgesteuertes Dual-Rail Bit) |
| REQ | Request |

| Ausgänge | Beschreibung |
|---|---|
| READY | Acknowledge |
| x | Output (flankengesteuertes Dual-Rail Bit) |

Es ist möglich, mehrere dieser Bauteile an einen einzigen **DDR_REG** anzuschließen. Dadurch kann ein gespeicherter Zustand in mehrere Richtungen requestet werden, ohne die Berechnung an anderer Stelle zu stören, was den Einsatz vom **DRBUFFER** auf ein Minimum reduziert.

Geht der Eingang `REQ` auf `high`, so wird der Ausgang `READY` nach kurzer Zeit auf `high` gelegt. Fällt `REQ` wieder auf `low`, so wird an `x` ein Flankenwechsel ausgelöst.

**REQ_DDR_REG**$x$

Dieses Bauteil wandelt den gespeicherten pegelgesteuerten Dual-Rail Zustand eines **DDR_REG**$x$ auf Anfrage wieder in einen flankengesteuerten Dual-Rail um. (Siehe Abb. 6.10).

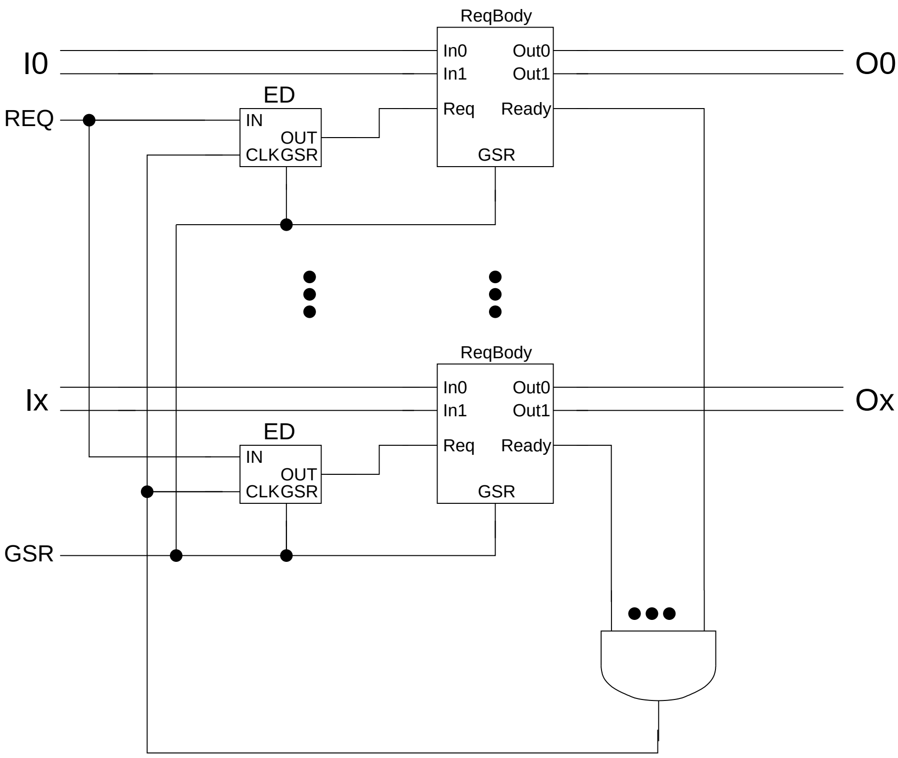



Abbildung 6.10: Schaltplan eines fertigen flankengesteuerten Dual-Rail Requestbausteines

| Eingänge | Beschreibung |
|---|---|
| `GSR` | Global State Reset |
| `I` | Input ($x$ pegelgesteuerte Dual-Rail Bits) |
| `REQ` | Request (flankengesteuerte Logik) |

| Ausgänge | Beschreibung |
|---|---|
| `O` | Output ($x$ flankengesteuerte Dual-Rail Bits) |

Bei einem eingehenden Flankenwechsel auf `REQ` werden die Zustände, die an `I` anliegen, in Flankenwechsel umgewandelt und an `O` ausgegeben.

**DDR_FASTOR**

Abb. 6.11 zeigt einen **DDR_FASTOR**-Multiplexer. Hierbei handelt es sich um ein Gatter, welches bei einem Flankenwechsel an einem Eingang sofort den ausgang flankt; es wartet also nicht, bis *a*lle Eingänge durch Flankenwechsel ihren logischen Zustand ausgegeben haben.

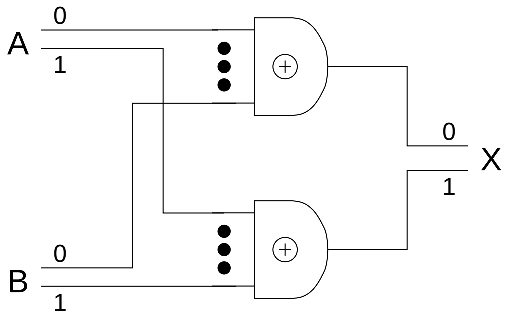



Abbildung 6.11: Schaltplan eines flankengesteuerten Dual-Rail Multiplexers

| **Eingänge** | **Beschreibung** |
|---|---|
| A | Input (Flankengesteuertes Dual-Rail Bit) |
| B | Input (Flankengesteuertes Dual-Rail Bit) |
| **Ausgänge** | **Beschreibung** |
| X | Output (Flankengesteuertes Dual-Rail Bit) |

Dieses Gatter wurde ausgehend von der Tatsache entwickelt, daß das Ergebnis der Verknüpfung $X = A \dot{\vee} B$ immer genau dann sein Ergebnis ändert, wenn genau eine der beiden Variablen $A$ oder $B$ ihren Zustand ändert.

Dadurch kommt dieses Bauteil ohne Edge-Detector und Edge-Generator aus, was eine enorme Einsparung von Chipfläche bedeutet. Allerdings müssen die vorgeschalteten Bauteile so gewählt sein, daß nur jeweils ein einziger ihrer Ausgänge einen Flankenwechsel aufweisen kann, da ansonsten das Ergebnis mit großer Wahrscheinlichkeit nicht mehr deterministisch ist.

Das übliche Decodieren einer Dualzahl, die angibt, welcher der Eingänge an den Ausgang gelegt werden soll, entfällt.

Obwohl es prinzipiell „nur“ eine Oder-Verknüpfung darstellt, haben wir es bei unserer asynchronen Implementierung dennoch überall dort verwendet, wo in der synchronen Variante Multiplexerbausteine befanden.

### DDR_FASTOR8

Ein Gatter, welches genau wie **DDR_FASTOR** funktioniert, allerdings für Busse von 8 Bit Breite gedacht ist.

| Eingänge | Beschreibung |
|---|---|
| A | Input (flankengesteuerter Dual-Rail-Bus von 8 Bit Breite) |
| B | Input (flankengesteuerter Dual-Rail-Bus von 8 Bit Breite) |

| Ausgänge | Beschreibung |
|---|---|
| X | Output (flankengesteuerter Dual-Rail-Bus von 8 Bit Breite) |

Statt einem Bit bestehen die Ein- und Ausgänge hier aus kompletten 8-Bit-Wörtern.

Dieselbe Eigenschaft, wie für XOR-Funktionen mit zwei Variablen, gilt auch für Funktionen mit mehr Variablen. Wenn genau eine ihren Zustand ändert, so ändert sich das Ergebnis. **FASTOR**-Multiplexer gibt es deswegen auch mit mehr als zwei Eingängen.

**DDR_BUFFER8**

Der in Abb. 6.12 dargestellte Buffer hat die Aufgabe, Flankenwechsel am Eingang an seine Ausgänge weiterzuleiten, oder sie zu verwerfen.

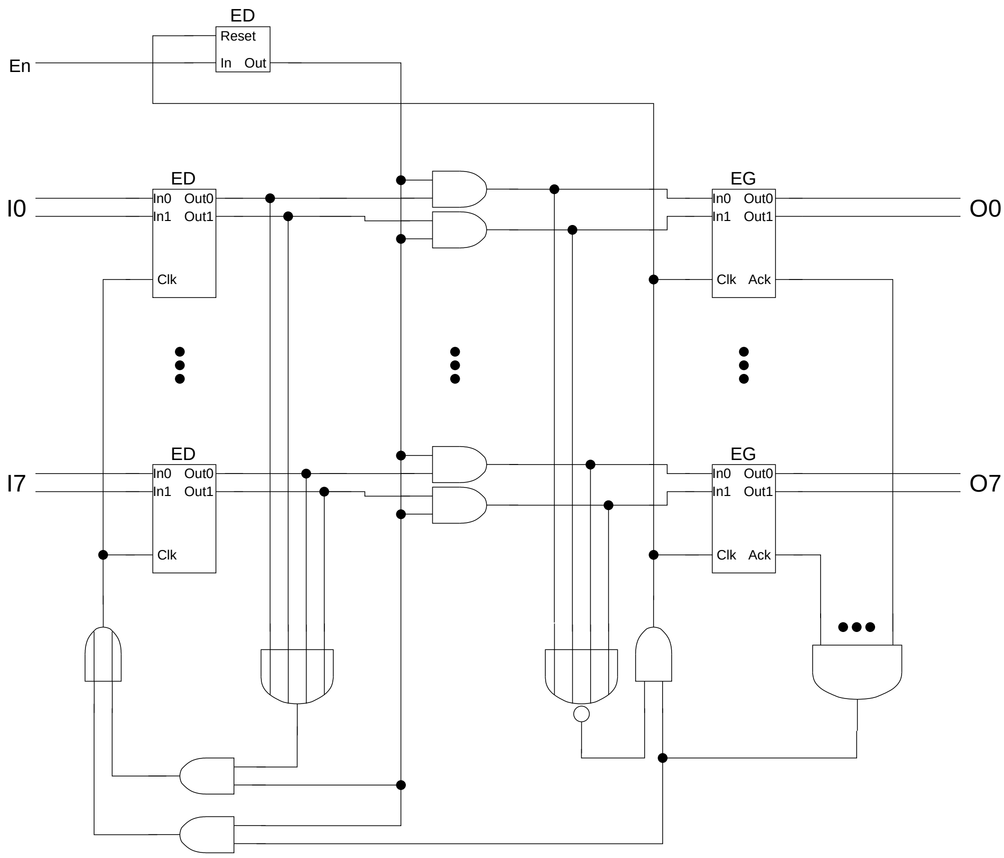



Abbildung 6.12: Schaltplan des **DDR_BUFFER8**

| Eingänge | Beschreibung |
|---|---|
| `I` | Input (flankengesteuerter Dual-Rail Bus von 8 Bit Breite) |
| `GSR` | Global State Reset |
| `En` | Enable (flankengesteuert) |

| Ausgänge | Beschreibung |
|---|---|
| `O` | Output (flankengesteuerter Dual-Rail Bus von 8 Bit Breite) |

Nachdem auf `En` ein Flankenwechsel an ein Bauteil gegangen ist, leitet es einen Flankenwechsel, der bei `I` ankommt, an `O` weiter.

Sollte an `I` ein Flankenwechsel auftreten, ohne daß `En` geflankt hat, so bleibt `O` auf den Pegeln, die vor diesem Flankenwechsel herrschten.

**ADDABLABH**

Abb. 6.13 zeigt einen Additionsbaustein für die indizierte Adressierung.

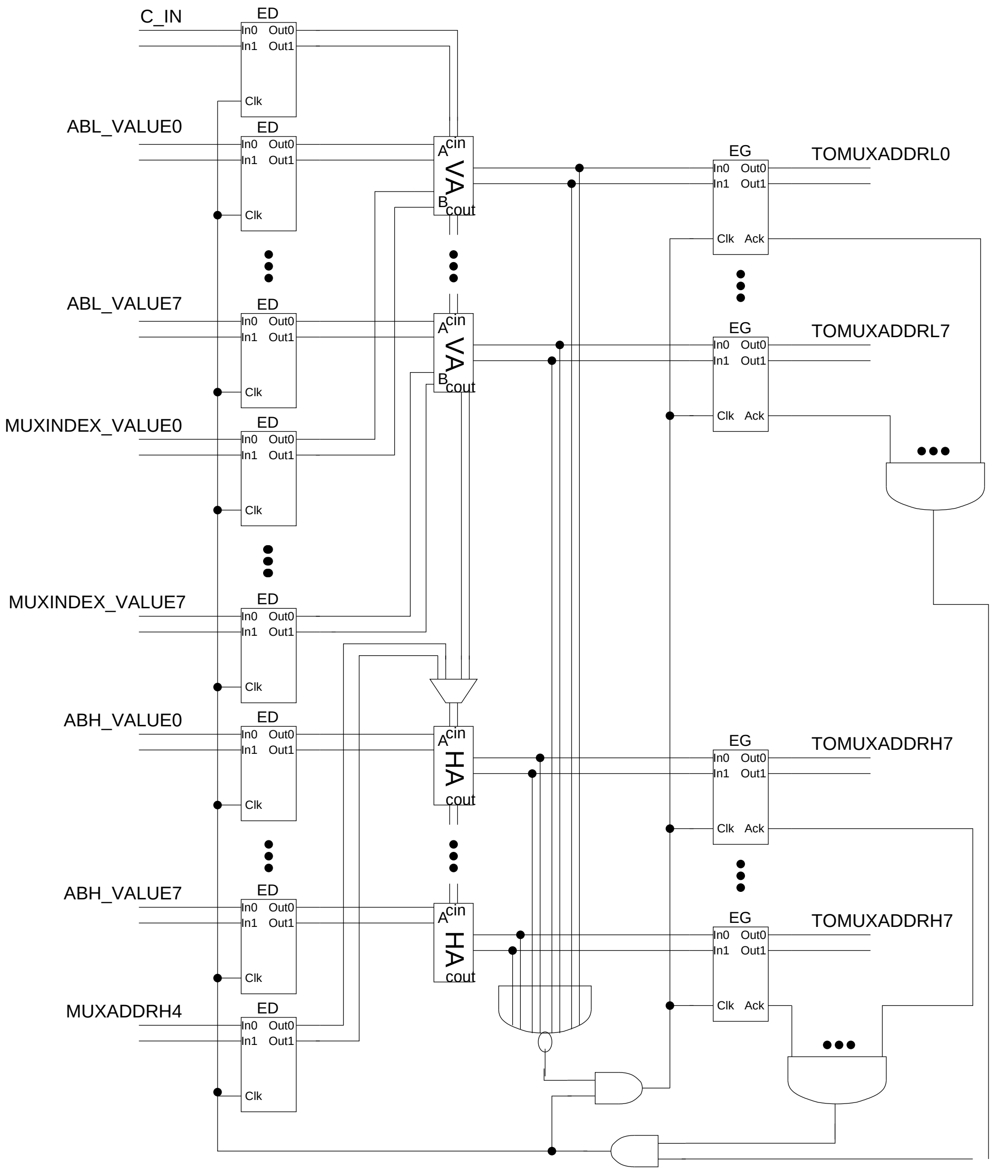



Abbildung 6.13: Schaltplan des **ADDABLABH**

| **Eingänge** | **Beschreibung** |
|---|---|
| GSR | Global State Reset |
| C_IN | Carry In (flankengesteuertes Dual-Rail Bit) |
| ABL_VALUE | Input (flankengesteuerter Dual-Rail Bus von 8 Bit Breite) |
| ABH_VALUE | Input (flankengesteuerter Dual-Rail Bus von 8 Bit Breite) |

| **Eingänge** | **Beschreibung** |
|---|---|
| MUXINDEX_VALUE | Input (flankengesteuerter Dual-Rail Bus von 8 Bit Breite) |
| MUXADDRH4 | Input (flankengesteuertes Dual-Rail Bit) |

| **Ausgänge** | **Beschreibung** |
|---|---|
| TOMUXADDRL | Output (flankengesteuerter Dual-Rail Bus von 8 Bit Breite) |
| TOMUXADDRH | Output (flankengesteuerter Dual-Rail Bus von 8 Bit Breite) |

Dieses Bauteil nimmt die Werte, die es an ABL_VALUE empfängt, und addiert sie mit MUXINDEX_VALUE und C_IN. Das Ergebnis wird an TOMUXADDRL ausgegeben.

Tritt bei dieser Addition ein Übertrag auf, und liegt MUXADDRH4 dabei auf 1, so wird zusätzlich noch der Wert von ABH_VALUE um eins erhöht und das Ergebnis an TOMUXADDRH ausgegeben.

**ADDPCLPCH**

Ein Additionsbaustein für den Programcounter, siehe Abb. 6.14.

