

Systemanalyse eines Mikroprozessors und
dessen Re-Design als transparente Struktur
zur dynamischen Visualisierung interner
Abläufe während des Betriebs unter
ProfiLab 4.

microprocessor@freenet.de

MÄRZ 2010

DATEIÜBERBLICK	9
VORWORT	11
Voraussetzungen	13
ALLGEMEINES	14
Haftungsausschluss	14
Schnellstart	15
Grenzen der Simulation	16
Terminologie	17
Prozessorwahl	17
Kompatibilität	18
Bauteil Auswahl	19
Projektuntergliederung	20
Vereinbarung	22
ALU/SHIFT - MODUL	24
Arithmetische/Logische Einheit (ALU)	24
ACCU und ALU-REG-IN	26
Status-Register	28
Decimal Adjust	31
Low Nibble-Auswertung	33
High Nibble-Auswertung	34
SHIFT - Funktion	36

RMB / SMB / BBR / BBS	39
Bit-Masken	40
Befehlsunterbrechung bzw. Sprungauslösung bei BBR / BBS	41
RMB x (Reset Memory Bit x)	41
SMB x (Set Memory Bit x)	43
Branch On Bit Reset (BBR)	45
Branch On Bit Set (BBS)	47
Manueller Betrieb der ALU/SHIFT - Unit	49

ADRESSGENERATOR 52

Adressierungarten	52
Implied: IMP	53
Accumulator: A	53
Immediate: IMM	53
Absolute: ABS	54
Absolute Indexed: ABS,X oder ABS,Y	55
Zero Page: ZP	56
Zero Page Indexed: ZP,X oder ZP,Y	57
Absolute Indirect: (ABS)	58
Absolute Indexed Indirect: (ABS,X)	60
Zero Page Indirect: (ZP)	62
Zero Page Indexed Indirect: (ZP,X)	64
Zero Page Indirect Indexed (ZP),Y	66
Relativ 1-Byte-Offset	68
Relativ 2-Byte-Offset	70
Struktursynthese	72
Vektorgenerator	73
Reset	73
NMI	73
INT	73
Stackpointer	74
SWI	74
High-Byte bei relativem Sprung mit 1 Byte Offset (R8)	75
Zero Page	75
Zusammenfassung	76
Y- und X-Register	77
TEMP-Register	77
Adressaddierer	77

Adressregister	77
Programmzähler	78
Stackpointer	78
PC CLK bei INT und NMI	79
RAM	80
RAM im Simulationsbetrieb	81
RAM im realen Betrieb	83
ROM	84
Manueller Betrieb des Adressgenerators	85
 ABLAUFSTEUERUNG	 88
Unterscheidungen von Steuerwerken	89
Takt	90
Taktung der Steuersignale	92
Ein-Phasen-Takt	93
Symetrische Teilschritte	95
Befehlsstruktur	96
Die Mikrocode-Speicher-Adressierung	98
Fetch-Cycle	101
Aktivierung eines neuen Befehls	104
CLK DATA IN [^] - und CLR ^o - Signal	107
Page-Bit-Ansteuerung	107
Ansteuerung des Command-Registers	107
Ansteuerung des μ Step-Counters	108
Sonderbefehle (P1-Befehle)	109
NMI - Aufbereitung	113

INT - Aufbereitung	114
0, 3, N und I - Signale	114
Nibble Dekoder	115
Pseudo-BRK	115
Bedingte Sprünge	116
BE8 - Signal	117
BE16-Signal	119
BI8- und BI16-Signal	120
Generierung der kompletten Pseudo-Befehle	121
Generierung der inkompletten Pseudo-Befehle	123
Start-Logik	124
Manueller Betrieb des Steuerwerks	125
Reset (1000h)	125
CLK	125
Laden eines neuen Befehls:	125
BRK (1010h)	126
NMI (1020h)	126
INT (1040h)	126
(Branch on CCR-Bit) mit 8-Bit-Offset (1080h)	126
(Branch on CCR-Bit) mit 16-Bit-Offset (1100h)	126
(Branch on Bit Set/Reset) mit 8-Bit-Offset (1080h)	127
(Branch on Bit Set/Reset) mit 16-Bit-Offset (1100h)	127
65C02Cumin.PRJ	128
 MAKRO-ASSEMBLER	 130
"HAND-ASSEMBLER"	130
MAKRO-ASSEMBLER	130

FETCH (F1 & F2)	FETCH	136
RESET	1000	137
BRK	1010	138
NMI	1020	140
INT	1040	141
SWI xx	02	142
BRA unconditional rel. 8-Bit-Offset	80	144
BRN conditional rel. 8-Bit-Offset	XX	145
BRA unconditional rel. 16-bit-Offset	83	146
BRN conditional rel. 16-bit-Offset	XX	147
BBR ZP, rel. 8-Bit-Offset	XX	148
BBR ZP, rel. 16-Bit-Offset	XX	149
BBS ZP, rel. 8-Bit-Offset	XX	150
BBS ZP, rel. 16-Bit-Offset	XX	151
JMP abs.	4C	152
JMP (abs.)	6C	153
JMP (abs.,X)	7C	154
JMP (ZP)	DC	155
JMP (ZP,X)	FC	156
JSR abs.	20	157
JSR rel. 8-Bit-Offset	22	158
JSR rel. 16-Bit-Offset	23	159
JSR (abs.)	43	160
JSR (abs.,X)	63	162
JSR (ZP)	42	164
JSR (ZP,X)	62	166
RTS	60	168
RTI	40	169
RMB ZP	XX	170
SMB ZP	XX	171
CLC	18	172
SEC	38	172
CLD	D8	173
SED	F8	173
CLI	58	174
SEI	78	174
CLV	B8	175
DEA	3A	176
INA	1A	176
DEX	CA	177
INX	E8	177
DEY	88	178
INY	C8	178
NOP	EA	179
PHA	48	180

PLA	68	180
PHP	08	181
PLP	28	181
PHX	DA	182
PLX	FA	182
PHY	5A	183
PLY	7A	183
TXA	8A	184
TAX	AA	184
TXS	9A	185
TSX	BA	185
TYA	98	186
TAY	A8	186
Befehle nach Adressierungsart		187
ACCU Addressing-Mode		193
Immediate Addressing-Mode		194
Absolute Addressing-Mode		196
Absolute,X Addressing-Mode		198
Absolute,Y Addressing-Mode		200
Zero Page Addressing-Mode		201
Zero Page,X Addressing-Mode		203
Zero Page,Y Addressing-Mode		205
(Zero Page) Addressing-Mode		206
(Zero Page,X) Addressing-Mode		208
(Zero Page),Y Addressing-Mode		210
Direction (Memory to Register)		212
AR ALU-IN		212
AR BIT ALU-IN (BIT-Command immediate only)		212
AR ACCU		212
AR XREG		212
AR YREG		213
AR SHIFT		213
Functions		214
FUNCTION: ADC H (ADC HEX-MODE)		214
FUNCTION: ADC D (ADC DECIMAL-MODE)		214
FUNCTION: AND		215
FUNCTION: ASL		215
FUNCTION: BIT ALU-IN		215
FUNCTION: CMP (A - M)		215
FUNCTION: CPX (X - M)		216
FUNCTION: CPY (Y - M)		216
FUNCTION: DEC ALU-IN / MEM		216
FUNCTION: DEC ACCU		217
FUNCTION: DEC X-REG		217
FUNCTION: DEC Y-REG		217

FUNCTION: EOR	218
FUNCTION: INC ALU-IN	218
FUNCTION: INA	218
FUNCTION: INC X-REG	219
FUNCTION: INC Y-REG	219
FUNCTION: LSR	219
FUNCTION: ORA	220
FUNCTION: ROL	220
FUNCTION: ROR	220
FUNCTION: SBC (HEX-MODE): SBC H	220
FUNCTION: SBC (DECIMAL-MODE): SBC H & DAS	221
Direction (Result to Memory/Accu)	222
ACCU TO MEM	222
XREG TO MEM	222
YREG TO MEM	222
ZERO TO MEM	222
SHIFT TO MEM	223
SHIFT TO ACCU	223

BEFEHLSMATRIX	224
----------------------	-----

AUSSICHTEN	226
-------------------	-----

QUELLENANGABEN	226
-----------------------	-----

Dateiüberblick

Für die Anzeige und die Simulation der folgenden Schaltungen ist das Programm ProfiLab-(Expert) Version 4 als Voll- bzw. Demo-Version (www.abacom-online.de/html/demoverionen.html) erforderlich.

65C02.PRJ	komplettes Projekt mit wahlweise vollautomatischem Betrieb oder manuellem Schritt-für-Schritt-Modus (Bedienung s. „Schnellstart“ in der beiliegenden Dokumentation
65C02DIS.TXT	Textdatei zeigt während der Programmausführung den Assembler-Mnemonic des aktuellen Befehls an. Die Datei muss, sofern der Pfad-Eintrag im Read-Line-Bauteil nicht modifiziert wird, im gleichen Verzeichnis wie 65C02.PRJ stehen.
65C02DOC.PDF	diese Datei, Dokumentation für alle im Folgenden aufgeführten 65C02xx.PRJ-Dateien

Die folgenden Dateien sind nur für diejenigen interessant, die sich weitergehend mit dem Thema auseinandersetzen möchten. Die Dateien sind jeweils Teilmengen von 65C02.PRJ und durch ihre geringere Größe schneller lad- und simulierbar. Alle Funktionen können per Schalter eingestellt und getestet werden. Für den automatischen Betrieb des Prozessors ist das grundsätzliche Verständnis dieser Teilschaltung Voraussetzung.

Um sich das Arbeiten zu erleichtern wird empfohlen, die jeweiligen PRJ-Dateien auszudrucken. Dadurch erspart man sich den ständigen Wechsel zwischen Projekt- und Dokumentationsdatei auf dem Bildschirm.

65C02AS.PRJ	ALU/SHIFT-Einheit (AS)
65C02AG.PRJ	Adressgenerator (AG)
65C02CU.PRJ	Control-Unit (CU), Ablaufsteuerung
65C02CUmin.PRJ	Minimalversion einer Ablaufsteuerung

65C02 μ C.PRJ	Ausgabe der Einzelschritte mit ihren Steuer- signalen für alle Befehle des Befehlsatzes
65C02 μ C.PDF	Tabellarische „Übersicht“ des Micro-Codes
LST2TXT.EXE	konvertiert die .LST-Ausgabe des CA65- Makro-Assemblers in das für ProfiLab erforderliche .TXT-Format
C.BAT	Schnellaufruf von LST2TXT.EXE
A.BAT	parametrisierte Batch für Makro-Assembler
AC.BAT	assemblieren und konvertieren (s.o.)
6502DEMO.ASM	ein kleines Demonstrationsprogramm inkl. NMI-, INT- und Subroutinen. Es ist im Original- Projekt 65C02.PRJ bereits ab Adresse 0300h geladen
6502DEMO.LST	Programmlisting des Demonstrationsprogramms
6502DEMO.TXT	konvertierter, in ProfiLab RAM/ROM ladbarer Programmcode des Demonstrationsprogramms

Vorwort

Es gibt inzwischen kaum ein elektrisches Gerät, in dem nicht ein Mikroprozessor enthalten ist, wobei zuweilen die Frage erlaubt sein muss, ob denn der Einsatz überhaupt überall Sinn macht. Wie dem auch sei: jeder nutzt mehr oder weniger bewusst diese Geräte, ohne sich Gedanken über die Grundlage, den Mikroprozessor, zu machen. Und so soll es im Prinzip auch sein, denn der Nutzen bzw. der Effizienzgewinn eines Gerätes sollten im Vordergrund stehen. Kein Mensch, der heute ein Auto benutzt, muss wissen wie ein Motor funktioniert.

Von der Erfindung des Rads, der Entwicklung des Buchdrucks, der Dampfmaschine, des Verbrennungsmotors, bis zur Elektrotechnik oder der Elektronik: nicht nur der Anwender hat sich mit zunehmender Komplexität des jeweiligen Geräts immer weiter von den dessen inneren Funktionen entfernt. Auch die Hard- und Software-Entwickler und so auch die Prozessor-Entwickler verlieren den direkten Kontakt zu ihrem Produkt. Die Zahl der Prozessor-Entwickler, die ein gesamtes Produkt überschauten, war früher schon naturgemäß gering. Durch die Komplexität der heutigen Prozessoren und der notwendigen Projektuntergliederung, der daraus resultierenden Arbeitsteilung sowie dem Einsatz von Werkzeugen wie VHDL oder Verilog für die Hardware-Entwicklung findet inzwischen auch in diesem Bereich eine Abstraktion des Arbeitsinhaltes und eine Arbeitsentfremdung wie in anderen Produktionsbereichen statt. Darüber hinaus unterliegen alle Ergebnisse dem Betriebsgeheimnis. Das führt zu der interessanten oder vielleicht auch erschreckenden Erkenntnis, dass wir die Grundlagen einer Technologie, die unser Leben (mit-)bestimmt, nicht mehr nachvollziehen können, selbst wenn wir es wollten.

Programmiersprachen wie C++, Visual-Sprachen und Entwicklungsumgebungen entbinden von der Notwendigkeit, den Kern der Sache, auf dem alles aufbaut, zu kennen. Für Hardware, Treiber, Betriebssysteme und Anwendungen sind Schnittstellen definiert, -zig Firmen liefern Gigabyte an Softwarekomponenten, die aufeinander aufsetzen und es ergibt sich ein System, dass in der Regel funktioniert, aber mit dem menschlichen Geist im Detail nicht mehr zu erfassen oder gar komplett zu testen ist.

Durch die Entwicklungstools ist die Zahl der Programmierer, die hardwarenah zu programmieren in der Lage sind, nach meiner Einschätzung seit den siebziger Jahren trotz der inzwischen weit verbreiteten Nutzung der EDV in allen Teilen der Gesellschaft weiter zurückgegangen.

Es stellt sich somit die Frage, ob zumindest die Basis, das Innere eines Mikroprozessors, in Gänze von einem einzelnen

Menschen, dazu noch Laie, nachzuvollziehen ist. Erforderlich wäre dafür zumindest ein kompletter Schaltplan mit Dokumentation. Recherchen im Internet und Fach- sowie Universitätsbibliotheken erbrachten nur unbefriedigende Ergebnisse. Selbst die Schalt-/Funktionspläne der ältesten 4- oder 8-Bit-Prozessoren wurden bisher von den einschlägigen Herstellern nicht veröffentlicht. In manchen Vorlesungsunterlagen der Fachbereiche Informatik oder Nachrichtentechnik findet man einzelne exemplarische Funktionsteile, die zudem oft nur als Blockschaltbild dargestellt sind. Ein vollständiges System war nicht zu finden. So entstand die Idee, einen kompletten, funktionsfähigen, dynamisch-transparenten, kompatiblen und dokumentierten Mikroprozessor aus Standard-ICs zu entwickeln. Dabei galten die in der Entwicklung üblichen Forderungen nach einer möglichst geringen Anzahl von Bauteilen und einem möglichst kompakten Schaltplan wobei versucht wurde, diese beiden Kriterien weitestgehend (aber nicht dogmatisch) einzuhalten. Wenn es für die Modularisierung der Schaltung oder die Übersichtlichkeit des Schaltplans erforderlich schien, wurde in wenigen Fällen auf die Umsetzung der Forderung verzichtet.

Voraussetzungen

Beim ersten Blick auf den Umfang der Schaltung und der Dokumentation mag der/die eine oder andere davor zurückschrecken, sich mit dem Projekt zu beschäftigen. Grundkenntnisse der Digitaltechnik, die Zugriffsmöglichkeit auf die Funktionsbeschreibungen von Standard-ICs und des 65C02 sowie Grundlagen der Assemblerprogrammierung reichten aus, um das Projekt zu realisieren und höchstens genau so viel technische Kenntnisse benötigt man, um es zu verstehen. Durch die Verwendung eines einfachen Taktes kommt die Dokumentation bis auf wenige Ausnahmen ohne Timingdiagramme aus. Kenntnisse der Bool'schen Algebra oder von KV-Diagrammen sind für das Verständnis der Schaltung nicht erforderlich.

Der höhere Aufwand liegt nicht in den fachlichen Voraussetzungen sondern in der Ausdauer, die gesamte Schaltung mit ihrer hohen Anzahl an Funktionen zu entwickeln bzw. im Kontext nachzuvollziehen.

Es handelt sich um eine rein akademische Schaltung, die der Visualisierung des internen Datenflusses eines Mikroprozessors während der Programmverarbeitung dient, d.h. einen gläsernen bzw. transparenten Prozessor.

Bis zu diesem Projekt habe ich mich weder privat noch beruflich mit Digitaltechnik oder Schaltungsentwurf beschäftigt. Lediglich während der Ausbildung kam ich kurzzeitig mit Digitaltechnik und Mikroprozessorprogrammierung in Verbindung. Hier stellte sich dann auch die Frage, ob es möglich sei, einen Prozessor selbst zu entwickeln. Ursprünglich hatte ich geplant, das Projekt in der Realität, sprich mit Platinen, ICs und LEDs zu realisieren. Rückblickend betrachtet glaube ich, dass mir das auf Grund der fehlenden Erfahrung in dieser Form nicht gelungen wäre. Zufällig stieß ich im Elektronikhandel auf ProfiLab und seit dem Update auf die Version 3 (mit Bustreibern) hat sich das Projekt bis auf den heutigen Stand entwickelt.

