VHDL-ENTWURF EINES PROZESSORS MIT RISC-ARCHITEKTUR

2. Studienarbeit

Andreas Gerstlauer
Matrikelnummer: 1491498

Institut für Mikroelektronik Stuttgart (IMS)
Prof. Dr. B. Höfflinger

Institut für Nachrichtenvermittlung und Datenverarbeitung (IND)
Universität Stuttgart
Prof. Dr. P. Kühn

Betreuer:
Eduard Bernath (IMS)
Kai Dieffenbach (IMS)
Albrecht Schwarz (IND)

12. Juli 1996

Kurzfassung

Im Zuge der kontinuierlich fortschreitenden Entwicklung im Bereich höchstintegrierter Schaltungen haben sich auch die Methoden zur Entwicklung solcher Schaltung so gewandelt, daß heute kundenspezifische Schaltkreise (ASICs) in vielen Schaltungen wirtschaftlich eingesetzt werden können. Das IMS unterstützt diesen Prozeß mit der kostengünstigen Entwicklung und Herstellung von ASICs auch in kleinen Stückzahlen. Speziell für die in der Industrie häufig zu findenden Steuerungsaufgaben existiert ein Mikrocontroller-Baukasten, bei dem je nach Applikation ein Prozessorkern mit der für die Aufgabe benötigten Peripherie individuell auf einem Chip integriert werden kann.

Im Rahmen dieser Arbeit wurde ein neuer, universell einsetzbarer Mikrocontrollerkern für den Baukasten am IMS entwickelt. Der Prozessormakro ist befehlssatzkompatibel zum Mikrocontroller PIC17 von Microchip Technology. Er besitzt eine RISC-Architektur mit getrennten Instruktions- und Datenbussen. Instruktionen haben eine Breite von 16 Bit und es können maximal 64k Worte Programmspeicher angesprochen werden. Der Datenspeicherbereich hat eine maximale Größe von 256 x 8 Bit. Neben einem Ereigniszähler unterstützt der Mikrocontrollerkern vier verschiedenen Interruptquellen unterschiedlicher Priorität.

Entsprechend den am IMS eingesetzten modernen Designmethoden erfolgte die Entwicklung des Mikrocontrollerkerns mit der Hardwarebeschreibungssprache VHDL. In einem ersten Schritt wurde die Spezifikation in eine abstrakte Verhaltensbeschreibung umgesetzt, simuliert und überprüft. Anschließend wurde darauf aufbauend eine Beschreibung des Prozessorkerns entwickelt, aus der automatisch eine Schaltung synthetisiert werden konnte. Abschließend wurde die so erhaltene Gatternetzliste erneut simuliert und getestet. Zum funktionalen Test des Mikrocontrollerkerns auf allen drei Ebenen wurde ein spezielles Testprogramm erzeugt und simuliert ausgeführt. Am Ende der Entwicklung steht die zur Produktion freigegebene Netzliste des Mikrocontrollerkerns.

Abstract

In recent years, the continuous development in the area of highly integrated circuits has lead to a change in the design methods used, making it possible to economically utilize application specific integrated circuits (ASICs) in many designs nowadays. At the IMS this process is supported by developing and fabricating ASICs at competitive prices even in low quantities. Especially for frequently needed controlling applications, the IMS has a library of microcontroller modules. Depending on the task to be performed, a processor kernel can be integrated on one chip together with the special peripherals needed for the application.

In this work a new microcontroller kernel module was developed. Its instruction set is compatible to the microcontroller PIC17 by Microchip Technology. The processor employs a RISC architecture with separate busses for instructions and data. Instructions are 16 bit wide and the program memory has a maximum size of 64k words. The maximum size of the data memory is 256 x 8 bits. Special features include a timer/counter module and support for four interrupt sources with different priorities.

Based on the modern design methods used at the IMS the microcontroller kernel was developed using the hardware description language VHDL. First, the specification was transferred into an abstract behavioral description which was then simulated and tested. Based on that, the next step was to develop a description of the microcontroller kernel which could be automatically synthesized into a circuit. Finally, the resulting gate level netlist was again simulated and tested. In order to perform a functional test of all three description levels of the microcontroller kernel a special test program was created. At the end of the process the development results in a netlist which can be used to fabricate a chip.

Inhaltsverzeichnis

Abkürzungs- und Formelzeichenverzeichnis iv

1 Einführung 1
1.1 Entwicklung integrierten Schaltungen 1
1.2 VHDL 2
1.3 ASIC-Entwicklung 3
1.4 Übersicht über die Arbeit 4

2 Spezifikation des Mikrocontrollerkerns 6
2.1 Überblick 6
2.2 Systemtaktschema 7
2.3 Reset-Logik 8
2.4 Speicherorganisation 9
2.4.1 Programmspeicher 9
2.4.2 Datenspeicher (Register) 10
2.5 Schnittstellenbeschreibung 13
2.5.1 Übersicht 13
2.5.2 Programmspeicher 15
2.5.3 Datenspeicher 18
2.6 Spezialregister 22
2.6.1 Register INDF0 (Adresse 00h) 24
2.6.2 Register FSR0 (Adresse 01h) 24
2.6.3 Register PCL (Adresse 02h) 25
2.6.4 Register PCLATH (Adresse 03h) 26
2.6.5 Register ALUSTA (Adresse 04h) 26
2.6.6 Register RTCSTA (Adresse 05h) 27
2.6.7 Register CPUSTA (Adresse 06h) 28
2.6.8 Register INTSTA (Adresse 07h) 29
2.6.9 Register INDF1 (Adresse 08h) 30
2.6.10 Register FSR1 (Adresse 09h) 30
2.6.11 Register W (Adresse 0Ah) 31
2.6.12 Register RTCCL (Adresse 0Bh) und RTCCH (Adresse 0Ch) 31
2.6.13 Register TBLPTRL (Adresse 0Dh) und TBLPTRH (Adresse 0Eh) 31
2.6.14 Register BSR (Adresse 0Fh) 32
2.7 Befehlssatzarchitektur 32
2.7.1 Adressierungsarten 32
2.7.2 Befehlssatz 35
2.7.2.1 Datentransferoperationen 36
2.7.2.2 Arithmetische und logische Operationen 38
2.7.2.3 Operationen zur Steuerung des Programmablaufs 44
2.7.2.4 Operationen zur Bitmanipulation 49
2.7.2.5 Sonstige Operationen 49
2.8 Befehlsausführung 50
2.8.1 Allgemein 50
2.8.2 Table-Operationen 54
2.8.3 Befehle zur Programmablaufsteuerung 56
2.8.3.1 Skips 57

2.8.3.2 Sprünge und Unterprogrammaufrufe ... 58
2.8.3.3 Rücksprünge ... 59
2.8.4 Spezialfälle ... 60
2.9 Stack ... 62
2.10 Interrupts ... 63
2.10.1 Allgemein ... 64
2.10.2 Interruptquellen ... 66
2.10.3 Interruptlogik ... 68
2.10.4 Interrupt-Timing ... 70
2.10.4.1 Table-Operationen ... 72
2.10.4.2 Befehle zur Programmablaufsteuerung ... 74
2.10.4.3 Spezialfälle ... 77
2.10.5 Besonderheiten ... 79
2.11 RTCC-Modul ... 80
2.11.1 Ereignisauswahl ... 82
2.11.2 Vorteiler ... 83
2.11.3 Synchronisation ... 84
2.11.4 Zählerregister ... 85

3 Architektur der RTL-Beschreibung ... 87
3.1 Grundlagen ... 87
3.1.1 Allgemeine Richtlinien ... 87
3.1.2 Datenpfad ... 88
3.1.3 Blockhierarchie ... 89
3.2 Taktgenerator ... 90
3.3 Schnittstellentiming ... 91
3.4 W Register ... 91
3.5 Registermultiplexer ... 92
3.6 ALU ... 93
3.7 ALU Status Register ... 100
3.8 Ergebnismultiplexer ... 101
3.9 Table Latches ... 102
3.10 Table Pointer ... 103
3.11 Interruptmodul ... 104
3.11.1 Interrupterkennung ... 105
3.12 RTCC Modul ... 106
3.12.1 Vorteiler ... 107
3.12.2 Synchronisation ... 107
3.12.3 RTCC Register ... 108
3.13 Programmzählermodul ... 108
3.13.1 Stack ... 111
3.14 PC Latch ... 112
3.15 Instruktionverarbeitung ... 113
3.16 Instruktionsdekoder ... 115

4 Synthese und Simulation ... 118
4.1 VHDL-Simulationen ... 118
4.1.1 Überblick ... 118
4.1.2 Testumgebung ... 120
4.1.2.1 RAM ... 121
4.1.2.2 ROM ... 121
4.1.2.3 Tracer ... 122
4.2 Synthese ... 122

4.3 Simulation auf Gatterebene 123
4.4 Ergebnisse 125
5 Zusammenfassung und Ausblick 128
Literaturverzeichnis 130

Abkürzungs- und Formelzeichenverzeichnis

ALU	Arithmetic Logical Unit
ALUSTA	ALU Status
ASIC	Application Specific Integrated Circuit
BS	Bank Select
BSR	Bank Select Register
C	Carry Flag
CLK	Clock
CMOS	Complementary Metall Oxide Semiconductor
CPU	Central Processing Unit
CPUSTA	CPU Status
DAddrSrc	Data Address Source
DAddrDest	Data Address Destination
DC	Digit Carry
DDataIn	Data Data Input
DDataOut	Data Data Output
DTris	Data Tristate
DSel	Data Select
EDIF	Electronic Design Interchange Format
EPROM	Electrically Programmable Read Only Memory
FS0...FS3	File Select 0...3
FSR	File Select Register
GLINTD	Global Interrupt Disable
IAddr	Instruction Address
IALE	Instruction Adress Latch Enable
IC	Integrated Circuit
IDataIn	Instruction Data Input
IDataOut	Instruction Data Ouput
IMS	Institut für Mikroelektronik Stuttgart
INT0	Interrupt 0
INT0IE	INT0 Interrupt Enable
INT0IR	INT0 Interrupt Request
INT1	Interrupt 1

INT1IE	INT1 Interrupt Enable
INT1IR	INT1 Interrupt Request
INTEDG	Interrupt Edge
INTSTA	Interrupt Status
I/O	Input/Output
IR	Instruction Register
ITris	Instruction Tristate
nDCE	Not Data Chip Enable
nDOE	Not Data Output Enable
nDWR	Not Data Write Enable
nIOE	Not Instruction Output Enable
nIWR	Not Instruction Write Enable
nMCLR	Not Master Clear
OV	Overflow
PC	Program Counter
PCH	PC High
PCL	PC Low
PCLATH	PC Latch High
PEIE	Peripheral Interrupt Enable
PEIR	Peripheral Interrupt Request
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RTCC	Real Time Clock Counter
RTCCH	Real Time Clock Counter High
RTCCL	Real Time Clock Counter Low
RTCIE	RTCC Interrupt Enable
RTCIR	RTCC Interrupt Request
RTCSTA	RTCC Status
RTEDG	RTCC Edge
RTPS0..3	RTCC Prescaler 0...3
STKAVL	Stack Available
t_{AH}	Address Hold Time
t_{AS}	Address Setup Time
t_{CP}	Chip enable Pulse Width
t_{DHW}	Data Hold Time after Write end

t_{DSW}	Data Setup Time to Write end
t_{HZDV}	Time from output High Impedance to Data Valid
t_{HZOE}	Output Enable High to output High Impedance Delay
t_{PRECH}	Minimum Precharge Time
t_{WP}	Write enable Pulse Width
TBLAT	Table Latch
TBLATH	Table Latch High
TBLATL	Table Latch Low
TBLPTR	Table Pointer
TBLPTRH	Table Pointer High
TBLPTRL	Table Pointer Low
TOS	Top Of Stack
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration
W	Working
Z	Zero

1 Einführung

1.1 Entwicklung integrierten Schaltungen

In den letzten Jahren ist die Entwicklung im Bereich der integrierten Schaltungen (engl. *Integrated Circuit*, *IC*) kontinuierlich fortgeschritten und ermöglichte immer komplexere Designs mit einer exponentiellen Zunahme der Anzahl Transistoren pro Chip. Zu Beginn dieses Prozesses war die Entwicklung von integrierten Schaltungen umständlich, zeitaufwendig, fehlerträchtig und teuer. Die einzelnen Masken für die Herstellung des Chips wurden von Hand durch Plazierung der verschiedenen Transistorelemente, Metallverbindungen, Vias, etc. am Computer entworfen. Aus diesem Grund lohnte sich dieser Aufwand nur für die Herstellung von Standardbausteinen in Massenproduktion. Bei der Schaltungsentwicklung wurden dann diese Standardbausteine mit anderen Bauteilen auf einer Platine kombiniert.