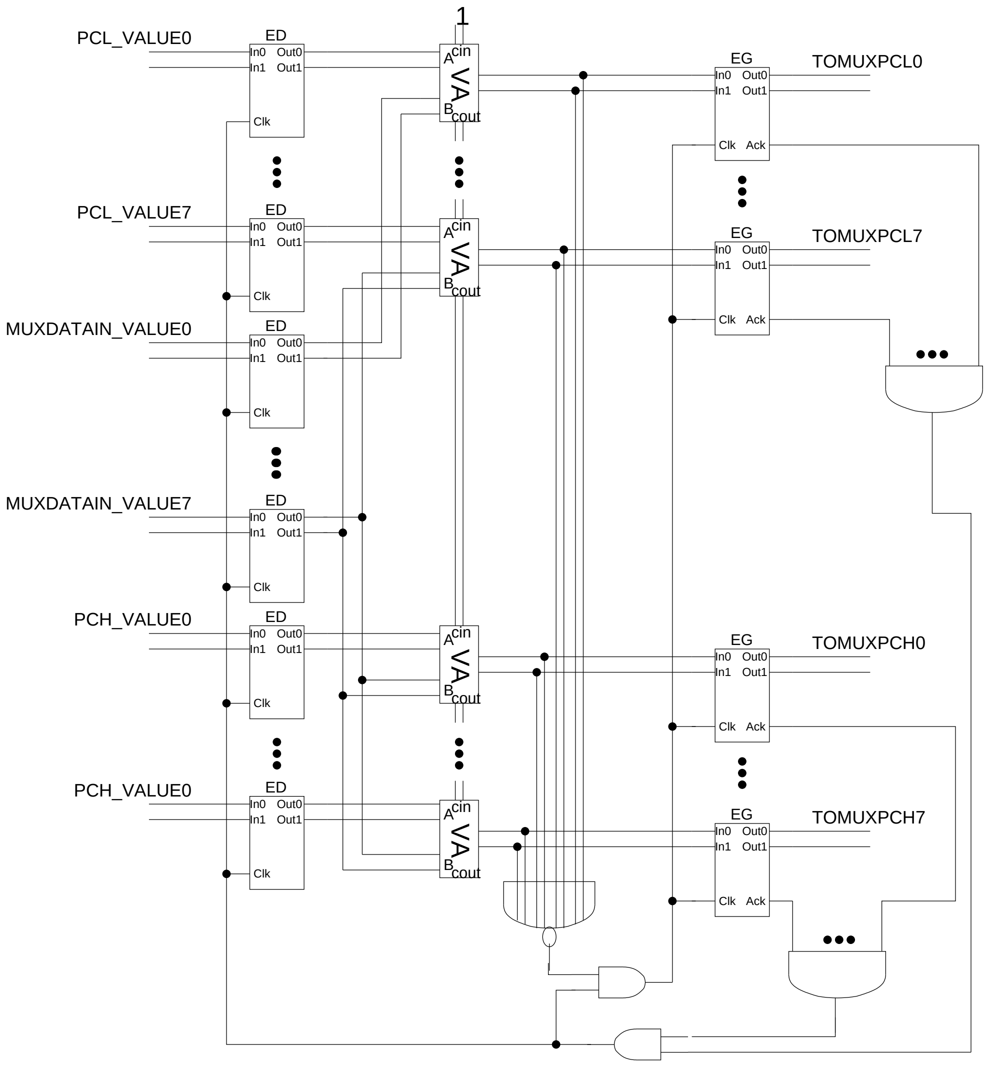



Abbildung 6.14: Schaltplan des **ADDPCLPCH**

| Eingänge | Beschreibung |
|---|---|
| GSR | Global State Reset |
| PCL_VALUE | Input (flankengesteuerter Dual-Rail Bus von 8 Bit Breite) |
| PCH_VALUE | Input (flankengesteuerter Dual-Rail Bus von 8 Bit Breite) |
| MUXDATAIN_VALUE | Input (flankengesteuerter Dual-Rail Bus von 8 Bit Breite) |

| Ausgänge | Beschreibung |
|---|---|
| TOMUXPCL | Output (flankengesteuerter Dual-Rail Bus von 8 Bit Breite) |
| TOMUXPCH | Output (flankengesteuerter Dual-Rail Bus von 8 Bit Breite) |

Dieses Bauteil nimmt die Werte, die es an PCL_VALUE und MUXDATAIN_VALUE bekommt, und addiert sie.

PCH_VALUE wird bei einem Übertrag der PCL-Addition in- bzw. dekrementiert, je nachdem, ob MUXDATA_VALUE ein Wert kleiner oder größer 128 ist, und das Ergebnis wird an TOMUXPCH ausgegeben.

**KONSTANTEN**

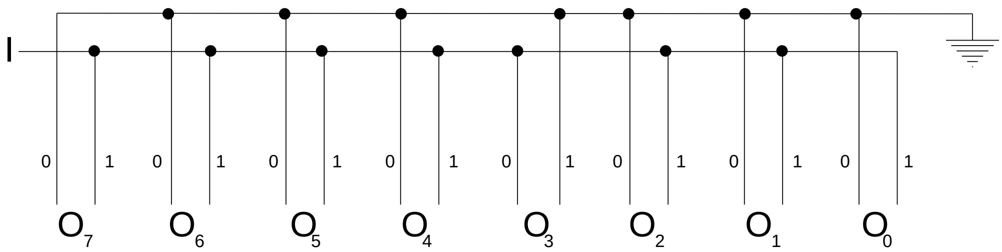



Abbildung 6.15: Beispiel zur Erzeugung einer flankengesteuerten Dual-Rail Konstanten

Konstanten sind nach flankengesteuerter Dual-Rail-Logik einfacher zu realisieren als nach konventioneller Logik, wo ein Eingang des Multiplexers explizit mit acht `high-low` Pegeln verbunden werden muß. Durch Select-Leitungen des Multiplexers werden diese Pegel bei Bedarf an dessen Ausgang gelegt, wo sie weiterverarbeitet werden.

Stattdessen werden nun die Konstanten durch Flankenwechsel immer genau dann erzeugt, wenn sie benötigt werden. Das Select-Signal an sich enthält somit bereits die Information, die es eigentlich auswählen sollte.

Obiges Beispiel erzeugt also bei einem Flankenwechsel an `I` am Outputbus `O` den flankengesteuerten Dual-Rail-Wert 11110111, oder `0xF7`.

## 6.2 Umstellung des Blockschaltbildes

Im Großen und Ganzen konnte bei der asynchronen Implementation des 6502 das Blockschaltbild der synchronen CPU (Abb. C.1) beibehalten bleiben. Allerdings darf nicht vergessen werden, daß die Busse nun nicht mehr in konventioneller Logik aufgebaut sind, sondern daß Ihre Werte nun durch Flankenwechsel codiert werden.

Auch ist es wichtig, die Reihenfolge und Anzahl der einzelnen Flankenwechsel zu kennen, damit nicht ein Bauteil plötzlich zuviele oder zuwenige Daten erhält, und das Ergebnis nicht mehr deterministisch ist.

Die wichtigste Umstellung des Blockschaltbildes liegt bei den Registern: Statt wie in der konventionellen Logik nur mit einem Bus als Ausgang, der sich an gegebener Stelle in weitere Busse verzweigt, hat bei diesem Blockschaltbild jedes Register mehrere Ausgangsbusse, bei denen durch die Requestbausteine bestimmt werden kann, welcher von ihnen einen Flankenwechsel aufweisen soll. Ein mögliches Blockschaltbild ist in Abb. C.2 dargestellt.

### Multiplexer

Die Multiplexer im Blockschaltbild wurden durch einfache **`FASTOR`**-Bausteine ersetzt. Aufgrund deren spezieller Eigenschaft, daß nur jeweils einer Ihrer Eingänge einen Flankenwechsel aufweisen darf, muß die äußere Beschaltung diesen Anforderungen genügen. Aus diesem Grunde wurden u.a. die zusätzlichen Spalten in der Decodertabelle eingeführt (siehe Abschnitt 6.3).

Unumgänglich ist auch der zusätzliche Multiplexer vor der ALU, mit dessen Hilfe garantiert wird, daß auch bei fehlenden Daten auf dem `DATAINBUS` ein notwendiger Flankenwechsel an sie geschickt werden kann, sollte es erforderlich sein, daß sie dennoch rechnet.

### Register

Statt 8 Flip-Flops als Register kommen nun **`DDRREG8`**-Bausteine zum Einsatz. Auch bekommen nicht mehr alle angeschlossenen Bauteile den gleichen gespeicherten Wert, sondern durch nachgeschaltete **`DDRREQ`**s werden die Daten dorthin gesendet, wo sie wirklich gebraucht werden.

### Buffer

An manchen Stellen ist es wichtig, daß die Multiplexer vor ankommenden Flankenwechseln geschützt werden. Kritische Busse, die an zuviele Multiplexer angeschlossen sind, sind in diesem Fall der `DATAINBUS` sowie der Ausgang der ALU.

Da auch die ALU Flankenwechsel ausgibt, die manchmal auf den `DATAOUTBUS` gelegt werden sollen, manchmal aber auch nicht, ist es sinnvoll, hier ebenfalls einen Buffer zu verwenden.

### Additionsbausteine

Die Additionsbausteine für die indizierte Adressierung und die Sprungadressen werden statt aus jeweils zwei Bausteinen nun aus einem einzigen gebildet, damit der Übertrag nicht überflüssigerweise in flankengesteuerte Dual-Rail Logik übersetzt werden muß, nur um ihn kurz darauf wieder zurückzuverwandeln.

Als problematisch hat sich hier die Zeropageadressierung erwiesen, da bei dieser der Inhalt des **ABH**-Registers nicht requested werden darf. Aus diesem Grunde wurde die Konstante `0x00` vom **MUXADDRH** entfernt und als Alternative in den **ADDABLABH** eingegeben. Durch ein zusätzliches AND-Gatter in dem Bauteil wurde außerdem dafür gesorgt, daß kein Übertrag auftreten kann, der diese `0x00` verändert.

## 6.3 Erweiterung der Decodertabelle

### Hinzufügen neuer Decoderausgänge

Die Umstellung des Blockschaltbildes macht es erforderlich, die Decodertabelle des asynchronen 6502 im Vergleich zur synchronen Lösung um einige neue Spalten zu erweitern. Unser Konzept sieht hierfür vor, daß nur die Bestandteile des Prozessors aktiviert werden, die auch tatsächlich für den entsprechenden Rechenschritt benötigt werden, da es ansonsten zu fehlgeleiteten Daten oder verfälschten Ergebnissen käme. Dies konnte aber mit der bestehenden Decodertabelle des synchronen 6502 nicht erreicht werden, da diese die entsprechende Funktionalität nicht bietet. So wurden die zusätzlichen Leitungen `ALU_EN` und `LINKS_NULL` erforderlich, um den neu hinzugekommenen Multiplexer vor der **ALU** korrekt ansteuern zu können.

In der konventionellen Logik ist es möglich, daß eine Spalte $x$ einen der drei möglichen Werte „0“ (für einen `low`-Pegel), „1“ (für einen `high`-Pegel), und „-“ (für einen beliebigen, legalen, halt „Don't Care“, Pegel) welcher immer dann eingetragen wird, wenn das zugehörige Bauteil nicht wirklich rechnen sollte, enthält. Für die asynchrone Decodertabelle haben wir daraus drei Spalten gemacht: $x$_0, $x$_1, $x$_DC. Dies ist eine Abkehr von der bisher verwendeten Dual-Rail-Logik; hin zu einer „Multi-Rail-Logik“, bei der Werte nicht mehr nur Binärzahlen durch `0` und `1` auf Gruppen von zwei Leitungen codiert werden, sondern die Anzahl der Leitungen der Anzahl der codierten Werte entspricht. Die naheliegendste Alternative, den „Don't Care“ Zustand einfach durch `low`-Pegel auf beiden Dual-Rail-Leitungen auszudrücken scheitert jedoch, da nur durch `high`-Pegel am Ausgang der DNF davon ausgegangen werden kann, daß sie fertig gerechnet hatte (Siehe dazu den Abschnitt 4.3).

Bei Spalten, die in der ursprünglichen Tabelle Gruppen gebildet haben, und gemeinsam eine Binärzahl darstellten, zum Beispiel für Multiplexer, wurden decodiert. Die drei Spalten, die z.B. **MUXALU** angesteuert hatten, enthielten Zahlen von null bis sechs. Daraus wurden in der asynchronen Variante eine Gruppe von sieben Spalten, plus eine DC-Spalte, die immer dann auf `high` liegt, wenn die ALU nicht rechnen darf. Ein `high` in dieser Spalte bedeutet also, daß die entsprechende Komponente deaktiviert ist. Nach dem selben Prinzip verfuhren wir mit allen anderen Spalten für Multiplexer.

Für die `ALU_OP` Spalten hätte dies einen allzugroßen Overhead an Edge-Generatoren produziert, weswegen wir hier jedes Bit einzeln umgesetzt haben.

### Vereinfachung und Implementierung des neuen Decoders

Nach Komplettierung und Überprüfung der Decodertabelle wurde der *VHDL*-Code mit Hilfe einiger Programme wie zum Beispiel Espresso automatisch aus einer Textversion der Decodertabelle erstellt. Eine genaue Beschreibung der Vorgehensweise findet sich im Abschnitt über den synchronen 6502-Decoder, da dort genauso verfahren wurde.

### Decoderwrapper

Aufgrund der erwähnten „Multi-Rail-Ansteuerung“ stellten sich auch neue Anforderungen an die Ansteuerung der Edge-Generatoren am Ausgang, da die **`DOPPELEG_DR`** Bausteine nur für Dual-Rail und damit für zwei Leitungen ausgelegt sind. Sobald eine der Leitungen einen Flankenwechsel auslöst, ist das Bauteil fertig. Hier allerdings sind es drei oder mehr Leitungen, bei denen immer genau eine einen Flankenwechsel auslöst, weswegen ein wenig Handarbeit nötig wurde.

Ein weiteres Problem lag an den Eingängen des Decoders. Weil der Kern des Decoders automatisch erzeugt wurde, war es nicht mehr gegeben, daß er mit seiner Berechnung warten würde, bis auch wirklich auf jedem Dual-Rail-Eingang genau ein Flankenwechsel stattgefunden hatte. Deswegen müssen alle Ausgänge der Edge-Detectoren durch nachgeschaltete AND-Gatterkünstlich auf `low` gezogen werden, bis an allen ihren Eingängen ein gültiges Datum anliegt.

## 6.4 Umstellung der ALU

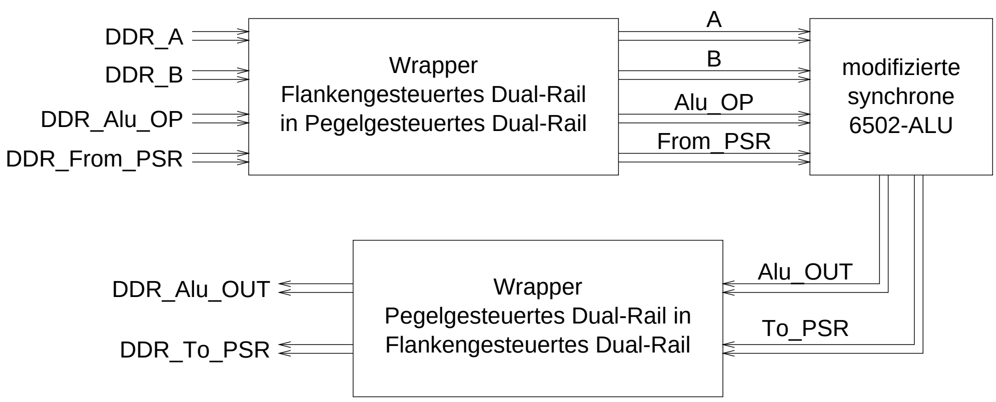



Abbildung 6.16: schematische Zeichnung der ALU mit Wrapper

Die Umsetzung der ALU in eine asynchrone Variante war nicht sehr kompliziert, da daß eigentliche Rechenwerk bis auf wenige Ausnahmen komplett übernommen werden konnte.

Da die ALU intern mit pegelgesteuertem Dual-Rail arbeitet, jedoch flankengesteuerte Dual-Rail-Signale erhält, mußte das Rechenwerk von einem Wrapper umgeben werden, der die Signale umsetzt. Dieser Wrapper besteht in der vorgeschaltete Komponente aus **DOPPELED_EDs**, die die ankommenden Leitungen konvertieren und die Pegel so lange halten, bis die zweite, nachgeschaltete Komponente des Wrappers ein gültiges Ergebnis erkannt und das `dRES`-Signal generiert hat.

Die nachgeschaltete Komponente, bestehend aus **DOPPELEG_DRs**, erzeugt aus den Ergebnissen der ALU, die in pegelgesteuertem Dual-Rail vorliegen, flankengesteuerte Dual-Rail-Signale, und gibt diese Flanken an die nachgeschalteten Bauteile weiter. Durch diese direkte Umsetzung der synchronen ALU in eine pegelgesteuerte Dual-Rail-ALU konnten die eigentlichen Änderungen des Rechenwerkes auf eine Umstellung der Leitungstypen im *VHDL*-Code beschränkt werden, durch die unsere Dual-Rail-Gatter eingebunden wurden. Lediglich Konstanten wie `0` und `1` mußten abhängig von Eingangssignalen generiert werden, um so zu garantieren, das immer nur gewünschte Flankenwechsel generiert werden, denn die ALU erhält nur dann Eingabesignale, wenn sie auch wirklich eine Berechnung durchführen muß. Statische Signale würden jedoch immer generiert, und dadurch das ganze Konzept der Flankenverarbeitung durcheinanderbringen.