Allgemeines

Haftungsausschluss

Es wurde sich bemüht, eine 65C02 - kompatible Schaltung zu entwickeln. Dennoch besteht keine Garantie auf Funktionsfähigkeit oder Kompatibilität.

Der Autor übernimmt keine Haftung für die Folgen aus dem Gebrauch der Schaltung. Dies gilt insbesondere für den Betrieb der Schaltung in Verbindung mit Hardware-Schnittstellen zur Steuerung oder Regelung realer Prozesse!

Schnellstart

Wer die Schaltung nicht analysieren sondern nur die Bits „flitzen“ sehen möchte, der gehe wie folgt vor:

Digital-ProfiLab Ver. 4 (Demo- oder Vollversion) starten

Simulation auf „Fast“
einstellen

Bauteile-Bibliotheksanzeige
(„<<<“) ausschalten

Datei 65C02.prj laden (dauert ca. 30 Sek.)

Pegelanzeige (HI/LO - Symbol)
Aktivieren

Simulation starten und (dauert ca. 3 Min.)
Frontplatte aktivieren

Modus A:

Clock-Mode-Schalter auf ON (automatischer Programmablauf,
default)

RST°-Schalter auf OFF und
zurück auf ON (Programmstart)

Sollte das Programm auf älteren, langsamen Rechnern fehlerhaft ablaufen, muss die Frequenz im CLK-Baustein entsprechend niedriger eingestellt werden.

oder Modus B:

Clock-Mode-Schalter auf OFF manueller Schritt-für-Schritt-
Betrieb

RST°-Schalter auf OFF und
zurück auf ON (Programmstart)

Clock-Schalter im Wechsel (Einzelschritt-Modus)

Grenzen der Simulation

Die Schaltung ist so umfangreich, dass die maximale Taktgeschwindigkeit für die Simulation im automatischen Modus bei Verwendung eines 3GHz Single-Core-Prozessor-Systems je nach Wahl der aktivierbaren Funktionen (Signalpegelanzeige, Mnemonic-Anzeige) im Bereich von 8 - 12Hz liegt. Das scheint auf den ersten Blick wenig, ist aber genau die richtige Geschwindigkeit, um einen Eindruck der dynamischen Abläufe im Prozessor während der Programmverarbeitung zu erhalten. Schneller visualisierte Prozessabläufe lassen sich ohnehin nicht mehr von Augen und Gehirn verarbeiten. Erhöht man den Takt, so reicht die Verarbeitungsgeschwindigkeit des PCs nicht mehr aus und es treten Fehler bei der simulierten Befehlsverarbeitung auf. Das gleiche passiert u.U. wenn während des automatischen Betriebs die Frontplatte mit der Maus verschoben oder die Pegelanzeige ein- bzw. ausgeschaltet wird. Die geringe Simulationsgeschwindigkeit liegt vor allem in der Größe der Projektdatei. Da aber eine Vielzahl an 7474 (D-Flip-Flops) zum Einsatz kommen und dieses Standard-Bauteil nicht in der internen Bauteile-Bibliothek von ProfiLab vorhanden ist, musste die Flip-Flop-Funktion durch ein aufwendiges Makro realisiert werden, das bei der Simulation entsprechend seinem Umfang und der erforderlichen Anzahl unnötiger Weise Ressourcen beansprucht.

Dokumentation

Bei der beiliegenden Dokumentation handelt es sich um eine Kurz-/Vorabversion. Sie dient in dieser Form vor allem dem Autor als Gedächtnisstütze für den Wiedereinstieg in das Projekt nach längeren Entwicklungspausen zur Vermeidung der Frage: „Was habe ich mir damals nur dabei gedacht?“. Einige Abschnitte beinhalten Beispiele und weitergehende Ausführungen, andere enthalten z. Zt. nur einige Sätze als Gedankenstütze.

Die Grundlagen der Digitaltechnik und der Assemblerprogrammierung, die Funktion der Standard-Bauteile der 74-Reihe oder das Basiswissen zum Thema 65C02 sind in der Beschreibung nicht enthalten. Diese Informationen findet man unter den jeweiligen Stichworten in großem Umfang im Internet. Dem interessierten Digitaltechniker sollte der Umfang der Dokumentation in Verbindung mit den im Internet verfügbaren Informationen aber ausreichen, die Schaltung und den Mikro-Code nachzuvollziehen.

Terminologie

In der Dokumentation werden technische Anglizismen verwendet, da die englischen Fachausdrücke zum einem üblich und zum anderen meist kürzer sind. Auch macht es macht zum Beispiel keinen Sinn, aus einem gängigen Begriff wie „Datenbus“ eine „Datensammelschiene“ zu machen usw.

Prozessorwahl

Es wurde bewusst ein 8-Bit Prozessor gewählt. Bezüglich der ALU/SHIFT-Unit bedeutet eine 16-Bit-Breite lediglich eine Verdoppelung der 8-Bit-Struktur, d.h. es führt zu keinen neuen Erkenntnissen sondern lediglich zu einer größeren Projektdatei mit der Folge zunehmender Unübersichtlichkeit und erhöhten Ladezeiten und geringerer Simulationsleistung.

Der Adressgenerator für einen 64KB-Adressbereich ist bei einem 8-Bit-System deutlich anspruchsvoller und damit interessanter als bei einem 16-Bit-Bus, da bei einem 8-Bit-Prozessor die 16-Bit-Adresse in zwei Schritten aus zwei Bytes gebildet werden muss. Wird ein Adressbereich größer als 64KB gewählt, muss wie bei einem 8-Bit-Prozessor die Adresse (z.B. 20 Bit für 1MB) entweder wieder in zwei Schritten gelesen werden oder vorher ein Segmentregister initialisiert werden, aus dem in Verbindung mit einer 16-Bit-Adresse die endgültige 20-Bit-Adresse gebildet wird. Auch hier führt eine Verdoppelung der Busbreite letztendlich nur zu einer Aufblähung der Schaltung und bringt für die Demonstration eines Prozessors im Wesentlichen keine neuen Informationen.

Die Wahl fiel auf den 6502, ein früher auf Grund seiner ökonomischen und technischen Vorteile sehr populären Prozessor, der heute noch u.a. in Fernbedienungen Verwendung findet. Vor allem seine komplexen Adressierungsarten machen ihn wegen seines effizienten Codes in der Anwendung und als Untersuchungsobjekt für dieses Projekt interessant. Darüber hinaus besitzt die spätere CMOS-Variante 65C02 für die damalige Zeit leistungsfähige Befehlserweiterungen, die heute zum Standard-Befehlsumfang von Prozessoren gehört (u.a. Bit-Test-Befehle und Branch-On-Bit-Befehle). Bereits in der Urfassung war eine Dezimalkorrektur implementiert, eine Funktion, die in der Praxis eher unbedeutend ist aber bei diesem Projekt für weitere Vielfalt sorgt. Mit den mannigfaltigen Adressierungsarten und Funktionen bietet der 65C02 eine gute Grundlage für die Entwicklung anderer, auch proprietärer Prozessoren.

Kompatibilität

Ziel des Projekts war es, die Schaltung in der Form zu realisieren, dass der gesamte Befehlsumfang des 65C02 enthalten ist und die Befehle in ihrer Funktion kompatibel zum Original sind. Eine weitere Forderung bestand darin, dass alle Befehle unter Berücksichtigung der Taktform in weniger oder maximal genauso vielen Teilschritten abgearbeitet werden wie bei dem Original. Andere Kompatibilitätskriterien wie z.B.

Pin-
Signal-
Befehls-
Bauform-
Zyklen-
Taktform-
Taktgeschwindigkeits-
Performance-
oder thermische

Kompatibilität, um nur einige zu nennen, konnten oder sollten keine Berücksichtigung finden.

Um eine bei 8-Bit-Prozessoren bestehende und in der Praxis umständlich zu umgehende Einschränkung zu beheben, wurde der Original-Befehlsumfang um einige zusätzlichen und damit inkompatible Anweisungen ergänzt. Bei dem Originalprozessor wird die Sprungweite für relative Verzweigungen mit einem Offset von einem vorzeichenbehafteten Byte angegeben, sodass Sprünge nur im Bereich -128/+127 möglich sind. Bei dem hier vorgestellten Projekt wurden alle relativen Sprungbefehle (Branch und Bit-Branch) zusätzlich mit 2-Byte-Offset implementiert, sodass relative Sprünge im Bereich -32768/+32767 möglich sind. Darüber hinaus wurden alle Sprungbefehle und Unterprogrammverzweigungen als relative Befehle mit 1 bzw. 2 Byte Offset realisiert, sodass der Programmcode weitestgehend relocierbar generiert werden kann. Insgesamt wurde der Befehlssatz um 34 neue Befehle ergänzt. Der hier vorgestellte Prozessor mit seinen Befehlserweiterungen wird dort, wo die Unterschiede der Befehlssätze kenntlich gemacht werden müssen, als 65C02+ bezeichnet.

An dieser Stelle wird ausdrücklich darauf hingewiesen, dass es sich hier um eine rein theoretische Arbeit handelt, da ein reales 65C02-basiertes System als Referenz für Quer-Tests nicht zu Verfügung stand.

Bauteilauswahl

Um die Möglichkeit zu erschließen, das simulierte Projekt ohne ein Re-Design in die Realität umzusetzen, wurde auf den Einsatz von Komponenten aus der internen ProfiLab-Bauteilebibliothek weitestgehend verzichtet, da dort nur einige Grundfunktionen enthalten und diese teilweise nicht pin- bzw. signalkompatibel zu realen Bauteilen sind. In manchen Fällen liegen auch Funktionsunterschiede vor.

Aus diesem Grund wurden nahezu alle Bausteine als Makros realisiert. Das hat zudem den Vorteil, dass sich die gesamte Schaltung bis auf Gatterebene verfolgen lässt. Lediglich die Standardgatter (jedoch nur im Umfang und Verfügbarkeit von realen Bausteinen), die Speicher und die Bustreiber, sofern diese als Byte- und nicht als Nibble-Treiber Verwendung fanden, wurden aus der Standardbibliothek entnommen. Nachteilig wirkt sich diese Vorgehensweise auf die Dateigröße des Projekts, die Ladezeit, die Startzeit für die Simulation und vor allem auf die Simulationsgeschwindigkeit aus.

Aufgrund der Verfügbarkeit in vielen Technologie-Familien (incl. CMOS) wurde die Schaltung mit Bauteilen der 54/74 - Reihe (meist in Nibble-Breite) entwickelt. Es wurde bewusst auf die Verwendung „moderner“ Bauteile (z.B. 16-Bit-Bustreiber oder 12-Bit-Zähler) verzichtet, um das Projekt ggfs. mit vorhandenen Standard-ICs realisieren zu können.

Da bei der Pinbezeichnung von Makros eine Verwendung von mehreren Fonts zur Darstellung von alphanumerischen Zeichen und Sondersymbolen nicht möglich ist und auf die Erstellung eines speziellen Fonts mit z.B. Invertierungsbalken über den Zeichen verzichtet wurde, gelten für Makros dieses Projekts folgende Vereinbarungen:

Makro-Pin ohne Zusatz : Signal active high, Pegel
Makro-Pin mit ^ : Signal active high, Flanke

Makro-Pin mit °
(symbolisch für „0“) : Signal active low

Auf Grund des Fehlens weiterer, auch bei kleiner Darstellung gut erkennbarer und eindeutiger Symbole bei den Standardfonts, wurde für active-low auf eine Differenzierung zwischen Pegel und Flanke verzichtet. In der Regel erkennt man an der Bezeichnung (CLK), dass es sich um einen flankensensitiven Eingang handelt.

Projektuntergliederung

Das gesamte Projekt ist modular aufgebaut. Die einzelnen Module sind optisch getrennt und funktional weitestgehend unabhängig voneinander. Übersichtlichkeit, vereinfachte Fehlersuche und schnellere Tests sind die bekannten Gründe für eine modulare Entwicklung. Der wesentliche Punkt ist in diesem Fall das schnellere Laden der Module und die kürzere Startzeit für die Simulation. Nach einer einzelnen Änderung werden für den Simulationsstart der Gesamt-Projektdatei 65C02.PRJ auf einem 3GHz-Rechner 3 Minuten. Eine zügige Schaltungsentwicklung ist somit nicht mehr möglich.

Der Grund für die hohe Ladezeit liegt zum einem in der Größe der Projektdatei (954 KB). Die Simulationsstartzeit hat sich aber nach dem Update von Digital-ProfiLab 3 auf Version 4 mehr als verdoppelt. Bereits in der Version 3 war kein einflankengetriggertes Flip-Flip in der internen Bauteilebibliothek enthalten, doch konnte diese Einschränkung mit einem zusätzlich beschaltetem Ein-Flanken-R/S-FF in Form eines Makros, das beim Simulationsstart noch zügig geladen wurde, umgangen werden. Die Version 4 beinhaltet überhaupt kein einflankengetriggertes Flip-Flop in der internen Bauteilebibliothek, sodass das 7474 komplette aus sechs Gattern realisiert werden muss. Dies hat zur Folge, dass bei jedem Simulationsstart das relativ aufwendige Makro analysiert und „durchgerechnet“ werden muss. In der Schaltung findet das 7474 insgesamt 172-mal Verwendung, d.h. der zusätzliche Zeitaufwand beim Simulationsstart steigt entsprechend.

Dateiname	Beschreibung	Ladezeit	Simulation starten	ProfiLab beenden
65C02.PRJ	Gesamtprojekt	0:30	3:00	2:35
65AS.PRJ	ALU/SHIFT-Modul	0:04	0:14	0:05
65AG.PRJ	Adressgenerator	0:07	0:38	0:12
65CU.PRJ	Control-Unit	0:04	0:03	0:01

Ladezeit, Simulations-Startzeiten und Beenden von ProfiLab mit geladener Projektdateien auf einem 3 GHz-System (Angaben in Minuten und Sekunden)

Die hier nicht veröffentlichte Schaltung mit Micro-Code-Editor benötigt 1 Minute zum Laden, 5 Minuten für den Simulationsstart und knapp zwei Minuten für das Beenden des Programms.

In der Gesamtprojektdatei 65C02.PRJ erkennt man auf der linken Seite vertikal die zehn Micro-Code-ROMs und darunter die externen Signaleingänge RST°, NMI°, IRQ°, CLK und CLK-Mode. Rechts der Micro-Code-ROMs wurde im oberen Bereich horizontal die ALU/SHIFT-Unit und darunter der Adressgenerator platziert. Zwischen diesen beiden Einheiten verläuft der Datenbus. Die Steuerleitungen für die ALU-SHIFT-Unit werden von oben, die für den Adressgenerator von unten an die Bauteile geführt. Unterhalb der Steuerleitungen des Adressgenerators findet man das Steuerwerk (Control-Unit).

Die Module 65C02AS.PRJ, 65C02AG.PRJ und 65C02CU.PRJ sind bis auf wenige Ausnahmen, die für den isolierten Betrieb erforderlich sind und auf die in den jeweiligen Abschnitten ggfs. gesondert hingewiesen wird, Eins-zu-eins-Kopien der entsprechenden Abschnitte des Gesamtprojekts. Die Ansteuerung der Schaltungskomponenten erfolgt in den einzelnen Modulen nicht automatisiert aus den Micro-Code-ROMs sondern über Schalter, die auf der rechten Seite der Schaltung platziert wurden. Dadurch wird ein manueller Testbetrieb ermöglicht.

Durch die Aufteilung in einzelne Module lassen sich u.U. Redundantien nicht vermeiden, d.h. in einigen wenigen Fällen sind Funktionen doppelt angelegt.

Da der größere Teil der Bauteile mit active-low Pegeln oder fallenden Flanke aktiviert werden, sind für eine höhere Erkennbarkeit der jeweils aktiven Schaltungszustände alle Steuersignale als active-low-Signale ausgelegt, was zur Folge hat, dass für die wenigen active-high-Eingänge die entsprechende Steuerleitungen invertiert werden müssen. Dieser Mehraufwand an Gattern ist bei einem Demonstrationssystem unerheblich im Vergleich zum Nutzen, da man mit einem Blick auf die Steuersignale und den ihnen zugeordneten LEDs erkennt, welche Leitungen (Schalter) aktiv sind. Auch von Seiten des Timings entstehen durch die in den Gatterlaufzeiten der Inverter begründeten Verzögerungen des Signals bei dem für das Projekt gewählten Taktverfahren keine Probleme. In einem professionellen System würde man natürlich auf die Inverter verzichten und das Signal invertiert im Micro-Code-ROM abspeichern.

Vereinbarung

Da die Plus- und Masse-Symbole im Schaltplan viel Platz benötigen und freie Eingänge bei ProfiLab standardmäßig „High“ sind, wurde vereinzelt auf die Plus-Symbole verzichtet. Bei einer Umsetzung des Projekts in die Realität sind die Eingänge auf „High“ zu legen! Wenn möglich wurden die Bussignale immer so verlegt, dass bei horizontaler Anordnung das niederstwertige Bit oben und das höchstwertige Bit unten und bei vertikaler Anordnung das niederstwertige Bit rechts und das höchstwertige Bit links liegen.

Für Schalter und Pegel werden die Bezeichnungen OFF, LOW, L und 0 für „AUS“ und ON, HIGH, H und 1 für „AN“ verwendet.