Im Laufe der Zeit kam man auf die Idee, das Prinzip der Zusammensetzung aus Standardelementen auch auf die eigentliche IC-Entwicklung anzuwenden, um diese dadurch drastisch zu vereinfachen. Bei diesem sog. *Semi-Custom-Design* werden bei der Entwicklung des ICs vom Halbleiterhersteller vorgegebene Standardzellen, wie z.B. logische Gatter oder Speicherzellen, plaziert und passend verdrahtet. Die Entwicklung der Schaltung kann dabei graphisch am Rechner erfolgen, indem der Schaltungsentwickler die graphische Repräsentation der Standardblöcke aus einer Bibliothek entnimmt und am Rechner in einem Schaltplan verdrahtet. Hinter den Standardzellen verbergen sich vordefinierte und ausgetestete Maskenblöcke, die dann bei der Chipherstellung gemäß dem Schaltplan auf dem Chip plaziert und über Metallebenen verbunden werden. Der große Vorteil dabei ist aber, daß den Standardzellen auch eine computerbasierende Beschreibung des Bausteinverhaltens zwischen den Ein- und Ausgängen, inklusive interner Verzögerungszeiten und Randbedingungen, wie z.B. Setup- und Hold-Zeiten, beigegeben wird, so daß noch vor der Plazierung auf dem Chip eine erste Simulation der graphisch entwickelten Schaltung am Rechner möglich ist. Dadurch war es erstmals möglich, die Schaltung vor der eigentlichen, kostenintensiven Fertigung fast vollständig zu testen.

Dieser Fortschritt ermöglichte schließlich überhaupt erst die Herstellung kundenspezifischer ICs in kleinen Stückzahlen, den sog. *ASIC* (engl. *Application Specific Integrated Circuit*). Ziel bei der Herstellung von ASICs war und ist es, in der Entwicklung von Schaltungen kleinerer oder größerer Projekte die bisher auf der Platine realisierte Schaltung möglichst komplett in einem oder mehreren ICs unterzubringen. Die Vorteile, die daraus resultieren sind geringerer Platz- und Stromverbrauch, geringere Anzahl Bauteile, Schutz des Schaltungsdesigns vor Einblicken, etc., weshalb sich heutzutage beim Blick auf moderne Platinen meistens nur noch wenigen Bauelemente neben einigen wenigen ASICs befinden.

Mit der ständig zunehmenden Komplexität der ASICs wurde aber auch die graphische Eingabe der Schaltung bis hinunter auf Gatterebene immer umfangreicher und damit unübersichtlicher und fehleranfälliger. In den letzten Jahren wurden deshalb sog. Hardwarebeschreibungssprachen (engl. *Hardware Description Language*, *HDL*) entwickelt, deren Einsatz sich bis zum heutigen Tage mehr und mehr durchgesetzt hat, so daß die Entwicklung von ASICs mit Hilfe von Hardwarebeschreibungssprachen aktueller Stand der Technik ist. Die bekanntesten Vertreter sind Verilog und VHDL, wobei sich abzeichnet, daß VHDL allgemein als Standardsprache akzeptiert und verwendet wird.

1.2 VHDL

Die Hardwarebeschreibungssprache VHDL (engl. *VHSIC Hardware Description Language*, VHSIC steht für *Very High Speed Integrated Circuit*) ist in den 80er Jahren über das amerikanische Verteidigungsministerium im Rahmen eines großangelegten Projekts zur Förderung der Entwicklung höchstintegrierten Schaltungen entstanden. Die Sprache wurde inzwischen vom IEEE genormt und standardisiert.

Hinter VHDL verbirgt sich im Prinzip eine Programmiersprache, mit der sich eine Schaltung auf verschiedenen Abstraktionsebenen beschreiben läßt. Die VHDL-Beschreibung einer Schaltung läßt sich einfach am Rechner simulieren und dient gleichzeitig zur Dokumentation der Schaltung. Einzelne Schaltungsmodule werden in VHDL als sog. *Entities* bezeichnet. Die Entity definiert dabei die Schnittstelle des Moduls, d.h. sämtliche Ein- und Ausgänge. Die Funktion und das Verhalten der Entity selbst wird in ihrer zugeordneten sog. *Architecture* beschrieben. In der Architecture einer Entity können wiederum weitere Unterblöcke (Entities) als sog. *Components* aufgenommen und verschaltet werden oder es wird das Verhalten des Moduls direkt beschrieben. Das Verhalten eines Moduls wird über einen oder mehrere sog. *Prozesse* beschrieben. Alle Prozesse eines Moduls arbeiten parallel, wobei ein Prozeß genau dann angestoßen wird, wenn sich eines seiner Eingangssignale ändert. Der Prozeß verarbeitet die Eingangssignale und evtl. interne Statussignale, um ggf. Ausgangs- oder Statussignale neu zu setzen. Innerhalb eines Prozesses stehen eine Vielzahl von Konstrukten, inklusive aller bekannten Programmierkonstrukte, wie z.B. logische, binäre und arithmetische Verknüpfungen, Zuweisungen, Schleifen, Abfragen, usw., zur Verfügung. Die in VHDL vorhandenen Möglichkeiten sind so vielfältig, daß auf eine weitere Beschreibung im Rahmen dieser Arbeit verzichtet werden muß und auf die entsprechende Literatur verwiesen wird [9,1].

Ausgehend von einer sog. *Top-Level-Entity* wird die Schaltung so immer weiter bis hin zu einer kompletten Beschreibung des Verhaltens in Unterblöcke unterteilt und modularisiert. Nach Fertigstellung eines Moduls kann dieses vor Integration in die Gesamtschaltung separat simuliert und getestet werden. Im Laufe der Zeit erhält man so eine Bibliothek von getesteten und funktionierenden Standardmodulen, die sich in zukünftigen Projekten wiederverwenden lassen.

VHDL erlaubt die Beschreibung eines Moduls auf sehr verschiedenen Abstraktionsebenen. Eine Entity kann dabei mehrere verschiedene Architectures besitzen, wobei die tatsächlich verwendete über eine sog. *Konfiguration* festgelegt werden muß. Ein Modul kann so zuerst auf einer sehr hohen Abstraktionsebene spezifiziert und simuliert werden und erst später, wenn seine Funktion genau verifiziert ist, nach und nach verfeinert, unterteilt und beschrieben werden.

Zu Beginn des Übergangs weg von der graphischen Schaltungseingabe und hin zu Hardwarebeschreibungssprachen bedeutete dies, daß am Ende der Schaltungsentwicklung jedes Modul der VHDL-Beschreibung bis auf Gatterebene hinab beschrieben wurde, d.h. die Beschreibungen der untersten Module bestanden aus der Strukturbeschreibung ihrer als Unterblöcke aufgenommenen und verschalteten Gatter. Der große Durchbruch kam erst mit dem Aufkommen sog. Synthesewerkzeuge, die direkt aus einer VHDL-Beschreibung eines Moduls eine Netzliste mit verschalteten Gattern generieren, d.h. synthetisieren können. Mit diesen Werkzeugen ist es möglich, aus einer relativ abstrakten VHDL-Beschreibung automatisch eine Gatterbeschreibung zu erhalten.

Da sich mit VHDL auch Prozesse beschreiben lassen, die nicht in eine Schaltung umsetzbar sind, unterstützen die Synthesewerkzeuge nur eine Untermenge aller VHDL-Konstrukte. Zur Synthese muß die Schaltung als sog. RTL-Beschreibung vorliegen (*Register Transfer Logik*). Eine RTL-Beschreibung einer Schaltung besteht im Prinzip aus der Beschreibung der benötigten Flip-Flops, Latches und der dazwischengeschalteten kombinatorischen Logik. Flip-Flops und Latches werden dazu über festgelegte und

vorgegebene If-Konstrukte beschrieben, wohingegen kombinatorische Logik durch allgemeine logische Operationen, Zuweisungen und Abfragen realisiert werden kann. Je nach verwendetem Synthesewerkzeug unterscheiden sich die zugelassenen und synthetisierbaren VHDL-Konstrukte u.U. deutlich. Moderne Werkzeuge gehen dabei teilweise auch schon über reine RTL-Beschreibungen hinaus und ermöglichen die Synthese komplexer Verhaltensbeschreibungen, wie z.B. Zustandsautomaten (engl. *State Machine*).

1.3 ASIC-Entwicklung

Mit der Einführung von VHDL und dem Entstehen von Synthesewerkzeugen hat sich die ASIC-Entwicklung stark verändert. Zum einen sind im Laufe der Jahre in den einzelnen Designhäusern aus den verschiedenen Projekten umfangreiche Bibliotheken mit getesteten und vorgefertigt synthetisierten Schaltungsblöcken verschiedenster Komplexität und Funktionalität entstanden. Im Idealfall muß bei der Entwicklung eines ASICs die Schaltung eines ganzen Systems nur noch aus den schon vorhandenen Blöcken, wie Prozessorkern, RAM, ROM, Businterface, etc. zusammengesetzt werden. Dadurch sind extrem kurze Entwicklungszeiten überhaupt erst möglich.

Auf der anderen Seite müssen aber in vielen Fällen neben den eingesetzten Standardmodulen je nach Applikation noch diverse spezielle Module zusätzlich entwickelt werden. Bei der Entwicklung eines Schaltungsmoduls unter Einsatz von VHDL wird dabei in mehreren Schritten vorgegangen. Nach der Spezifikation der Funktionalität des Moduls wird dieses Pflichtenheft in einem ersten Schritt in eine reine VHDL-Verhaltensbeschreibung auf sehr hohem Abstraktionsniveau umgesetzt. Diese Verhaltensbeschreibung soll sich nach außen entsprechend der spezifizierten Funktionsweise des Moduls verhalten, ohne dabei auf Details der Implementierung Rücksicht zu nehmen. Durch den Einsatz aller möglichen VHDL-Sprachmittel soll die Verhaltensbeschreibung möglichst einfach gehalten werden, so daß die Fehleranfälligkeit minimiert und die Verifikation vereinfacht wird. Die Verhaltensbeschreibung wird dann simuliert, um sicherzustellen, daß das Modul ordnungsgemäß funktioniert.

In dem nächsten, zentralen Schritt wird die Schaltung anschließend nach und nach in eine VHDL-Beschreibung auf RTL-Ebene umgewandelt. In diesem Schritt wird, ausgehend von einem Blockschaltbild, die Architektur zur Umsetzung der Spezifikation in eine reale Schaltung festgelegt. Dazu wird das Modul ggf. in weitere Blöcke unterteilt, die der Reihe nach auf RTL-Ebene implementiert werden. Wenn nötig können die einzelnen Unterblöcke auch zuerst auf einem höheren Niveau beschrieben werden, um sie dann der Reihe nach, bei ständiger Simulation des Gesamtsystems, in eine RTL-Beschreibung umzuwandeln. Abschließend wird die komplette RTL-Beschreibung genaustens simuliert. Bei allen Simulation werden Verletzungen der Spezifikation oder sonstige Fehler durch Vergleich der Simulationsergebnisse mit den Protokollen der Simulation der Verhaltensbeschreibung aufgedeckt.

Zur VHDL-Simulation der Verhaltensbeschreibung, der RTL-Beschreibung und aller Zwischenschritte wird die zu testende VHDL-Beschreibung des entwickelten Moduls in eine VHDL-Testumgebung, die sog. *Testbench*, eingebettet. In dieser Testbench wird die Umgebung des Moduls an seinen Schnittstellen entsprechend den realen Verhältnissen im späteren Betrieb simuliert und somit verifiziert, daß sich das Modul korrekt verhält. Dabei macht man sich zu Nutze, daß sich auch komplexe Verhaltensformen und Zeitkriterien in VHDL recht einfach unter Einsatz hochsprachlicher Konstrukte sehr abstrakt beschreiben lassen. Die Testbench steuert also alle Eingänge des Moduls an und überprüft die vom Modul gelieferten Ausgangswerte. Zusätzlich wird durch die Testbench der genaue Ablauf der Simulation in einer Datei protokolliert, so daß sich einzelne Simulationen, z.B. verschiedener Beschreibungsformen oder nach Änderungen am Modul, leicht automatisch vergleichen lassen.