## 6.5 Check-Machine-State (kurz: „CMS“)

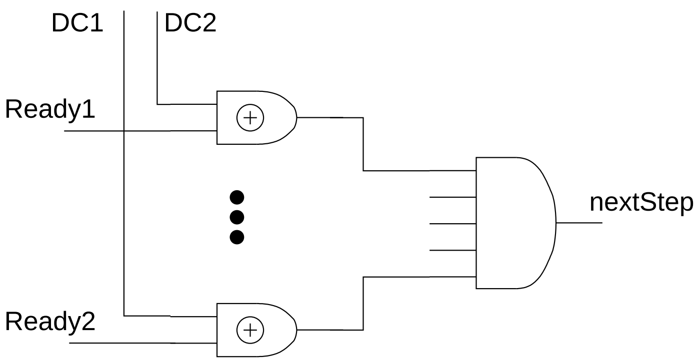



Abbildung 6.17: Das Check-Machine-State des asynchronen 6502

| **Eingänge** | **Beschreibung** |
|---|---|
| `DC` | „Don’t care“ Decoder-Signal; Bauteil soll nicht beachtet werden |
| `Ready` | Bauteil-Signal; `high`, wenn das Bauteil fertig ist |
| **Ausgänge** | **Beschreibung** |
| `nextStep` | veranlaßt den Decoder den nächsten Zyklus zu starten |

Tabelle 6.32: Anschlüsse des CMS

Anmerkung zu Tabelle 6.32: `DC` und `Ready` müssen für jedes zu beachtende Bauteil vorhanden sein.

Das Check-Machine-State (oder auch kurz „CMS“ genannt) trägt dafür Sorge, das eigentliche Signal zu erzeugen, welches den Takt ersetzt. Es muß genau dann ein Signal `nextStep` liefern, wenn alle Bauteile den aktuellen Zyklus abgeschlossen haben. Dafür bekommt es von dem Decoder das Signal `DC`, welches symbolisiert, ob ein Bauteil im aktuellen Zyklus berücksichtigt werden muß. Auf der anderen Seite melden die Bauteile ein `Ready`, wenn sie fertig sind mit ihrer Operation. Also wird das gewünschte `nextStep` erreicht, indem die beiden Signale, die Informationen über ein Bauteil liefern, durch ein XOR verknüpft werden. Die XOR-Gatter müssen nun noch durch ein flankengesteuertes Dual-Rail AND-Gatter zusammengefaßt werden, und das Ergebnis ist das geforderte Signal `nextStep`.

## 6.6 Abschließende Bemerkungen

Unsere Implementierung des 6502 mit den genannten Bauteilen und dem Blockschaltbild in Abb. C.2 benötigte 3091 von 3136 CLBs. Die restlichen zur Verfügung stehenden CLBs reichten nicht mehr aus, um Übersetzungsbausteine von konventioneller in flankengesteuerte Dual-Rail Logik hinzuzufügen, weswegen ein Benchmark im Vergleich zur synchronen Variante leider entfallen muß.

# Kapitel 7

# Fazit und Ausblick

Dieses Kapitel faßt die Aufgaben und Ziele des Projektseminars *PsalM* zusammen und schildert kurz die Hauptprobleme und die erreichten Ergebnisse. Am Ende soll ein Ausblick auf Lösungen beim Design asynchroner Schaltungen gewagt werden.

## 7.1 Implementierung des asynchronen Prozessors: Teilziele und Rahmenbedingungen

Während des Projektseminars „*PsalM*“ wurde der Versuch unternommen, von bestehenden synchronen Prozessoren asynchrone Versionen zu entwerfen und diese auf programmierbaren Logikchips zu implementieren. Dies geschah unter der Prämisse, durch die Asynchronizität Geschwindigkeitsvorteile gegenüber den synchronen Varianten zu erzielen.

### Entscheidung zur Implementierung auf einem FPGA

Wir haben uns entschlossen, unsere Prozessoren auf einem FPGA zu implementieren, da FPGAs die einzigen uns zur Verfügung stehenden Bauteile waren, die unseren Anforderungen genügten. Verlangt war eine einfache Möglichkeit, verschiedene Prozessoren auf einem einzigen Chip zu implementieren (Rekonfigurierbarkeit), dabei verschiedene Ansätze zu verfolgen (Flexibilität), diese einfach zu testen (Kommunikation mit einem Host-System) und die verschiedenen Ansätze vergleichen zu können (hohe Schaltgeschwindigkeit der Einzelzellen). Es hat sich allerdings herausgestellt, daß die uns zur Verfügung stehenden FPGAs nicht optimal zur Implementierung von asynchronen Schaltungen geeignet sind.

### Testweise Implementierung einer Minimal-CPU

Als erste Test-CPU für die verschiedenen asynchronen Ansätze wurde die DEP-CPU gewählt, da diese ein minimalistischer Prozessor ist, der nur über die grundlegendsten arithmetischen und logischen Operationen verfügt. Dieser Prozessor wurde erst synchron implementiert und anschließend in die vier asynchronen Ansätze portiert. Hierzu wurden für jeden Ansatz erst eine passende Gesamtstruktur und dann eine Reihe benötigter Grundbausteine wie Gatter, Konverter, Speicher, etc. entwickelt.

Schon dabei zeigte sich die grundlegende Problematik beim Entwurf zeitkritischer Schaltelemente (wie etwa Laufzeitketten und Verzögerungselemente) auf FPGAs. So schied während dieser Phase einer der Ansätze – das Micro-Pipelining-Verfahren – aus, da hierfür zwei Voraussetzungen nötig waren, die sich aber nicht realisieren ließen. Zum einen müssen Laufzeitverzögerungen erzeugt werden, deren Dauer über der Schaltzeit einzelner Gatter, aber deutlich unter den Periodendauern der angeschlossenen Taktgeneratoren liegen. Solche Verzögerungen ließen sich nach den Optimierungen der Entwicklungsumgebung nicht mehr generieren. Zum anderen muß beim Einsatz dieses Verfahrens aber auch das zeitliche Verhalten der eigentlichen CPU-Logik *gemessen* oder im Simulationen exakt genug abgeschätzt, beziehungsweise *vorhergesagt* werden. Die Messung so kurzer Zeiten war mit den zur Verfügung stehenden Mitteln nicht möglich und eine Simulation auf Entwurfsebene (*VHDL* oder *Schematic Editor*) mit Hilfe der Programme der *Xilinx Foundation Series 3.1i* stellte sich als zu unzuverlässig und ungenau heraus. Ein manuelles Untersuchen auf CLB-Ebene wäre zwar (bei eingeschränkter Genauigkeit) mit der *Xilinx Foundation Series 3.1i* möglich, bei der Entwicklung einer kompletten CPU aber zu aufwendig. Der Micro-Pipelining-Ansatz könnte mit Hilfe zuverlässiger und genauer Software-Werkzeuge oder entsprechender Labor-Hardware (Logic-Analyzer mit mehreren hundert MHz Meßauflösung) weiterverfolgt und realisiert werden. Ergebnis der ersten Phase war die Entwicklung eines eigenen asynchronen Ansatzes – einer Mischung aus pegel- und flankengesteuertem Dual-Rail. In dieser Schaltungstechnik wurde dann eine asynchrone Variante des 6502 entwickelt.

### Entscheidung zum 6502

Die Wahl des eigentlich zu entwickelnden Prozessors wurde bestimmt durch die Komplexität des Prozessors, dessen Verbreitung und nicht zuletzt durch die Größe des Prozessors. Eine Abschätzung der Größe war notwendig, da ein FPGA nur eine begrenzte Anzahl von programmierbaren Logikelementen besitzt. Leider kam aus Platzgründen der wesentlich interessantere Z80 nicht mehr in Frage. Dieser verfügt über 158 Befehle, darunter sogar 16Bit-Arithmetikbefehle und vier Byte lange Operationen (siehe Zilog Datenblatt, zum Beispiel in [ELEKTOR]). So fiel die Wahl auf den ebenfalls sehr bekannten und weit verbreiteten Prozessor 6502 von *MOS Technologies*. Mit 56 Befehlen und einer maximalen Befehlslänge von drei Bytes paßt dessen Implementierung gerade noch auf den von uns genutzten FPGA.

## 7.2 Probleme bei der Implementierung asynchroner CPUs auf FPGAs

### Probleme bei der Design-Entwicklung

Bei der Implementierung des synchronen 6502 mußte als erstes der grobe Aufbau des Prozessors festgelegt werden. Dokumentation über die Softwareseite des 6502 war zwar reichlich zu finden, allerdings nicht zum detaillierten Hardwareaufbau wie z.B. Busaufbau. Dies mußte neu entwickelt werden, wobei der Platzbedarf ein große Rolle spielte.

Die Umsetzung der Implementierung des 6502 ins Asynchrone brachte zunächst viel zu große Prozessoren, worauf hauptsächlich auf Platz und nicht auf Performance optimiert wurde. Nach mehreren Platzoptimierungen mittels des im Anhang beschriebenen „Tableshrinkers“ und ein paar zusammengefaßter Komponenten konnte der Platzbedarf des Prozessors soweit reduziert werden, daß seine asynchrone Implementierung auf den FPGA paßt.

### Probleme der verwendeten Software

Dabei zeigte sich, daß die *Xilinx Foundation Series 3.1i* im Falle zu großer Implementierungen nur bis zu einer maximalen Größe erstellte und die noch nicht untergebrachte Logik einfach abschneidet. Eine Warnung wird – mit den Standardeinstellungen – dabei nicht gezeigt.

Ein weiteres Problem zeigte sich mit den Unterschieden zwischen der Simulation des Prozessors mittels Software und dem Test auf dem konfigurierten FPGA. Besonders die für das Projekt wichtige Simulation des Timings stimmte in vielen Fällen nicht mit dem Verhalten des FPGAs überein.

In der asynchronen Variante der 6502-ALU wurden sämtliche `CASE`-Anweisungen durch Gatterschaltungen ersetzt, da diese im Verdacht standen, bei der Kompilierung Schwierigkeiten zu verursachen.

### Probleme mit der verwendeten Dokumentation und Hardware

Gab es schließlich doch eine komplette, auf den FPGA passende CPU, so tauchten Probleme mit der *V.C.C.*-Schnittstelle zwischen erzeugtem Prozessor und PCI-Bus auf. Mal waren Teile der Schnittstelle nicht oder nicht ausreichend dokumentiert. Und mal gab es sogar gravierende Fehler in der Dokumentation. Dieser Umstand machte es nötig, mittels „Trial-and-Error“ die tatsächlichen Gegebenheiten herauszufinden, was sich als sehr zeitraubend und umständlich erwies.

## 7.3 Gesamtergebnisse und ein Ausblick

Die Ergebnisse der ersten Projektphase führten zu einem asynchronen Ansatz, der versucht, die geringe Größe des pegelgesteuerten Dual-Rail mit der Geschwindigkeit des flankengesteuerten Dual-Rail zu verbinden. Dieser Ansatz zeigte sich auch bei einer größeren CPU wie dem 6502 als anwendbar und erfolgversprechend.

Des weiteren ist in der ersten Hälfte des Projektseminars eine Entwurfsmethodik und (für jeden Dual-Rail-Ansatz) ein Satz von Grundelementen entwickelt worden, mit denen der danach entwickelte synchrone 6502 mit deutlich geringerem Aufwand in eine asynchrone Variante umgewandelt werden konnte. Die Vorgehensweise und die entwickelten Komponenten sind dabei so allgemein, daß sie auf die meisten synchronen

Implementationen von CPUs ähnlicher Größe und mit ähnlichem Aufbau angewendet werden und diese mit relativ geringem Aufwand in asynchrone Implementationen umgeschrieben werden können.

Der asynchrone 6502 benötigte dabei 3091CLBs, also nur knapp das Dreifache der synchronen Implementierung. Dies übertrifft die Erwartungen bei weitem – unsere Abschätzungen zu Beginn des Projektseminars lagen bei einem vielfachen davon. Hier muß jedoch noch beachtet werden, daß die Optimierungseinstellungen in beiden Fällen unterschiedlich waren: Der synchrone 6502 wurde mit wesentlich weniger Optimierungen durchkompiliert, um die Kompilierzeit möglichst gering zu halten, während der asynchrone mit maximaler Optimierung durchkompiliert wurde, um auf den sehr knappen CLBs den gesamten Prozessor unterzubringen. Außerdem verläuft die Optimierung unterschiedlich gut und abhängig von der Größe der unterzubringenden Logik. Der tatsächliche Größenunterschied liegt also über dem Faktor drei.

Weitere Ergebnisse konnten wir auch bei der Frage des Design-Stils und daraus resultierender Größenunterschiede feststellen: Die synchrone 6502 CPU benötigte zusammen mit dem PCI-Wrapper und dem PCI-Interface 1107CLBs. Ein zum Vergleich kompilierter, in Microcode implementierter und kaum optimierter 6502 des „Free IP Project“ [FREEIP] verbrauchte 1118CLBs – und das sogar ohne den PCI-Wrapper! Dieser Vergleich zeigt, daß bei gleicher Funktionsweise, mit einer etwas aufwendigeren *VHDL*-Beschreibung gut fünf Prozent eingespart werden können.

Neben Verwendung eines besseren „Programmierstils“ könnte auch die Wahl der benutzten Beschreibungssprache und damit die dafür verfügbaren Compiler eine Rolle spielen. So könnte man – was wir jedoch nicht weiter untersucht haben – die verwendete Hardwarebeschreibungssprache *VHDL* gegen das komfortablere *Verilog* austauschen.

Leider konnte ein direkter Geschwindigkeitsvergleich zwischen asynchroner und synchroner CPU aus den genannten Gründen nicht mehr durchgeführt werden. Wie erste Simulatorläufe jedoch zeigten, laufen manche Verarbeitungsschritte deutlich schneller als der maximal zu verwendende Takt ab. Jedoch zeigte sich schon sehr schnell das Hauptproblem asynchroner CPUs: Es gibt nur viel zu kurze interne Verarbeitungsschritte und viel zu häufig Zugriffe auf externe Komponenten (meistens das RAM). Auch wenn mit dem „DEP-Format“ eine technische Möglichkeit zur Beschleunigung mancher RAM-Zugriffe gefunden wurde (auf Kosten der Speicherausnutzung), kann eine CPU (der Größe eines 6502) in einer Standardumgebung mit asynchroner Logik kaum Geschwindigkeitsgewinne erreichen. Bei den heute aktuellen Prozessoren mit wesentlich komplexerer interner Logik und längeren Berechnungen zwischen den Speicherzugriffen sieht das aber sicherlich anders aus.

Ideal – und zu empfehlen für folgende Projekte in diesem Bereich, unabhängig von der CPU-Größe – wäre eine Kombination von asynchroner CPU und einem asynchronen Cache auf einem Chip.

Abschließend ist zu sagen, daß man asynchrone Logik besser auf FPGAs implementiert, die dafür ausgelegt sind oder aber direkt auf ASICs zurückgreift. Ein Beispiel hierfür sind die Amulet1 und Amulet2e Prozessoren [MANCH].

# Anhang A

# Tools

Neben den in Kapitel 2 beschriebenen Programmen der *Xilinx Foundation Series 3.1i* haben wir – zur Beschleunigung der Entwicklung, für eine bequemere Arbeit oder um überhaupt zu einer Lösung zu kommen – ein paar freie und ein paar selbstentwickelte Programme verwendet. Diese werden in den kommenden Abschnitten beschrieben.

## A.1 *PsalM*-Desk

Zu ihrer PCI-Karte (siehe Kapitel 2.4) bietet die Firma *V.C.C.* mehrere Programme und DLLs. Hierzu zählt auch das Programm *Testcommander*, das dem Konfigurieren des FPGAs auf der PCI-Karte, dem Debuggen der entwickelten Logik und dem Austauschen von Daten zwischen FPGA und Host-Anwendung dient.

Der *Testcommander* ist allerdings nur sehr unbequem über eine Kommandozeile zu bedienen und erlaubt kein automatisches Messen von Antwortzeiten, wie wir es für einen Vergleich der CPUs brauchen. Deshalb wurde ein eigenes Programm „*PsalM*-Desk“ geschrieben, das die von uns benötigten Funktionsweisen zur Verfügung stellt.

Hierzu zählen neben den Grundfunktionen wie dem Konfigurieren des FPGA und dem Datenaustausch mit der PCI-Karte auch spezielle Anpassungen an die von uns in diesem Projektseminar entwickelten Designs. So gibt es in jedem Projekt – da es sich bei allen Designs um CPUs handelt – eine RESET- und meistens ein oder mehrere Interrupt-Leitungen. Auf diese häufig genutzten Funktionen kann im *PsalM*-Desk besonders einfach zugegriffen werden. Hinzu kommen noch SingleStep-Funktionen für ein einfacheres Debuggen der CPUs, Geschwindigkeitsmessungen um die Designs vergleichen zu können und weitere für unsere Entwicklungen benötigte Funktionalitäten.