Die Schalter für den manuellen Betrieb sind mit unterschiedlichen Farben versehen.

Blau gekennzeichnete Schalter öffnen eine Datenquelle, die den Bus beeinflusst. Aus diesem Grund darf bei einer Aktion zur gleichen Zeit nur ein blauer Schalter aktiv sein, um Datenkollisionen auf dem Bus zu vermeiden.

Die unterschiedlichen Funktionen einzelner Bausteine werden mit den schwarz gekennzeichneten Schaltern selektiert. Diese Schalter haben keinen Einfluss den Bus, sodass mit ihnen theoretisch gleichzeitig mehrere Steuereingänge in unterschiedlichen Funktionsbereichen geschaltet werden könnten, da nur die durch eine blauen Schalter aktivierte Funktionseinheit ein korrektes Ergebnis auf den Bus leitet. Die anderen Bereiche würden zwar gleichzeitig mit der für sie i.d.R. sinnlosen „schwarzen“ Bitkombination angesteuert werden, was aber durch die nicht aktivierte Ausgabe keinen Einfluss auf den ordnungsgemäßen Ablauf hätte. Bei diesem Projekt wurde aus didaktischen Gründen für jede Funktionsauswahl ein separater Schalter vorgesehen. Bei professionellen Systemen würde man sicherlich versuchen, die Funktionsauswahlleitungen mehrfach zu verwenden, um die Gesamtheit der Steuerleitungen und damit die ROM-Breite für den Mikro-Code zu minimieren.

Nach der Auswahl der Datenquelle und der Voreinstellung der gewünschten Funktion wird mit dem entsprechenden Clock-Signal (rot-markierte Schalter, active low!) eine Aktion durchgeführt oder das Ergebnis einer Aktion in ein Zielregister oder in eine Speicherstelle eingetaktet.

Grün gekennzeichnete Schalter haben keine Funktion oder sind für spätere Erweiterungen reserviert. Ihre Anzahl ergibt sich aus der Anzahl der tatsächlich benötigten Steuerleitungen und der nächsten ohne Rest durch acht teilbaren Zahl, wodurch sich

auch die Menge der benötigten 8-Bit-ROMs für die Speicherung des Mikro-Codes errechnet.

Für die ALU/SHIFT-Einheit und den Adressgenerator werden jeweils 40 Schalter/ 5 ROMs eingesetzt. Auch wenn von 80 Steuersignalen nicht alle zum Einsatz kommen, so erscheint die Zahl im Vergleich zu anderen 8-Bit-Prozessoren, sofern darüber Angaben vorliegen, eher hoch und damit ineffizient. Dies liegt aber, wie bereits oben ausgeführt, an der Anzahl der den einzelnen Funktionen dediziert zugeordneten Steuersignalen. Im Gegensatz zu professionellen System, bei denen die Steuersignale teilweise direkt aus dem Befehlsbyte abgeleitet werden (partielle RISC-Architektur), sind bei diesem Demonstrationsmodell fast alle Befehle über eine Firmwaresteuerung realisiert, wodurch der Hardwareaufwand für die Befehlsdekodierung gering und die Möglichkeit zur flexiblen Anordnung der Befehle bzw. zur Verwendung des Dekoders für andere Prozessoren hoch ist. Nur einige wenige Befehle wurden zur Demonstration hardware-abhängig umgesetzt.

Um die Busstrukturen beizubehalten und damit die bessere Lesbarkeit des Schaltplans und die einfachere Signalverfolgung zu gewährleisten, wurde die ProfiLab-Funktion „Aufräumen“ bewusst nicht angewandt.

Die Makro-Bezeichnungen berücksichtigen die Anzahl der in einem realen Baustein der 74-Reihe vorhandenen Funktionen. Die Makro-Bezeichnung 7474½ zum Beispiel bedeutet, dass das Makro ein einzelnes Flip-Flop eines 7474-Bausteins enthält, der in der Realität zwei Flip-Flop-Funktionen beinhaltet. Durch die Trennung in singuläre Komponenten lassen sich die notwendigen Funktionen mengenmäßig exakt an der Stelle im Schaltplan platzieren, an der sie benötigt werden. Damit werden umständliche Leitungsführungen zu einem noch nicht verwendeten Bauteil an irgendeine unpassende Stelle im Plan sowie das evtl. Übersehen noch nicht verwendeter Komponenten vermieden. Über die Stücklisten-Funktion lässt sich dann der tatsächliche IC-Bedarf ermitteln.

Der externe 8-Bit-breite Datenbus (bis zu den beiden Bustreibern 74245) wird mit „Datenbus“ bezeichnet. Die internen Datenverbindungen werden dagegen nur mit „Bus“ bezeichnet und ggfs. mit High Bus (High Byte) bzw. Low Bus (Low Byte) genauer spezifiziert.

ALU/SHIFT - Modul

Das ALU/SHIFT - Modul (65C02AS.PRJ) umfasst folgende Funktionsblöcke:

- Arithmetisch/Logische Einheit (ALU)
- Akkumulator (ACCU)/ALU-REG-IN
- Status- bzw. Bedingungsregister (Conditional Code Register, CCR)
- Dezimalkorrektur (Decimal Adjust, DAA)
- Schiebereinheit (SHIFT-Register)
- Bit Set/Reset - Branch on Bit Set/Reset

Die Schaltung ist am Datenbus um zwei über einen Bustreiber zuschaltbare HEX-Eingaben ergänzt, um Testdaten auf den Bus auszugeben. Es empfiehlt sich, diesen Eingabeblock ggfs. an passende Stellen umzuplatzieren, damit man sich den Umweg über den ACCU spart, z.B. für den DAA-Test an den Channel B oder für den CCR-Test hinter die ALU.

Arithmetische/Logische Einheit (ALU)

Im Zentrum der Schaltung findet man die Arithmetische/Logische Einheit (ALU). Die erforderliche 8-Bit-Breite erhält man durch das Kaskadieren zweier Standard-4-Bit-ALUs vom Typ 74181. Damit stehen zwei 8-Bit-Eingänge für die Operanden mit den Bezeichnungen Channel A und Channel B zu Verfügung. Der Übertrags-Bit-Ausgang der Low-Nibble-ALU C_{n+4} wird auf den Übertrags-Bit-Eingang C_n der High-Nibble-ALU geführt.

Die ALU kann entweder mit invertierten Daten (active low) und Carry = true oder mit nicht invertierten Daten (active high) und invertiertem Carry betrieben werden. Für die hier vorliegende Schaltung wurde der Betrieb mit active-high-Daten gewählt. Um den invertierten Carry-Bit-Eingang zu umgehen, könnte zur Realisierung eines active-high-Eingangssignals ein zusätzlicher Inverter eingesetzt werden, doch wurde darauf aus Laufzeit- und aus Ressourcengründen verzichtet.

Mit den 5 Steuerleitungen **ALU S0** - **ALU S3** und **ALU MODE** lassen sich jeweils 16 arithmetische und 16 logische Funktionen ausführen. Im Folgenden ist eine Untermenge von Funktionen aufgeführt, die für die Umsetzung von Befehlen gängiger Mikroprozessoren, so auch des 65C02, eingesetzt werden:

<u>MODE</u>	<u>S3</u>	<u>S2</u>	<u>S1</u>	<u>S0</u>	<u>Funktion</u>	
					Cn° = H (no Carry)	Cn° = L (with Carry)
0	0	0	0	0	A	A PLUS 1
0	0	1	1	0	A-B-1	A-B
0	1	0	0	1	A PLUS B	A PLUS B PLUS 1
0	1	1	1	1	A-1	A
1	0	0	0	0	A°	
1	0	0	0	1	A NOR B	
1	0	0	1	0	A° AND B	
1	0	0	1	1	0	
1	0	1	0	0	A NAND B	
1	0	1	0	1	B°	
1	0	1	1	0	A EXOR B	
1	1	0	1	0	B	
1	1	0	1	1	A AND B	
1	1	1	1	0	A OR B	
1	1	1	1	1	A	

Dem Übertragsbit-Eingang Cn der Low-Nibble-ALU ist ein 1-aus-4-Multiplexer (74153) vorgeschaltet, der durch Schalten der Signale **MUX ALU S0** und **MUX ALU S1** das für die jeweilige Funktion notwendige Eingangssignal liefert. Bei einigen Befehlen wird für die korrekte Ausführung eine echte „Null“ bzw. „Eins“ benötigt (Inkrementieren bzw. Dekrementieren). Bei der Addition und der Subtraktion hingegen muss der Wert des Carry-Bits berücksichtigt werden. Für die Subtraktion beim 65C02 ist das invertierte Carry (Borrow) erforderlich ($A - B - \text{Carry}^\circ$), bei anderen Prozessoren das nicht invertierte Carry ($A - B - \text{Carry}$). Der invertierte Cn - Eingang an der ALU erfordert für die Befehlsausführung in diesem Zusammenhang besondere Konzentration (Details s. Mikro-Code ADC, SBC, INC, DEC).

MUX-ALU		Signal	
S1	S0	an Cn°	
0	0	0	echte „Null“
0	1	1	echte „Eins“
1	0	C-Bit	Carry-Bit
1	1	C-Bit°	invertiertes Carry-Bit

Das Resultat der ALU wird über das Signal **ALU TO BUS°** über einen 74244 auf den Bus ausgegeben.

Der Ausgabe des ACCU-Inhalts bzw. des ALU-IN-Registerinhalts an die ALU, die Verarbeitung in der ALU und das Schreiben des Ergebnisses in ein internes Register bzw. in den Speicher wird in der Simulationsschaltung in einem einzigen Taktzyklus durchgeführt. Besonders die ALU-Verarbeitung und das Abspeichern in das RAM benötigen verhältnismäßig viel Zeit, so dass ein entsprechend langer Taktzyklus erforderlich wird (s. Takt). Zur Vermeidung dieses Problems in der Realität würde man das ALU-Ergebnis zunächst in einem Ausgangsregister (z. Bsp. 74273) auffangen und erst in einem weiteren Takt abspeichern. Die dafür erforderlichen Signalleitungen **CLK ALU OUT°** ist bereits vorgesehen. Der Mikro-Code muss dann natürlich entsprechend geändert werden.

ACCU und ALU-REG-IN

Auf der linken Seite der Schaltung findet man den Akkumulator. Der Akku erhält seine Daten vom Datenbus und leitet sie an Channel-A der ALU und an die Dezimal-Korrektur DAA (Decimal Adjust Accu) weiter.

Werte aus dem Arbeitsspeicher, die mit Hilfe der ALU verarbeitet werden sollen, werden vom Datenbus in das Eingangsregister ALU-REG-IN übernommen. Von dort werden die Daten bei Befehlen, die in Verbindung mit dem Accu-Inhalt ausgeführt werden (z.B. Addiere Accu-Inhalt mit Wert), auf den ALU-Channel-B ausgegeben, bei alleiniger Bearbeitung jedoch i.d.R. auf den ALU-Channel-A, da dort im Gegensatz zum Kanal B alle benötigten ALU-Funktionen zu Verfügung stehen.

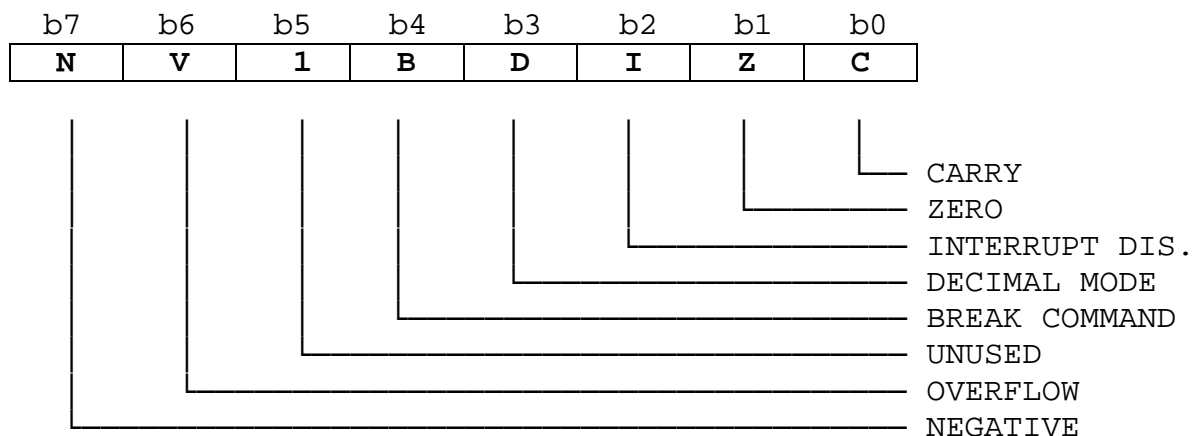
Anmerkung zu Ringbefehlen:

Bei der hier gewählten Form der ACCU/ALU-Aktion innerhalb eines Taktzyklus handelt es sich um einen klassischen Ringbefehl. Mit

der steigenden Flanke wird der Wert des ACCUs an die ALU ausgegeben und das ALU-Ergebnis mit der fallenden Flanke sofort wieder in den ACCU eingetaktet, sodass der neue Wert wieder zum Eingabewert der ALU wird. Berücksichtigt man in realen Systemen die Gatterlaufzeiten, so kann in dieser Form durchaus eine schnelle und ordnungsgemäße Befehlsausführung realisiert werden. Bei digitalen Simulationen sollte sicherheitshalber auf diese Lösung verzichtet werden, da die Arbeitsweise des Simulationsprogramms (Reihenfolge der Teilschritte) nicht bekannt und vorhersehbar ist. In beiden Fällen, real oder simuliert, erhöht sich die Gefahr für ein falsches Timing und damit falsche Ergebnisse, wenn der ALU aus zwei unterschiedlichen Quellen mit evtl. unterschiedlichen Signallaufzeiten Daten zugeführt werden. Dies trifft insbesondere bei der Addition und Subtraktion zu, bei denen zusätzlich zu den Operanden das Carry-Bit, das selbst auch einer Ringfunktion unterliegt, als Eingangswert berücksichtigt werden muss. Es empfiehlt sich deshalb, Ringbefehle zu vermeiden und durch Zwischenspeichern des ALU-Ergebnisses eine zeitliche Entkopplung vorzunehmen. Bei diesem Projekt wird das Zwischenspeichern des ALU-Ergebnisses und damit ein Taktzyklus eingespart und die Entkopplung an anderer Stelle vorgenommen. Durch die doppelte Ausführung des ACCUS, des Y- bzw. X-Registers und des Carry-Bits in Form von hintereinander geschalteten Registern bzw. Flip-Flops, die zeitversetzt durch die beiden unterschiedliche Flanken des jeweiligen Clock-Signals getaktet werden, ist sichergestellt, dass die ursprünglichen Inhalte noch stabil an der ALU anstehen während das ALU-Ergebnis und das Carry sicher in die jeweils vorgelagerten Register bzw. Flip-Flops eingetaktet werden.

Status-Register

Das Status-Register (Bedingungsregister, Condition Code Register, CCR) des 65C02 verfügt über acht Status-Bits, von denen sieben Positionen Verwendung finden. Das achte Bit (b5) ist immer „1“.



Das Status-Register des Prozessors wird aus 7 einzelnen D-Flip-Flops vom Typ 7474 realisiert. Die Status-Bits können dadurch separat über Steuerleitungen gesetzt (getaktet) werden, so wie es der jeweilige Befehl erfordert. Da die Steuerleitungen

```

CLK C°
CLK Z°
CLK I°
CLK D°
CLK B°
CLK V°
CLK N°
  
```

vereinbarungsgemäß als „active low“ definiert sind, ist jedem Takteingang der Flip-Flops ein Inverter vorgeschaltet. Bei den Additions-/Subtraktions- und den Schiebebefehlen ROL und ROR wird das C-Bit „gleichzeitig“ gelesen und geschrieben. Normalerweise genügen die Gatterlaufzeiten, um eine konfliktfreies Lesen des Ausgangs und Schreiben des Eingangs zu ermöglichen. Wie in der Einleitung angemerkt, soll die Schaltung aber ohne Berücksichtigung der Gatterlaufzeiten innerhalb eines Arbeitsschrittes realisiert werden. Deshalb wurde dem C-Bit-Flip-Flop ein weiteres 7474 nachgeschaltet, das mit dem Rücksetzen des CLK C° - Signals (steigende Flanke) eine

Kopie des C-Bits anlegt. Durch die zeitliche Entkopplung ist ein sicheres Lesen des C-Bits vor dem Schreiben sichergestellt (s.a. ALU/Ringbefehle). Beide Flip-Flops werden mit demselben Steuersignal getaktet, sodass die Erstellung der C-Bit-Kopie automatisch und zeitnah geschieht und bei der späteren Mikro-Code-Erstellung nicht explizit berücksichtigt werden muss. Diese Methode dient zur Vermeidung von Fehlern, hat aber, wie sich bei den Schiebebefehlen zeigen wird, einen kleinen Nachteil.

Vor den Dateneingängen liegt jeweils ein Multiplexer, der jedem Flip-Flop das gemäß den mit den Steuerleitungen MUX-CCR-S0 bis MUX-CCR-S2 gewählten Signalen aus den verschiedenen Datenquellen zuführt.