Nachdem die RTL-Beschreibung des Moduls fertig entwickelt, simuliert und getestet ist, wird durch Synthese der Schaltung die Gatternetzliste erzeugt. Die Gatternetzliste wird anschließend ebenfalls simuliert und damit überprüft, daß sich die Netzliste wie die VHDL-

Beschreibung verhält. Bei der Simulation auf Gatterebene werden außerdem zum ersten Mal auch die Laufzeiten der verwendeten Gatter berücksichtigt, so daß sich hier Fehler im Design, die durch die Laufzeiten verursacht werden, zeigen. Zusätzlich können durch Simulation und Analyse des Designs auf Gatterebene die Grenzen der Schaltung bezüglich Taktfrequenz und anderer Arbeitsbedingungen ausgelotet werden. Gegebenenfalls muß das Design durch Änderungen der RTL-Beschreibung oder des Syntheseprozesses optimiert bzw. korrigiert werden. Nach erneuter Synthese und Simulation wiederholt sich dieser Vorgang solange bis die endgültige Netzliste feststeht.

Ist die Gatternetzliste schließlich zur Produktion freigegeben, können die Gatter auf einem Chip plaziert und verdrahtet werden. Dieser Vorgang geschieht größtenteils automatisch durch ein entsprechendes Werkzeug am Computer, wobei in kritischen Fällen von Hand eingegriffen werden muß. Nach der Plazierung und Verdrahtung stehen Informationen über die Signallängen zwischen den Gattern zur Verfügung, die in einer nachfolgenden, erneuten Simulation auf Gatterebene zusätzlich berücksichtigt werden. So wird sichergestellt, daß das Timing des Chips auch mit der zusätzlichen Belastung der Leitungskapazitäten noch stimmt. Dies wird mit abnehmenden Transistorgrößen immer wichtiger, da dabei die Leitungskapazitäten im Verhältnis zu den Gatterkapazitäten zunehmend stärker ins Gewicht fallen. Erst nachdem auch diese Simulation erfolgreich verlaufen ist, erfolgt die Freigabe, und der Chip kann tatsächlich produziert werden.

1.4 Übersicht über die Arbeit

Am Institut für Mikroelektronik Stuttgart (IMS), einer Stiftung des öffentlichen Rechts, werden ASICs für die mittelständische Industrie auch in kleinen Stückzahlen kostengünstig angeboten. In vielen Fällen handelt es sich um spezielle Mikrocontroller als zentraler Bestandteil industrieller Meß-, Steuer- und Regelungsschaltungen. Aus diesem Grund existiert am IMS ein Mikrocontroller-Baukasten, bestehend aus vorgefertigten Modulen für eine Vielzahl von Aufgaben. Je nach gewünschter Applikation wird so ein ausgewählter Prozessorkern zusammen mit der benötigten Peripherie, wie z.B. RAM, ROM, UART, Pulsbreitenmodulation, A/D- und D/A-Wandler, uvm. auf einem Chip integriert.

Die einzelnen Module des Mikrocontrollerbaukastens werden am IMS in VHDL unter Einsatz diverser Synthese- und Simulationswerkzeuge entwickelt, simuliert und getestet. Zur Herstellung eines ASICs werden die Gatternetzlisten der ausgewählten Module am IMS dann kombiniert und auf dem Chip plaziert und verdrahtet. Nach der Produktionsfreigabe wird der ASIC dann ebenfalls am IMS in den gewünschten Stückzahlen produziert. Das IMS erhält dafür vorgefertigte Silizium-Waver, auf denen bereits Blöcke von n- und p-Transistoren vorgefertigt vorhanden sind. Am IMS werden diese dann durch Aufbringen der entsprechenden Metallverbindungen nur noch passend verdrahtet. Bisher bildete die Basis eine 1,2 µm-CMOS-Technologie [2], wobei im Augenblick gerade der Umstieg auf eine 0,8 µm-CMOS-Technologie erfolgt.

In dieser Arbeit wurde ein neuer Mikrocontrollerkern für den Mikrocontrollerbaukasten am IMS entwickelt. Es handelt sich dabei um einen reinen Prozessorkern, der über externe Schnittstellen erst mit zusätzlicher Peripherie, wie z.B. RAM, ROM oder Steuermodule, beschaltet wird. Die Schnittstellen des Kerns sind so ausgelegt, daß die externe Beschaltung des Kerns, je nach Applikation, sehr flexibel erfolgen kann. Der RISC-Mikrocontrollerkern ist befehlssatzkompatibel zum Mikrocontroller PIC17 vom Microchip Technology [8,10] und besitzt eine Harvard-Architektur mit getrennten Bussen für Daten und Instruktionen.

Die Entwicklung des Mikrocontrollerkerns erfolgte in VHDL. Ausgehend von der genauen Spezifikation des Mikrocontrollerkerns, die sich nach dem Vorbild PIC17 richtet, wurde in einem ersten Schritt eine VHDL-Verhaltensbeschreibung des Kerns entwickelt und ausführlich simuliert. Anschließend wurde die Architektur des Prozessormakros festgelegt und in eine VHDL-Beschreibung auf RTL-Ebene umgesetzt. Nach der Verifikation der RTL-

Beschreibung durch Simulation und Vergleich mit den Simulationsdaten der Verhaltensbeschreibung wurde die endgültige Schaltung synthetisiert und auf Gatterebene erneut simuliert. Abschließend wurde der entwickelte Mikrocontrollerkern auf Basis der Gatternetzliste auf seine technischen Daten hin untersucht.

Für die Synthese der Schaltung wurde das Programm Synopsys [12] benutzt. Bei der Entwicklung der RTL-Beschreibung wurde aber darauf geachtet, daß keine speziellen Konstrukte verwendet werden, so daß die Schaltung auch von anderen Syntheseprogrammen verarbeitet werden kann. Zur Kontrolle und zur Fehlersuche wurde die Schaltung auch auf die erfolgreiche Synthese mit Compass [4] hin untersucht. Als VHDL-Simulator kam das Programm V-System [7] zum Einsatz. Die Simulation der Gatternetzlise erfolgte wiederum mit Compass. Der Austausch der Gatternetzliste zwischen Synopsys und Compass wurde auf Basis des gemeinsamen, standardisierten EDIF-Formats durchgeführt.

2 Spezifikation des Mikrocontrollerkerns

In diesem Kapitel wird die externe Spezifikation des Mikrocontrollerkerns aus Anwendersicht beschrieben, d.h. es werden die Eigenschaften und Fähigkeiten des Prozessormakros dargelegt, die zur Programmierung und zum Einsatz des Kerns in einer realen Schaltungsumgebung benötigt werden.

In einem ersten Schritt dieser Arbeit wurde diese Spezifikation in eine VHDL Verhaltensbeschreibung umgesetzt. Die Verhaltensbeschreibung bietet dabei nach außen hin die hier definierten Schnittstellen des späteren realen Mikrocontrollerkerns, implementiert aber die dazu notwendige Funktionalität mit einer Beschreibung auf relativ abstrakter Ebene, die nicht direkt in eine Schaltung umgesetzt werden kann. Der Schritt über eine Beschreibung auf Verhaltensebene hat den Vorteil, daß sich aufgrund der darin zur Verwendung kommenden hochsprachlichen Beschreibungsmittel die Umsetzung der Spezifikation und die Validierung dieser Umsetzung leichter bewerkstelligen läßt.

Da für den PIC17 kein Simulator auf PC- oder Workstationebene zur Verfügung stand, mußte die Umsetzung der Spezifikation in die Verhaltensbeschreibung getestet und verifiziert werden, indem ein Testprogramm entwickelt wurde, das alle Funktionen des Kerns austestet und dessen korrekte Ausführung in der Verhaltensbeschreibung von Hand simuliert und überprüft werden mußte. Später dient die verifizierte Verhaltensbeschreibung als Referenz bei der Überprüfung der eigentlichen RTL-Beschreibung, welche automatisch durch Vergleich der Ausführung eines Testprogramms erfolgen kann (siehe Kapitel 4).

2.1 Überblick

Die Vorgabe bei der Entwicklung des Mikrocontrollerkerns bestand darin, einen Prozessormakro zu entwerfen, der befehlssatzkompatibel zu dem Mikrocontroller PIC17C42 von Microchip Technology ist, um bei der Programmentwicklung für den Prozessor auf die entsprechenden, für den PIC17 bereits vorhandenen Entwicklungswerkzeuge, wie Simulatoren, Assembler, Debugger, etc., direkt zurückgreifen zu können. Um eine möglichst hohe Kompatibilität sicherstellen zu können, ist die prinzipielle Architektur des Mikrocontrollerkerns stark an den Aufbau des PIC17-Mikrocontrollers angelehnt [8]. Dabei beinhaltet der Prozessormakro aber nur den eigentlichen Kern des Mikrocontrollers. Nicht im Mikrocontrollerkern enthalten sind die beim PIC17C42 bereits auf dem Chip integrierten Peripheriekomponenten und -bausteine, wie Programmspeicher-EPROM, allgemein verwendbarer Datenspeicher bzw. Register (bis auf die im Kern direkt benötigten Spezialregister), digitale I/O-Ports, serielle Schnittstelle, Pulsbreitenmodulationsmodul, diverse Timer und Zähler (bis auf das zum Kern zählende RTCC[1]-Modul, siehe Abschnitt 2.11) und die verschiedenen Spezialfunktionen (Oszillator, Watch-Dog Timer, Sleep Mode, Start-Up Timer, Power-Timer und -Reset, Code- und Schreibschutz, etc.). Es kann davon ausgegangen werden, daß in Zukunft von Microchip Technology weitere Mitglieder der PIC17-Familie entwickelt werden, die sich vom bereits existierenden PIC17C42 in der Auswahl der auf dem Chip integrierten Peripheriekomponenten unterscheiden, wobei der eigentliche Kern, laut Datenbuch inklusive RTCC-Modul, stets der gleiche sein wird. Der Prozessormakro umfaßt deshalb genau diesen Kern und stellt entsprechende Schnittstellen zum Anschluß beliebiger Peripheriekomponenten zur Verfügung. Durch entsprechende äußere Beschaltung des

[1] Engl. *Real Time Clock Counter*

Mikrocontrollerkerns mit zusätzlichen Bausteinen können beliebige Konfigurationen, z.B. bezüglich Größe und Aufteilung der Programm- und Datenspeicherbereiche oder bezüglich integrierten Peripheriemodulen, bei der Herstellung eines kompletten integrierten Mikrocontroller-Chips verwirklicht werden. Dadurch ist neben der Befehlssatzkompatibilität gleichzeitig eine größtmögliche Flexibilität sichergestellt.

Der Prozessormakro basiert auf einer sog. *RISC*-Architektur (engl. *Reduced Instruction Set Computer* [6]), bei der eine Reihe der Eigenschaften von RISC-Mikroprozessorarchitekturen zum Einsatz kommt. Der Befehlssatz des Mikrocontrollerkerns besteht aus wenigen einfachen, orthogonalen Befehlen, die sehr schnell ausgeführt werden können. Durch eine zweistufige Pipeline, in der parallel das Holen eines Befehls und die Befehlsausführung überlappt werden, wird erreicht, daß alle Befehle, bis auf Sprünge und Befehle zum Austausch von Daten zwischen Programm- und Datenspeicher, nur einen Zyklus zur Ausführung benötigen. Der Mikrocontrollerkern besitzt eine sog. Harvard-Architektur, bei der, im Gegensatz zur klassischen von-Neumann Architektur, Programm- und Datenpfade getrennt sind und auf Programme und Daten in getrennten Speichern zugegriffen wird. Dadurch wird einerseits die Speicherbandbreite erhöht und andererseits können Instruktions- und Datenbusse unterschiedlich breit gewählt werden. Beim Mikrocontroller besitzen die Instruktionen selbst und die Instruktionsadressen eine Breite von 16 Bit, d.h. es können 64k x 16 Bit Programmspeicher adressiert werden. Datenwörter besitzen eine Breite von 8 Bit und es können 256 Speicherplätze (entsprechend einer Datenadreßbreite von 8 Bit) direkt oder indirekt angesprochen werden. Es gibt keine Unterscheidung zwischen Registern und Datenspeicherbereich, d.h. es ist nur der einheitlich adressierbare Datenspeicherbereich von 256 Registern verfügbar. Dabei sind alle Spezialregister des Kerns zur Ansteuerung in den Datenspeicherbereich eingeblendet und werden wie die übrigen Allzweckregister angesprochen. Aufgrund des orthogonalen Befehlssatzes, kann im Prinzip jeder Befehl auf jedes der 256 Register, inklusive Spezialregister, angewendet werden.