Genau wie die *V.C.C.*-Programme setzt auch *PsalM*-Desk ein MS-Win32-Betriebssystem[1] voraus. Getestet wurde es nur unter MS-Windows NT 4.0.

[1]MS-Windows 9X, ME, NT 4 oder 2000

### A.1.1 H.O.T.-DLLs

Für ein Konfigurieren des FPGAs aus einer eigenen Host-Anwendung heraus und um Daten zwischen einer Host-Anwendung und dem FPGA auszutauschen, stellt *V.C.C.* drei DLLs bereit (HOT.dll, HOT2.dll und HOTpci.dll). Zu den DLLs gibt es Header- und LIB-Dateien, die einen objektorientierten Zugriff von eigenem C++-Code heraus ermöglichen.

Vor einem Zugriff auf den FPGA muß eine Anwendung eine Instanz einer HOT-Klasse erzeugen. Dann kann das Programm über Methoden dieser Instanz den FPGA konfigurieren, das RAM auf der Karte auslesen und beschreiben und über bestimmte Leitungen des FPGAs Daten mit der konfigurierten Logik austauschen. Danach muß die Instanz wieder über ihren Destruktor gelöscht werden.

Auch das *PsalM*-Desk benutzt (allerdings über eine Zwischenstufe – siehe nächsten Abschnitt) die H.O.T.-DLLs, weswegen sich diese drei Dateien entweder im selben Verzeichnis wie die *PsalM*-Desk.exe oder im Windows-Suchpfad befinden müssen.

Leider sind die H.O.T.-DLLs und/oder die entsprechenden Teile des HOT_CORE (siehe Kapitel 2.5.1) nicht ganz fehlerfrei, weswegen es zum Absturz des Host-PC kommen kann, wenn mit demselben Programm mehrmals nacheinander der FPGA konfiguriert wird (dies gilt natürlich auch für *PsalM*-Desk). Als weiterer Fehler macht sich ein falsches Auslesen der RAM-Bänke bemerkbar, sobald man den FPGA konfiguriert hat. Da die RAM-Bänke bei einem Programmneustart nicht gelöscht werden, sollte die Host-Anwendung (zum Beispiel *PsalM*-Desk) erst beendet und neu gestartet werden, wenn der RAM nach einem FPGA-Zugriff ausgelesen werden soll.

### A.1.2 OwnHOT-DLL

*PsalM*-Desk greift nicht direkt auf die Methoden der H.O.T.-DLLs zu. Als Zwischenstufe wurde die Datei OwnHOT.dll entwickelt, die beim Aufruf von *PsalM*-Desk geladen wird und die deshalb – wie die H.O.T.-DLLs – entweder im selben Verzeichnis wie die *PsalM*-Desk.exe oder im Windows-Suchpfad liegen muß. Für die Verwendung einer eigenen DLL als Zwischenstufe sprachen folgende Gründe:

- Der objektorientierte Zugriff, wie er von den H.O.T.-DLLs zur Verfügung gestellt wird, ist sehr umständlich. Die Ein-/Ausgabe-Aufrufe erfolgen bei den Host-Anwendungen in der Regel aus verschiedenen Programmteilen und müssen, obwohl schon in den DLLs zusammengefaßt, zuvor künstlich über eine Objektreferenz kanalisiert werden. Hinzu kommt, daß die Vorteile objektorientierter Programmierung überhaupt nicht genutzt werden können, da es weder gleichzeitig mehrere Instanzen geben kann (beziehungsweise nie ein gleichzeitiger Zugriff erfolgen *darf*), noch eine Vererbung gebraucht wird.
- In den Host-Anwendungen werden meist jeweils nur manche der H.O.T.-Routinen und nicht alle der möglichen Parameter gebraucht. Dies erlaubt eine kleinere und einfacher zu benutzende Schnittstelle (realisiert mit der OwnHOT-DLL).
- Über die OwnHOT-DLL läßt sich (beispielsweise per winsocket) ein verteilter Zugriff mehrerer PCs innerhalb eines lokalen Netzes auf eine einzelne PCI-

Karte steuern. Dies ist in der aktuellen Version noch nicht implementiert, kann aber später ohne Änderung der Anwendungen nachgeholt werden.

- Um Host-Anwendungen auch ohne PCI-Karte laufen zu lassen, wurde ein Simulationsmodus in die OwnHOT-DLL integriert. Dieser kann über eine Funktion an- und abgeschaltet werden und überspringt in Funktionsaufrufen der Anwendung die eigentlichen H.O.T.-Routinen. So können ohne Änderung der Aufrufe die sonstigen Funktionen der Anwendung genutzt werden, auch wenn keine PCI-Karte installiert ist.

- Schließlich begrenzt die OwnHOT-DLL die Parameter der H.O.T.-Aufrufe auf ungefährliche Werte. Die H.O.T.-DLLs gestatten zum Beispiel ein Löschen des Flash-EEPROMs auf der PCI-Karte, was bei den meisten Anwendungen nicht gebraucht wird und bei fehlerhaften Funktionsaufrufen schnell die Karte unbrauchbar macht.

### A.1.3 Die *PsalM*-Desk Oberfläche

Die Oberfläche vom *PsalM*-Desk gliedert sich in ein Hauptmenü am oberen Fensterrand, einen Pufferbereich in der linken und einen PCI-Board-Bereich in der rechten Fensterhälfte und eine Statuszeile am unteren Fensterrand.

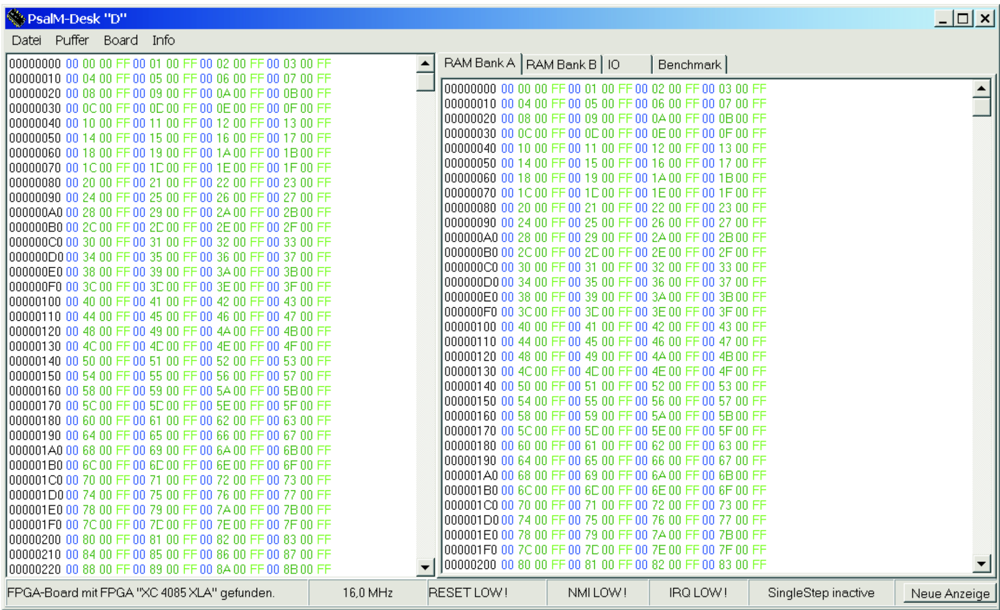



Abbildung A.1: Die *PsalM*-Desk Oberfläche

#### Der Pufferbereich

Der Pufferbereich ermöglicht ein Vorbereiten, Konvertieren und Ansehen der in die RAM-Bänke zu ladenden Daten und ein Abspeichern der aus den RAM-Bänken gelesenen Daten. Die Daten des Puffers werden als Hex-Dump angezeigt und können bei

Bedarf farblich gekennzeichnet werden. Dies ist bei Verwendung des DEP-Formats hilfreich. Am linken Rand wird die Adresse der jeweiligen Datenzeile (ebenfalls als Hex-Dump) gezeigt.

### Der PCI-Board-Bereich

Der PCI-Board-Bereich (rechts) gliedert sich in vier sich überlappende, über Reiter zugängliche Seiten auf. Die ersten beiden Seiten „RAM-Bank A“ und „RAM-Bank B“ zeigen die Daten in den beiden RAM-Bänken. Die möglichen Darstellungsweisen sind die gleichen wie im Pufferbereich. Zweierlei muß hier beachtet werden:

1. Die angezeigten Daten sind nur so aktuell, wie dies vom Benutzer gewünscht wird. Beim Programmstart und beim Laden aus dem Puffer werden die Seiten automatisch aktualisiert, danach muß dies der Benutzer selbst machen. Nach der Konfiguration des FPGAs werden – durch einen Fehler in der H.O.T.-Schnittstelle, siehe Kapitel A.1.1 – die Daten zudem falsch aktualisiert. Möchte man den RAM-Inhalt nach der Konfiguration lesen, so muß das Programm neu gestartet werden[2].

2. Beim manuellen Aktualisieren und nach dem Programmstart weiß *PsalM*-Desk nicht, wie groß der „interessierende“ Speicherbereich ist. Deshalb werden in diesen Fällen die Daten der Standardgröße (die ersten 256kB) angezeigt.

Die dritte Seite „IO“ des PCI-Board-Bereiches ermöglicht den Datenaustausch mit dem FPGA. Sie gliedert sich in einen Eingabebereich (oben) und einen Ausgabebereich (unten). Im Eingabebereich kann rechts die Zieladresse (also das Zielregister auf dem FPGA[3]) ausgewählt werden. Hier steht neben den Standardports 1 bis 4 (Adressen 0, 4, 8, 12) auch ein freies Eingabefeld zur Verfügung. Ist der Zielport gewählt, kann das zu sendende Datum in der Mitte eingegeben werden. Besteht das Datum nur aus Ziffern, so wird die Zahl als Dezimalzahl interpretiert. Sind auch Buchstaben (A bis F) enthalten, so wird eine Eingabe im Hexadezimalformat angenommen. Möchte man eine nur aus Dezimalziffern bestehende Hexadezimalzahl eingeben, so muß man der Zahl „0x“ oder „$“ voranstellen.

Die Eingabe darf in beiden Fällen (dezimal/hexadezimal) nicht größer als $2^{32}$ sein. Zum Senden des Datums kann entweder der Button „Senden“ rechts neben dem Eingabefeld oder ein einfaches „RETURN“ im Eingabefeld verwendet werden. Alle erfolgten Ausgaben werden in dem Fenster über dem Eingabefeld zusammen mit der Adresse aufgelistet.

Die untere Hälfte der Datenaustauschseite zeigt vom FPGA gesendete Daten. Genau wie beim Senden von Daten zum FPGA stehen rechts die verschiedenen Ports/Register

[2] Es gibt einen anscheinend meist funktionierenden Umweg, indem der Board-Takt vor dem Aktualisieren aus- und danach wieder eingeschaltet wird. Dieses Vorgehen wurde jedoch nicht mit *PsalM*-Desk getestet und ein Erfolg kann nicht garantiert werden.

[3] In der *V.C.C.*-Dokumentation ist immer von „Ports“ und „Registern“ die Rede. Da die Eingaben vom Host asynchron kommen, empfiehlt es sich auch tatsächlich, Register für jeden benutzten Port zu verwenden. Letztendlich bedeutet ein gewählter „Port“ aber lediglich das Setzen einer von 16 Steuerleitungen im FPGA. Wie und ob dieses Setzen interpretiert wird, hängt natürlich vom jeweiligen Design ab.

zur Verfügung. Jetzt können auch mehrere gleichzeitig ausgewählt werden. Je nachdem, welche Adressen gewählt wurden, werden in der Liste links die zugehörigen Spalten mit empfangenen Daten gefüllt oder bleiben leer. Unter der Adressauswahl kann eingestellt werden, ob in der Liste nur neue, geänderte Daten erscheinen oder ob bei jeder Aktualisierung eine neue Zeile angehängt wird, unabhängig davon, ob sich irgendein Eingabedatum geändert hat oder nicht. Vorteil von „Laufend aktualisieren“ ist, daß im Nachhinein der Zeitpunkt einer Änderung festgestellt werden kann. Dafür wird die Liste jedoch auch sehr lang.

Über das Eingabefeld unten kann die Zeit bestimmt werden, die zwischen zwei Portabfragen abgewartet wird. Dieses Verfahren des „Polling“ (siehe Glossar) hat jedoch ein paar Nachteile: Ist die eingegebene Wartezeit zwischen den Portabfragen zu lang, so könnte es (bei schnellen Änderungen) Ausgaben vom FPGA geben, die nicht angezeigt werden. Ist die gewählte Zeit jedoch zu kurz, so könnten auch kurze Änderungen nicht stabiler Zustände gelistet werden. Da aber selbst die kürzeste Zeit (eine Millisekunde) noch sehr lang ist, empfiehlt sich eine kurze Einstellung. Dann sollte allerdings auch „Nur Änderungen anzeigen“ ausgewählt werden, damit die Liste der empfangenen Daten nicht zu lang wird. Benötigt man für ein eigenes Design eine exakte Messung kurzer Zeitabstände oder ein Speichern sich schnell ändernder FPGA-Ausgaben, so muß man dies über eine Hardware-Lösung erreichen[4]. Entsprechende Lösungen waren im Rahmen des Projektseminars nicht nötig oder nicht praktikabel. Für die Geschwindigkeitsmessungen wurde statt dessen ein „Handshaking-Protokoll“ auf Software-Ebene entwickelt. Auf der „IO“-Seite ganz unten finden sich schließlich noch die Schaltflächen zum Starten und Beenden der Portabfragen, zum Speichern und zum Löschen der aktuellen Liste.

Die vierte Seite „Benchmark“ im PCI-Board-Bereich ist speziell für unsere CPUs und für bestimmte Testprogramme gedacht. Ziel ist ein Messen und Vergleichen von Antwortzeiten nach Bearbeitung bestimmter Berechnungen. Es werden – nach dem Laden des Testprogramms und Konfigurieren des FPGAs – über einen längeren Zeitraum Startsignale an den FPGA geschickt (dies kann auch ein RESET sein) und dann die Zeit bis zum Empfang eines bestimmten Datums gemessen. Diese Antwortzeiten werden in der Liste (oben) und einem Chartdiagramm (unten) eingetragen und danach eine erneute Berechnung angestoßen. Da der Datenaustausch genau wie auf der vorherigen Seite beschrieben – der Empfang also insbesondere asynchron per „Polling“ – geschieht, ist die Genauigkeit bei kurzen Testprogrammen zu gering. Bei längeren Testprogrammen kann aber durch das wiederholte Ausführen über eine längere Zeit hinweg eine akzeptable Genauigkeit erreicht werden.

### Die Statuszeile

Die Statuszeile am unteren Fensterrand zeigt das erkannte PCI-Board, die aktuelle Frequenz des programmierbaren Clock-Generators, den aktuellen Pegel der 6502-

[4] Zeitmessungen sind mit einem Binärzähler am 100MHz Takt möglich. Ebenso kann der 100MHz Takt dem Triggern zusätzlicher Register zum Speichern von Design-Ausgaben dienen. Vor allem letzteres kostet jedoch sehr viele CLBs, so daß gegebenenfalls Design-spezifische Lösungen entwickelt werden müssen. Eine Möglichkeit hierzu ist das Erzeugen von PCI-Interrupts, das mit der von uns verwendeten H.O.T.-Schnittstelle jedoch nur unzuverlässig funktioniert. Bei manchen Designs kann die Schaltung auf dem FPGA auch auf die Port-Abfragen vom Host „warten“.

Eingangsleitungen `RESET`, `NMI` und `IRQ` und den momentanen Status des Single-Step. Ganz rechts befindet sich noch eine Schaltfläche zum Aktualisieren dieser Anzeigen. Eine automatische Aktualisierung findet nach dem Senden von Daten zum FPGA (nicht zum RAM!), nach dem Ändern von einer der angezeigten Funktionen per Menü und beim Programmstart statt. Die Pegel der drei Leitungen `RESET`, `IRQ` und `NMI` können entweder über das Menü „Board“ oder durch direktes Klicken mit der linken Maustaste auf die jeweilige Anzeige invertiert werden. Der Status vom Single-Step kann nur durch direktes Klicken auf die SingleStep-Fläche geändert werden. Bei jedem Mausklick wird der Pegel des SingleStep-Clock einmal invertiert – für einen vollen Clock-Zyklus muß also zweimal geklickt werden. Ist SingleStep noch nicht aktiviert, so geschieht dies zuvor.

Im Simulationsmodus (ohne Zugriff auf eine PCI-Karte) sind die Anzeigen der Statuszeile ohne Bedeutung und zeigen konstant „(Sim)“.

### A.1.4 Die *PsalM*-Desk Menüstruktur

Wie in Bild A.1 zu sehen ist, gliedert sich das Hauptmenü am oberen Bildrand in die Menüpunkte „Datei“, „Puffer“, „Board“ und „Info“.