Für die Befehle CLC, CLD, CLI und CLV (Rücksetzen des jeweiligen Bits) wird dem jeweiligen Flip-Flop eine logische „0“ über den Eingang D0 des Multiplexers, für die Befehle SEC, SED und SEI (Setzen des jeweiligen Bits) eine logische „1“ über den Eingang D1 zugeführt.

Die dritte Datenquelle für das Status-Register ist das Wiederherstellen des Registers mit den Befehlen PHP und RTI. Dabei wird der auf den Stack gerettete ursprüngliche Inhalt des CCRs über den Bus und den Multiplexer-Eingang D2 wieder in die Statusregister-Flip-Flops zurück geschrieben.

Am häufigsten findet die Änderung der Status-Bits nach ALU- oder Transfer-Befehlen statt. Dabei wird bei Transfer-Befehlen und bei der Ausgabe des ALU-Ergebnisses auf den Bus (ggfs. Mit ALU TO BUS^o) das N-Bit direkt vom Bus abgegriffen und auf den Eingang des D3 des entsprechend Multiplexers geführt. Das Z-Bit wird in Ermangelung eines realen NORs mit 8 Eingängen durch zwei NORs mit 4 Eingängen, deren beide Ausgänge über ein AND verknüpft werden, realisiert.

Das V-Bit und das C-Bit werden vor bzw. hinter der ALU erzeugt bzw. abgegriffen.

Das V-Bit wird aus zwei EXOR und einem AND ermittelt. Diese Schaltung vergleicht die Vorzeichen der Operanden mit dem Vorzeichen des Ergebnisses.

Das C-Bit wird direkt vom Übertrags-Ausgang der High-Nibble-ALU abgegriffen.

Obwohl das C- und das V-Bit nicht aus Bus-Signalen ermittelt werden, wird die Datenquelle aus Transport- und ALU-Befehlen als „Bus-Quelle“ bezeichnet (D3 an den Multiplexern).

Auch wenn das N- und das Z-Bit nur zwei Eingangswerte (RTI/SP und ALU/BUS) besitzen und man bei entsprechender Anordnung der Eingangssignale mit 2-Kanal-Multiplexer (74154) auskäme, wurden, wie auch für die CCR-Bits V,B,D und I, 4-Kanal-Multiplexer vom Typ 74153 gewählt und für das achte und beim 65C02 nicht benutzte CCR-Bit der entsprechende Platz freigehalten. Damit ist es möglich, die gesamte Struktur ggfs. für andere Prozessortypen ohne aufwendiges Re-Design übernehmen zu können.

Für das C-Bit, das einen fünften und sechsten Kanal für die Datenquelle „SHIFT“ benötigt, wird ein 74151 mit 8 Kanälen eingesetzt. Somit werden drei Steuerleitungen für die Kanalauswahl benötigt.

Das N- und Z-Bit werden bei der Ausgabe des Schiebeergebnisses auf den Bus und damit zeitunabhängig von der C-Bit-Auswertung über den Kanal D3 (ALU/BUS) aktualisiert, sodass für diese beiden Bits bei SHIFT-Aktionen auf die Kanäle 4 und 5 verzichtet werden kann (Details s. Abschnitt SHIFT).

Im Folgenden wird noch einmal tabellarisch gezeigt, wie die Schalterstellung für die Steuersignale an den CCR-Multiplexern die entsprechende Datenquelle selektiert und welche Status-Register-Bits (Flip-Flops) zur Übernahme der Daten getaktet werden müssen.

MUX-CCR-S2	MUX-CCR-S1	MUX-CCR-S0	QUELLE	CCR-BITS
0	0	0	„0“	V, I, D, C
0	0	1	„1“	I, D, C
0	1	0	RTI/PLP	N, V, B, D, I, Z, C
0	1	1	BUS/ALU	N, V, Z, C
1	0	0	SHIFT D0	C
1	0	1	SHIFT D7	C
1	1	0	No Function	
1	1	1	No Function	

Für das Speichern der CCR-Bits auf den Stack nach einem Interrupt oder dem PHP-Befehl werden die Statusregister-Bits durch das Aktivieren der Leitung CCR TO BUS^o über einen 74244 auf den Bus geleitet. Bit 5 ist am Eingang des Bustreiber nicht beschaltet (bei realer Schaltung „H“ anlegen!) und damit bei

der Ausgabe auf den Bus „H“. So wird gewährleistet, dass der auf dem Stack abgelegte CCR-Wert kompatibel zu dem bei einem realen 65C02-System ist.

Decimal Adjust

Der 65C02 bietet die Möglichkeit zur BCD-Addition bzw. Subtraktion mit der erforderlichen Dezimalkorrektur.

Eine Korrektur, d.h. eine Addition bzw. Subtraktion mit dem Wert 6, ist bei folgenden Situationen erforderlich:

Unteres Halbbyte:

nach einer Operation liegt das Ergebnis im Bereich Ah - Fh.
Bsp.: die Addition von 9 + 3 ergibt Ch, erwartet wird 12d.

nach einer Operation liegt das Ergebnis zwar im Bereich 0 - 9, die Operation erzeugte aber einen Stellenüberlauf (Half-Carry).
Bsp.: die Addition von 9 + 9 ergibt 12h, erwartet wird 18d.

Oberes Halbbyte:

nach einer Operation liegt das Ergebnis im Bereich Ah - Fh.
Bsp.: die Addition von 50 + 60 ergibt B0h, erwartet wird 10d mit Carry = 1.

nach einer Operation liegt das Ergebnis zwar im Bereich 0 - 9, die Operation erzeugte aber einen Stellenüberlauf (Carry).
Bsp.: die Addition von 90 + 90 ergibt 120h, erwartet wird 80d Mit Carry = 1.

Ein Sonderfall liegt vor, wenn die Korrektur des unteren Nibbles das obere Halbbyte auf Ah setzt.

Bsp.: die Addition von 95 + 7 ergibt 9Ch, erwartet wird 102, wobei die Hunderterstelle im Carry angezeigt werden und am Ausgang der ALU 02 anstehen soll. Wird auf Ch im unteren Halbbyte zur Korrektur 6 addiert, so ergibt dies für das Low Nibble zwar 2, für das High Nibble jedoch Ah, d.h. in einem zweiten Schritt ist die Korrektur des oberen Halbbytes erforderlich (Ah + 6 = 0 und Carry = H). Um diesen zweiten

Schritt zu vermeiden wird eine Look-Ahead-Schaltung mit folgender Funktion realisiert:

liegt das Ergebnis einer Addition im Bereich 9Ah bis 9F so werden sowohl das untere als auch das obere Halbbyte korrigiert.

Ist das D-Bit bei einem Additions- bzw. Subtraktionsbefehl gesetzt und muss das Ergebnis korrigiert werden, so ist entsprechend der oben aufgeführten Bedingungen ein zweiter Rechenschritt zur Dezimalkorrektur erforderlich. Durch eine entsprechend aufwendige Schaltung kann ermittelt werden, ob überhaupt eine Korrektur erforderlich ist, d.h. es kann ggfs. ein Taktzyklus eingespart werden. Neben dem Schaltungsaufwand ist durch die im Laufe des Programms nur bedingt vorhersehbaren Ergebnisse der Addition bzw. Subtraktion die Kalkulation der Taktzyklen zur Berechnung von Programmlaufzeiten (z.B. Warteschleifen) nahezu unmöglich. Deutlich einfacher ist die Realisierung, wenn auf jeden Fall ein Korrekturzyklus durchgeführt wird, selbst wenn keine Korrektur erforderlich ist, d.h. es findet eine Pseudo-Korrektur mit „0“ statt.

Der bei der Addition bzw. Subtraktion ermittelte und anschließend im ACCU abgelegte Wert wird wieder der entsprechend eingestellten ALU (Channel A) zugeführt. Auf den Eingang B der ALU wird der erforderliche Korrekturwert gelegt. Für das High- und das Low-Nibble werden jeweils eine „6“ für die Korrektur und eine „0“ für die Pseudo-Korrektur benötigt. Diese Werte werden durch entsprechend beschaltete Bustreiber realisiert, deren Enable-Eingängen (G°) jeweils ein ODER-Gatter vorgeschaltet ist. Ist das **DAA $^\circ$** - Signal aktiv, wird durch das Low-Schalten des zweiten ODER-Eingangs je nach Erfordernis die „0“ bzw. die „6“ aktiviert.

Für die Entscheidung, ob eine Korrektur des unteren Halbbytes erforderlich ist, muss nach der ursprünglichen Addition bzw. Subtraktion auch der Überlauf der Low-Nibble-ALU ausgewertet werden. Das Half-Carry-Bit ist im Statusregister der 65C02 nicht berücksichtigt. Deshalb wird der ALU-Überlauf des unteren Halbbytes auf den Eingang eines Flip-Flops geführt, das das Half-Carry-Bit für eine spätere Auswertung speichert. Da das H-Bit nach einer Addition bzw. Subtraktion gleichzeitig mit dem

C-Bit aktualisiert werden muss und es ausschließlich für die Dezimalkorrektur und nicht für Sprungentscheidungen herangezogen wird, benötigt es kein eigenes Clock-Signal sondern kann mit dem C-CLOCK-Signal getaktet werden.

Der Überlauf Cn4° der ALU verhält sich bei arithmetischen Operationen wie folgt:

	Wenn kein Überlauf	wenn Überlauf	
Addition	1	0	Negative Logik
Subtraktion	0	1	Positive Logik

Am Eingang des H-Bit Flip-Flops liegt der von der ALU ausgegebene invertiert Überlauf, der auch in dieser Form in das Flip-Flop übernommen wird. Mit dem invertierenden Ausgang Q° steht das H-Bit für die Additionskorrektur auch in positiver Logik zu Verfügung.

Das H-Bit und das C-Bit sowie ihre invertierten Zustände werden jeweils auf die beiden Eingangskanäle zweier Multiplexer geleitet, deren Ausgänge über die S0-Leitung der ALU selektiert werden. Für eine Subtraktionskorrektur (S0 = „L“) werden die in positiver Logik vorliegenden Signale weitergeleitet. Um die DAA-Auswertung einheitlich mit positiver Logik durchführen zu können, werden bei einer Additionskorrektur (S0 = „H“) die am Ausgang der H- und C-Bit-Flip-Flops nochmals invertierten und damit jetzt auch in positiver Logik vorliegenden Signale weitergeleitet.

Low Nibble-Auswertung

Durch ein NOR-Gatter und einen Inverter wird aus dem im ACCU liegenden Wert mittels eines weiteren NOR-Gatters der Bereich Ah - Fh des Low Nibbles (LN) dekodiert. Zusammen mit dem H-Bit erfolgt nun die Auswertung für die Dezimalkorrektur.

A-F LN	H-Bit	OR	NOR
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

Nur wenn das untere Halbbyte im gültigen Zahlenbereich 0 - 9 liegt (A-F LN = „0“) und kein Übertrag aufgetreten ist (H-Bit = „L“), wird der Ausgang des OR-Gatters „L“. Dieses „L“ in Verbindung mit einem aktivierten DAA°-Signal („L“) schaltet das nachfolgende OR-Gatter ebenfalls auf „L“, sodass der Treiber den Wert 0h für die Pseudokorrektur auf das untere Halbbyte des Datenbusses ausgibt.

Liegt das untere Halbbyte im Bereich Ah - Fh (A-F LN = „H“) und/oder ist ein Überlauf aufgetreten (H-Bit = „H“) wird der Ausgang des NOR-Gatters „L“. Dieses „L“ in Verbindung mit einem aktivierten DAA°-Signal („L“) schaltet das nachfolgende OR-Gatter ebenfalls auf „L“, sodass der Treiber den Wert 6h für die Dezimalkorrektur auf das untere Halbbyte des Datenbusses ausgibt.

High Nibble-Auswertung

Durch ein OR-Gatter und ein AND-Gatter wird aus dem im ACCU liegenden Wert der Bereich Ah - Fh des High Nibbles (HN) dekodiert.

Für den Look-Ahead-Bereich 9Ah - 9Fh wird die High-Nibble-9 aus zwei Invertern und einem OR-Gatter (s.o.) dekodiert und mit den bereits teilausgewerteten Signalen für den Low-Nibble-Bereich Ah-Fh auf ein fünf-fach-NOR geleitet.

Zusammen mit dem C-Bit erfolgt nun die Auswertung für die Dezimalkorrektur.

A-F HN	9A-9F	C-Bit	OR	NOR
0	0	0	0	1
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

Nur wenn das obere Halbbyte im gültigen Zahlenbereich 0 - 9 (A-F HN = „0“) und das Ergebnis nicht im Bereich 9Ah - 9F liegt und kein Übertrag aufgetreten ist (C-Bit = „L“), wird der Ausgang des OR-Gatters „L“. Dieses „L“ in Verbindung mit einem aktivierten DAA°-Signal („L“) schaltet das nachfolgende OR-Gatter ebenfalls auf „L“, sodass der Treiber den Wert 0h für

die Pseudokorrektur auf das obere Halbbyte des Datenbusses ausgibt.

Liegt das obere Halbbyte im Bereich Ah - Fh (A-F LN = „H“) und/oder das Ergebnis im Bereich 9Ah - 9Fh (9A-9F = „H“) und/oder ist ein Überlauf aufgetreten (C-Bit = „H“), wird der Ausgang des NOR-Gatters „L“. Dieses „L“ in Verbindung mit einem aktivierten **DAA°**-Signal („L“) schaltet das nachfolgende OR-Gatter ebenfalls auf „L“, sodass der Treiber den Wert 6h für die Dezimalkorrektur auf das obere Halbbyte des Datenbusses ausgibt.

SHIFT - Funktion

Der 65C02 besitzt die vier Schiebebefehle ASL, LSR, ROL und ROR.

Mit den Schiebebefehlen können der Inhalt des Accus bzw. einer Speicherstelle um eine Position nach links bzw. nach rechts verschoben werden:

ASL :

c ←	b7	b6	b5	b4	b3	b2	b1	b0	← 0
-----	----	----	----	----	----	----	----	----	-----

LSR :

0 →	b7	b6	b5	b4	b3	b2	b1	b0	→ c
-----	----	----	----	----	----	----	----	----	-----

ROL :

c ←	b7	b6	b5	b4	b3	b2	b1	b0	← c
-----	----	----	----	----	----	----	----	----	-----

ROR :

c →	b7	b6	b5	b4	b3	b2	b1	b0	→ c
-----	----	----	----	----	----	----	----	----	-----

Bei der Analyse der Befehle erkennt man, dass entweder das höchst- oder das niederstwertige Bit beim Schiebevorgang in das Carry-Bit übertragen wird.

Bereits die Möglichkeit, 8-bit-breite Daten einzulesen und wieder auszugeben, schränkt die Bausteinauswahl bei den Standard-Schieberegistern auf wenige Möglichkeiten ein, wobei unter Berücksichtigung der Verfügbarkeit aus möglichst vielen Produktfamilien die Wahl auf den 74299 fällt.

Auch das Schieben in beide Richtung und das Einziehen von Daten in das höchst- bzw. das niederstwertige Bit ist bei diesem Baustein gegeben. Da das Ergebnis des Schiebevorgangs beim 74299 erst durch die Ausgabe der Daten auf den Bus im Folgetakt erkennbar wird, wurde bei dem Makro zur unmittelbaren Kontrolle die zusätzlichen Ausgang-Pins LED A - LED H herausgeführt.

Ein besonderes Problem stellt bei den Schiebebefehlen ROR und ROL die Verarbeitung des Carry-Bits dar. Zum einen muss der Originalwert des C-Bits zum Schieben stabil an den Eingängen SR/SL anstehen und zum anderen wird beim Schieben der Originalwert mit dem Wert von D0 bzw. D7 überschrieben. Bei Berücksichtigung der Gatterlaufzeiten von Schieberegister und C-Bit-Flip-Flop kann durchaus eine funktionierende Lösung realisiert werden, bei der das Original-Carry bereits für das Schieben berücksichtigt wurde bevor es mit dem neuen D0 bzw. D7 aktualisiert wird. Da bei digitalen Simulationen Gatterlaufzeiten nicht determinierbar sind (s.a. Ringbefehle), wird hier ein etwas umständlicherer aber dafür sicherer Weg gewählt, indem das C-Bit-Lesen und -Schreiben zeitlich entkoppelt werden. Eine Variante wäre, das C-Bit vor dem Schieben zwischenspeichern, D0 bzw. D7 des zu schiebenden Wertes über einen freien Kanal des C-Bit-Multiplexers (z.B. Eingang D4 für D0 und Eingang D5 für D7) in das Carry-Bit-Flip-Flop zu takten und anschließend den Schiebevorgang mit dem zwischengespeicherten Original-Carry durchzuführen. Für die hier vorliegende Schaltung wurde eine andere Variante gewählt, da sie durch die Verwendung der bereits vorhandenen Carry-Kopie einen Schritt weniger benötigt.

Die Lade- und Schiebeaktionen sollen mit der fallenden Flanke des Taktes ausgeführt werden, sodass das **CLK-SHIFT^o**-Signal, das standardmäßig „H“ ist, einmal invertiert werden muss, um den mit positiver Logik ausgelegten CLK-Eingang zum richtigen Zeitpunkt zu takten.

Sind die Steuersignale S0 und S1 des Schieberegisters auf „H“ (Standard-Ruhestellung), befindet sich der Baustein im Ladezustand, d.h. die Datenleitungen stehen auf „Eingang“ und die internen Daten wirken nicht auf den Bus (Tri-State). Aus diesem Grund kann auf die separate Ausgangskontrolle mittels G1^o und G2^o verzichtet werden. Beide Eingänge werden deshalb mit GND auf „Dauerbetrieb“ geschaltet.