Neben den für den Instruktions- und Datenpfad benötigten Hauptkomponenten enthält der Kern zusätzlich noch folgende Blöcke: Erzeugung der internen Taktsignale aus dem externen Takt (siehe Abschnitt 2.2), Interrupt-Logik (Abschnitt 2.10) und das RTCC-Modul (Abschnitt 2.11).

2.2 Systemtaktschema

Zum Betrieb des Mikrocontrollerkerns muß ein extern erzeugter Takt an den entsprechenden Takteingang des Kerns (*CLK*) angelegt werden. Die verschiedenen beim PIC17 vorhanden Oszillatoren und die Konfiguration der Oszillatormodi durch Programmieren diverser Bits im EPROM stehen hier nicht zur Verfügung, d.h. der Prozessormakro entspricht einem PIC17 im externen Oszillatormodus. Eine entsprechende Oszillatorschaltung muß also ggf. zusätzlich zum Mikrocontrollerkern und eventueller Peripherie noch auf dem Chip untergebracht werden.

Analog zum PIC17 verwendet auch der Mikrocontrollerkern intern ein Taktschema, bei dem der an den Kern angelegte Takt (*CLK*) heruntergeteilt und in vier Phasen (Q1, Q2, Q3 und Q4) zerlegt wird. Für jede der vier Phasen wird ein entsprechender Takt (*CLK1*, *CLK2*, *CLK3* und *CLK4*) mit einem Tastverhältnis von 1:3 erzeugt, der in der zugehörigen Phase einen „High“-Pegel annimmt (siehe Abbildung 2.1). Ein kompletter Instruktionszyklus setzt sich aus den vier Taktphasen Q1 bis Q4 zusammen, so daß die interne Taktfrequenz ein Viertel der externen Taktfrequenz beträgt.

Zusätzlich zu den vier internen Taktsignalen, entsprechend den vier internen Taktphasen, wird ein weiteres Taktsignal, *CLKOUT*, erzeugt, das der internen Taktfrequenz (d.h. der durch vier geteilten externen Taktfrequenz) in einem Tastverhältnis von 1:1 entspricht. Dieses Taktsignal wird über die Schnittstelle nach außen geführt und kann so in einer externen Beschaltung verwendet werden, falls dort ein Takt entsprechend den internen Taktverhältnissen benötigt wird.

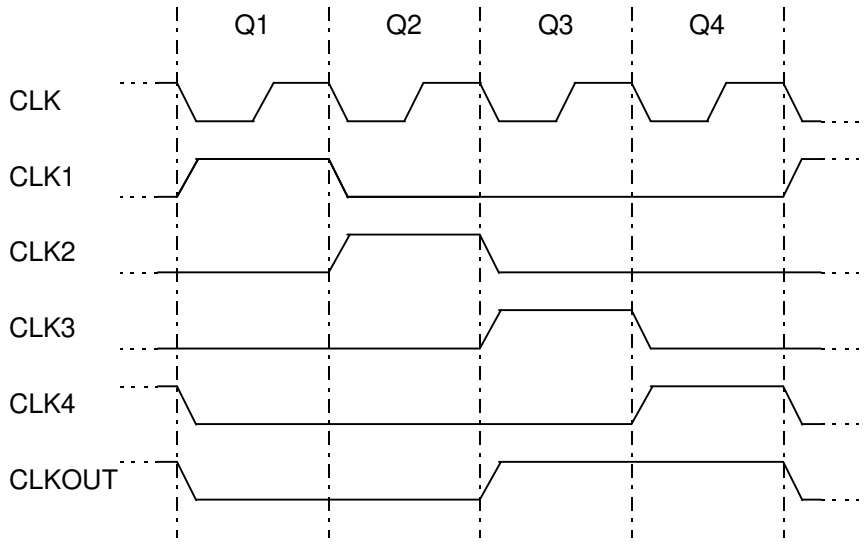


Abbildung 2.1: Internes Taktschema.

Auf der einen Seite ist die interne Teilung des Taktes natürlich mit zusätzlichem Aufwand für eine entsprechende Logik innerhalb des Controllerkerns zur Erzeugung der vier internen Takte aus dem externen Systemtakt verbunden. Auf der anderen Seite hat dieses Verfahren aber den Vorteil, daß die diversen benötigten Kontrollsignale, z.B. zur Ansteuerung von externem RAM, direkt aus den vier internen Taktsignalen abgeleitet werden können und so auch ein komplexeres Zeitverhalten leicht verwirklicht werden kann. Andernfalls, bei nur einem internen Systemtakt, müßten solche zeitkritischen Steuersignale mit Hilfe von Verzögerungsschaltungen o.ä. aus dem Systemtakt abgeleitet werden. Neben der Schwierigkeit der Entwicklung und Synthese einer entsprechenden Schaltung bringt dieses Vorgehen eine Reihe anderer Nachteile mit sich, z.B. die Abhängigkeit der zeitlichen Zusammenhänge von den Verzögerungszeiten der verwendeten Gatter und damit eine eventuell nötige Änderung der Verzögerungsschaltung bei einem Wechsel der Technologie.

2.3 Reset-Logik

Im Gegensatz zum Vorbild PIC17 implementiert der Mikrocontrollerkern dieser Arbeit nur eine recht einfache Reset-Logik. Als einzige Reset-Quelle steht der externe Eingang *nMCLR* zur Verfügung, der einen Reset auslöst, sobald er auf logisch Null geht (low aktiv). Die beim PIC17 bereits auf dem Chip zusätzlich realisierte Reset-Logik, wie Power-Up Reset oder Reset durch Watch-Dog Timeout, zählen nicht direkt zum Kern und werden aufgrund der höheren Flexibilität nicht unmittelbar im Prozessormakro unterstützt. Werden solche Funktionen benötigt, so müssen entsprechende Schaltungen extern an den Reset-Eingang des Prozessormakros angeschaltet und auf dem Chip zusätzlich integriert werden.

In Abbildung 2.2 ist das Timing für einen Reset wiedergegeben. Mit der fallenden Flanke von *nMCLR* wird der Reset aktiv, die Ausführung des gerade ablaufenden Befehls wird sofort abgebrochen und der Prozessorkern begibt sich in den Reset-Status, der der Taktphase Q1 entspricht. Solange der Reset aktiv ist (*nMCLR* low) wird der Kern in der Taktphase Q1 gehalten, mit dem entsprechenden Zustand der diversen Signale an den Schnittstellen. Mit der steigenden Flanke von *nMCLR* wird der Reset inaktiv und der Prozessorkern fährt mit der Ausführung an dieser Stelle durch Übergang in die nächste Taktphase (Q2) fort.

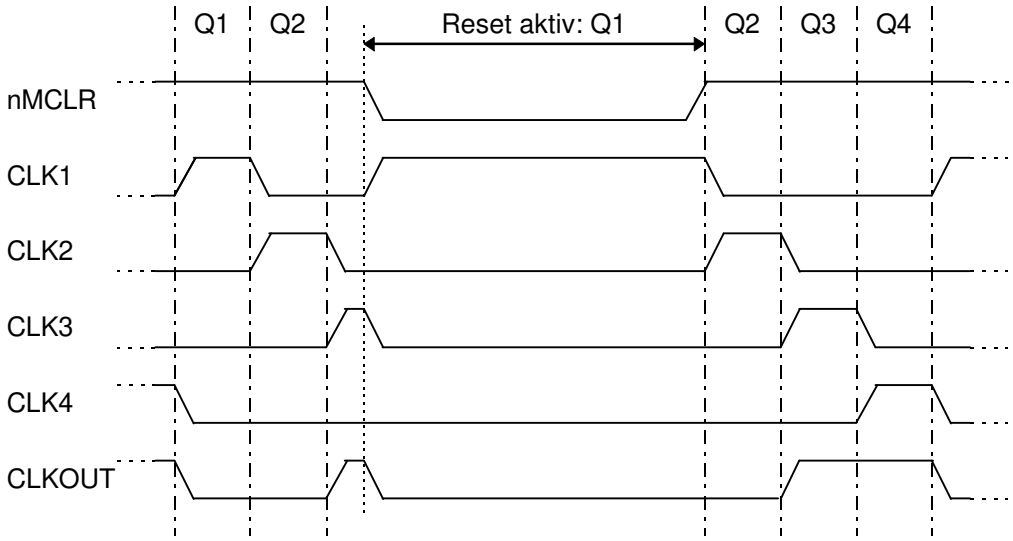


Abbildung 2.2: Reset-Timing.

2.4 Speicherorganisation

2.4.1 Programmspeicher

Von den 64k x 16 Bit Programmspeicher (Adreßbereich 0000h - FFFFh), die aufgrund der 16 Bit breiten Instruktionsadreß- und Instruktionsdatenbusse angesprochen werden können, sind beim originalen PIC17-Mikrocontroller bereits 2k-Worte als EPROM direkt auf dem Chip realisiert. Zusätzlich können bis zu 64k x 16 Bit externer Programmspeicher an den Chip angeschlossen werden. Der PIC17 Prozessor unterstützt dabei prinzipiell drei verschiedene Betriebsmodi bezüglich der Konfiguration von externem und internem Programmspeicher: Betrieb mit internem Speicher, d.h. es können nur die internen 2k-Worte Programmspeicher angesprochen werden (sog. *Microcontroller Mode*: Zugriffe auf Adressen oberhalb 2k liefern 0000h bzw. haben keine Schreibzugriffe zur Folge), Mischbetrieb mit internem Programmspeicher bis 2k-Worte und externem Programmspeicher darüber (sog. *Extended Microcontroller Mode*) oder reiner Betrieb aus externem Programmspeicher, d.h. alle Programmspeicherzugriffe werden nach außen gegeben (sog. *Microprocessor Mode*). Weiterhin existiert noch ein sog. *Protected Microcontroller Mode*, bei dem der Datenaustausch zwischen Datenspeicher und internem Programmspeicher gesperrt ist, um den Programmcode zu schützen. Die (dauerhafte) Einstellung des Betriebsmodus beim PIC17 Mikrocontroller erfolgt durch Programmieren verschiedener spezieller Zellen des internen EPROMs. Alle Zellen zur Konfiguration des PIC17-Mikrocontrollers (Einstellung des Betriebsmodus, Konfiguration des Watchdog Timers und Auswahl der Oszillatorschaltung) und der Festspeicher, der die Test- und Bootprogramme enthält, sind im Mikrocontroller Modus in den oberen Bereich (Adressen FE00h - FFFFh) des Programmspeicheradreßraums eingeblendet. In den anderen Betriebsmodi kann auf die Konfigurationszellen und auf die Test- und Bootprogramme nicht zugegriffen werden.

Der in dieser Arbeit entwickelte Prozessormakro enthält nur den reinen Mikrocontrollerkern und beinhaltet keinen internen Programmspeicher, d.h. der komplette Programmspeicherbereich wird über eine Schnittstelle des Prozessormakros extern an den Kern angeschaltet, wobei der Programmspeicher dabei in beliebiger Kombination und Größe auf dem Chip oder außerhalb des Chips realisiert werden kann. Alle Programmspeicherzugriffe beim Lesen von Instruktionen oder beim Austausch von Daten zwischen Programm- und

Datenspeicher werden über diese Schnittstelle abgewickelt. Deshalb muß innerhalb des Kerns keine Unterscheidung zwischen externen oder internen Programmspeicherbereichen und zwischen den verschiedenen Betriebsmodi des PIC17 gemacht werden. Da der Mikrocontrollerkern weiterhin auch keine Test- oder Bootprogramme und keine programmierbaren Konfigurationsmöglichkeiten enthält, kann natürlich auch nicht auf entsprechende Daten zugegriffen werden. Diese Festlegungen stellen aber keine Einschränkungen der Kompatibilität im Hinblick auf den Einsatz von PIC17 Entwicklungswerkzeugen (Assembler, Simulator, etc.) dar, da der Mikrocontrollerkern einem PIC17 im Microprocessor Modus entspricht.