#### Dateimenü

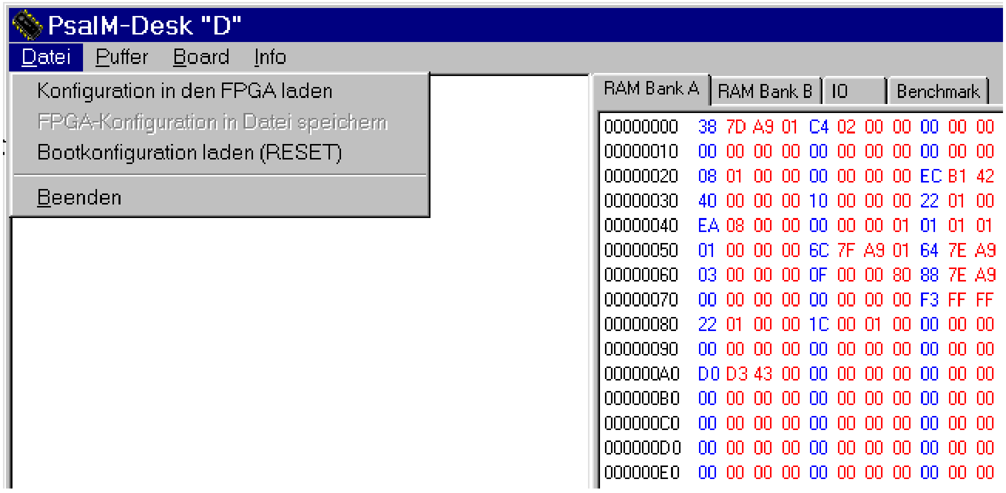



Abbildung A.2: Das Dateimenü

Das Dateimenü besitzt die Menüpunkte „Konfiguration in den FPGA laden“, „FPGA-Konfiguration in Datei speichern“, „Bootkonfiguration laden (RESET)“ und „Beenden“. Nach Wahl des ersten Punktes erscheint ein Dateiauswahldialog, über den eine HOT-Datei ausgewählt werden kann. Diese Datei wird dann in den FPGA geladen.

Der zweite Menüpunkt ist für den umgekehrten Weg gedacht. Jedoch war es im Laufe des Projektseminars nie nötig, die aktuelle Konfiguration des FPGA wieder in eine Datei zu speichern, weswegen dieser Punkt zur Zeit noch nicht auswählbar ist.

Der dritte Menüpunkt erlaubt ein Zurücksetzen des Boards in die Grundkonfiguration. Nach einem solchen RESET befindet sich an einem IO-Port ein einfaches Register und an einem anderen ein mit der User-Clock getakteter Zähler. Die RAM-Bänke und der Cache werden nicht gelöscht.

Der letzte Menüpunkt schließt schließlich die Kommunikation mit dem FPGA-Board und beendet *PsalM*-Desk.

### Puffermenü

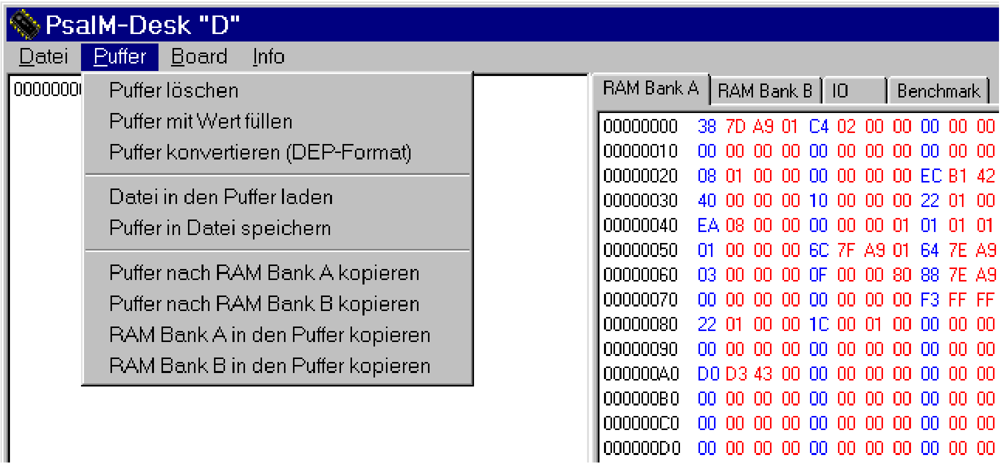



Abbildung A.3: Das Puffermenü

Das zweite Menü „Puffer“ ist in drei Bereiche aufgeteilt. Die drei Menüpunkte des ersten Bereichs dienen dem Verändern, Initialisieren und Normieren/Formatieren des Puffers. Als Standard-Speicherformat findet hierbei das „DEP-Format“ Verwendung (siehe Glossar).

Der zweite Bereich deckt den Datenaustausch vom/zum Puffer ab. Bei beiden Menüpunkten erscheinen die üblichen Datei-/Pfadauswahldialoge. Über die Menüpunkte des dritten Bereichs erfolgt der Datenaustausch mit den RAM-Bänken.

### Boardmenü

Der dritte Punkt des Hauptmenüs ist in vier Bereiche gegliedert und erlaubt Einstellungen auf dem PCI-Board. Etwas heraus fällt der erste Bereich. Die beiden Menüpunkte normieren/formatieren die RAM-Bänke in der gleichen Weise, wie dies mit dem Puffer über den zweiten Hauptmenüpunkt möglich ist. Vor dem Konfigurieren des FPGA mit einer unserer CPUs muß die RAM-Bank A unbedingt in das DEP-Format gebracht werden. Andernfalls ist das Verhalten der CPU undeterministisch.

Der dritte Menüpunkt erlaubt das Ändern des einstellbaren User-Clock. Erlaubt sind Eingaben von 0,5 bis 100 (in MHz).

Über den dritten Bereich können die User-Clock-Einstellungen der H.O.T.-Schnittstelle vorgenommen werden. Siehe auch [VCCSOFTAPI], Seite 39. Aufgrund der Funk-

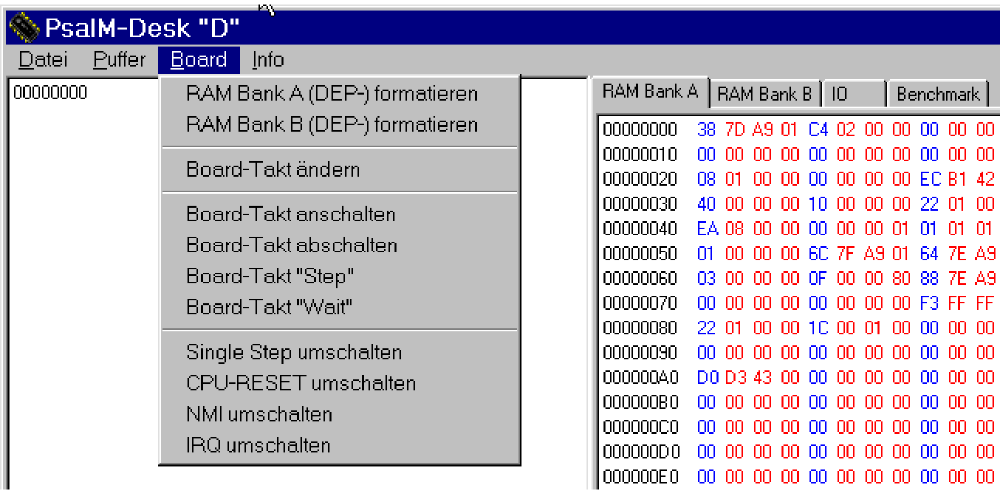



Abbildung A.4: Das Boardmenü

tionen des von uns verwendeten PCI-Wrappers haben wir von diesen Funktionen jedoch keinen Gebrauch gemacht.

Die letzten vier Punkte schalten schließlich die für unsere CPUs wichtigen Signale (CPU-)`RESET`, `NMI`, `IRQ` und die SingleStep-Betriebsart um. Siehe hierzu auch die Beschreibung der Statuszeile.

### Info

Über den einzigen Menüpunkt „Über *PsalM*-Desk „D““ des Info-Menüs kann ein Info-Dialog angezeigt werden. Zu sehen ist ein einzelnes Fenster mit der Versionsnummer und den Mitwirkenden des *PsalM*.


Abbildung A.5: Der Info-Dialog

Zu bemerken sei hier noch, daß die Reihenfolge der *PsalM*-Entwickler zufällig ist und bei jedem Neuzeichnen des Fensters geändert wird.

### A.1.5 Typische Anwendungsfälle

Nach dem Aufruf von *PsalM*-Desk wird zunächst gefragt, ob die Zugriffe auf eine PCI-Karte nur simuliert oder tatsächlich durchgeführt werden sollen. Die Voreinstellung „YES“/„JA“ muß bei PCs übernommen werden, in denen keine H.O.T.-PCI-Karte eingebaut ist. In diesem Fall haben die meisten Funktionen von *PsalM*-Desk natürlich keinen Sinn – diese werden meist mit „Nur Simulation“ bestätigt. Manche Programmteile können aber uneingeschränkt genutzt werden und zwar auf allen PCs (siehe Einleitung zu *PsalM*-Desk).

#### Vorbereitungen

Nach dem Programmstart sollen in der Regel erst eine oder beide RAM-Bänke beschrieben werden, um der Logik definierte Startzustände (oder, wie in diesem Projektseminar, ein abzuarbeitendes Programm) zu geben. Hierzu werden zunächst die gewünschten Daten in den Puffer geladen. Dies kann entweder über das Kontextmenü des Puffers geschehen (rechte Maustaste über dem Puffer drücken - „Datei in den Puffer laden“ wählen), oder über das Menü „Puffer“ - „Datei in den Puffer laden“. Gegebenenfalls kann (wieder über Kontextmenü oder Hauptmenü) der Pufferinhalt in das DEP-Format konvertiert werden.

Danach müssen die Daten aus dem Puffer in die jeweilige RAM-Bank kopiert werden. Am einfachsten geht dies über das Kontextmenü der jeweiligen RAM-Bank (rechte Maustaste über der gewählten Bank) und dem Menüpunkt „Mit Pufferinhalt überschreiben“. Alternativ gibt es im Menü „Puffer“ die Auswahl „Puffer nach RAM Bank kopieren“.

Als weitere Vorbereitung vor dem Konfigurieren des FPGA muß eventuell die Taktfrequenz der programmierbaren „User Clock“ eingestellt werden. Diese Einstellung ist über das Menü „Board“ - „Board-Takt ändern“ möglich. Zu beachten ist der angegebene Bereich der Taktfrequenz (0,5 bis 100MHz).

Ebenfalls vor dem Konfigurieren sollten die Einstellungen zum Datenempfang vorgenommen werden. Hierzu muß im PCI-Board-Bereich (rechts) auf die dritte Seite „IO“ gewechselt werden. Im unteren Teil müssen die zu verfolgenden Adressen über „Ausgabeauswahl“ eingestellt werden. Desweiteren der Abfragemodus und die Zeit zwischen zwei Portabfragen.

#### Konfigurieren des FPGA

Nun kann der FPGA konfiguriert werden, was über das Menü „Datei“ - „Konfiguration in den FPGA laden“ möglich ist. Hierauf erscheint ein Dateiauswahldialog, über den als Voreinstellung „.HOT“-Dateien ausgewählt werden können. Über „Dateityp“ - „Alle Dateien“ können auch andere Formate geladen werden. Möglich sollen auch „.BIT“-Dateien sein, empfohlen wird aber das „.HOT“-Format.

Die Umwandlung von „.BIT“ nach „.HOT“ ist mit dem *V.C.C.*-Programm „make-HOT“, beziehungsweise „winMakeHOT“ möglich (siehe auch [VCCSOFTAPI]).

**Nach der Konfiguration**

Ist der FPGA konfiguriert, sollten manche Signale – je nach Design – auf „Ruhepegel“ gesetzt werden. RESET-Leitungen sollten schon im Design per Konfigurations-RESET gesetzt sein, aber beispielsweise ein SingleStep oder bestimmte IO-Ports müssen erst nach der Konfiguration zurückgesetzt werden.

Nun sollte auch der Empfang von Daten des FPGA gestartet werden und das Debuggen oder der Programmdurchlauf kann beginnen.

Das Auslesen aus dem RAM kann wieder über die Kontextmenüs der RAM-Bänke oder das Menü „Puffer“ erfolgen. Hier gibt es jedoch die schon in Kapitel A.1.3 geschilderten Probleme.

## A.2 *PsalM*-VR

*PsalM*-VR ist ein kleines Windows-Programm, daß die in den meisten *VHDL*-Editoren fehlende Funktion des mehrfachen Ein- und Auskommentieren ermöglicht. Je nach verwendetem Editor können die zusätzlichen Funktionen sehr schnell und fast transparent zum Beispiel aus dem System-Tray benutzt werden.

## A.3 Espresso

Bei Espresso handelt es sich um ein Programm zur Vereinfachung logischer Funktionen mit Hilfe von Heuristiken, das Anfang der 80er Jahre an der Berkeley Universität entwickelt und im Rahmen der sogenannten Octtools (siehe [OCT1]) veröffentlicht wurde.

Bei der Suche nach einer besten Vereinfachung logischer Funktionen handelt es sich um ein NP-vollständiges Problem, das mit Verfahren wie z.B. Quine-McCluskey nur mit sehr hohem Aufwand zu lösen ist. Espresso bedient sich daher heuristischer Verfahren, um trotz der NP-Vollständigkeit des Problems noch zu einer näherungsweisen Lösung zu gelangen; dies hat zur Folge, daß Espresso im allgemeinen eine Vereinfachung liefert, die wesentlich kürzer als die Eingabe ist, aber keine Aussage darüber machen kann, ob nicht noch bessere Lösungen für das Problem existieren. Dabei wendet Espresso diesen heuristischen Algorithmus iterativ an, solange die Ausgabe eine bessere Lösung darstellt als im vorhergehenden Durchlauf, und erzielt auf diesem Wege Ergebnisse, die den jeweils „besten“ Lösungen ziemlich nahe kommen. Sowohl Ein- als auch Ausgabe von Espresso ist jeweils eine Liste (1..n) von die Tabelle abdeckenden Blöcken „cubes“, rechteckigen Ausschnitten der Wahrheitstabelle, die aber dennoch in der Summe die Informationen der gesamten Wahrheitstabelle enthalten. Im folgenden eine kurze Beschreibung der einzelnen Schritte, die von Espresso durchgeführt werden.

1. EXPAND: Expresso expandiert die in der Tabelle plazierten Blöcke so weit wie möglich; hierbei auftretende Überschneidungen zwischen den einzelnen Blöcken sind explizit erwünscht.

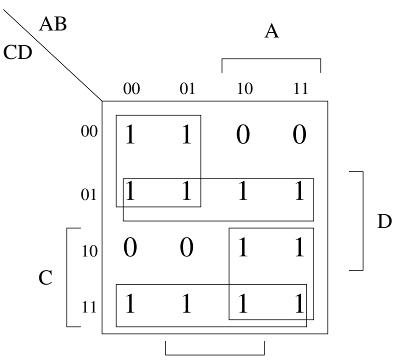
2. IRREDUNDANT COVER: Eine neue Liste von Blöcken wird aus der expandierten Tabelle extrahiert, die dennoch die kompletten Informationen der gesamten Tabelle enthält. Diese Liste stellt im allgemeinen schon eine recht gute Lösung dar.

3. REDUCE: Rückgängigmachen der Expansion der Blöcke.

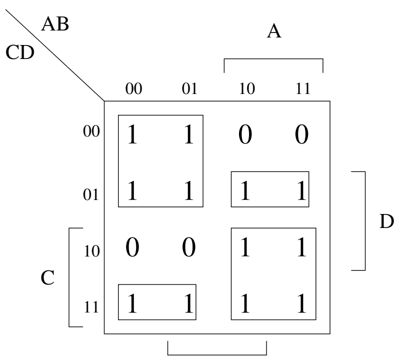
4. Die jetzt vorhandene Lösung ist typischerweise noch bei weitem nicht optimal, daher werden die Schritte REDUCE, EXPAND und IRREDUNDANT COVER in einer Art und Weise wiederholt, daß andere Blöcke verwendet werden, um zu einer besseren Lösung zu gelangen. Diese Schritte werden in unterschiedlicher Reihenfolge wiederholt, solange das Ergebnis eines Durchlaufes eine Verbesserung in Bezug auf den vorhergehenden Durchlauf darstellt.

5. Espresso benutzt desweiteren noch einige andere Methoden zur Verbesserung des Ergebnisses oder zur Beschleunigung der Berechnung. Diese beinhalten unter anderem eine frühe Identifikation und Extraktion essentieller Bestandteile der Wahrheitstabelle, so daß diese bei Durchführung des vierten Schritts nicht mehr ins Gewicht fallen, oder auch die Verwendung des Funktionskomplementes zur effizienten Überprüfung innerhalb des EXPAND-Schritts, in wie weit eine Überdeckung von Blöcken innerhalb der Wahrheitstabelle vorhanden ist.