Die „Einzieh“-Pins SL und SR werden zusammengeschaltet, da je nach Schieberichtung nur der jeweilig benötigte Pin berücksichtigt wird. Mit einem vorgeschalteten Multiplexer wird dem Befehl entsprechend entweder das Carry-Bit oder eine „0“ an SL/SR geführt. Die Auswahl erfolgt über das Signal **MUX SH**.

Das Laden des zu schiebenden Wertes vom Bus in das Schieberegister erfolgt mit steigender Flanke wenn die Steuersignale S1 und S0 „H“ sind. Mit diesen drei über ein UND-Gatter verknüpften Signalen werden Bit0 und Bit7 beim Ladevorgang des Schieberegisters in die Hilfs-Flip-Flops

getaktet. Für den eigentlichen Schiebevorgang werden mit den Signalen S1 und S0 die Schieberichtung und mit MUX SH das Einziehbit ausgewählt. Bei den ROL/ROR-Befehlen wird das Original-Carry von der Carry-Kopie abgegriffen. Anschließend wird mit **CLK SHIFT°** der Schiebevorgang ausgelöst und mit **CLK C°** bei entsprechend eingestellten C-Bit-Multiplexer D0 bzw. D7 in das Carry-Bit übertragen. Erst mit dem Rückstellen von CLK C°, also nach dem vollendeten Schiebevorgang, wird das neue Carry-Bit in das C-Bit-Kopie-Flip-Flop übernommen.

Bei der anschließenden Ausgabe des geschobenen Wertes auf den Bus (S1 und/oder S0 = „L“) werden das N- und das Z-Bit aktualisiert.

RMB / SMB / BBR / BBS

RMB x / SMB x,

Bei dieser Befehlsgruppe wird in der Zero-Page-Speicherstelle, die dem Befehlsbyte folgt, eine bestimmte Bit-Position auf „low“ bzw. „high“ gesetzt.

Für die Befehle RMB x und SMB x wird der gewählte Zero-Page-Speicherinhalt mit einer entsprechenden Bitmaske über die ALU logisch verknüpft, um das gewünschte Ergebnis zu erzielen und es anschließend wieder in den Zero-Page-Speicher zurück zu schreiben.

BBR x (8) / BBS x(8)

Bei dieser Befehlsgruppe wird aus dem Inhalt der Zero-Page-Speicherstelle, die dem Befehlsbyte folgt, mittels einer Bit-Maske eine bestimmte Bit-Position auf ihren Zustand (0 oder 1) überprüft.

Das Testergebnis bestimmt, ob ein relativer Sprung entsprechend dem dritten Befehlsbyte (8-Bit-Offset) erfolgt oder der Programmablauf hinter dem dritten Befehlsbyte mit dem nächsten Befehl fortgeführt wird.

Befehlsaufbau BBR x / BBS x (8-bit-Offset)

1. Befehl
2. Zero-Page-Adresse
3. 8-bit-Offset

BBR x(16) / BBS x(16)

Hierbei handelt es sich um die gleiche Funktion wie zuvor, jedoch mit 2-Byte Offset (16-Bit). Damit besteht die Möglichkeit auch relative Sprungweiten größer 127 zu realisieren. Es handelt sich somit um einen 4-Byte-Befehl mit folgendem Aufbau:

1. Befehls-Byte
2. Zero-Page-Adresse
3. Offset Low Byte
4. Offset High Byte

Diese Befehlsgruppe ist eine Weiterentwicklung und damit nicht Bestandteil des offiziellen Befehlssatzes bzw. von Makroassemblern oder Compilern!!!

Bit-Masken

Die benötigten Masken werden durch die Dekoder-Bausteine 74138 und 74238 realisiert. Der 74138 setzt abhängig von der Kombination seiner drei Eingangssignale A, B und C eine von acht Bitposition an seinem Ausgang auf "low" (Single Low), der 74238 eine von acht Bitposition auf "high" (Single High). Aus dem Befehlsbyte im Befehlsregister werden die Bits b4, b5 und b6, die die Position des zu testenden Bits kodieren (s.u.) an die Eingänge der Dekoder geführt. Je nach Befehlsgruppe wird der entsprechende Dekoder über den jeweils zugeordneten Bustreiber frei geschaltet (**SMB/BBS°** bzw. **RMB/BBR°**) und die benötigte Maske an den Eingang von ALU-A geleitet. Gleichzeitig wird die bereits vorher in das ALU-IN-Register transportierte Kopie der Speicherstelle an den Eingang der ALU-B ausgegeben. An der ALU wird die passende logische Verknüpfung eingestellt, um das gewünschte Ergebnis zu ermitteln. Für die bedingten Sprungbefehle BBR und BBS wird mit der Zero-Test-Logik das für die weitere Programmablaufsteuerung notwendige Signal ermittelt und weitergeleitet.

Die Anordnung der Befehle RMB/SMB, BBR/BRS sind in der Befehlsmatrix des 65C02 spaltenweise in Gruppen hintereinander angeordnet (x7h bzw. xFh), so dass die Bit-Masken-Selektion direkt aus dem Befehls-Byte ohne weitere Dekodierung möglich ist. Für die zusätzliche Befehlsgruppe BBR/BBS mit 16-Bit-Offset wurde die freie Spalte xBh verwendet. Für andere Mikroprozessor-Projekte, bei denen diese Befehlsgruppen nicht vorhanden sind aber für einen proprietären Prozessor implementiert werden sollen, ist u.U. eine andere Vorgehensweise erforderlich. Sollten in der Befehlsmatrix keine 16 Plätze in einer Reihe oder Spalte sondern nur verstreut verfügbar sein, so kann die Masken-Dekodierung mit den drei reservierten Leitungen **BIT S/R/T°** (Bit Set/Bit Reset/Bit Test) aus dem Mikro-Code erfolgen.

Befehlsunterbrechung bzw. Sprungauslösung bei BBR / BBS

Durch das „Low“- Setzen der Leitung **BIT BRN°** wird dem System mitgeteilt, dass es sich um einen Befehl aus der Gruppe BBR bzw. BBS handelt. Das Signal wird auf den Eingang eines OR - Gatters geführt.

Der Ausgang des Zero-Tests wird auf den zweiten Eingang des OR-Gatters geleitet. Wurde bei dem Bit-Test festgestellt, dass ein Sprung ausgeführt werden soll (Zero-Test = 0), so wird in Verbindung mit der aktivierten **BIT BRN°** - Leitung der Ausgang des OR-Gatters auf „0“ gesetzt und das BI° Signal ausgelöst. BI° steht für „Branch Internal“ und teilt der Programmablaufsteuerung mit, dass aus dem aktuellen, noch nicht abgeschlossenen Mikro-Befehl heraus mit dem nächsten Befehlstakt ein Sprung in einen anderen Mikro-Befehl (1080 bzw. 1100, relativer Sprung) erfolgen soll.

RMB x (Reset Memory Bit x)

Das zu bearbeitende Byte wird mit einer Bit-Maske logisch verknüpft, so dass das gewünschte Bit gelöscht wird und die übrigen Bits nicht beeinflusst werden. Die gewünschte Bitposition ist in dem jeweiligen Befehlsbyte kodiert.

<u>Befehlsbyte</u>	
hex	binär
	<u>b7<---->b0</u>

Befehl	07h	(0000	0111b)	lösche Bit 0
Befehl	17h	(0001	0111b)	lösche Bit 1
Befehl	27h	(0010	0111b)	lösche Bit 2
Befehl	37h	(0011	0111b)	lösche Bit 3
Befehl	47h	(0100	0111b)	lösche Bit 4
Befehl	57h	(0101	0111b)	lösche Bit 5
Befehl	67h	(0110	0111b)	lösche Bit 6
Befehl	77h	(0111	0111b)	lösche Bit 7

Beispiel:

Befehlsbyte	37h	lösche Bit 3 aus dem Byte
Zero-Page-Adresse	10h	der Adresse \$0010

Beispiel 1 für RMB:

Das Bit b3 eines Bytes (in diesem Fall "1") soll auf "Low" gesetzt werden:

	b7<----->b0							
Byte-Wert	0	0	1	1	1	1	0	0
Testmaske (74138)	1	1	1	1	0	1	1	1
	0	0	1	1	0	1	0	0

Beispiel 2 für RMB:

Das Bit b3 eines Bytes (jetzt "0") soll auf "Low" gesetzt werden:

	b7<----->b0							
Byte-Wert	X	X	X	X	0	X	X	X
Testmaske (74138)	1	1	1	1	0	1	1	1
	0	0	0	0	0	0	0	0

Anhand der beiden Beispieltabellen erkennt man, dass die bitweise AND-Verknüpfung des Testwerts und der Testmaske die gestellte Anforderung erfüllt:

alle irrelevanten Bits b7, b6, b5, b4, b2, b1, b0 bleiben durch die "1" an der jeweiligen Stelle unverändert. Das Testmaskenbit b3 = 0 erzwingt als Ergebnis immer eine "0".

SMB x (Set Memory Bit x)

Das zu bearbeitende Byte wird mit einer Bit-Maske logisch verknüpft, so dass das gewünschte Bit gesetzt wird und die übrigen Bits nicht beeinflusst werden. Die gewünschte Bitposition ist in dem jeweiligen Befehlsbyte kodiert.

<u>Befehlsbyte</u>	
hex	binär
<u>b7<---->b0</u>	

Befehl	87h	(1000	0111b)	setze Bit 0
Befehl	97h	(1001	0111b)	setze Bit 1
Befehl	A7h	(1010	0111b)	setze Bit 2
Befehl	B7h	(1011	0111b)	setze Bit 3
Befehl	C7h	(1100	0111b)	setze Bit 4
Befehl	D7h	(1101	0111b)	setze Bit 5
Befehl	E7h	(1110	0111b)	setze Bit 6
Befehl	F7h	(1111	0111b)	setze Bit 7

Beispiel:

Befehlsbyte	B7h	setze Bit 3 aus dem Byte
Zero-Page-Adresse	12h	der Adresse \$0012

Beispiel 1 für SMB:

Das Bit b3 eines Bytes (in diesem Fall "0") soll auf "High" gesetzt werden:

	b7<----->b0
Byte-Wert	0 0 1 1 0 1 0 0
<u>Testmaske (74238)</u>	0 0 0 0 1 0 0 0
	0 0 1 1 1 1 0 0

Beispiel 2 für RMB:

Das Bit b3 eines Bytes (jetzt "1") soll auf "High" gesetzt werden:

	b7<----->b0
Byte-Wert	0 0 1 1 1 1 0 0
<u>Testmaske (74238)</u>	0 0 0 0 1 0 0 0
	0 0 1 1 1 1 0 0

Anhand der beiden Beispieltabellen erkennt man, dass die bitweise OR-Verknüpfung des Testwerts und der Testmaske die gestellte Anforderung erfüllt:

alle irrelevanten Bits b7, b6, b5, b4, b2, b1, b0 bleiben durch die "0" an der jeweiligen Stelle unverändert. Das Testmaskenbit b3 = 1 erzwingt als Ergebnis immer eine "1".

Branch On Bit Reset (BBR)

Das zu bearbeitende Byte wird mit einer Bit-Maske logisch verknüpft, so dass die irrelevanten Bits des Testwertes auf "low" gesetzt werden und das relevante Bit entsprechend seines Wertes festgestellt wird. Die gewünschte Bitposition ist in dem jeweiligen Befehlsbyte kodiert.

(Springe wenn Bit = "low", springe nicht wenn Bit = "high")

<u>Befehlsbyte</u>	
hex	binär
<u>b7<--->b0</u>	

Befehl	0Fh	(0000	1111b)	teste Bit 0 auf „0“
Befehl	1Fh	(0001	1111b)	teste Bit 1 auf „0“
Befehl	2Fh	(0010	1111b)	teste Bit 2 auf „0“
Befehl	3Fh	(0011	1111b)	teste Bit 3 auf „0“
Befehl	4Fh	(0100	1111b)	teste Bit 4 auf „0“
Befehl	5Fh	(0101	1111b)	teste Bit 5 auf „0“
Befehl	6Fh	(0110	1111b)	teste Bit 6 auf „0“
Befehl	7Fh	(0111	1111b)	teste Bit 7 auf „0“

Beispiel:

Befehlsbyte	3Fh	Teste Bit 3 aus dem Byte
Zero-Page-Adresse	12h	der Adresse \$0012
Relativer Offset	05h	springe um 5 Adressen vorwärts wenn b3 "low" ist

Beispiel 1 für BBR:

Das Bit b3 eines Bytes (in diesem Fall "0") soll auf "Low" getestet werden:

	b7<----->b0							
Zu testender Wert	X	X	X	X	0	X	X	X
Testmaske (74138)	1	1	1	1	0	1	1	1
	0	0	0	0	1	0	0	0

Das Ergebnis der logischen Verknüpfung ist ungleich "0" und das Z-Bit somit ebenfalls "0", daraus folgt:
Bedingung (b3 = 0) erfüllt, Z-Test = 0, d.h. Sprung

Beispiel 2 für BBR:

Das Bit b3 eines Bytes (jetzt "1") soll auf "Low" getestet werden:

	b7<----->b0							
Zu testender Wert	X	X	X	X	1	X	X	X
Testmaske (74138)	1	1	1	1	0	1	1	1
	0	0	0	0	0	0	0	0

Das Ergebnis der logischen Verknüpfung ist gleich "0" und das Z-Bit damit gleich "1", daraus folgt:
Bedingung (b3 = 0) nicht erfüllt, Z-Test = 1, d.h. kein Sprung.

Anhand der beiden Beispieltabellen erkennt man, dass die bitweise NOR-Verknüpfung des Testwerts und der Testmaske die gestellten Anforderungen erfüllt:

Alle irrelevanten Bits b7, b6, b5, b4, b2, b1, b0 werden durch die "1" an der jeweiligen Stelle ausgeblendet und auf "0" gesetzt. Das Testmaskenbit b3 = 0 bringt als Ergebnis eine 1, d.h. Z-Test = "0" wenn die Bedingung erfüllt (Sprung) bzw. eine "0" d.h. Z-Test = "1" wenn die Bedingung nicht erfüllt ist (kein Sprung).

Die Weiterverarbeitung des Z-Test-Ergebnisses (BI° - Sprung) erfolgt im Kapitel "Automatische Befehlsverarbeitung".

Branch On Bit Set (BBS)

Das zu bearbeitende Byte wird mit einer Bit-Maske logisch verknüpft, so dass die irrelevanten Bits des Testwertes auf "low" gesetzt werden und das relevante Bit entsprechend seines Wertes festgestellt wird. Die gewünschte Bitposition ist in dem jeweiligen Befehlsbyte kodiert.

(Springe wenn Bit = "high", springe nicht wenn Bit = "low")

<u>Befehlsbyte</u>	
hex	binär
<u>b7<---->b0</u>	

Befehl	8Fh	(1000 1111b)	teste Bit 0 auf „1“
Befehl	9Fh	(1001 1111b)	teste Bit 1 auf „1“
Befehl	AFh	(1010 1111b)	teste Bit 2 auf „1“
Befehl	BFh	(1011 1111b)	teste Bit 3 auf „1“
Befehl	CFh	(1100 1111b)	teste Bit 4 auf „1“
Befehl	DFh	(1101 1111b)	teste Bit 5 auf „1“
Befehl	EFh	(1110 1111b)	teste Bit 6 auf „1“
Befehl	FFh	(1111 1111b)	teste Bit 7 auf „1“

Beispiel:

Befehlsbyte	EFh	Teste Bit 6 aus dem Byte
Zero-Page-Adresse	12h	der Adresse \$0012
Relativer Offset	FAh	springe um 5 Adressen rückwärts wenn b6 "high" ist

Beispiel 1 für BBS:

Das Bit b6 eines Bytes (in diesem Fall "1") soll auf "high" getestet werden:

	b7<----->b0							
Zu testender Wert	X	1	X	X	X	X	X	X
Testmaske (74238)	0	1	0	0	0	0	0	0
	0	1	0	0	0	0	0	0

Das Ergebnis der logischen Verknüpfung ist ungleich "0" und das Z-Test somit ebenfalls "0", daraus folgt:
Bedingung (b6 = 1) erfüllt, Z-Test = 0, d.h. Sprung.

Beispiel 2 für BBS:

Das Bit b6 eines Bytes (jetzt "0") soll auf "high" getestet werden:

	b7<----->b0							
Zu testender Wert	X	0	X	X	X	X	X	X
Testmaske (74238)	0	1	0	0	0	0	0	0
	0	0	0	0	0	0	0	0

Das Ergebnis der logischen Verknüpfung ist gleich "0" und das Z-Bit damit gleich "1", daraus folgt:
Bedingung (b6 = 1) nicht erfüllt, Z-Test = 1, d.h. kein Sprung.

Anhand der beiden Beispieltabellen erkennt man, dass die bitweise AND-Verknüpfung des Testwerts und der Testmaske die gestellten Anforderungen erfüllt:

Alle irrelevanten Bits b7, b5, b4, b3, b2, b1, b0 werden durch die "0" an der jeweiligen Stelle ausgeblendet und auf "0" gesetzt. Das Testmaskenbit b6 = 1 bringt als Ergebnis eine 1, d.h. Z-Test = "0" wenn die Bedingung erfüllt (Sprung) bzw. eine "0" d.h. Z-Test = "1" wenn die Bedingung nicht erfüllt ist (kein Sprung).