Wie beim PIC17 werden folgende Adressen im Programmspeicher als Interrupt- und Resetvektoren festgelegt, d.h. nach einem Reset bzw. bei einem Interrupt verzweigt die Programmausführung an die entsprechende Adresse und wird dort fortgesetzt:

- Adresse 0000h: Reset-Routine,
- Adresse 0008h: Interrupt-Serviceroutine des ersten externen Interrupts,
- Adresse 0010h: RTCC-Überlauf Interrupt-Serviceroutine,
- Adresse 0018h: Interrupt-Serviceroutine des zweiten externen Interrupts,
- Adresse 0020h: Peripherieinterrupt-Serviceroutine.

2.4.2 Datenspeicher (Register)

Zur Speicherung von Daten besitzt der Mikrocontrollerkern einen einheitlich adressierbaren Datenspeicherbereich von maximal 256 Registern (entsprechen der Breite von 8 Bit für die Adressierung der Datenzugriffe) mit einer Registerbreite von jeweils 8 Bit. Es können nur diese 256 Datenspeicherplätze angesprochen werden und ein zusätzlicher, getrennt adressierbarer Datenspeicher (RAM) ist nicht vorhanden. Jedes der 256 Register kann als Argument für jeden der Befehle, die ein Registerargument benötigen, eingesetzt werden. Der Registerbereich kann im Prinzip in zwei Blöcke unterteilt werden:

Spezialregisterbereich (Adressen 00h bis 0Fh): Dieser Bereich enthält die in den Datenspeicherbereich eingeblendeten 16 Spezialregister (Statusregister, Programmzähler, W-Register, etc.) des Mikrocontrollerkerns analog zum PIC17 Mikrocontroller (siehe Abschnitt 2.6). In diesem Bereich sind auch die beiden Steuerregister des RTCC-Moduls, das beim PIC17 Mikrocontroller zum Kern zählt, eingeblendet. Da die Register alle direkt im Kern für den normalen Kontrollfluß benötigt werden und deshalb an verschiedenen Stellen innerhalb des Kerns realisiert sind, werden Zugriffe auf diese Register innerhalb des Kerns gehandhabt und nicht über eine Schnittstelle nach außen geführt.

Analog zum PIC17-Mikrocontroller sind Zugriffe auf diese Register teilweise mit gewissen Einschränkungen bzw. Sonderfunktionen verbunden. Außerdem werden einige dieser Register aufgrund ihrer Spezialfunktion bei vielen Befehlen implizit manipuliert bzw. werden implizit als Argument, d.h. als Quelle oder Ziel verwendet.

Peripherieregister/Allzweckregister (Adressen 10h bis FFh): Beim originalen PIC17 Mikrocontroller ist dieser Bereich weiter unterteilt: In den unteren 8 Speicherplätze dieses Bereichs (Adressen 10h bis 17h) sind dort die Peripherieregister zur Ansteuerung und Konfiguration der auf dem Chip integrierten Peripheriebausteine, außer des RTCC-Moduls, eingeblendet. Da diese 8 Speicherplätze aber nicht alle Peripherieregister aufnehmen können ist beim PIC17-Mikrocontroller zusätzlich noch für den Bereich von Adresse 10h bis Adresse 17h eine Speicherbankumschaltung möglich, wobei die Bankumschaltung aber nur für diesen Bereich und nicht für den restlichen Adreßbereich

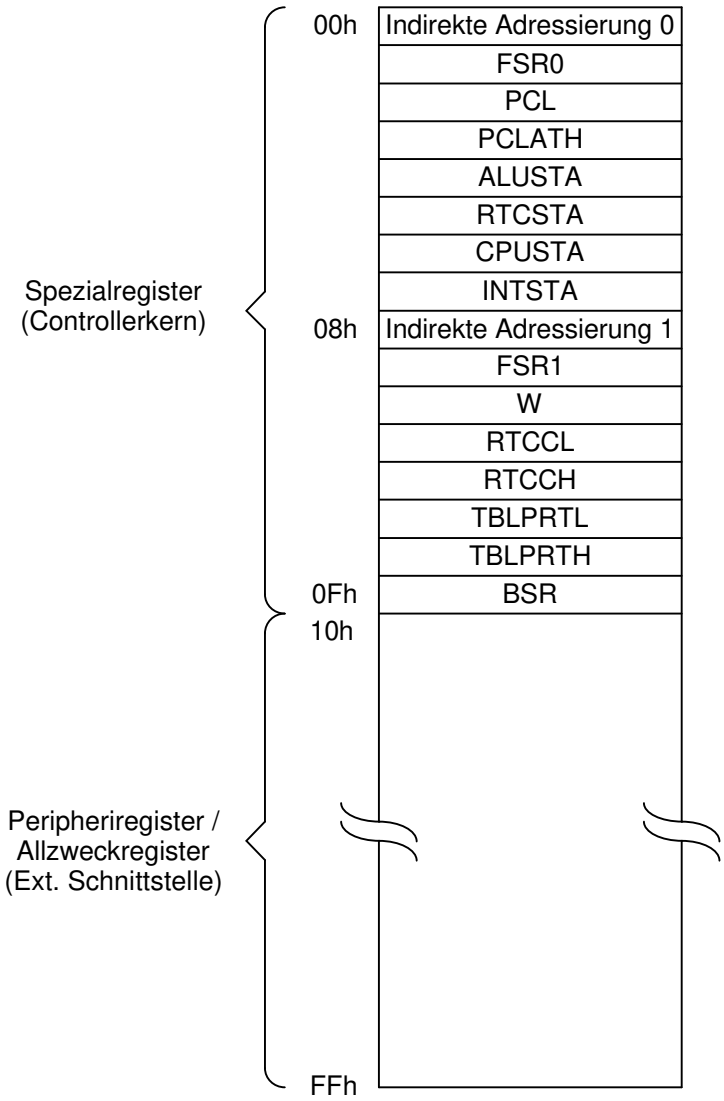


Abbildung 2.3: Organisation des Datenspeicher-/Registerbereichs.

gilt. Die Auswahl der Peripheriespeicherbank erfolgt dabei über das sog. *Bank Select Register* (*BSR*), d.h. um ein Peripherieregister in einer anderen Bank zu adressieren, muß erst das BSR mit dem entsprechenden neuen Wert geladen werden. Beim PIC17C42 werden für die Register der integrierte Peripherie nur die unteren 2 Bit des BSR zur Auswahl aus 4 Bänken benötigt. Für zukünftige Erweiterungen des Peripherieumfangs sind beim PIC17-Mikrocontroller sogar 4 Bit des BSR für maximal 16 mögliche Bänke implementiert. Prinzipiell besitzt das BSR aber eine Datenbreite von 8 Bit, so daß es möglich wäre bis zu 256 Bänke zu verwirklichen.

Im restlichen oberen Adreßbereich (Adressen 18h bis FFh) sind schließlich beim PIC17 Mikrocontroller die Allzweckregister untergebracht. Dieser Registerbereich ist nicht mit irgendwelchen Spezialfunktionen vorbelegt und bildet damit den Teil der allgemein verwendbaren Datenregister.

Da bei dem in dieser Arbeit entwickelten Prozessormakro die angesprochene Peripherie nicht im Kern integriert ist und damit auch keines der beim PIC17 dort eingeblendeten Peripherie-Spezialregister bereits im Kern vorhanden sein kann (die Register des im Kern vorhandenen RTCC-Moduls liegen nicht in diesem Bereich, sondern im Spezialregisterbereich des Kerns von 00h bis 0Fh), entfällt diese starre Unterteilung in Peripherieregister und Allzweckregister hier und die beiden Blöcke

werden zu einem gemeinsamen Registerbereich zusammengefaßt. Zugriffe auf diesen Bereich werden über eine Schnittstelle des Prozessormakros mit Daten-, Adreß- und Steuerbus nach außen geführt. Dabei läßt sich extern, falls gewünscht, anhand einzelner Bits des Adreßbusses immer noch leicht eine Differenzierung der Bereiche zwischen Peripherieregistern und Allzweckregistern vornehmen. Durch die Abwicklung der Zugriffe über eine externe Schnittstelle wird bei einer endgültigen Realisierung eines auf dem Prozessormakro basierenden Mikrocontroller-Chips eine größtmögliche Flexibilität bezüglich der tatsächlichen Datenregisterkonfiguration gewährleistet. Allein die externe Beschaltung des Makros mit RAM und Peripherie auf dem Chip entscheidet über Lage und Größe von Peripherie- und Allzweckregistern im Datenbereich.

Um zusätzlich auch eine möglichst große Flexibilität bezüglich der Bankauswahl zur ermöglichen, kann beim Prozessormakro das BSR beliebig breit implementiert werden (über sog. VHDL *Generics* und Neu-Synthese des Makros) und wird komplett nach außen geführt. Über eine entsprechende Dekodierung des BSR und der Adreßleitungen können beliebige Konfigurationen bezüglich Bankauswahl und Anzahl Register von beliebiger extern angeschalteter Peripherie erreicht werden. Zusätzlich besteht dabei auch die Möglichkeit, das BSR extern auch bei Zugriffen auf andere Speicherbereiche (d.h. bei Zugriffen auf Allzweckregister) auszuwerten und so, durch Erweiterung diverser allgemeiner Datenspeicherbereiche per Bankumschaltung, auf Kosten der Kompatibilität eine Vergrößerung des Datenspeicherbereichs zu erreichen. Auf der anderen Seite können Zugriffe auf andere Speicherbereiche das BSR natürlich ignorieren und so absolute Kompatibilität sicherstellen.

Insgesamt gesehen entscheidet alleine die externe Beschaltung über die Konfiguration des gesamten Bereichs von 10h bis FFh im Hinblick auf Lage, Größe, Bankauswahl, etc. der Peripherieregister und Allzweckregister. Erst bei der Programmierung des fertigen Chips muß die tatsächlich zur Verfügung stehende externe Konfiguration beachtet werden.

2.5 Schnittstellenbeschreibung

2.5.1 Übersicht

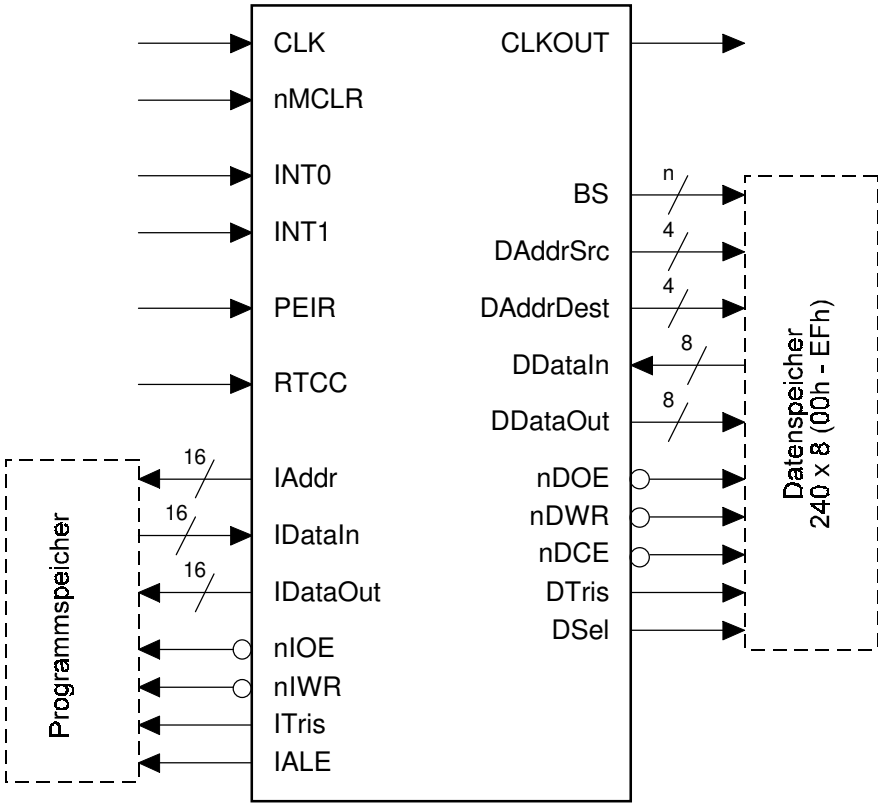


Abbildung 2.4: Schnittstellen des Mikrocontrollerkerns.