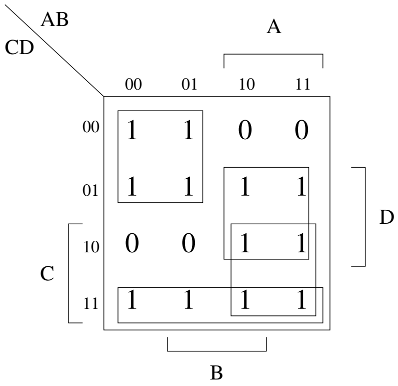
Die nachfolgenden Graphiken zeigen die Position und Anzahl der Blöcke nach Durchführung der verschiedenen Schritte von Espresso (von [OCT1]).



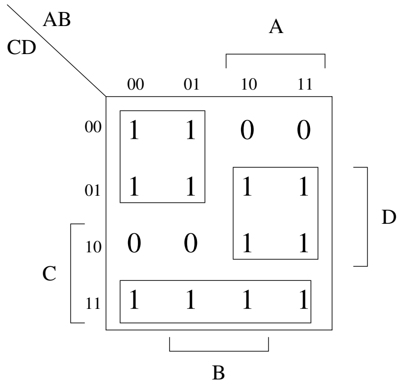
Anfangssituation



Zustand nach "REDUCE"



Zustand nach "EXPAND"



Zustand nach "IRREDUNDANT COVER"

## A.4 depasm

Bei `depasm` handelt es sich um einen Assembler für die als ersten Test implementierte DEP-CPU. Das Programm `depasm` benötigt drei Parameter. Zum einen die Datei mit dem Quellcode, zum anderen den Dateinamen zweier Ausgabedateien. Das Format der ersten Ausgabedatei ist dabei der Binärcode, so wie ihn die DEP-CPU versteht, in der zweiten Ausgabedatei ist dieser Binärcode in unserem properitären Format für das PSalM-Desk verpackt.

Die Eingabedatei muß in folgendem Format vorliegen:

```
Assemblerprogramm = { [Labelname ":" Space]
                      [Befehl] {Space}
                      [";" Kommentar] }.

Space             = (" ") {Space}.

Zahl              = ("0" | "1" | ... | "9").

Hexzahl           = ("0" | "1" | ... | "F")
                    ("0" | "1" | ... | "F").

Buchstabe         = ("A" | ... | "Z").

Labelname         = ( ( Buchstabe | Zahl )
                    { ( Buchstabe | Zahl ) }.

Label             = ( Labelname | Hexzahl ).

Kommentar         = {Buchstabe}.

Adresse           = Hexzahl.
```

```
Befehl              = ("BCS"  Space Label |
                       "BEQ"  Space Label |
                       "IN"   Space Adresse |
                       "OUT"  Space Adresse |
                       "STA"  Space Adresse |
                       "LDA"  Space Adresse |
                       "AND"  Space Adresse |
                       "OR"   Space Adresse |
                       "XOR"  Space Adresse |
                       "ADD"  Space Adresse |
                       "ADC"  Space Adresse |
                       "SBC"  Space Adresse |
                       "LDAI" Space Hexzahl |
                       "ANDI" Space Hexzahl |
                       "ORI"  Space Hexzahl |
                       "XORI" Space Hexzahl |
                       "ADDI" Space Hexzahl |
                       "ADCI" Space Hexzahl |
                       "SBCI" Space Hexzahl).
```

Dabei ist zu beachten, daß die ersten beiden Zeichen eines Labels keine Hexadezimalzahl ergeben dürfen. Ein Label ist zum Beispiel dann gültig, wenn es mit einem `G` beginnt. Ein Label „FEHLER“ führt hingegen zu einem Fehler.

Auch müssen die Labels, die hinter den Sprungbefehlen `BEQ` und `BCS` verwendet werden, an irgendeiner Stelle im Quelltext definiert werden.

Der DEP-Assembler ist ein einfacher 2-Pass Assembler. Im ersten Durchgang werden die Labels im Quellcode gesucht und ihre Adresse relativ zum Programmbeginn `0x00` im Speicher berechnet, im zweiten Durchgang werden die Opcodes der gefundenen Befehle nach der Tabelle im Kapitel 3 als Bytes in die Ausgabedateien geschrieben. Hinter jedem Opcode wird das Label, die Adresse oder die Hexzahl in die Ausgabedateien geschrieben.

Die Binärdateien werden später als Abbild des ROM-Speichers an die CPU übergeben. Diese sieht dann abwechselnd eine Instruktion, die sie in das **`IWR`** laden, sowie einen Operanden, mit dem sie das **`OPR`** versorgen kann.

## A.5 Der 6502-Assembler

Um Testprogramme für den 6502 zu schreiben, benutzten wir `as6502`[5], einen einfachen Assembler zum Übersetzen von 6502-Assemblerprogrammen in ein ASCII-Format – laut Versionsanzeige mit `-v` in der Version 4.1b vom 22.11.1984.

Im Kommentar ist zu lesen:

Assembler for the MOS Technology 650X series of microprocessors
Written by J. H. Van Ornum (201) 949-1781

[5] `http://www.geocities.com/oneelkruns/6502asm.html`

AT&T Bell Laboratories
Holmdel, NJ

Anpassungen des Programms waren eigentlich nicht nötig, `gcc` Version 2.95 hatte keine Schwierigkeiten, den alten Programmcode zu übersetzen. Dennoch haben wir eine Hilfeseite hinzugefügt und den Source so umgeschrieben, daß er keine Compiler-Warnungen mehr provoziert.

Der Assembler hat keine Möglichkeit, eine Binärdatei zu erzeugen, darum entstand ein weiteres Programm `asm2bin`, welches die Ausgabe des Assemblers, die er mit der Option `-o` erzeugt, in eine 64kB große Datei konvertiert. Diese Datei entspricht dem Speicherinhalt, wie er dann dem 6502 bei seinem „Einschalten“ vorgegeben wird und spielt damit auch eine Rolle bei der Initialisierung des Prozessors, denn die erste Aktion eines 6502 ist das Laden des Reset-Vektors – aus diesem Grund setzt `asm2bin` auch die NMI-, Reset- und IRQ-Vektoren.

Zur einfacheren Verwendung in *PsalM*-Desk kam noch die Option `-d` hinzu, die den Speicherinhalt im DEP-Format in einer 256kB großen Datei ablegt.

## A.6 drconv/drcalc

Beim Hantieren mit Dual-Rail-codierten Zahlen sind zusätzliche Tools praktisch, die das Umrechnen von verschiedenen Zahlensystemen in und aus Dual-Rail unterstützen. Das Programm `drconv` entstand, um Zahlen aus dem hexadezimalen, dezimalen und binären System in Dual-Rail-codierte Zahlen in hexadezimaler, dezimaler und binärer Darstellung zu überführen (und umgekehrt).

Ein weiteres Tool, welches für denselben Zweck geschrieben wurde, ist `drcalc`. Es ist weniger flexibel als `drconv`, allerdings bietet es die Möglichkeit, von einem vorgegebenen Dual-Rail-Zustand ausgehend einen Folgezustand zu berechnen.

Es gibt zwei Möglichkeiten das Programm zu benutzen. Ausgehend von einem Dual-Railzustand kann man einmal berechnen, welches der nachfolgende Zustand ist, wenn man eine bestimmte Zahl codieren möchte, oder andersherum, welche Zahl codiert wurde.

Um einen Folgezustand zu berechnen, gibt man dem Programm den Ausgangszustand als 16-Bit-Wort, sowie ein 8-Bit-Wort, welches codiert werden soll:

```
> ./drcalc AFFE 23

1010111111111110
1111011010100100
F6A4
```

Die hexadezimale Zahl am Ende der Ausgabe ist der gewünschte Folgezustand.

Andersherum gibt man dem Programm zwei Zustände als 16-Bit-Worte ein:

```
> ./drcalc AFFE CAA7

1010111111111110
```

```
1100101010100111
0 1 0 0 0 0 1 0
 66 (42)
```

Die beiden Zahlen in der letzten Zeile geben die im DualRail-Format eingegebene Zahl einmal als Dezimal- und (in Klammern) als Hexadezimalwert an.

Die beiden Tools `drconv` und `drcalc` entstanden unabhängig voneinander von zwei verschiedenen Autoren aus der Not, „normale“ Zahlen in Dual-Rail-Darstellung überführen zu müssen, allerdings bei der Verfolgung unterschiedlicher Dual-Rail-Ansätze: `drconv` entstand bei der Entwicklung des pegelgesteuerten Dual-Rails (siehe Abschnitt 4.2.2) und `drcalc` bei der *PsalM*-Dual-Rail-Entwicklung (Abschnitt 4.3).

## A.7 Tableshrinker

Die Aufgabe des sogenannten Tableshrinkers besteht darin, die große und unhandliche Decodertabelle des asynchronen 6502 zu vereinfachen, indem es zuverlässig identische Spalten findet, welche anschließend zu einer zusammengefügt werden. Ebenso können einfache Abhängigkeiten zwischen den Spalten erkannt und zu einer Optimierung der Schaltfunktion genutzt werden. Im Unterschied zu Verfahren wie Quine-McCluskey bzw. darauf aufbauenden Tools wie Espresso wird hier also nicht eine Vereinfachung und Zusammenfassung der *Zeilen angestrebt, sondern das Programm arbeitet vielmehr auf den* Spalten der Decodertabelle.

Dabei muß gewährleistet sein, daß

- Die Tabelle im Textformat vorliegt
- Nur die erste Zeile Text enthält, der Rest sollte aus 0,1 oder - bestehen
- Einzelne Spalten durch Tabulatoren (Char `0x09`) getrennt sind
- Einzelne Zeilen durch Linefeeds (Char `0x0A`)
- Niemals zwei Tabulatoren oder Linefeeds direkt hintereinander auftreten

Zusätzlich sollten noch die Dimensionen der Tabelle bekannt sein, man muß genau wissen, wieviele Zeilen sie enthält, wieviele Spalten, und wieviele davon Eingabespalten sind. Diese Dimensionen müssen im Quellcode als `MAXZEILEN`, `MAXSPALTEN` und `INPUTSPALTEN` definiert werden.

Gestartet wird das Programm, nach einem Compile, mit dem Dateinamen der Tabelle als Parameter. Zunächst gibt es die Überschriften der einzelnen Spalten aus, die es in der ersten Zeile findet, und numeriert sie durch. Danach benutzt es nur diese Nummern, um identische Spalten anzuzeigen.

Als Folge davon wird die Decodertabelle, und damit auch der Decoder an sich, kleiner, was bei der Implementierung Chipfläche einspart.

## A.8 StarOffice

Ein weiteres Tool, das uns große Arbeitsersparnis brachte, war das Programm StarOffice Version 5.2a der *Sun Microsystems, Inc.* Insbesondere das Teilprogramm „StarCalc“, welches früher auch getrennt vertrieben wurde, kann unter anderem wesentlich zur Übersicht beitragen.

Eingesetzt wurde die Tabellenkalkulation, um einen Überblick über die recht große Decodertabelle des 6502 zu erhalten. Bei der DEP-CPU erübrigt es sich, solche Programme einzusetzen, da der Decoder (siehe Tabelle C.1) ohne Probleme übersichtlich gestaltet werden kann. Bei größeren Projekten wie dem 6502, dessen synchrone Decodertabelle schon mit 28.000 Zellen droht, unübersichtlich zu werden, hilft solch ein Programm sehr. Es bietet die Möglichkeit, schnell in der Tabelle zu navigieren, Einträge zu ändern und vor allem die Tabelle schnell nach bestimmten Kriterien umzusortieren. Hierdurch kann wesentlich Zeit eingespart werden, da ähnliche Befehle (zum Beispiel alle arithmetischen Berechnungen) sich meistens in vielen Schritten ähneln. Selbst die unterschiedlichen Zeilen können bis auf wenige Zellen gleich sein, da sich bei arithmetischen Berechnungen z.B. nur der ALU-Op unterscheidet.

Ebenfalls praktisch sind die weiteren Funktionen, die StarOffice bietet. So kann man schnell eine Spalte von binär auf dezimal codiert umstellen.

Die Wahl fiel auf StarOffice, da dies das einzige uns zur Verfügung stehende Produkt war und ohne Lizenzkosten genutzt werden kann. Ähnliche Produkte, wie FreeOffice, AbiWord, KOffice etc. bringen sicher die gleichen Erleichterungen.

## A.9 Testprogramme

Es wurden für die kleine DEP-CPU und auch den 6502 Testprogramme entwickelt, um die Prozessoren auf ihre Funktionstüchtigkeit zu testen.

Beide Programme testen Befehl für Befehl, inkrementieren einen Zähler für jeden Befehl, brechen bei einem Fehler die Ausführung ab und geben den Zähler über den Befehl „OUT“ aus. Aufgrund des beschränkten Adreßraums der DEP-CPU wurde das Testprogramm für diesen Prozessor in zwei Teile aufgespalten, die nacheinander ausgeführt, auch alle Befehle testen.

# Anhang B

# Alternative, nicht realisierte Dual-Rail-Ansätze

Auf der Suche nach möglichst schnellen und kleinen Dual-Rail-Realisierungen gab es noch ein paar Ansätze, die jedoch entweder aus zeitlichen Gründen nicht weiter verfolgt werden konnten oder zu nicht lösbaren Problemen führten.

## B.1 Versuch: Halb flankengesteuert - Halb pegelgesteuert

Eine Mischung der beiden realisierten Techniken „flankengesteuert“ und „pegelgesteuert“ ist dieser Ansatz. Hier wird davon ausgegangen, daß beim Reset alle Leitungen `low` sind. Gibt es nun einen Flankenwechsel auf den Eingangsleitungen, so springen diese auf `high`. Um gegebenenfalls eine Flanke am Ausgang zu erzeugen, ist genauso ein Wechsel nach `high` nötig. Es läßt sich also zunächst dieselbe Logik wie beim pegelgesteuerten Dual-Rail (siehe Abschnitt 4.2.2) einsetzen. Die Idee ist nun, nach dem erfolgreichen Ausgeben einer Flanke, die Eingänge, die `high` wurden, zu invertieren und die durch diesen Schritt zurückgesetzte Ausgangsleitung ebenfalls zu invertieren, so daß die interne Logik wieder „statisch“ arbeitet, die Ausgänge jedoch korrekt sind. Dies könnte theoretisch ohne ein speicherndes Element möglich sein, eine Lösung wurde allerdings noch nicht gefunden. Nimmt man den Aufwand von Flip-Flops (zum Beispiel am Ausgang) in Kauf, so findet sich sogar recht schnell eine Lösung, allerdings ist diese dann ähnlich aufwendig wie die oben beschriebenen.

## B.2 Versuch: *VHDL* machen lassen!

Ein weiterer Ansatz war, das gewünschte Verhalten einfach (!) in *VHDL* zu beschreiben, was recht kompakt möglich ist. Bei der resultierenden Logik könnte man sich auf die Optimierungen des Compilers verlassen und ihn möglichst kleine Gatter erzeugen lassen.

Verwendet man alle Möglichkeiten, die *VHDL* einem theoretisch bietet, kommt man zum Beispiel zu folgendem Ansatz (Beispiel AND-Gatter):

```
entity DR_AND is
  port(
    RESET  : in std_logic;
    Alow   : in std_logic;
    Ahigh  : in std_logic;
    Blow   : in std_logic;
    Bhigh  : in std_logic;
    Qlow   : out std_logic;
    Qhigh  : out std_logic;
  );
end DR_AND;

architecture ARCH of DR_AND is
signal intQlow, intQhigh : std_logic;
signal gotAhigh, gotBhigh : boolean;
begin
  Qlow  <= intQlow;
  Qhigh <= intQhigh;

  process(RESET, Alow, Blow) is
  begin
    if RESET='1' then
      intQlow <= '0';
    elsif Alow'event or Blow'event then
      intQlow <= not intQlow;
    end if;
  end process;

  process(RESET, intQlow, intQhigh, Ahigh) is
  begin
    if RESET='1' or intQhigh'event or intQlow'event then
      gotAhigh <= '0';
    elsif Ahigh'event then
      gotAhigh <= '1';
    end if;
  end process;

  process(RESET, intQlow, intQhigh, Bhigh) is
  begin
    if RESET='1' or intQhigh'event or intQlow'event then
      gotBhigh <= '0';
    elsif Bhigh'event then
      gotBhigh <= '1';
    end if;
  end process;

  process(RESET, gotAhigh, gotBhigh) is
```

```
  begin
    if RESET='1' then
      intQhigh <= '0';
    elsif gotAhigh'event or gotBhigh'event then
      if (gotAhigh='1') and (gotBhigh='1') then
        intQhigh <= not intQhigh;
      end if;
    end if;
  end process;
end ARCH;
```

Jedoch ist kein *VHDL*-Compiler bekannt, bei dem alle hier verwendeten Konstrukte erlaubt sind. So verlangt `event` stets die Angabe einer bestimmten Flanke (auf- oder absteigend).

Auch wenn der Quelltext so verändert würde, daß ein *VHDL*-Compiler ihn akzeptieren würde, taucht ein weiteres Problem auf: Uns ist kein FPGA bekannt, das Flip-Flops mit zwei (oder mehr) Clock-Eingängen zuläßt. Und bei den uns zur Verfügung stehenden FPGAs kommen weitere Einschränkungen hinzu, da nicht gleichzeitig asynchrones Set und Reset oder gleichzeitig synchrones Set und Reset der Flip-Flops möglich ist.