Die Weiterverarbeitung des Z-Test-Ergebnisses (BI° - Sprung) erfolgt im Kapitel „Automatische Befehlsverarbeitung“.

Manueller Betrieb der ALU/SHIFT - Unit

An Hand des folgenden Beispiels wird gezeigt, wie Befehle schrittweise mit der manuellen Steuerung abgearbeitet werden.

Aufgabe:

Es soll eine Addition durchgeführt werden. Dazu wird der erste Wert in den ACCU geladen, aus dem ACCU-Inhalt und einem zweiten Wert wird die Summe gebildet und diese wieder in den ACCU gespeichert. Die Befehlsfolge in Assembler-Schreibweise lautet:

```
LDA #07 lade ACCU mit dem Wert 07h
ADC #05 addiere ACCU-Inhalt + 05h + Carry-Bit
      Und speichere das Ergebnis in den ACCU
```

Dazu ist die Datei 65AS.PRJ zu laden, die Simulation zu starten, die Bauteile-Bibliotheksanzeige („<<<“) auszuschalten und die Pegelanzeige (HI/LO - Symbol) sowie die Fontplatte zu aktivieren.

Es empfiehlt sich, vor jedem Teilschritt alle Schalter auf „AN“ in die Ruhestellung zu versetzen. Dadurch wird vermieden, dass ein von einem vorhergehenden Schritt noch aktivierter Schalter die nächste Aktion verfahrenswidrig beeinflusst. Man erkennt den vollständigen Ruhezustand der Schaltung daran, dass alle Steuerleitungen auf „H“ stehen.

Auf der Frontplatte sind die Signalzustände für die bessere Lesbarkeit invertiert dargestellt, d.h. alle LEDS sind im Ruhezustand ausgeschaltet („L“). In diesem Zusammenhang wird noch einmal daran erinnert, dass alle Signale „active low“ ausgelegt sind und mit dem Kennzeichen ° versehen wurden. Wenn im Folgenden bei einer Aktion z.B. CLK C° angegeben ist, bedeutet dies, dass der Schalter mit dieser Bezeichnung auf „OFF“ geschaltet und damit aktiviert werden muss. Eine Angabe wie „MUX CCR to 001“ bedeutet, dass der Multiplexer für die Bedingungs-Bits folgendermaßen einzustellen ist: S2 auf 0 (höchstwertige Leitung steht links), S1 auf 0 und S0 auf 1 (niedrigstwertige Leitung steht rechts). Dem großgeschriebene „TO“ innerhalb einer Signalbezeichnung folgt die Zielbezeichnung, an die die Daten weitergeleitet werden. Das kleingeschriebene „to“ bedeutet: „setze das/die angegebene Signal(e) auf die folgenden Werte.“

Mit der HEX-Eingabe und dem darüber liegenden „to BUS°“ - Schalter werden die Daten, die bei automatischem Betrieb aus dem Speicher gelesen werden, auf den Datenbus gelegt.

Die Übersicht für alle Schalterstellungen kann dem Abschnitt „Micro-Code“, Unterabschnitt „ACT“. entnommen werden.
Erster Befehl (LDA), erster Teilschritt:

<u>Aktion</u>	<u>Kommentar</u>
mit der HEX-Eingabe den Wert 07h einstellen	simuliert Speicherzugriff
to BUS°	Ausgabe der Daten auf den Bus
MUX CCR to 011	CCR-Bit-Multiplexer auf Einstellung: Bits vom Bus lesen
CLK N°	N-Bit, und Z-Bit dem Eingangswert
CLK Z°	entsprechend aktualisieren
CLK ACCU°	Wert in ACCU takten

Alle Schalter wieder in die Ruhestellung setzen!!!!

Zweiter Befehl (ADC), erster Teilschritt:

<u>Aktion</u>	<u>Kommentar</u>
mit der HEX-Eingabe den Wert 05h einstellen	simuliert Speicherzugriff
to BUS°	Ausgabe der Daten auf den Bus
CLK ALU IN°	zweiten Wert in ALU-IN takten

Alle Schalter wieder in die Ruhestellung (ON) setzen!!!!

Zweiter Befehl (ADC), zweiter Teilschritt:

<u>Aktion</u>	<u>Kommentar</u>
ACCU TO ALU A°	ACCU-Wert an ALU-Kanal A
ALU-IN TO ALU B°	zweiter Wert an ALU-Kanal B
MUX ALU to 00	Wert des C-Bits bei Addition berücksichtigen
ALU to 01001	ALU auf A + B + Carry
MUX CCR to 011	CCR-Bit-Multiplexer auf Einstellung: Bits vom Bus lesen
CLK N°	N-Bit, Z-Bit und C-Bit dem
CLK Z°	Ergebnis entsprechend
CLK C°	aktualisieren
CLK ACCU°	Ergebnis in ACCU speichern

Alle Schalter wieder in die Ruhestellung (ON) setzen!!!!

In dieser Form lassen sich alle Befehle per Hand ausführen und testen. Die Schalterstellungen für alle Funktionen/Befehle sind im Kapitel „Micro-Code“, Abschnitt „Functions“ aufgeführt.

Adressgenerator

Mit Hilfe des Adressgenerators werden die Adressen für die charakteristischen Adressierungsarten des Prozessors gebildet.

Der Adressgenerator wird anhand der Datei 65C02AG.PRJ erläutert.

Am oberen Rand der Schaltung findet man horizontal den Datenbus, der den Arbeitsspeicher (RAM und ROM oben rechts) mit dem Adressgenerator verbindet. Oben rechts wurden zwei HEX-Eingaben hinzugefügt, sodass die Möglichkeit besteht, bei laufender Simulation Testdaten einzugeben. Um eine Datenkollision mit Werten aus dem RAM/ROM zu verhindern, wurde der Arbeitsspeicher zusätzlich mit einem Bustreiber vom Datenbus abgekoppelt. Mit „DATABUS PRESET“ auf high (default) ist der Arbeitsspeicher aktiv, steht der Schalter auf low, können über die HEX-Eingabe Testdaten auf den Datenbus ausgegeben werden.

Der Datenbus wird am linken Rand der Schaltung über zwei bidirektionale Bustreiber (75245) in den Low-Bus und darunter in den High-Bus des Adressgenerators aufgeteilt. Man erkennt die beiden Strukturen an den mit doppeltem Abstand angelegten horizontalen Leitungen.

Adressierungsarten

Im Folgenden werden zunächst an Hand jeweils eines exemplarischen Beispiels die Adressierungsarten des 65C02 und ihre Zerlegung in Teilschritten erläutert. Die hier vorgenommenen groben Unterteilungen dienen dazu, die erforderlichen Strukturen, die Voraussetzung für die Implementierung aller Adressierungsarten sind, zu erkennen.

Adressen werden mit „\$“ oder eckigen Klammern, Werte mit geschweiften Klammern, die indirekte Adressierung durch das Einschließen der Adressierungsmethode in runde Klammern „()“ gekennzeichnet. „Absolut“ steht für eine endgültige (direkte) Adresse.

Implied: IMP

Bei der impliziten Adressierung handelt es sich um Ein-Byte-Befehle (ohne Argument-Byte), in dem bereits alle Informationen über die auszuführenden Aktionen kodiert sind.

Accumulator: A

Obwohl es sich ebenfalls um Ein-Byte-Befehle gemäß der Definition von Implied-Befehlen handelt, werden Befehle, die den Inhalt des ACCUs beeinflussen, separat in dieser Kategorie geführt.

Immediate: IMM

Die unmittelbar auf das Befehlsbyte folgende Adresse enthält das zu ladende Argument. Dabei handelt es sich um einen Wert und nicht um eine Adresse. Der Programmzähler PC wurde bereits beim Laden und Dekodieren des Befehls auf diese Adresse eingestellt und adressiert das Argument, das in ein Register des Prozessors übertragen wird. In der Literatur wird diese Adressierungsart oftmals als „direkt“ bezeichnet, da das Argument direkt (ohne Angabe einer Speicheradresse) zu Verfügung steht. Dies führt jedoch u.U. zu falschen Schlüssen. Steht nach dem Befehlsbyte als Argument eine Adresse, die nicht das Zielbyte sondern zunächst wiederum eine Adresse adressiert, die dann ihrerseits das Zielbyte adressiert, spricht man von indirekter Adressierung. Ausgehend von dieser in der Mikroprozessorwelt üblichen Bezeichnung müssen Adressierungsarten, bei denen im Befehl eine Adresse angegeben ist, die das Zielbyte ohne Umweg adressiert, als direkt bezeichnet werden. Dazu gehören die absolute Adressierung und die Zero-Page-Adressierung aber auf keinen Fall die Immediate-Adressierung, die deshalb als „unmittelbare Adressierung“ bezeichnet werden sollte.

Absolute: ABS

Aus dem zweiten (LB) und dem dritten Byte (HB) des Befehls wird die absolute Adresse gebildet.

Befehl:

0208	AD	LDA \$2030
0209	30	LB
020A	20	HB

Die endgültige Adresse ist 2030h

Schritt 0:

PC [0209] auf AB
LB {30} in TEMP
PC auf nächste Adresse erhöhen [020A]

Schritt 1:

PC [020A] auf AB
TEMP {30} auf Bus

HB {20} auf High-Bus
LB {30} aus TEMP auf Low-Bus

AR laden
PC auf nächsten Befehl erhöhen [020B]

Absolute Indexed: ABS,X oder ABS,Y

Die absolute Adresse im zweiten und dritten Befehlsbyte (LB, HB), erhöht um den Inhalt des X- oder Y-Registers, ergibt die endgültige Adresse. Der Übertrag vom LB-Addierer auf den High-Byte-Addierer wird berücksichtigt.

Wert im X-REG:

02

Befehl:

```
0208    BD    LDA $2030,X
0209    30    LB
020A    20    HB
```

Die endgültige Sprungadresse für X-REG = 2 ist 2032h

Schritt 0:

PC [0209] auf AB

LB {30} in TEMP

PC auf nächste Adresse erhöhen [020A]

Schritt 1:

PC [020A] auf AB

HB {20} an ADD HB B

TEMP {30} an ADD LB B

X {02} an ADD LB A

{00} an ADD HB A

Addierer Eingang B 20 30

Addierer Eingang A 00 02

=====

Summe in AR laden 20 32

PC auf nächsten Befehl erhöhen [020B]

Zero Page: ZP

Aus dem zweiten Byte (LB) des Befehls und einer vom Prozessor vorgegeben „00“ für das HB wird die absolute Adresse gebildet.

Befehl:

0208	A5	LDA \$30
0209	30	LB

Die endgültige Adresse ist 0030h

Schritt 0:

PC [0209] auf AB
LB {30} auf Low-Bus
HB „00“ aus Prozessor auf High-Bus

AR laden
PC auf nächste Adresse erhöhen [020A]

Zero Page Indexed: ZP,X oder ZP,Y

Aus dem zweiten Byte (LB) des Befehls und einer vom Prozessor vorgegeben „00“ für das HB, erhöht um den Inhalt des X- oder Y-Registers, wird die endgültige Adresse gebildet. Der Übertrag vom LB-Addierer auf den High-Byte-Addierer wird nicht berücksichtigt.

Wert im X-REG:
02

Befehl:

0208	B5	LDA \$30,X
0209	30	LB

Die endgültige Sprungadresse für X-REG = 2 ist 0032h

Schritt 0:

PC [0209] auf AB

LB	{30}	an ADD LB A
	{00}	an ADD HB A
X	{02}	an ADD LB B
	{00}	an ADD HB B

Addierer Eingang A	00 30
Addierer Eingang A	00 02
	=====
Summe in AR laden	00 32

PC auf nächsten Befehl erhöhen [020A]

Absolute Indirect: (ABS) (1/2)

Die absolute Adresse im zweiten (LB) und dritten (HB) Byte des Befehls, stellt eine indirekte Adresse dar, in der das Low-Byte der endgültigen Adresse enthalten ist. Auf der folgenden indirekten Adresse liegt das High-Byte der endgültigen Adresse. Diese Adressierungsart gilt nur für den Sprungbefehl JMP (ABS) {6C}.

Befehl:

0208	6C	JMP (2032)
0209	32	LB indirekt
020A	20	HB indirekt

Speicherauszug \$2030 - \$2035:

2030	12	LB Sprungadresse
2031	1A	HB Sprungadresse
2032	17	LB Sprungadresse
2033	2F	HB Sprungadresse
2034	23	LB Sprungadresse
2035	65	HB Sprungadresse

Die endgültige Sprungadresse ist 2F17h

Absolute Indirect (ABS) (2/2)

Schritt 0:

PC [0209] auf AB
LB indirekt {32} in TEMP

PC auf nächste Adresse erhöhen [020A]

Schritt 1:

PC [020A] auf AB
HB indirekt {20} an AR HB
TEMP {32} an AR LB

AR laden

Schritt 2:

AR [2032] auf AB
LB Sprungadresse {17} in TEMP
AR auf [2033] erhöhen

Schritt 3:

AR [2033] auf AB
HB Sprungadresse {2F} an PC HB
TEMP {17} an PC LB

PC laden

Absolute Indexed Indirect: (ABS,X) (1/2)

Die absolute Adresse im zweiten (LB) und dritten (HB) Byte des Befehls, erhöht um den Inhalt des X-Registers, ergibt eine indirekte Adresse, in der das Low-Byte der endgültigen Adresse enthalten ist. Auf der folgenden indirekten Adresse liegt das High-Byte der endgültigen Adresse. Diese Adressierungsart gilt nur für den Sprungbefehl JMP (ABS,X) {7C}. Der Übertrag vom LB-Addierer auf den High-Byte-Addierer wird berücksichtigt.

Wert im X-REG:

02

Befehl:

0208	7C	JMP (2030,X)
0209	30	LB indirekt
020A	20	HB indirekt

Speicherauszug \$2030 - \$2035:

2030	12	LB Sprungadresse für X = 00
2031	1A	HB Sprungadresse für X = 00
2032	17	LB Sprungadresse für X = 02
2033	2F	HB Sprungadresse für X = 02
2034	23	LB Sprungadresse für X = 04
2035	65	HB Sprungadresse für X = 04

Die endgültige Sprungadresse für X-REG = 2 ist 2F17h

Absolute Indexed Indirect (ABS,X) (2/2)

Schritt 0:

PC [0209] auf AB
LB indirekt {30} in TEMP
PC auf nächste Adresse erhöhen [020A]

Schritt 1:

PC [020A] auf AB
HB indirekt {20} an ADD HB B
TEMP {30} an ADD LB B
X {02} an ADD LB A
Vektor ZP {00} an ADD HB A

Addierer Eingang B	20 30
Addierer Eingang A	00 02
	=====
Summe in AR laden	20 32

Schritt 2:

AR [2032] auf AB
LB Sprungadresse {17} in TEMP
AR auf [2033] erhöhen

Schritt 3:

AR [2033] auf AB
HB {2F} an PC HB
TEMP {17} an PC LB

AR laden
PC auf nächste Adresse erhöhen [020B]

Zero Page Indirect: (ZP) (1/2)

Die Zero Page Adresse im zweiten Byte des Befehls (LB) zeigt aus eine indirekte Adresse im Zero-Page-Bereich, in der die endgültige Adresse (Low Byte) enthalten ist. In der darauf folgenden indirekten Adresse ist das High Byte der endgültigen Adresse enthalten.

Befehl:

0208	B2	LDA (92)
0209	92	Zero Page Adresse

Speicherauszug Zero-Page \$0090 - \$0095

0090	0A	LB Sprungadresse
0091	02	HB Sprungadresse
0092	0B	LB Sprungadresse
0093	02	HB Sprungadresse
0094	0C	LB Sprungadresse
0095	02	HB Sprungadresse

Die endgültige Adresse ist 020Bh

Zero Page Indirect: (ZP) (2/2)

Schritt 0:

PC [0209] auf AB
Indirekte Adresse LB {92} auf Low-Bus
HB „00“ aus Prozessor auf High-Bus

AR laden

Schritt 1:

AR [0092] auf AB

LB direkt {0B} in TEMP

Schritt 2:

AR auf HB indirekt [0093] erhöhen

Schritt 3:

AR [0093] auf AB
HB Adresse {02} an AR HB
TEMP {0B} an AR LB

AR laden
PC auf nächste Adresse erhöhen [020A]

Zero Page Indexed Indirect: (ZP,X) (1/2)

Die Zero Page Adresse im zweiten Byte des Befehls (LB), erhöht um den Inhalt des X-Registers, zeigt auf eine indirekte Adresse im Zero-Page-Bereich, in der das LB der endgültigen Adresse enthalten ist. In der darauf folgenden Zero-Page-Adresse ist das HB der indirekten Adresse enthalten. Der Übertrag vom LB-Addierer auf den High-Byte-Addierer wird nicht berücksichtigt.