Name	Typ	Breite	Beschreibung
CLK	In	1	Eingang für den Systemtakt, aus dem die internen Taktsignale abgeleitet werden.
CLKOUT	Out	1	Ausgang für den internen Takt, d.h. der durch vier heruntergeteilte angelegte Systemtakt.
nMCLR	In	1	Reset-Eingang. Active low.
INT0	In	1	Eingang für ersten externen Interrupt. Interrupt konfigurierbar bei fallender oder steigender Flanke.
INT1	In	1	Eingang für zweiten externen Interrupt. Interrupt konfigurierbar bei fallender oder steigender Flanke.
PEIR	In	1	(Peripheral Interrupt Request) Eingang für Anforderung eines Peripherie-Interrupts.
RTCC	In	1	(Real Time Clock Counter) Externer Takteingang für Ereigniszähler.

Tabelle 2.1: Allgemeine Steuersignale.

Name	Typ	Breite	Beschreibung
IAddr	Out	16	Instruktionsadreßbus. Programmspeicheradresse für Speicherzugriff.
IDataIn	In	16	Instruktionsdatenbus-Eingang. Daten-/Befehlswort, das aus dem Programmspeicher gelesen wird.
IDataOut	Out	16	Instruktionsdatenbus-Ausgang. Daten-/Befehlswort, das in den Programmspeicher geschrieben wird.
ITris	Out	1	Steuersignal für Tristate-Treiber des Instruktionsdatenbus. Low: IDataOut treibt den Bus (Tristate-Treiber aktiv). High: Bus hochohmig (Tristate-Treiber inaktiv).
IALE	Out	1	(Address Latch Enable) Steuersignal für Einlatchen gültiger Programmspeicheradresse und für Adressen-Tristatetreiber. High: Programmspeicher übernimmt Adresse, IAddr treibt gemultiplexten Adreß-/Datenbus. Low: Adreß-Tristatetreiber hochohmig (inaktiv).
nIOE	Out	1	(Output Enable) Steuersignal Programmspeicherausgabe. Low: Programmspeicher treibt den Datenbus (Read/Output). High: Programmspeicherausgabe inaktiv (hochohmig).
nIWR	Out	1	(Write Enable) Steuersignal Programmspeicher Schreiben. Low: Datenwort in Programmspeicher schreiben (Write). High: Schreiben Programmspeicher inaktiv.

Tabelle 2.2: Schnittstelle zum Programmspeicher.

Name	Typ	Breite	Beschreibung
DAddrSrc	Out	8	Datenadreßbus Quelle. Datenspeicheradresse für Lesezugriff. (Adreßbereich 00h bis EFh).
DAddrDest	Out	8	Datenadreßbus Ziel. Datenspeicheradresse für Schreibzugriff. (Adreßbereich 00h bis EFh).
DSel	Out	1	Steuersignal für Multiplexen des Adreßbusses. High: Quelladresse auf Adreßbus selektieren. Low: Zieladresse auf Adreßbus selektieren.
DDataIn	In	8	Datenspeicherdatenbus-Eingang. Datenwort, das aus dem Datenspeicher gelesen wird.
DDataOut	Out	8	Datenspeicherdatenbus-Ausgang. Datenwort, das in den Datenspeicher geschrieben wird.
DTris	Out	1	Steuersignal für Tristate-Treiber des Datenspeicherdatenbus. Low: DDataOut treibt den Bus (Tristate-Treiber aktiv). High: Bus hochohmig (Tristate-Treiber inaktiv).
nDOE	Out	1	(Output Enable) Steuersignal Datenspeicherausgabe. Low: Datenspeicher treibt den Datenbus (Read/Output). High: Datenspeicherausgabe inaktiv (hochohmig).
nDWR	Out	1	(Write Enable) Steuersignal Datenspeicher Schreiben. Low: Datenwort in Datenspeicher schreiben (Write). High: Schreiben Datenspeicher inaktiv.

nDCE	Out	1	(Chip Enable) Steuersignal Datenspeicheraktivierung. Low: Speicherbaustein aktiv (Enable). High: Speicherbaustein inaktiv (Precharge).
BS	Out	n	(Bank Select) Auswahl der Speicherbank. Inhalt des BSR. Die Breite des Ausgangs ist variable und hängt von der Konfiguration des Mikrocontrollerkerns ab (maximal 8).

Tabelle 2.3: Schnittstelle zum Datenspeicher.

2.5.2 Programmspeicher

Die Schnittstelle des Prozessormakros zum Programmspeicher besteht aus Adreß-, Daten- und Steuerbus. Der Adreßbus hat eine Breite von 16 Bit entsprechend den maximal adressierbaren 64k Programmspeicher. Der Datenbus hat eine Breite von 16 Bit, d.h. pro Schreib- oder Lesezyklus wird jeweils ein 16 Bit breiter Befehl oder ein 16 Bit breites Datenwort zwischen Programmspeicher und Prozessormakro transferiert. Da der Befehlssatz des PIC17 Mikrocontrollers über die Instruktionen `TABLRD` und `TABLWT` einen Austausch von Daten zwischen Programm- und Datenspeicher vorsieht, muß die Möglichkeit gegeben sein, den Programmspeicher auch zu beschreiben. Um dies zu ermöglichen, sind in der Schnittstelle zum Programmspeicher entsprechende Steuerleitung und ein bidirektionaler Datenbus vorhanden. Wird extern ein Programmspeicher angeschlossen, der nicht beschrieben werden kann (ROM), so muß durch entsprechende externe Beschaltung der Programmspeicherschnittstelle oder durch entsprechende Programmierung des Mikrocontrollers dafür gesorgt werden, daß Schreibzugriffe auf den Programmspeicher keine Auswirkungen haben bzw. gar nicht erst vorkommen können.

Um eine größtmögliche Flexibilität bezüglich der Auswahl von geeigneten Tristate-Treibern für den bidirektionalen Datenbus bzw. bezüglich der tatsächlichen Realisierung des Datenbusses am externen Programmspeicher zu haben, wird der Datenbus nicht als bidirektionaler Tristate-Bus aus dem Mikrocontrollerkern herausgeführt. Statt dessen wird separat eine Steuerleitung für eventuell vorhandene Bus-Tristatetreiber (*ITris*) und für jedes der 16 Bit des Datenbusses jeweils eine Eingangs- und eine Ausgangsleitung (*IDataIn[0:15]* und *IDataOut[0:15]*) herausgeführt. Damit kann der Datenbus extern beliebig realisiert werden und es ist z.B. auch möglich, Datenbuseingang und Datenbusausgang unterschiedliche Funktionen zuzuweisen. Um auf Seite des Mikrocontrollerkerns einen bidirektionalen Tristate-Datenbus zu erhalten, müssen die Datenbusleitungen des Prozessormakros, wie in Abbildung 2.5 gezeigt, extern mit Tristate-Treibern beschaltet werden.

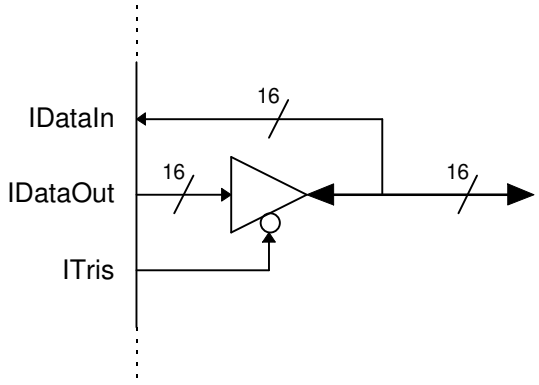


Abbildung 2.5: Externe Beschaltung des Prozessormakros mit Tristate-Treibern für einen bidirektionalen Instruktionsdatenbus.

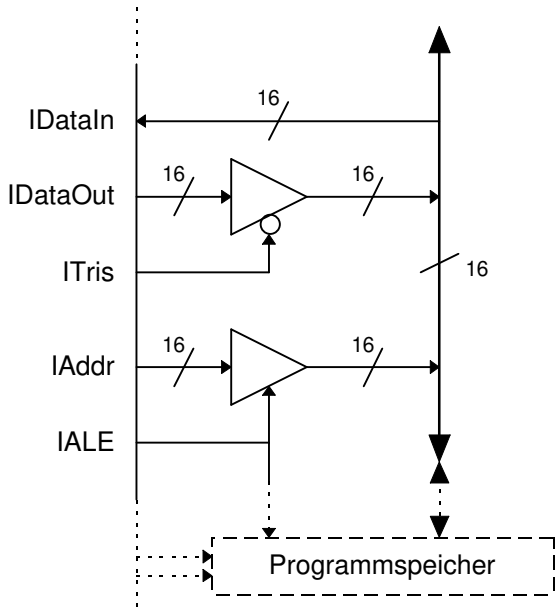


Abbildung 2.6: Externe Beschaltung des Prozessormakros mit Tristate-Treibern für einen gemultiplexten, bidirektionalen Instruktionsadreß- und Datenbus.

Zusätzlich zu einem externen bidirektionalen Datenbus kann mit dem vom Mikrocontrollerkern zur Verfügung gestellten Steuersignal *IALE (Address Latch Enable)* extern ein gemultiplexter Adreß- und Datenbus verwirklicht werden, um bei einer Verwirklichung des Programmspeichers außerhalb des Chips die Anzahl der Pins des Mikrocontrollers so gering wie möglich halten zu können. Das Signal *IALE* dient dabei zur Steuerung der Adreßübernahme des Programmspeicheradreßlatches, d.h. dem Programmspeicher wird mit aktivem *IALE* signalisiert, daß die Daten auf dem gemeinsamen Bus gültige Adressen darstellen und vom Programmspeicher übernommen werden müssen. Um auf Mikrocontrollerkernseite den gemultiplexten Adreß- und Datenbus zu verwirklichen, besteht auf der einen Seite die Möglichkeit dies mit Hilfe von Tristate-Treibern zu erreichen. In diesem Fall dient das Signal *IALE* auf Mikrocontrollerkernseite dazu, die Tristate-Treiber der Adreßleitungen auf dem gemeinsamen Bus zu steuern. Zusätzlich zu der Beschaltung mit Tristate-Treibern für die Datenleitungen müssen zur Verwirklichung eines gemeinsamen Adreß- und Datenbusses dann auch die Adreßleitungen mit Tristate-Treiber, wie in Abbildung 2.6 gezeigt, beschaltet werden. Diese Art der externen Beschaltung hat aber den Nachteil, daß Probleme auftreten können, wenn bei überlappenden *IALE*- und *ITris*-Signalen beide Treiber gleichzeitig versuchen den Bus zu treiben. Gegebenenfalls muß das Signal *ITris* entsprechend verzögert werden, um dies zu vermeiden. Deshalb, und auch aufgrund von Geschwindigkeits- und Platzvorteilen, wird eine externe Verwirklichung des gemeinsamen Adreß- und Datenbusses mit Hilfe eines Multiplexers bevorzugt (siehe Abbildung 2.7). Dabei wird das Signal *IALE* als Steuerleitung des Multiplexers zur Auswahl von Adressen oder Daten verwendet. Das Ausgangssignal des Multiplexers geht dann auf die Bustreiber, die aktiviert werden durch *IALE* high oder *ITris* low.

Zur Steuerung der Schreib- und Lesezugriffe auf den Programmspeicher werden vom Mikrocontrollerkern folgende Steuerleitung an der Schnittstelle zum Programmspeicher zur Verfügung gestellt:

nIOE (*Output Enable*): Active low, schaltet die Datenbusausgangstreiber des Programmspeichers aktiv, d.h. *nIOE* = low bei einem Lesezyklus veranlaßt den Programmspeicher, die adressierten Daten auf den Datenbus zu gegeben (Read/Output).

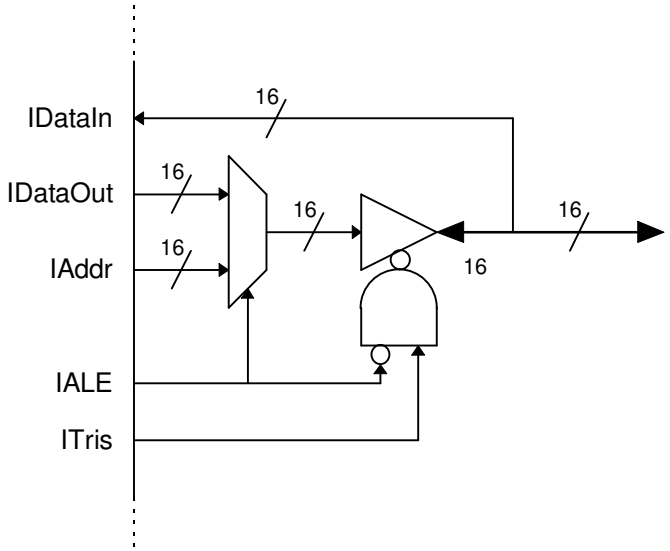


Abbildung 2.7: Externe Beschaltung des Kerns mit Multiplexer für einen gemeinsamen bidirektionalen Adreß- und Datenbus.

nIWR (*Write Enable*): Active low, steuert das Schreiben des Programmspeichers, d.h. nIWR = low bei einem Schreibzyklus veranlaßt den Programmspeicher, die auf dem Datenbus anliegenden Daten in die adressierte Speicherstelle zu schreiben (Write).