So zeigte sich leider recht schnell, daß in unserer Entwicklungsumgebung mit den von uns genutzten *VHDL*-Bibliotheken und den von uns verwendeten FPGAs komplexere, aber kurze Design-Beschreibungen unmöglich sind. Damit wurde die Beschreibung eines flankengesteuerten Dual-Rail-AND-Gatter viel zu detailliert und umfangreich, um die Vorteile einer einfachen Designeingabe oder einer effektiven Optimierung spüren zu können.

## B.3 Versuch: Das Ideal: In Flanken denken

Wenn man sich das flankengesteuerte Dual-Rail-Gatter ansieht, so erkennt man eine traditionelle Logik im Innern und eine Konvertierung von/nach flankengesteuert außen. Diese Umsetzung wurde gewählt, weil wir es gewohnt sind (und es uns leicht fällt) in Logik*zuständen* zu denken. Der Idealfall wäre nun aber dann gegeben, wenn wir von vornerein *in Flanken denken* und Gatter - wie auch komplexe Schaltungen - ohne Umwandlungen designen würden. Wir modellieren also (zunächst mathematisch) wie folgt: Eine Funktion $f$ (Flanke) gibt an, daß auf ihrem Parameter ein Flankenwechsel stattfindet/stattzufinden hat. Für die Modellierung eines flankengesteuerten Dual-Rail-AND sind nun folgende Gleichungen nötig/gegeben: (`0`-Leitung, deren Pegelwechsel `0` symbolisiert, `1`-Leitung, deren Pegelwechsel `1` symbolisiert)

$$\begin{aligned} f(Q_1) &= f(A_1) \wedge f(B_1) \\ f(Q_0) &= f(A_0) \vee f(B_0) \end{aligned}$$

Einen vielversprechenden Ansatz stellt das XOR-Gatter da, daß einem „in Flanken denkendem“ OR-Gatter entspricht. Ausnahme wäre ein Wechsel auf beiden Eingangsleitungen gleichzeitig, was allerdings tatsächlich vernachlässigt werden müßte. So ist die zweite Gleichung – unter dieser Prämisse – einfach zu realisieren mit der Schaltung:

$$Q_0 \quad = \quad A_0 \ \dot{\vee} \ B_0$$

Es fehlt dann nur noch das Flanken-AND-Gatter für die erste Gleichung, doch genau da liegt das Problem. Hier benötigt man wieder speichernde Elemente, die in Form von Flip-Flops das Design stark aufblähen. Außerdem müßte es ein Einbeziehen der `0`-Leitungen geben, was einem einfacheren Denkmodell aber widersprechen würde.

Trotzdem erscheint das Etablieren einer eigenen Denk- und/oder Notationsweise für diese Art der Logik intuitiv als möglich und ist vielleicht sogar nötig. Ein weiteres Verfolgen dieses Ansatzes würde jedoch den Rahmen dieses Projektes sprengen und muß in einer eigenen Arbeit erfolgen.

# Anhang C

# Blockschaltbilder und Decodertabellen

**Decoder ein- und ausgänge des asynchronen 6502-Prozessors**

| **Eingänge** | **Beschreibung** |
|---|---|
| IW7 | IWR, Bit 7 |
| IW6 | IWR, Bit 6 |
| IW5 | IWR, Bit 5 |
| IW4 | IWR, Bit 4 |
| IW3 | IWR, Bit 3 |
| IW2 | IWR, Bit 2 |
| IW1 | IWR, Bit 1 |
| IW0 | IWR, Bit 0 |
| FROM_PSR7 | PSR, Bit 7 |
| FROM_PSR6 | PSR, Bit 6 |
| FROM_PSR1 | PSR, Bit 1 |
| FROM_PSR0 | PSR, Bit 0 |
| RST | Reset |
| NMI | Nicht-maskierbarer Interrupt |
| IRQ | Interrupt |
| MSR_IN2 | MSR, Bit 2 |
| MSR_IN1 | MSR, Bit 1 |
| MSR_IN0 | MSR, Bit 0 |

| **Ausgänge** | **Beschreibung** |
|---|---|
| MSR_OUT2_0 | MSR-Ausgang, Bit 2, Wert 0 |
| MSR_OUT2_1 | MSR-Ausgang, Bit 2, Wert 1 |
| MSR_OUT1_0 | MSR-Ausgang, Bit 1, Wert 0 |
| MSR_OUT1_1 | MSR-Ausgang, Bit 1, Wert 1 |
| MSR_OUT0_0 | MSR-Ausgang, Bit 0, Wert 0 |
| MSR_OUT0_1 | MSR-Ausgang, Bit 0, Wert 1 |
| MUX_PCL_0 | Select MUX_PCL, Wert 0 |

| **Ausgänge** | **Beschreibung** |
|---|---|
| MUX_PCL_1 | Select MUX_PCL, Wert 1 |
| MUX_PCL_2 | Select MUX_PCL, Wert 2 |
| MUX_PCL_DC | Select MUX_PCL, keiner der obigen Werte |
| MUX_PCH_1 | Select MUX_PCH, Wert 1 |
| MUX_PCH_DC | Select MUX_PCH, keiner der obigen Werte |
| MUX_INDEX_0 | Select MUX_INDEX, Wert 0 |
| MUX_INDEX_1 | Select MUX_INDEX, Wert 1 |
| MUX_INDEX_2 | Select MUX_INDEX, Wert 2 |
| MUX_INDEX_DC | Select MUX_INDEX, keiner der obigen Werte |
| MUX_ZP_0 | Select MUX_ZP, Wert 0 |
| MUX_ZP_1 | Select MUX_ZP, Wert 1 |
| MUX_ADDRH_0 | Select MUX_ADDRH, Wert 0 |
| MUX_ADDRH_1 | Select MUX_ADDRH, Wert 1 |
| MUX_ADDRH_2 | Select MUX_ADDRH, Wert 2 |
| MUX_ADDRH_3 | Select MUX_ADDRH, Wert 3 |
| MUX_ADDRH_4 | Select MUX_ADDRH, Wert 4 |
| MUX_ADDRL_1 | Select MUX_ADDRL, Wert 1 |
| MUX_DATAIN_0 | Select MUX_DATAIN, Wert 0 |
| MUX_DATAIN_1 | Select MUX_DATAIN, Wert 1 |
| MUX_DATAIN_DC | Select MUX_DATAIN, keiner der obigen Werte |
| CE_ABL_0 | Chip Enable ABL-Register, Wert 0 |
| CE_ABL_1 | Chip Enable ABL-Register, Wert 1 |
| CE_ABH_0 | Chip Enable ABH-Register, Wert 0 |
| CE_ABH_1 | Chip Enable ABH-Register, Wert 1 |
| MUX_ALU_0 | Select MUX_ALU, Wert 0 |
| MUX_ALU_1 | Select MUX_ALU, Wert 1 |
| MUX_ALU_2 | Select MUX_ALU, Wert 2 |
| MUX_ALU_3 | Select MUX_ALU, Wert 3 |
| MUX_ALU_4 | Select MUX_ALU, Wert 4 |
| MUX_ALU_5 | Select MUX_ALU, Wert 5 |
| MUX_ALU_6 | Select MUX_ALU, Wert 6 |
| MUX_ALU_DC | Select MUX_ALU, keiner der obigen Werte |
| CE_TEMPZ_0 | Chip Enable TEMPZ-Register, Wert 0 |
| CE_TEMPZ_1 | Chip Enable TEMPZ-Register, Wert 1 |
| CE_PCL_0 | Chip Enable PCL, Wert 0 |
| CE_PCL_1 | Chip Enable PCL, Wert 1 |
| CE_PCH_0 | Chip Enable PCH, Wert 0 |
| CE_PCH_1 | Chip Enable PCH, Wert 1 |
| C_IN_0 | Carry In, Wert 0 |
| C_IN_1 | Carry IN, Wert 1 |
| CE_ACCR_0 | Chip Enable ACCR, Wert 0 |
| CE_ACCR_1 | Chip Enable ACCR, Wert 1 |
| CE_X_0 | Chip Enable X-Register, Wert 0 |
| CE_X_1 | Chip Enable X-Register, Wert 1 |
| CE_Y_0 | Chip Enable Y-Register, Wert 0 |
| CE_Y_1 | Chip Enable Y-Register, Wert 1 |

| Ausgänge | Beschreibung |
|---|---|
| CE_SP_0 | Chip Enable SP, Wert 0 |
| CE_SP_1 | Chip Enable SP, Wert 1 |
| CE_PSR_0 | Chip Enable PSR, Wert 0 |
| CE_PSR_1 | Chip Enable PSR, Wert 1 |
| CE_TMP_0 | Chip Enable TMP-Register, Wert 0 |
| CE_TMP_1 | Chip Enable TMP-Register, Wert 1 |
| LINKS_NULL_0 | Select MUX_ALU_A, Wert 0 |
| LINKS_NULL_1 | Select MUX_ALU_A, Wert 1 |
| ALU_EN_0 | Chip Enable ALU, Wert 0 |
| ALU_EN_1 | Chip Enable ALU, Wert 1 |
| ALU_OP4_0 | ALU-Opcode, Bit 4, Wert 0 |
| ALU_OP4_1 | ALU-Opcode, Bit 4, Wert 1 |
| ALU_OP3_0 | ALU-Opcode, Bit 3, Wert 0 |
| ALU_OP3_1 | ALU-Opcode, Bit 3, Wert 1 |
| ALU_OP2_0 | ALU-Opcode, Bit 2, Wert 0 |
| ALU_OP2_1 | ALU-Opcode, Bit 2, Wert 1 |
| ALU_OP1_0 | ALU-Opcode, Bit 1, Wert 0 |
| ALU_OP1_1 | ALU-Opcode, Bit 1, Wert 1 |
| ALU_OP0_0 | ALU-Opcode, Bit 0, Wert 0 |
| ALU_OP0_1 | ALU-Opcode, Bit 0, Wert 1 |
| RNOTW_0 | Read/Not Write, Wert 0 |
| RNOTW_1 | Read/Not Write, Wert 1 |
| RNOTW_DC | Read/Not Write, Wert egal |
| MUX_DATA_0 | Select MUX_DATA, Wert 0 |
| MUX_DATA_1 | Select MUX_DATA, Wert 1 |
| MUX_DATA_2 | Select MUX_DATA, Wert 2 |
| MUX_DATA_3 | Select MUX_DATA, Wert 3 |
| MUX_DATA_DC | Select MUX_DATA, keiner der obigen Werte |
| CE_IWR_0 | Chip Enable IWR-Register, Wert 0 |
| CE_IWR_1 | Chip Enable IWR-Register, Wert 1 |
| INTVEC_LSB2_0 | Select INTVEC_LSB, Bit 2, Wert 0 |
| INTVEC_LSB2_1 | Select INTVEC_LSB, Bit 2, Wert 1 |
| INTVEC_LSB2_DC | Select INTVEC_LSB, Bit 2, Wert egal |
| INTVEC_LSB1_0 | Select INTVEC_LSB, Bit 1, Wert 0 |
| INTVEC_LSB1_1 | Select INTVEC_LSB, Bit 1, Wert 1 |
| INTVEC_LSB0_0 | Select INTVEC_LSB, Bit 0, Wert 0 |
| INTVEC_LSB0_1 | Select INTVEC_LSB, Bit 0, Wert 1 |

Tabelle C.3: Ein- und Ausgänge des Decoders, asynchroner 6502

| Inst. | Instruction Word | | | | | | | | CCR | | MSR | MSR | IWR | OPR | Mux | Mux | AccR | CCR | PCR | ¬RAM | ¬ROM | ¬IO | R/¬W | Alu- |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ST | LD | BCS | BEQ | IMM | Al2 | Al1 | Al0 | Z | C | out | in | | | AB | PC | | | | | | | | Op |
| | – | – | – | – | – | – | – | – | – | – | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | – |
| | – | – | – | – | – | – | – | – | – | – | 1 | 2 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | – |
| STA | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | – | – | 2 | 0 | 0 | 0 | 0 | – | 0 | 0 | 0 | 0 | 1 | 1 | 0 | – |
| LDA | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | – | – | 2 | 3 | 0 | 1 | 0 | – | 0 | 0 | 0 | 0 | 1 | 1 | 1 | – |
| | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | – | – | 3 | 0 | 0 | 0 | – | – | 1 | 1 | 0 | 1 | 1 | 1 | 1 | – |
| LDAI | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | – | – | 2 | 0 | 0 | 0 | – | – | 1 | 1 | 0 | 1 | 1 | 1 | 1 | – |
| BCS | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | – | 0 | 2 | 0 | 0 | 0 | – | – | 0 | 0 | 0 | 1 | 1 | 1 | 1 | – |
| | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | – | 1 | 2 | 0 | 0 | 0 | – | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | – |
| BEQ | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | – | 2 | 0 | 0 | 0 | – | – | 0 | 0 | 0 | 1 | 1 | 1 | 1 | – |
| | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | – | 2 | 0 | 0 | 0 | – | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | – |
| IN | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | – | – | 2 | 3 | 0 | 1 | 0 | – | 0 | 0 | 0 | 1 | 1 | 0 | 1 | – |
| | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | – | – | 3 | 0 | 0 | 0 | – | – | 1 | 1 | 0 | 1 | 1 | 1 | 1 | – |
| OUT | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | – | – | 2 | 0 | 0 | 0 | 0 | – | 0 | 0 | 0 | 1 | 1 | 0 | 0 | – |
| Alu | 0 | 1 | 0 | 0 | 0 | a | b | c | – | – | 2 | 3 | 0 | 1 | 0 | – | 0 | 0 | 0 | 0 | 1 | 1 | 1 | abc |
| | 0 | 1 | 0 | 0 | 0 | a | b | c | – | – | 3 | 0 | 0 | 0 | – | – | 1 | 1 | 0 | 1 | 1 | 1 | 1 | abc |
| AluI | 0 | 1 | 0 | 0 | 1 | a | b | c | – | – | 2 | 0 | 0 | 0 | – | – | 1 | 1 | 0 | 1 | 1 | 1 | 1 | abc |