Wert im X-REG:

02

Befehl:

0208	A1	LDA (90,X)
0209	90	Zero Page Adresse

Speicherauszug Zero-Page \$0090 - \$0095

0090	0A	LB Sprungadresse für X = 00
0091	02	HB Sprungadresse für X = 00
0092	0B	LB Sprungadresse für X = 02
0093	02	HB Sprungadresse für X = 02
0094	0C	LB Sprungadresse für X = 04
0095	02	HB Sprungadresse für X = 04

Die endgültige Sprungadresse für X-REG = 2 ist 020Bh

Zero Page Indexed Indirect: (ZP,X) (2/2)

Schritt 0:

PC [0209] auf AB

LB	{90}	an	ADD	LB	A
	{00}	an	ADD	HB	A
X	{02}	an	ADD	LB	B
	{00}	an	ADD	HB	B

Addierer Eingang A	00 90
Addierer Eingang A	00 02
	=====
Summe in AR laden	00 92

Schritt 1:

AR [0092] auf AB

LB direkt {0B} in TEMP

Schritt 2:

AR auf HB indirekt [0093] erhöhen

Schritt 3:

AR [0093]	auf	AB
HB Adresse {02}	an	AR HB
TEMP {0B}	an	AR LB

AR laden

PC auf nächste Adresse erhöhen [020A]

Zero Page Indirect Indexed (ZP),Y (1,2)

Die Zero-Page-Adresse im zweiten Byte des Befehls zeigt auf das LB einer indirekten Adresse im Zero-Page-Bereich. Das HB der indirekten Adresse ist in der folgenden Zero-Page-Adresse enthalten. Die endgültige Adresse ergibt sich aus dem Wert der indirekten Adresse, erhöht um den Wert im Y-Register. Der Übertrag vom LB-Addierer auf den High-Byte-Addierer wird berücksichtigt.

Wert im Y-REG:

02

Befehl:

0208	A1	LDA (92),Y
0209	92	Zero Page Adresse

Speicherauszug Zero-Page \$0090 - \$0095

0092	0B	LB Adresse
0093	02	HB Adresse

Speicherauszug Zero-Page \$020B - \$020D

020B	xx	Adresse für Y-REG = 0
020C	yy	Adresse für Y-REG = 1
020D	zz	Adresse für Y-REG = 2

Die endgültige Sprungadresse für Y-REG = 2 ist 020Dh

Zero Page Indirect Indexed (ZP),Y (2,2)

Schritt 0:

PC [0209] auf AB
Indirekte Adresse LB {92} auf Low-Bus
HB „00“ aus Prozessor auf High-Bus

AR laden {0092}

Schritt 1:

AR [0092] auf AB

LB direkt {0B} in TEMP

Schritt 2:

AR auf HB direkt [0093] erhöhen

Schritt 3:

AR [0093] auf AB

HB	{02}	an	ADD	HB	A
TEMP	{0B}	an	ADD	LB	A
Y	{02}	an	ADD	LB	B
	{00}	an	ADD	HB	B

Addierer Eingang A 02 0B
Addierer Eingang B 00 02
 =====

Summe in AR laden 0B 92

PC auf nächste Adresse erhöhen [020A]

Relativ 1-Byte-Offset (1/2)

Das zweite Byte im Befehl bestimmt mit 7 Bit die Sprungweite bezogen auf den Stand des Programmzählers. Dieser steht vor Beginn der Berechnung bereits auf dem folgenden Befehlsbyte. Für Vorwärtssprünge ist das höchstwertige Bit gleich „0“, die übrigen sieben Datenbits enthalten die Sprungweite (maximal 127) und werden zu dem Wert des Programmzählers addiert. Zur Unterscheidung von Vorwärts- und Rückwärtssprüngen muss der Wert für einen Rückwärtssprung im Zweier-Komplement angegeben werden, der dann für die Sprungberechnung ebenfalls addiert wird. Das höchstwertige Bit ist in diesem Fall immer „1“, die maximale Sprungweite liegt bei -128. Um auch relative Sprungbefehle mit 2-byte-Offset realisieren zu können, wird bei diesem System ein 16-Bit-Addierer für die Adressberechnung eingesetzt (s.u.). Das hat zur Folge, dass bei 8-Bit-Additionen die Bits D8 - D15 entsprechend der Sprungrichtung gesetzt werden müssen, um ein korrektes Ergebnis zu erhalten (00 bei Vorwärts- und FFh bei Rückwärtssprüngen). Das Übertragsbit wird ignoriert!

Befehl:

0207	88	DEY
0208	B0	BCS #FD
0209	FD	Sprungweite (3 Schritte rückwärts)
020A	—	nächstes Befehlsbyte

Berechnung der Sprungweite:

Programmzähler vor Sprung	020A
Offset addieren	FFFD
	====
Programmzähler nach Sprung (1)	0207

Relativ 1-Byte-Offset (2/2)

Schritt 0:

PC [0209] auf AB

Offset {FD} in TEMP laden

PC auf nächste Adresse erhöhen [020A]

Schritt 1:

FF {00} an ADD HB A (FFh bei Rückwärtssprung !)

TEMP {FD} an ADD LB A

PC LB {02} an ADD LB B

PC HB {00} an ADD HB B

Addierer Eingang A FF FD

Addierer Eingang B 02 0A

=====

Summe in PC laden (1)02 07

Relativ 2-Byte-Offset (1/2)

Diese Adressierungsart ist kein Bestandteil des Original 65C02! Das zweite Byte (LB) und das dritte Byte im Befehl bestimmt mit 15 Bit die Sprungweite bezogen auf den Stand des Programmzählers. Dieser steht vor Beginn der Berechnung bereits auf dem folgenden Befehlsbyte. Für Vorwärtssprünge ist das höchstwertige Bit gleich „0“, die übrigen 15 Datenbits enthalten die Sprungweite (maximal 32767) und werden zu dem Wert des Programmzählers addiert. Zur Unterscheidung von Vorwärts- und Rückwärtssprüngen muss der Wert für einen Rückwärtssprung im Zweier-Komplement angegebenen werden, der dann für die Sprungberechnung ebenfalls addiert wird. Das höchstwertige Bit ist in diesem Fall immer „1“, die maximale Sprungweite liegt bei -32768. Das Übertragsbit wird ignoriert.

Befehl:

0207	88	DEY
0208	B3	BCS #FEF9
0209	F9	Sprungweite LB
020A	FE	Sprungweite HB
020B	—	nächstes Befehlsbyte

Berechnung der Sprungweite:

Programmzähler vor Sprung	020B
Offset addieren	FEF9
	====
Programmzähler nach Sprung (1)	0104

Relativ 2-Byte-Offset (2/2)

Schritt 0:

PC [0209] auf AB
LB Offset {FF} in TEMP

PC auf nächste Adresse erhöhen [020A]

Schritt 1:

PC [020A] auf AB

OFFSET HB	{FE}	an	ADD HB A
TEMP	{FD}	an	ADD LB A
PC LB	{0B}	an	ADD LB B
PC HB	{02}	an	ADD HB B

Addierer Eingang A	FE	FD
Addierer Eingang B	02	0A *
	=====	
Summe in PC laden	(1)01	0A

* Mit dem Programmzähler = 020Ah wird das Offset-HB adressiert und direkt für die Summenbildung an den Addierer geleitet, da für das HB keine Möglichkeit zur temporären Speicherung besteht und der PC vor der Addition nicht auf die Adresse des folgenden Befehlsbytes inkrementiert werden kann (Referenzadresse). Aus diesem Grund erfolgt die Summenbildung mit dem Stand des nicht inkrementierten PCs. Die Korrektur des PCs erfolgt nachträglich im Schritt 2!

Schritt 2:

PC zur Korrektur inkrementieren

Struktursynthese

Aus der Unterteilung der Adressierungsarten in kleine Teilschritte lässt sich die Anordnung der notwendigen Bauteile ableiten. Die 8-Bit-Register (X-, Y- und TEMP) erhalten ihre Daten vom Low Bus und geben sie auch wieder auf ihn aus. Über den Low Bus können die Daten zur Adressmanipulation auf den Eingang A des Adressaddierers geleitet werden, über einen separaten Bus an den Eingang B.

Programmzähler PC, Adressregister AR und der Stackpointer SP erhalten ihre Daten für das Low Bus über den Low Bus, PC und AR das High Byte über den High Bus. Das High Byte des Stackpointers ist fix auf 01h eingestellt. Alle drei geben ihre Daten zur Adressierung auf den Adressbus aus. Der Programmzähler hat jeweils eine Verbindung zum Low- und High-Bus, um seinen Zählerstand bei Programmunterbrechungen auf den Stack zwischenspeichern zu können. Der SP hat ebenfalls die Möglichkeit sein LB auf den Bus auszugeben (TSX-Befehl).

Für die Berechnung neuer Adressen bei Sprungbefehlen muss auf den aktuellen Programmzählerstand ein Wert (z.B. relativer Offset) addiert werden. Aus diesem Grund kann der Inhalt des PCs auf den Eingang des Adressaddierers ausgegeben werden. Auch der Ausgang des Adressregisters hat eine Verbindung zum Adressaddierer, um geladene Adressen mit den Werten von X- bzw. Y-Register modifizieren zu können.

Der Adressaddierer erhält seine Daten vom Bus, vom Programmzähler und vom Adressregister und gibt sein Ergebnis an den PC und das AR zurück. Für die Addition einer 16-Bit-Adresse und einem 8-Bit-Wert aus dem X-, Y- bzw. TEMP-Register kann das High Byte beim Addiervorgang auf „00“ gesetzt werden. Für einige Adressierungsarten wird der Übertrag vom Low- zum High Byte bei der Addition nicht berücksichtigt, sodass dieser durch ein geschaltetes AND-Gatter ignoriert werden kann.

Vektorgenerator

Unterhalb der beiden Bustreiber liegt ein aus mehreren Bauteilen bestehender Funktionsblock, über den vom Hersteller vorgegebene, fest eingestellte (Teil)-Adressen (Resetvektor, Interruptvektoren usw.) ausgewählt werden können. Da beim 65C02 nur relativ wenige Adressen erforderlich sind, wurden die Werte über beschaltete Bustreiber und eine explizite Dekodierung realisiert. Für Systeme, die eine Vielzahl an Vektoradressen benötigen (Timer-Interrupts, Watchdog usw.), ist es sinnvoll, die Adressen in einem ROM abzulegen.

Reset

Nach einem Reset muss der Programmzähler mit der Anfangsadresse des Anwenderprogramms initialisiert werden. Der Prozessor lädt während seiner Reset-Sequenz diese Startadresse aus einer definierten Adresse innerhalb des Arbeitsspeichers und überträgt sie in den Programmzähler. In einem realen System muss die Startadresse in einem Festwertspeicher (ROM, EPROM, EEPROM) abgelegt sein, damit sie beim Abschalten des Systems nicht verloren geht. Beim 65C02 ist der Vektor für die Startadresse auf FFFCh (LB) und FFFD (HB) festgelegt.

NMI

Liegt ein NMI vor, wird nach Beendigung des laufenden Befehls die NMI-Sequenz ausgeführt, die aus den Adressen FFFAh (LB) und FFFB (HB) die Startadresse für die NMI-Routine in den Programmzähler lädt.

INT

Wurde eine Interrupt-Anforderung registriert und ist das I-Bit auf „low“ (Interrupt enabled), so wird nach Beendigung des laufenden Befehls die INT-Sequenz ausgeführt, die aus den Adressen FFFEh (LB) und FFFF (HB) die Startadresse für die INT-Routine in den Programmzähler lädt.

Stackpointer

Der Stapelspeicher (Stackpointer, SP) adressiert einen innerhalb des Arbeitsspeichers reservierten Bereich, in dem bei Programmunterbrechungen (Interrupts, Unterprogrammaufrufe) die Rücksprungadressen und ggfs. Registerinhalte gesichert werden. Beim 65C02 liegt der Stackbereich zwischen 0100h - 01ffh und kann, im Gegensatz zu anderen Prozessoren, vom Anwender nicht frei gewählt (User-Stack) oder verschoben werden. Aus diesem Grund ist das High-Byte durch einen entsprechend beschalteten Bustreiber fest auf 01h eingestellt. Während der Reset-Sequenz muss der Zähler, der das variable Low Byte beinhaltet, über den Low Bus auf den Wert FFh initialisiert werden.

SWI

Bei dem SWI-Befehl handelt es sich um eine über den Standardbefehlssatz des 65C02 hinausgehende Erweiterung. Der Befehl für einen Softwareinterrupt setzt sich aus dem Befehlsbyte 02h, gefolgt von einer Interruptnummer, zusammen. Die Interruptnummer adressiert einen Speicherplatz in der Address-Page 02 (HB), in der das LB der auszuführenden Interruptroutine abgelegt sein muss. Auf der ihr folgenden Adresse muss das HB der Interruptroutine liegen. Das bedeutet, dass der Befehl SWI 00 die Startadresse aus den Speicherstellen 0200h (LB) und 0201h (HB) und der Befehl SWI 01 die Startadresse der Interruptroutine aus den Speicheradressen 0202h (LB) und 0203 (HB) lädt. Somit sind 128 Software-Interrupts (00h - 7F), deren zugehörigen Vektoradressen auf den gerade Adressen abgelegt sein müssen (0200h, 0202h, 0204h, usw.), möglich.

Der SWI-Befehl hat keine bevorzugte Ausführungspriorität. Er besitzt dieselbe Wertigkeit wie alle anderen Standardbefehle. Bei der Initialisierungssequenz muss für die Page-2-Adressierung als HB der Wert 02 ausgegeben werden, der zusammen mit dem Argument des SWI-Befehls, der Interruptnummer, die Vektor-Adresse bildet.

High-Byte bei relativem Sprung mit 1 Byte Offset (R8)

Beim Original-65C02 sind nur relative Sprünge mit einem Byte Offset möglich. Sieben Bits des Offset-Bytes bestimmen die Sprungweite, das 8. Bit (D7) gibt die Sprungrichtung an. Ist D7 gleich 0 (positiver Wert, wird der Sprung vorwärts ausgeführt (Programmzähler + Offset), ist D7 gleich 1 (negativer Wert) erfolgt der Sprung rückwärts (Programmzähler - Offset). Die Addition bzw. Subtraktion erfolgt vermutlich über die 8-Bit-ALU, da bei einem Seitenüberlauf ein weiterer Takt für die Befehlsausführung erforderlich ist (Korrektur des Programmzählers bei Carry-Überlauf).

Bei dem hier vorgestellten System wurden auch relative Sprünge mit 2-Byte-Offset implementiert. Dazu (und zur Vermeidung von Korrekturtakten, s.o.) wird ein separater 16-Bit-Adress-Addierer verwendet, mit dem absolute und relative 16-Bit-Adress-Operationen möglich sind. Um aber die Originalbefehle für relative Sprünge mit einem Byte Offset auf der 16-Bit-Struktur korrekt ausführen zu können, muss das 1-Byte Offset auf zwei Byte erweitert werden und zwar mit Nullen im HB bei Vorwärtssprüngen und Einsen im HB bei Rückwärtssprüngen (Zweier Komplement!). Beim Vorliegen eines R8-Befehls (relativ 8-Bit) muss also durch die Auswertung von Bit D7 das HB entweder auf 00 oder FF gesetzt werden.

Zero Page

Bei Zero-Page-Befehlen folgt hinter dem Befehlsbyte ein Byte mit der LB-Adresse in der Zero-Page. Das System muss den Wert „00“ auf dem High-Bus zu Verfügung stellen, um eine 16-Adresse zu erzeugen.

Zusammenfassung

In der folgenden Tabelle sind alle für den Betrieb erforderlichen Adressen zusammengefasst, aufgeteilt nach High- und Low-Bus bzw. Nibble (Halbbyte) und den für die jeweiligen Zustände festgelegten Schalterstellungen:

		HB	HB	LB	LB	
74238 IN	74238 OUT	N4	N3	N2	N1	
111	Y7	-	-	-	-	HZ (no output)
110	Y6	F	F	F	E	INT
101	Y5	F	F	F	C	RESET
100	Y4	F	F	F	A	NMI
011	Y3	-	-	F	F	STACK
010	Y2	0	2	-	-	SWI
001	Y1	0	0	-	-	R8 (D7 = 0)
001	Y1	F	F	-	-	R8 (D7 = 1)
000	Y0	0	0	-	-	ZERO PAGE

Die drei Schalter **VECTOR S2°**, **S1°** und **S0** werden auf die Eingänge A, B und C eines 74238 3-zu-8-Dekoders geführt, der für jede Schalterkombination einen spezifischen Ausgang aktiviert (high). An Hand dieses Signals werden mittels der nachgeschalteten Gatter für alle vier Adress-Nibble die entsprechenden Bustreiber mit ihren voreingestellten Werten frei geschaltet. Zwischen dem Low-Bus und dem High-Bus befinden sich drei Bustreiber, von denen die zwei oberen die Werte Fh, Eh, Ch und Ah auf das Adress-Nibble 1 ausgeben. Aus der oberen Hälfte des dritten Treiber wird Fh für das Nibble 2 bereitgestellt. Da das untere Halbbyte des dritten Treibers für den LB-Bus-Bereich nicht mehr benötigt wird, findet er für die Ansteuerung des Nibble 3 Verwendung (2H für SWI).

Die beiden Treiber unterhalb des HB-Busses schalten nach Bedarf jeweils 0h bzw. Fh auf Nibble 3 und Nibble 4.

Für R8-Befehle (Y1) wird über die Auswertung von Bit D7 des Low Busses bei relativen Sprüngen mit einem Byte Offset der jeweils korrekte Wert (00h oder FFh) als Ergänzung zur 16-Bit-Breite selektiert.