Sind beide Steuerleitung high (inaktiv), so findet keine Aktion statt.

Ein Schreib- oder Lesezugriff auf den externen Programmspeicher umfaßt einen Instruktionszyklus, d.h. alle vier internen Taktphasen Q1 bis Q4. Abbildung 2.8 zeigt das Timing für Schreib- und Lesezugriffe. Während Phase Q2 werden die gültigen Adressen auf den Bus gelegt und *IALE* aktiv gesetzt. Mit der fallenden Flanke von *IALE* übernimmt der Programmspeicher die anliegenden Adressen. Bei einem Instruktions- oder Datenlesezyklus wird während der Phasen Q3 und Q4 nIOE aktiviert und der Programmspeicher gibt die adressierten Daten auf den Bus, die daraufhin vom Prozessormakro übernommen werden. Zu beachten ist, daß Probleme entstehen können, wenn die Datentreiber des Speichers bereits aktiv werden, während die Bustreiber des Kerns noch nicht ganz ausgeschaltet sind. Um dies

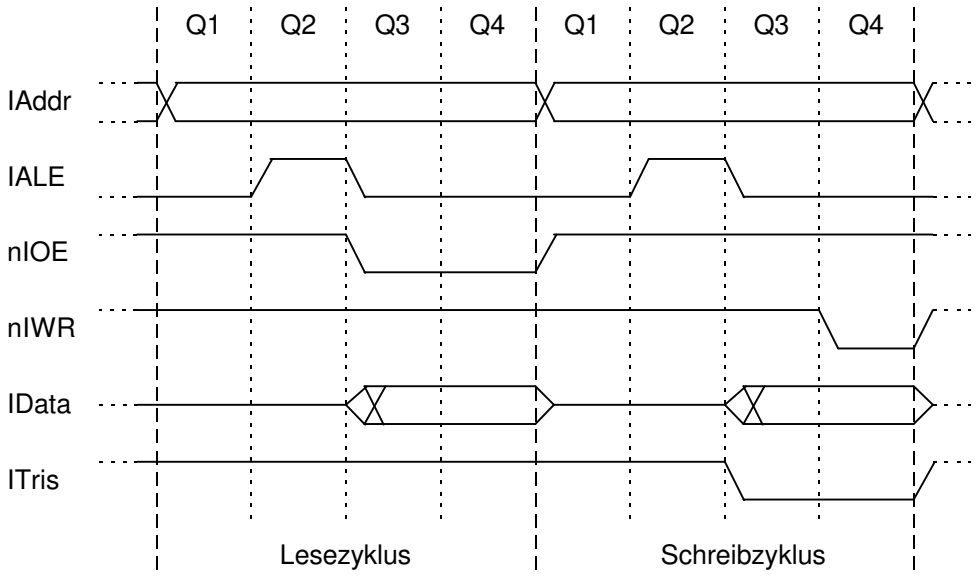


Abbildung 2.8: Timing für Schreib- und Lesezugriffe auf Programmspeicher.

zu verhindern, muß ggf. das Signal *nIOE* so verzögert werden, daß sichergestellt ist, daß die Treiber auf beiden Seiten nicht gleichzeitig aktiv sein können. Durch die zwischen der steigenden Flanke von *nIOE* und der erneuten Adreßausgabe liegende Taktphase Q1 wird auf jeden Fall sichergestellt, daß die Datenausgabetreiber des Programmspeichers sicher abgeschaltet sind, wenn die Adreßtreiber des Kerns wieder aktiviert werden.

Bei einem Schreibzyklus werden die Datentreiber auf Mikrocontrollerseite während der Phase Q3 und Q4 aktiviert und damit die zu schreibenden Daten auf den Bus gelegt. Auch hier ist zu beachten, daß beim Übergang von Phase Q2 zu Phase Q3 auf Seite des Mikrocontrollerkerns, bei Einsatz von Tristate-Treibern zur Verwirklichung des gemeinsamen Adreß- und Datenbusses, die Treiber für Adressen und Daten bei teilweise überlappenden Steuersignalen *ITris* und *IALE* nicht gegeneinander treiben. Dies kann entweder wiederum durch eine Verzögerung des Signals *ITris* oder durch eine entsprechende Beschaltung des Kerns mit Multiplexern statt Tristate-Treibern (s.o., Abbildung 2.7), bei der dieses Problem gar nicht erst auftritt, verhindert werden. Während der Phase Q4 wird schließlich *nIWR* aktiv und die Daten können vom Programmspeicher entweder mit der fallenden oder mit der steigenden Flanke von *nIWR* übernommen werden, da in beiden Fällen gültige Daten anliegen.

2.5.3 Datenspeicher

Analog zur Programmspeicherschnittstelle besteht auch die Schnittstelle für Peripherie- und Allzweckregister aus Adreß-, Daten- und Steuerbus. Entsprechend der Daten- und Registerbreite des Mikrocontrollerkerns hat der Datenbus hier eine Breite von 8 Bit und ist wie bei der Schnittstelle zum Programmspeicher nicht als bidirektionaler Tristate-Bus herausgeführt, sondern es werden separat eine Steuerleitung für evtl. Tristate-Treiber (*DTris*) und Dateneingangs- (*DDataIn[0:7]*) und Datenausgangsbusse (*DDataOut[0:7]*) bereitgestellt. Durch entsprechende externe Beschaltung, wie in Abschnitt 2.5.2, Abbildung 2.7 gezeigt, kann auf Seite des Mikrocontrollerkerns ein bidirektionaler Tristate-Datenbus erhalten werden.

Der Adreßbus der Datenspeicherschnittstelle ist 8 Bit breit, wobei von den 256 möglichen Speicherplätzen nur maximal 240 (Datenspeicherbereich der normalen Register, ohne den bereits im Kern dekodierten Bereich der 16 Spezialregister), entsprechend einem möglichen Adreßbereich von Adresse 10h bis Adresse FFh, angesprochen werden können. Um die Realisierung der externen Adreßdekodierung ggf. zu vereinfachen, kann der ansprechbare Adreßbereich durch eine Invertierung der Adreßleitungen außerhalb des Kerns in den Bereich von 00h bis EFh transformiert werden.

Da durch Verwendung der Instruktionen `MOVPF` und `MOVFP` mit indirekter Adressierung (siehe dazu Abschnitt 2.7.1) während eines Instruktionszyklus bzw. während eines Read-Modify-Write Zyklus an unterschiedlichen Adressen innerhalb des gesamten Adreßbereichs gelesen und geschrieben werden kann, besteht der Adreßbus aus zwei separaten, jeweils 8 Bit breiten Bussen für Lese- und Schreibadresse. Da beide Adressen schon zu Beginn eines Instruktionszyklus zur Verfügung stehen, wird sichergestellt, daß die Adressen auf dem Adreßbus so früh wie möglich anliegen und sich während der Dauer eines Instruktionszyklus nicht ändern. Dies ermöglicht eine große Flexibilität bezüglich der Wahl der geeigneten anzuschließenden Speicherart, auch im Hinblick auf Zugriffsgeschwindigkeiten, und es ist z.B. auch möglich ein dual-ported-RAM einzusetzen. Andernfalls, bei nur einem gemeinsamen Adreßbus für Quell- und Zieladresse, müßte die Adresse auf dem Adreßbus zwischen Lese- und Schreibzugriff umgeschaltet werden. Um dieses Verhalten aber auch mit der bestehenden Schnittstelle erreichen zu können, wird das Steuersignal *DSel* von der Schnittstelle bereitgestellt. Das Signal kann dazu benutzt werden, die getrennten Quell- und Zieladreßbusse auf einen gemeinsamen Adreßbus zu multiplexen. Mit dem Signal wird gewählt, welche der beiden Adressen auf den gemeinsamen Bus gelegt wird.

Zur Steuerung der Speicherzugriffe sind analog zur Programmspeicherschnittstelle die Steuersignale *nDOE* und *nDWR* vorhanden. Mit diesen beiden Steuersignalen besteht die

Möglichkeit, Latches über einen bidirektionalen Datenbus an diese Schnittstelle anzuschließen. Das Signal *nDWR* wird dabei zur Steuerung der Latches verwendet, d.h. das Latch ist bei *nDWR* low transparent und bei *nDWR* high hält es die eingelatchten Daten. Das Signal *nDOE* hingegen kann bei Einsatz eines bidirektionalen Datenbusses dazu eingesetzt werden, die Datenbustreiber der Latches zu steuern.

Auf der anderen Seite, aus Gründen der sich ergebenden Platzvorteile, soll die Datenspeicherschnittstelle des Mikrocontrollerkerns aber auch die Möglichkeit bieten, RAM anzuschließen. Am IMS steht ein RAM-Compiler zur Verfügung, der ein statisches RAM beliebiger Größe (derzeit nur als dual-ported RAM, zukünftig auch als normales, single-ported RAM) auf Basis der IMS-Gate-Forest Technologie als Modul, zur Kombination z.B. mit dem Mikrocontrollerkern auf einem gemeinsamen Chip, erzeugen kann [11]. Der damit erzeugte Datenspeicher hat den Vorteil, daß er deutlich weniger Platz auf dem Chip beansprucht als ein Datenspeicher gleicher Größe, der mit Latches realisiert ist. Entsprechend der Schnittstelle des am IMS zur Verfügung stehenden RAMs stellt der Mikrocontrollerkern deshalb zusätzlich noch die Steuerleitung *nDCE* (*Chip Enable*) zur Verfügung. Diese wird benötigt, um vor dem Lesen und Schreiben der Daten ein sog. *Precharge* des RAM durchzuführen. Da die Datenleitungen wesentlich schneller auf den Low-Pegel als auf den High-Pegel gezogen werden können, werden sie beim Precharge bereits auf High-Pegel aufgeladen und müssen beim anschließenden Lesen bzw. Schreiben ggf. nur noch auf den Low-Pegel gezogen werden, wodurch sich eine Beschleunigung der Lese- und Schreibzugriffe ergibt. Wenn die Steuerleitung *nDCE* high ist, dann ist der RAM-Baustein deaktiviert, d.h. die Steuerleitungen *nDWR* und *nDOE* werden ignoriert, und das RAM führt einen Precharge durch.