Tabelle C.1: DEP-CPU -Decoder-Tabelle

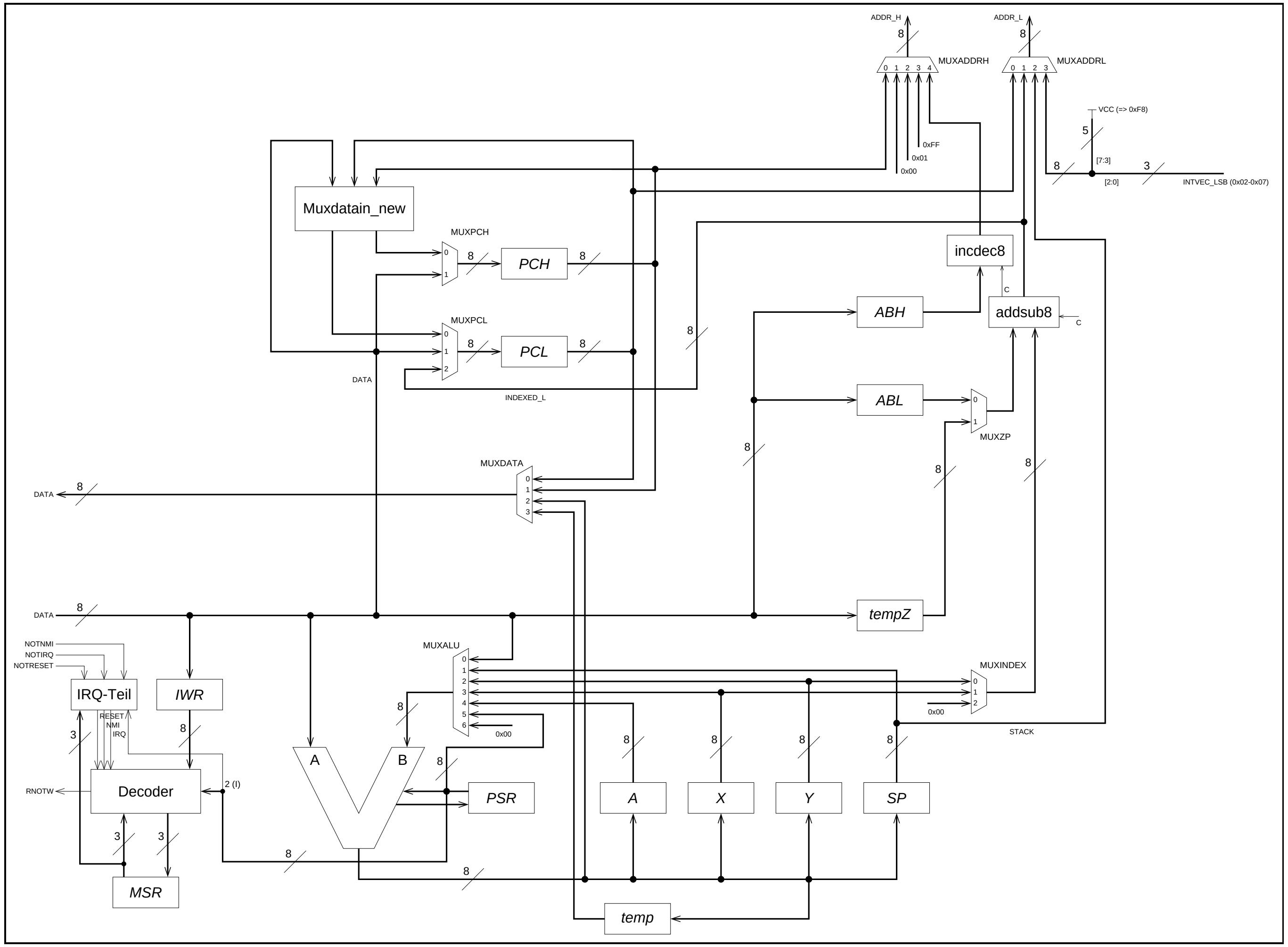



Abbildung C.1: Blockschaltbild des synchronen 6502

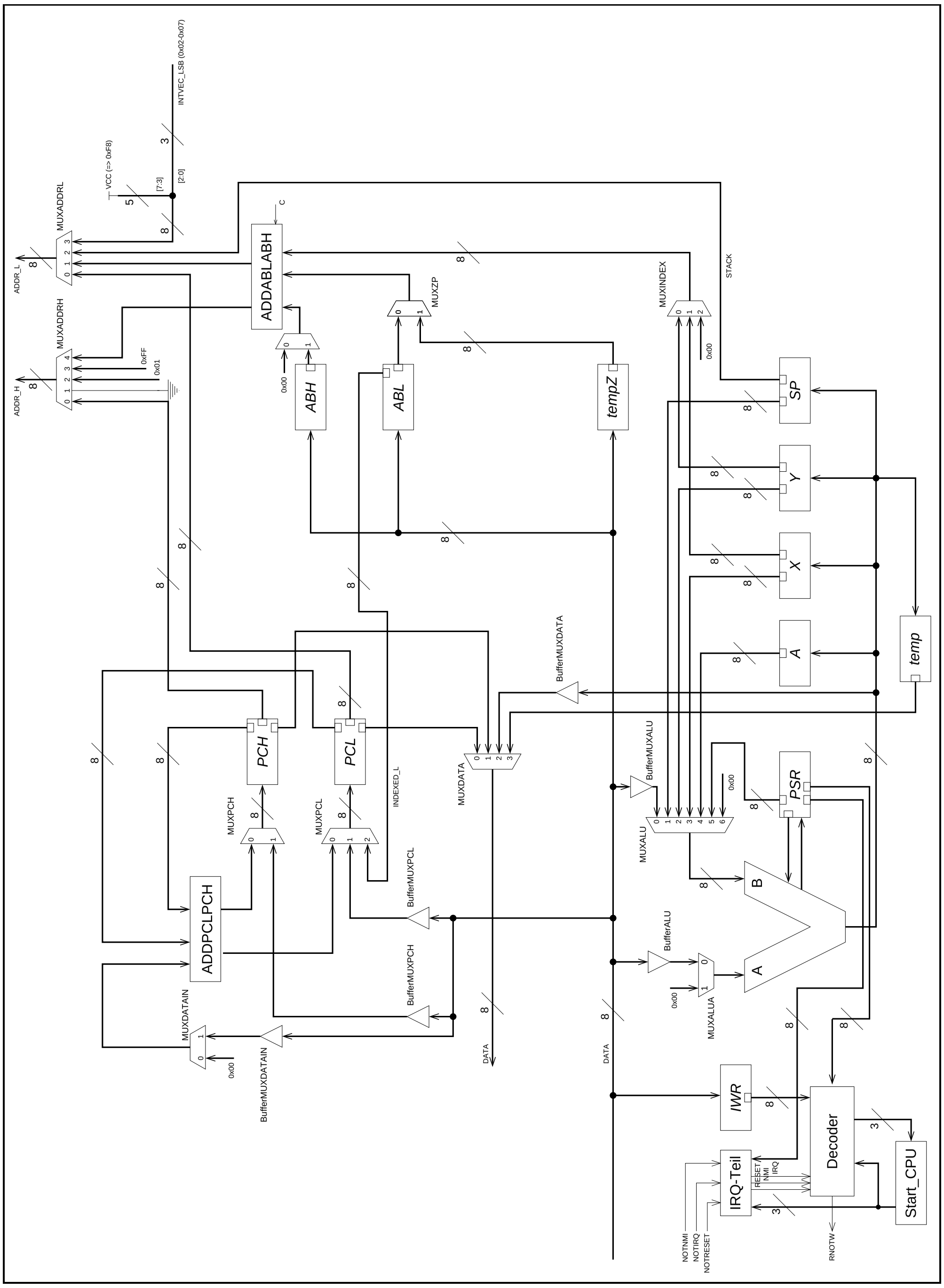



Abbildung C.2: Blockschaltbild des asynchronen 6502

# Literaturverzeichnis


[CT99] Wirtgen, Jörg, Bleul, Andreas: *Wie taktlos... Die Rückkehr asynchroner Prozessoren* Heise Verlag, c't 17/99, 1999

[MOORE] Moore, Gordon E.: *Cramming more components onto integrated circuits*, Research and Development Laboratories, Fairchild Semiconductor, 1965
12.10.2001 <http://www.intel.com/research/silicon/mooreslaw.htm>

[HUMBERT] Humbert, Hans-Jürgen: *64'er Hardware-Buch* Markt & Technik Verlag AG, 1992

[MUELLER] Müller, Helmut, Waltz, Lothar: *Mikroprozessortechnik* Vogel Verlag, 1990

[SCHUMNY] Schumny, Harald: *Mikroprozessoren* Vieweg Verlag, 1983

[ELEKTOR] Hogenboom, Paul (Hrsg.): *Mikroprozessor-Datenbuch 1* Elektor Verlag, 1988

[VCCSOFTAPI] Virtual Computer Corporation: *H.O.T. II Software API Guide Version 2.3* Virtual Computer Corporation, 1999

[VCCHARDAPI] Virtual Computer Corporation: *H.O.T. II Hardware API Guide Version 2.3* Virtual Computer Corporation, 1999

[VCCHARD] Virtual Computer Corporation: *H.O.T. II Hardware Guide Version 2.0* Virtual Computer Corporation, 1999

[HAUCK] Hauck, Scott: *Asynchronous Design Methodologies: An Overview* Proceedings of the IEEE, Vol. 83, No. 1, Seite 69-93, Januar 1995
27.08.2001 <http://www.elen.utah.edu/~cmyers/ee547/ho4.ps>

[MANCH] *The Asynchronous Logic Home Page*, University of Manchester
27.08.2001 <http://www.cs.man.ac.uk/async/>

[AM2E] *The AMULET2e Microprocessor* University of Manchester
11.10.2001 <http://www.cs.man.ac.uk/amulet/projects/AMULET2_uP.html>

[CALIF] Prof. Akella, Venkatesh: *ASRG Home Page*, Department of Electrical and Computer Engineering, University of California
27.08.2001 <http://www.ece.ucdavis.edu/cerl/asrg.html>

[ASYMT] Kapilan Maheswaran: *Hazard-Free Implementation of the Self-Timed Cell Set in a Xilinx FPGA*, Department of Electrical and Computer Engineering, University of California, 1994
12.10.2001 <ftp://ftp.ece.ucdavis.edu/pub/akella/fpga95.ps.Z>

[UTAH] *Timed Circuit Research Group*, University of Utah
27.08.2001 <http://www.async.elen.utah.edu/>

[DEPSCRIPT] Glöckner, Knoll, Zhang: *Materalien zum Digitalelektronischen Praktikum im Sommersemester 2000*, AG Technische Informatik, Universität Bielefeld, 2/1999
12.10.2001 <http://www.techfak.uni-bielefeld.de/ags/ti/lehre/DEP/depskriptS00.pdf>

[DKEAR] Kearney, David: *Asynchronous Logic*, School of Computer and Information Science, University of South Australia
27.08.2001 <http://www.cis.unisa.edu.au/~cisdak/nResearch/Async.html>

[FU650X] Commodore Semiconductor Group: *NMOS 6500 Series Microprocessors*, Finnish University and Research network FUNET
27.08.2001 <ftp://ftp.funet.fi/pub/cbm/documents/chipdata/650x/index.html>

[OP6502] West, John, Makela, Marco: *Documentation for the NMOS 65xx/85xx Instruction Set*
29.08.2001 <http://www.cs.cmu.edu/~dsladic/vice/doc/64doc.txt>

[FREEIP] *The Free IP Project*
08.10.2001 <http://www.free-ip.com/>

[KATZ1] Katz, Randy H.: *Contemporary Logic Design*, Electrical Engineering and Computer Science Department, University of California
27.08.2001 <http://www.redbrick.dcu.ie/academic/CLD/>

[OCT1] Gigascale Silicon Research Center: *Octtools*
01.10.2001 <http://www.gigascale.org/pubs/downloads/octtools/index.htm>

# Index

# Glossar

**2-Pass Assembler** Ein Assembler, der beim Übersetzen in die Maschinensprache in zwei Schritten vorgeht. Im ersten Pass wird das Programm gelesen und die Adressen für eventuelle Sprungbefehle im Zielprogramm werden berechnet. Im zweiten Pass werden die Assemblerausdrücke und Operatoren in den Binärcode übersetzt.

**AGU** Adress Generation Unit; Teil der CPU, der die nachfolgende Adresse generiert.

**Akkumulator** kurz auch Akku oder AccR; der eigentliche Ergebnisspeicher einer CPU.

**ALU** Arithmetic Logic Unit; das eigentliche Rechenwerk einer CPU. Hier werden Manipulationen von Registern ausführt.

**ASIC** „Application Specific Integrated Circuit“. Ein ASIC ist – im Gegensatz zum (Standard-) IC – auf eine spezielle Anwendung ausgelegt und meist von einem einzelnen Kunden für diesen speziell in Auftrag gegeben. ASICs lohnen sich natürlich erst ab größeren Stückzahlen, aber auch nur bei ausreichender Spezialisierung (= nicht zu große Stückzahl). In diesem Fall (asynchrone CPU) stimmt „spezielle Anwendung“allerdings nicht so richtig: Als CPU ist die Anwendungsspezialisierung natürlich möglichst gering. Aber aufgrund der geringen Nachfrage nach asynchronen CPUs und dem derzeitigen Entwicklungsstadium nutzt man den ASIC-Vorteil geringer Kosten.

**Benchmark** Zeit, die ein spezielles Programm braucht, um abzulaufen. Möglichkeit, zwei CPUs zu vergleichen.

**Bitfile** wird normalerweise von einem Compiler erzeugt und auf einen FPGA übertragen.

**Boolesche Verknüpfungen** in logischen Gleichungen bedeutet das Zeichen „$\wedge$“eine AND-Verknüpfung, „$\neg$“eine NOT-Verknüpfung, „$\vee$“eine OR-Verknüpfung und „$\dot{\vee}$“eine XOR-Verknüpfung.

**Bug** (engl.: Laus, Bazillus; engl. to $\sim$ : nerven, stören.) Fehler in Soft- oder Hardware.

**CCR** Code Condition Register; speichert das Carry- und Zero-Flag in der DEP-CPU.

**Check-Machine-State** kurz: CMS; erzeugt das Signal, das den Takt bei unserer asynchronen Umsetzung ersetzt.

**CLB** Configurable logic block; kleinste Logikeinheit auf den FPGAs.

**Decoder** decodiert einen Befehl und schaltet die Bauteile einer CPU entsprechenden.

**DEP** Digitalelektronisches Praktikum; eine Veranstaltung der Universität Bielefeld im Grundstudium des Studiengangs „Naturwissenschaftliche Informatik“zum Erlernen der grundsätzlichen Eigenschaften der Elektronik.

**DEP-CPU** Der abschließende Versuch des DEPs besteht in der Entwicklung dieser CPU mit Hilfe des *Schematic Editor*s von *Xilinx, Inc.*

**DEP-Format** Der Inhalt der RAM-Bänke wird im „DEP-Format“erwartet. Dieses Format belegt für jedes zu speichernde Byte ein DWord (32Bit). Im ersten dieser vier Bytes wird das eigentliche Nutzdatum gespeichert, in den mittleren zwei Bytes die 16Bit Adresse dieses Datums und im obersten Byte „`0xFF`“. Es können also maximal 64kB je RAM-Bank gespeichert werden, die 256kB Platz belegen. Vorteil dieses Formates ist, daß der PCI-Wrapper (siehe Kapitel 2.5.2) erkennen kann, welche Zelle das RAM gerade ausgibt. Zwar heißt eine Übereinstimmung mit der gewünschten Adresse noch nicht, daß auch schon das eigentliche Datum am Ausgang anliegt, nimmt man jedoch noch die Schaltzeit des Komparators hinzu, kann – ohne festen Takt oder eine feste Wartezeit – eine möglichst kurze RAM-Zugriffszeit erreicht werden.

**DNF** Disjunktive Normalform; Disjunktion von min-Termen.

**Flag** ein 1-Bit-Speicher, der einen Zustand symbolisiert.

**Flankenwechsel** Wechsel von `low` nach `high` oder von `high` nach `low`.

**FPGA** Field programmable gate array; rekonfigurierbarer Block logischer Elemente.

**IOB** I/O block, Interface zwischen FPGA und Außenwelt.

**IWR** Instruction Word Register; speichert den Befehl, der vom Decoder entsprechend verarbeitet wird.

**LUT** Look-Up Table; gespeicherte Tabelle, mit deren Hilfe eine logische Funktion realisiert wird.

**MMU** Memory Management Unit; Bauteil, das den Zugriff auf externen Speicher verwaltet.

**Mnemonic** Die Assemblerrepräsentation eines Maschinenbefehls, abkürzende Funktionsbeschreibung (z. B. `LDA` **L**oa**D** **A**ccumulator).

**MSR** Maschinenstatusregister; steuert den Ablauf des aktuellen Befehls.

**Nullwelle** Nullwellen sind im pegelgesteuerten Dual-Rail nötig, um die Logik zwischen Berechnungschritten zurückzusetzen.

**Octtools** Eine Sammlung von nützlichen Programmen und Programmbibliotheken rund um das Chipdesign, entwickelt an der UC Berkeley; siehe auch [OCT1].

**Offset** Position innerhalb eines Datenbereichs, ein Offset von $0$ bezeichnet den Anfang des Bereichs.

**Opcode** Abkürzung für „Operation Code“, Maschinencode. Bitmuster, das von einem Prozessor ausgewertet wird und in ihm eine Funktion auslöst.

**OPR** OPerand Register; speichert den Operanden eines Befehls bei der DEP-CPU.

**PCR** Program Counter Register: Hier wird gespeichert, in welchem Schritt des Programms man sich gerade befindet.

**Pointer** (engl.: Zeiger) Verweis auf einen Speicherbereich.

**Polling** (engl.: Stimmenabgabe, Wahl, Umfrage) Das Polling ist eine asynchrone Art des Datenaustausches. Sender und Empfänger – hier FPGA und Host-System – können jederzeit Daten senden, unabhängig vom Zustand des Empfängers. Ist der Empfänger zum Empfang von Daten bereit, fragt er selbst nach Daten (er „pollt“) und bekommt die zuletzt vom Sender geschickten Daten. Hierdurch können allerdings auch Daten verloren gehen, wenn der Sender schneller neue Daten sendet, als der Empfänger diese abfragt.

**Prozessor** siehe CPU

**RAM** Random Access Memory; beschreibbarer Speicher.

**ROM** Read Only Memory; nicht veränderbarer Speicher.

**Schaltgeschwindigkeit** Zeit, die eine Einheiten braucht, um das gültige Ergebnis zu erzeugen und am Ausgang anzulegen.

**Schematic Editor** Ein Teil der Software *Xilinx Foundation Series 3.1i*, die es ermöglicht, Schaltkreise mit Hilfe eines GUI zu designen.

**Tristate** Ein Buffer-Element, das neben den beiden Zuständen „leitend“und „sperrend“noch einen dritten, hochohmigen Zustand besitzt. Dieses ist für bidirektionale Busse wichtig, da so angeschlossene Bauteile davor bewahrt werden, durch auf Ausgänge gelegte Spannung zerstört zu werden.

**VHDL** VHSIC (Very High Speed Integrated Circuits) Hardware Description Language: eine Hardwarebeschreibungssprache, mittels derer sich alle möglichen Schaltwerke erstellen lassen.

**Zweierkomplementdarstellung** Die am häufigsten verwendete Darstellung für positive und negative Dualzahlen. Duale Werte der Zweierkomplementzahl stimmen mit $x$ überein, wenn $x \geq 0$. Ist $x$ negativ, dann wird der Betrag von $x$ von $2^n$ abgezogen, damit wieder ein positiver Wert entsteht.