Y- und X-Register

Im Anschluss an den Vektor-Generator sind am Low-Bus die 8-Bit breiten Y- und X-Register platziert. Sie sind durch zwei 74273 realisiert. Die Daten der beiden Register werden wahlweise auf den Bus (PUSH-Befehle) und an den Addierer geleitet (LDA ZP,X bzw. LDA ZP,Y). Darüber hinaus besteht eine direkte Verbindung zur ALU, sodass für das Inkrementieren und Dekrementieren des Registerinhalts der Umweg über den Bus und ALU-REG-IN und damit ein zusätzlicher Takt eingespart werden kann. Da dieser Vorgang nun innerhalb eines Taktes abgearbeitet wird, wurde zur Vermeidung von Dateninkonsistenzen (s. Ringbefehle) jeweils ein weiteres Auffangregister nachgeschaltet, dass die neuen Daten mit einem halben Takt Verzögerung übernimmt (zeitliche Entkopplung der Datenausgabe und Eintakten neuer Daten, Master/Slave-Funktion).

TEMP-Register

Das TEMP-Register dient beim Laden zweier aufeinander folgender Bytes aus dem Arbeitsspeicher zur Bildung einer 16-Bit Adresse zur Zwischenspeicherung des zuerst geladenen Low Bytes.

Adressaddierer

Der 16-Bit-Adressaddierer ADD (je 2* 4-Bit Addiererbaustein 74283 für Low- und High-Bus) dient der Berechnung neuer Adressen bei indexed und indirekter Adressierung oder Sprüngen. Die jeweiligen Übertrag-Bits werden standardmäßig auf den entsprechenden Eingang des nächsten, höherwertigen Zählerbausteins geführt. Da aber bei manchen Adressierungsarten das Übertragsbit zwischen LB und HB nicht berücksichtigt wird, kann der Übertrag durch das Low-Schalten des AND-Gatters bei Bedarf unterbunden werden.

Adressregister

Im 16-Bit breiten Adressregister AR (je 2* 4-Bit ladbare Zählerbaustein 74163 für Low- und High-Bus) werden die aufbereiteten Adressen zwischengespeichert. Da bei dieser Funktion immer die aufbereitete Adresse sowie die darauf folgende Adresse auf den Adressbus ausgegeben werden, genügen

an dieser Stelle einfache Aufwärtzzähler als Speicher. Die jeweiligen Übertrag-Bits werden auf den entsprechenden Eingang des nächsten, höherwertigen Zählerbausteins geführt. Da der AR-Inhalt zur Adressmodifikation auf den Adressaddierer geleitet und dessen Ergebnis zurück in den Programmzähler geschrieben werden kann, wurde zur Vermeidung von Dateninkonsistenzen (s. Ringbefehle) ein Auffangregister nachgeschaltet, dass die neuen Programmzählerdaten mit einem halben Takt Verzögerung übernimmt (zeitliche Entkopplung der Datenausgabe und Eintakten neuer Daten).

Programmzähler

Der 16-Bit breite Programmschrittzähler (Program-Counter, PC), der mit seinem Inhalt das aktuelle Byte des auszuführenden Anwendungsprogramms adressiert, ist aus vier ladbaren 4-Bit Zählern vom Typ 74163 für Low- und High-Bus aufgebaut. Da bei der linearen Programmverarbeitung der Zähler ausschließlich inkrementiert wird, genügen an dieser Stelle einfache Aufwärtzzähler als Speicher. Die jeweiligen Übertrag-Bits werden auf den entsprechenden Eingang des nächsten, höherwertigen Zählerbausteins geführt. Da der PC-Inhalt zur Adressmodifikation auf den Adressaddierer geleitet und dessen Ergebnis zurück in den Programmzähler geschrieben werden kann, wurde zur Vermeidung von Dateninkonsistenzen (s. Ringbefehle) ein Auffangregister nachgeschaltet, dass die neuen Programmzählerdaten mit einem halben Takt Verzögerung übernimmt (zeitliche Entkopplung der Datenausgabe und Eintakten neuer Daten).

Stackpointer

Das High-Byte des Stackpointers ist durch einen entsprechend beschalteten Bustreiber fest auf 01h eingestellt. Da der Stackpointer die zu sichernden Werte von 01ffh abwärts speichert und die Werte beim Wiederherstellen des ursprünglichen Zustands in aufsteigender Richtung adressiert werden (LIFO), ist hier der Einsatz von 2 ladbaren Auf-/Abwärtzzähler (74169), die das Low Byte darstellen, erforderlich. Um den Stackpointer während seiner Ausgabephase inkrementieren bzw. dekrementieren zu können (s. Micro-Code), werden seinem CLK-Eingang zur Vermeidung von Dateninkonsistenzen zwei zusätzliche Inverter als Verzögerungen vorgeschaltet (zeitliche Entkopplung des Lese- und Schreibvorgangs).

PC CLK bei INT und NMI

Die Auswertung von NMI und INT wird bei diesem System nicht während des letzten Execution-Cycles sondern während des Fetch-Cycles durchgeführt. Da der Programmzähler im Fetch-Cycle F2 inkrementiert wird, würde zu Beginn der Interruptsequenz eine falsche Rücksprungadresse auf den Stack abgespeichert werden. Durch den Einsatz von Vor-/Rückwärtszähler für den Program-Counter könnte der PC-Stand während der INT-Initialisierungssequenz durch einmaliges Dekrementieren korrigiert werden. Hier wird das Problem gelöst, indem der Fetch-Cycle analysiert und das CLK PC^o - Signal bei einem NMI bzw. INT unterdrückt wird.

RAM

Die Eigenschaften des RAM-Bausteins aus der ProfiLab-Bauteile-Bibliothek entsprechen nicht denen eines realen statischen Speichers wie zum Beispiel eines 2114. Es fehlt ihm die Möglichkeit an einem Bus betrieben zu werden, ein CS° - Eingang ist nicht vorhanden und seine Ausgänge lassen sich somit nicht in den hochohmigen Tri-State-Zustand schalten. Darüber hinaus wird zur einfacheren Handhabung des Timings der Schreibvorgang nicht wie bei einem realen Baustein zustandsgesteuert während $\text{WE}^\circ = \text{low}$ sondern mit einer (fallenden) Flanke durchgeführt.

Auch mit den eingeschränkten Möglichkeiten eines Ein-Phasen-Taktes mit der Vereinbarung

steigende Flanke = Adress- und Datenausgabe bis zum Ende des Taktes und


fallende Flanke = stabilisierte Daten schreiben (Eintakten)

kann das erforderliche Timing sowohl für den Simulationsbetrieb als auch mit Einschränkungen für den Betrieb mit einem echten RAM realisiert werden. Dazu wird das Original-RAM der Bibliothek um einige Bauteile erweitert und in einem Makro mit den zusätzlichen Pins WE° (Write Enable) und CS° (Chip Select) angespeichert.

RAM im Simulationsbetrieb

Für das Schreiben in den Arbeitsspeicher (RAM) muss der Prozessor das Signal **WE°** generieren. Dazu wird die Steuerleitung **R/W°** für die Dauer eines Taktzyklus auf „low“ gesetzt. Durch die ODER-Verknüpfung von **R/W°** mit **Clock** entsteht das erforderliche Signal **WE°** für den Schreibvorgang. Der Pegel ist während der ersten Takthälfte, in der die Adresse und die Daten auf den jeweiligen Bus gegeben werden, „high“ und liefert bei T/2, wenn alle Daten stabil anstehen, die fallende Flanke für das Schreiben. Die Daten bleiben bis zum Taktende, also über den Zeitpunkt des Schreibens hinaus, aktiv. Ist die **R/W°**-Leitung „high“ (Lesebetrieb) so bleibt **WE°** für den gesamten Takt ebenfalls „high“.

SCHREIBEN:

Clock 


R/W° 

Clock OR R/W°:


WE° 

Lesen:

Clock 

R/W° 
.....

Clock OR R/W°:

WE° 
.....

Der RAM-Baustein wird auf der Ausgangsseite durch einen Bustreiber erweitert, sodass die Daten gesteuert auf den Bus ausgegeben werden können. Der Treiber soll nur aktiviert werden wenn der RAM-Baustein selektiert ist und Daten gelesen werden sollen (**CS°** = „low“ und **WE°** = „high“). Beim Schreiben der Daten (**CS°** und **WE°** = „low“) muss der Bustreiber deaktiviert sein.

RAM-Bustreiber

WE°	CS°	OUT
0	0	HZ
0	1	HZ
1	0	READ
1	1	HZ

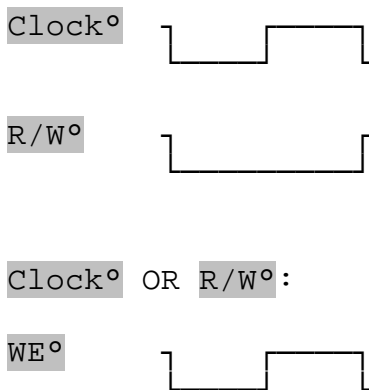
Aus der Funktionstabelle ist zu erkennen, dass das Enable-Signal (active high) für den Bustreiber durch die UND-Verknüpfung des invertierten CS^o-Signals und des WE^o-Signals abgeleitet werden kann.

Während eines Schreibtaktes ist in der ersten Hälfte des Taktes das RAM selektiert (CS^o = „low“) und WE^o auf „high“, sodass sich das RAM zunächst im Read-Mode befindet. Mit der fallenden Flanke von WE^o wird der Bustreiber vom Read-Mode in den High-Zero-Mode umgeschaltet und der Schreibvorgang durchgeführt. Um zu gewährleisten, dass beim Pegelwechsel von WE^o der Bustreiber sicher abgeschaltet ist und keine RAM-Daten den Bus beeinflussen, wird das WE^o-Signal über zwei hintereinander geschaltete Inverter um eine Gatterlaufzeit im Vergleich zur Bustreiber-Ansteuerung verzögert.

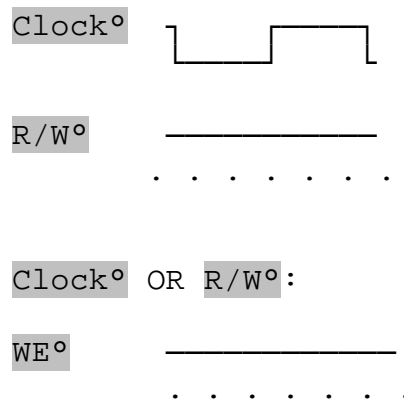
RAM im realen Betrieb

Bei einem realen, statischen Speicher müssen die Adresse und die Daten stabil an den Eingängen anliegen bevor das $\overline{\text{WE}}$ -Signal zur Aktivierung auf „low“ gezogen wird. Adressen und Daten müssen darüber hinaus nach dem Zeitraum des Schreibens für eine gewisse Dauer weiter anstehen. Dieses Zeitverhalten ist mit einem Ein-Phasen-Takt nur über Umwege zu realisieren.

SCHREIBEN:



Lesen:



Da das $\overline{\text{WE}}$ -Signal bereits zu Taktbeginn aktiv wird und die Adresse und Daten zu diesem Zeitpunkt noch nicht stabil an den Eingängen des Speicherbausteins anstehen, werden u.U. falsche Daten in falsche Adressen geschrieben. Dies lässt sich umgehen, indem das $\overline{\text{WE}}$ -Signal oder das $\overline{\text{CS}}$ -Signal künstlich (s.o.) verzögert wird. Eine andere Möglichkeit wäre wieder die Verlagerung des $\overline{\text{WE}}$ -Signals in die zweite Takthälfte, wobei dann das Signal vor Ende des Taktes wieder auf „high“ wechseln müsste, um vor dem Taktende (Datenwechsel auf dem Bus) den Schreibvorgang abzuschließen. Das ließe sich nur durch den Einsatz eines Monoflops realisieren, was wegen der Taktunabhängigkeit (fixe Zeiteinstellung) wenig praktikabel wäre. Eine andere Möglichkeit mit $\overline{\text{WE}}$ in der zweiten Takthälfte wäre die Verzögerung der Daten um einige Gatterlaufzeiten in den nächsten Takt, sodass diese über das Ende von $\overline{\text{WE}}$ am Taktende hinaus sicher anstehen. Hier zeigen sich wieder deutlich die Nachteile eines Ein-Phasen-Taktes, bei dem es sich zudem mit den hier aufgeführten Umgehungen der Einschränkungen nicht mehr um einen reinen Ein-Phase-Takt handeln würde. Eine saubere Lösung ist somit nur mit der Verdoppelung des Eingangstakts und seiner anschließenden Halbierung zur Erzeugung zusätzlicher Flanken möglich (s. Takt).

ROM

Auch das ROM aus der ProfiLab-Bauteile-Bibliothek ist nicht für den Busbetrieb geeignet. An den Datenausgängen wird deshalb ein Bustreiber angeschlossen, dessen Enable-Eingang über den invertierten Makro-Eingang CS^o (Chip Select) gesteuert wird.

Manueller Betrieb des Adressgenerators

An Hand des folgenden Beispiels wird gezeigt, wie Adressierungen schrittweise mit der manuellen Steuerung abgearbeitet werden.

Dazu ist die Datei 65C02AG.PRJ zu laden, die Simulation zu starten, die Bauteile-Bibliotheksanzeige („<<<“) auszuschalten und die Pegelanzeige (HI/LO - Symbol) sowie die Fontplatte zu aktivieren.

Es empfiehlt sich, vor jedem Teilschritt alle Schalter auf „AN“ in die Ruhestellung zu versetzen. Dadurch wird vermieden, dass ein von einem vorhergehenden Schritt noch aktivierter Schalter die nächste Aktion verfahrenswidrig beeinflusst. Man erkennt den vollständigen Ruhezustand der Schaltung daran, dass alle Steuerleitungen auf „H“ stehen.

In diesem Zusammenhang wird noch einmal daran erinnert, dass alle Signale „active low“ ausgelegt sind und mit dem Kennzeichen ° versehen wurden. Wenn im Folgenden bei einer Aktion z.B. CLK PC° angegeben ist, bedeutet dies, dass der Schalter mit dieser Bezeichnung auf „OFF“ geschaltet und damit aktiviert werden muss. Dem großgeschriebene „TO“ innerhalb einer Signalbezeichnung folgt die Zielbezeichnung, an die die Daten weitergeleitet werden. Das kleingeschriebene „to“ bedeutet: „setze das/die angegebene Signal(e) auf die folgenden Werte.“

Die Übersicht für alle Schalterstellungen kann dem Abschnitt „Micro-Code“, Unterabschnitt „Adressierungsarten“ entnommen werden.

Zur Demonstration soll der Befehl LDX \$0306 ausgeführt werden. Steht der LDX Befehl auf der Adresse 0313h und ist das Befehlsbyte bereits dekodiert, so adressiert der Programmzähler das Low Byte der absoluten Adresse im Ladebefehl (0314h). Da für das Beispiel mitten in ein Programm bzw. einen Befehl eingestiegen wird und der Programmzähler noch nicht durch vorhergehende Befehle gesetzt wurde, muss dieser vor Beginn des Beispiels manuell initialisiert werden. Dies geschieht mittels folgender Vorgehensweise, die nicht Teil des Beispiels ist, sondern nur der Vorbereitung dient.

Erster Teilschritt:

<u>Aktion</u>	<u>Kommentar</u>
HEX-Eingabe aktivieren 14h einstellen	Programmzähler Low Byte
ENL°	über Low Bus an TEMP-Register
CLK TEMP°	Low Byte in TEMP zwischenspeichern
Alle Schalter wieder in die Ruhestellung setzen!!!!	

Zweiter Teilschritt:

<u>Aktion</u>	<u>Kommentar</u>
HEX-Eingabe aktivieren 03h einstellen	Programmzähler High Byte
ENH°	über High Bus an PC HB
TEMP TO BUS°	Low Byte aus TEMP an PC LB
PC LD°	Vorbereitung Laden von PC
CLK PC°	PC laden
Alle Schalter wieder in die Ruhestellung setzen!!!!	

Speicherauszug für das Beispiel LDX \$0306

```
0306
.
.
.
0313 AE  LDX $0306
0314 06
0315 03
```

Erster Teilschritt:

<u>Aktion</u>	<u>Kommentar</u>
PC TO BUS° ENL°	Ausgabe des PCs auf den Adressbus Low Byte auf Low Bus
CLK TEMP° CLK PC°	Low Byte in TEMP zwischenspeichern PC auf HB der Adresse einstellen

Alle Schalter wieder in die Ruhestellung setzen!!!!

Zweiter Teilschritt:

<u>Aktion</u>	<u>Kommentar</u>
PC TO BUS° ENH° TEMP TO BUS° AR LD°	Ausgabe des PCs auf den Adressbus High Byte über High Bus an AR HB Low Byte aus TEMP an AR LB Vorbereitung Laden von AR
CLK AR°	AR mit Zieladresse laden

Alle Schalter wieder in die Ruhestellung setzen!!!!

Dritter Teilschritt:

<u>Aktion</u>	<u>Kommentar</u>
AR TO BUS° ENL°	Ausgabe des ARs auf den Adressbus Wert über Low Bus an X-Register
CLK X° CLK PC°	X-Register mit Wert laden Programmzähler auf nächsten Befehl

Alle Schalter wieder in die Ruhestellung setzen!!!!