Abbildung 2.9 zeigt das sich ergebende prinzipielle Timing der vom Mikrocontrollerkern erzeugten Signale an der Schnittstelle zu einem single-ported RAM. Je nach Art der Instruktion fährt der Mikrocontrollerkern entweder einen allgemeinen Read-Modify-Write Zyklus oder einen reinen Lesezyklus, wenn kein Wert in den Datenspeicher geschrieben werden soll. Es wird stets eine dieser beiden Zyklen gefahren, auch wenn zur Ausführung der Instruktion eigentlich kein Wert aus dem RAM gelesen werden müßte. In diesem Fall wird dann ein beliebiger Wert an einer zufälligen Adresse im ersten Teil des Zyklus gelesen und intern nicht weiter verarbeitet, d.h. das RAM kann in diesem Fall auch einen beliebigen, nicht

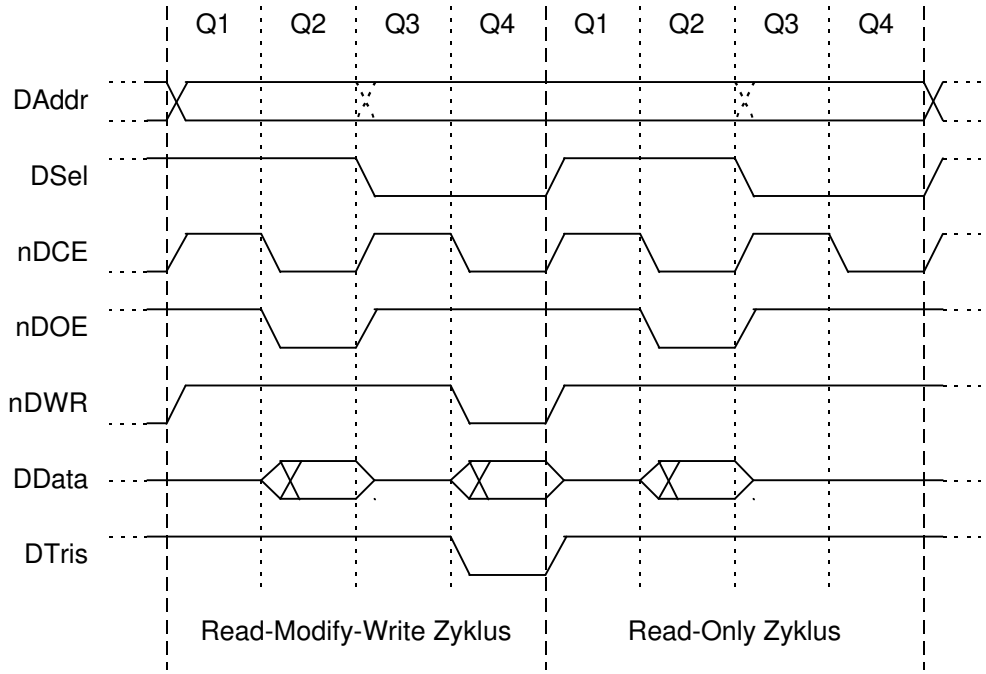


Abbildung 2.9: Timing für Zugriffe auf Datenspeicher.

sinnvollen Wert liefern (z.B. bei zufälligen Zugriffen auf nicht real vorhandene Adreßbereiche).

Jeder Zyklus umfaßt die vier Taktphasen Q1 bis Q4 und damit einen kompletten Instruktionszyklus. Während der ersten Taktphase Q1 wird der Precharge für den Lesezugriff durchgeführt. Während der Taktphase Q2 wird der Lesezugriff selbst ausgeführt und das RAM gibt die adressierten Daten auf den Datenbus. In der dritten Taktphase Q3 wird anschließend der Precharge für den Schreibzugriff durchgeführt, und es werden, falls benötigt, die Adressen auf dem Adreßbus von Leseadresse auf Schreibadresse umgeschaltet. In Q4 wird ggf. der abschließende Schreibzugriff durchgeführt. Dazu werden die Datenbustreiber auf Mikrocontrollerseite aktiviert und damit die Daten des zu schreibenden Ergebnisses auf den Bus gelegt. Durch Aktivierung von *nDWR* wird schließlich das Schreiben gesteuert und die Daten werden vom RAM mit der steigenden Flanke von *nDWR* übernommen. Bei einem reinen Lesezyklus entfällt die Aktivierung der Datenbustreiber und des Signals *nDWR*, so daß kein Schreibzugriff durchgeführt wird. Die Schreibadresse und der Wert auf dem Datenausgangsbus des Mikrocontrollerkerns nehmen dabei zufällige Zustände an.

Eine detailliertere Ansicht des nötigen Timings und der einzuhaltenden Timingkriterien auf Seite des IMS-RAMs zeigt Abbildung 2.10. Die angegebenen Grenzwerte für die einzelnen Zeiten sind die Angaben in ns für ein 256 x 8 Bit RAM als Beispiel. Die einzelnen Zeitkriterien haben dabei folgende Bedeutung:

t_{AS} (Address Setup): Zeit, die angibt, wie lange die Adressen vor der fallenden Flanke von *nCE*, *nOE* oder *nWR* bereits gültig waren. Der vom RAM geforderte Mindestwert muß vom Mikrocontrollerkern eingehalten werden.

t_{AH} (Address Hold): Zeit, die angibt, wie lange die Adressen nach der steigenden Flanke von *nCE*, *nOE* oder *nWR* noch gültig bleiben. Der vom RAM geforderte Mindestwert muß vom Mikrocontrollerkern eingehalten werden.

t_{PRECH} (Precharge): Zeitdauer des precharge, d.h. High-Pegel-Zeitdauer von *nCE*. Vom RAM geforderter Mindestwert darf nicht unterschritten werden.

t_{CP} (Chip Enable Pulse Width): Zeitdauer des Chip-Enable Impulses, d.h. Low-Pegel-Zeitdauer von *nCE*. Mindestdauer muß vom Mikrocontrollerkern eingehalten werden.

t_{WP} (Write Enable Pulse Width): Zeitdauer des Write-Enable Impulses, d.h. Low-Pegel-Dauer von *nWR*. Mindestwert muß eingehalten werden.

t_{HZDV} (High Z to Data Valid): Zeit, die vom RAM ab der fallenden Flanke von *nCE* bzw. *nOE* benötigt wird, bis die gültigen Daten auf dem Datenbus anliegen. Mindestwert gibt an, wie schnell das RAM die Daten maximal bereitstellen kann.

t_{HZOE} (Output Enable High to High Z): Zeitdauer, die das RAM benötigt, bis es den Datenbus nach der steigenden Flanke von *nCE* bzw. *nOE* wieder in den hochohmigen Zustand versetzt hat. Der Maximalwert gibt an, wie lange das RAM maximal braucht bis es seine Datentreiber deaktiviert hat.

t_{DSW} (Data Setup): Zeit, die angibt, wie lange vor der steigenden Flanke von *nCE* bzw. *nWR* die vom Mikrocontrollerkern auf den Datenbus gelegten Daten bereits gültig gewesen sein müssen. Der vom RAM geforderte Mindestwert muß eingehalten werden.

t_{DHW} (Data Hold): Zeit, die angibt, wie lange die Daten auf dem Datenbus nach der steigenden Flanke von *nCE* bzw. *nWR* noch gültig bleiben müssen. Der Mindestwert muß vom Mikrocontrollerkern eingehalten werden.

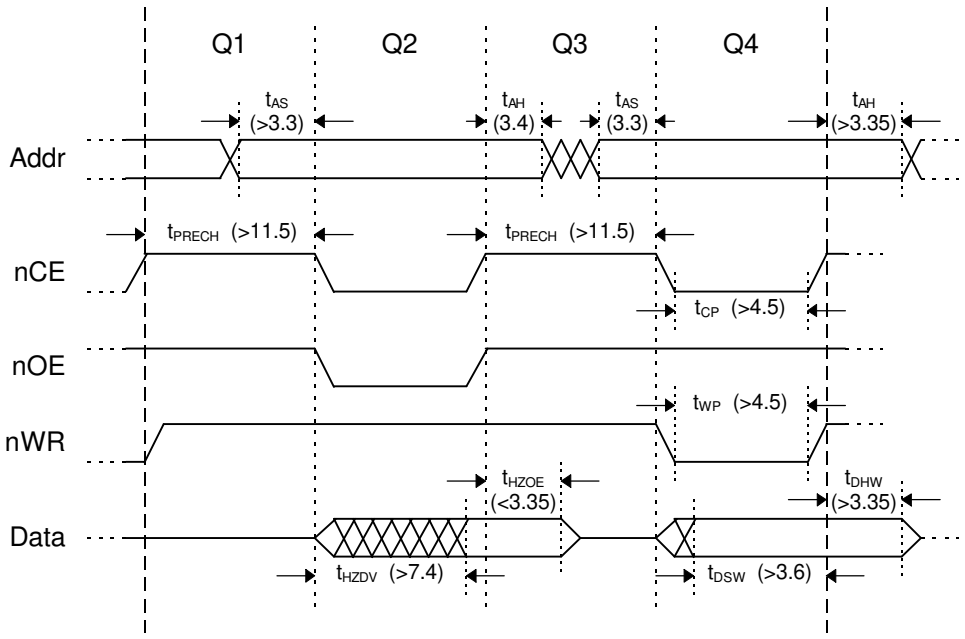


Abbildung 2.10: Timingkriterien des IMS-RAM für einen Read-Modify-Write Zyklus mit Beispielzeiten in ns für ein 256 x 8 Bit RAM.

Für das vom Mikrocontrollerkern erzeugte Timing an der Schnittstelle (siehe Abbildung 2.9) ergeben sich daraus eine Reihe von zusätzlich zu beachtenden Kriterien. Im einzelnen ergibt sich für die Mindestdauer der einzelnen Taktphasen:

Phase Q1: Mindestdauer = t_{PRECH}. Zu beachten ist, daß die Adressen rechtzeitig vor Ende der Phase gültig sind, damit sich keine Verletzung der Adressen-Setup-Zeit bei Beginn von Q2 und damit bei der fallenden Flanke von *nCE* und *nOE* ergibt.

Phase Q2: Mindestdauer = max (t_{CP}, t_{HZDV}).

Phase Q3: Mindestdauer = t_{PRECH}. Bei einem bidirektionalen Datenbus muß in der Mindestdauer auch noch t_{HZOE} berücksichtigt werden, um sicherzustellen, daß die Datenbustreiber des RAM sicher abgeschaltet sind, wenn die Datenbustreiber des Kerns bei Beginn von Q4 aktiviert werden. Wird zusätzlich während des Precharge in dieser Phase eine Umschaltung von Quell- auf Zieladresse gemacht, so müssen ggf. in der Mindestdauer auch die Setup- und Hold-Zeiten des Adreßbusses in ihrer Summe berücksichtigt werden. Unter Umständen muß auch die Umschaltung der Adressen so verzögert werden, daß die Adressen noch lange genug gültig bleiben und die Adreß-Hold-Zeit eingehalten wird, gleichzeitig aber die Adreß-Setup-Zeit dadurch nicht unterschritten wird.

Phase Q4: Mindestdauer = max (t_{CP}, t_{WP}). Zusätzlich muß in dieser Phase auch die maximale Signallaufzeit der Daten auf dem kritischen Pfad im Mikrocontrollerkern in die Mindestdauer einbezogen werden, da sich die Daten entsprechend verzögern und damit ggf. erst später gültig werden, auch wenn die Datenbustreiber auf Seite des Kerns schon lange vollständig aktiviert sind. Das bedeutet, daß die Mindestdauer alternativ auch durch den Zeitpunkt des Eintreffens der Daten unter Berücksichtigung der geforderten Daten-Setup-Zeit t_{DSW} bestimmt werden kann.

Schließlich muß bei der Ansteuerung des RAMs noch beachtet werden, daß nach Ende der Phase Q4 die Daten und Adressen auf den Bussen noch lange genug für das RAM gültig bleiben, so daß sich keine Verletzung der Daten- und Adreß-Hold-Zeiten ergibt. Für die

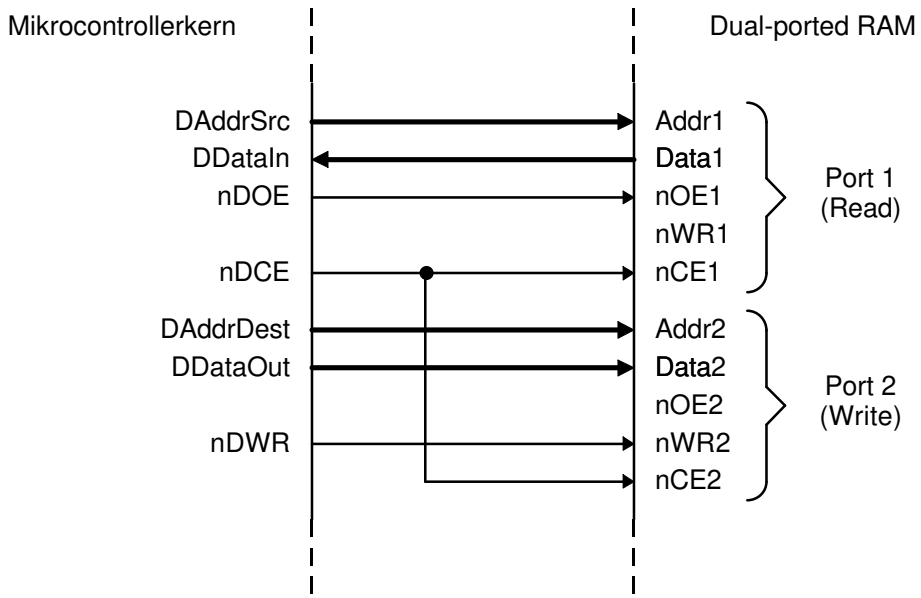


Abbildung 2.11: Kopplung eines dual-ported RAM an die Datenspeicherschnittstelle des Kerns.

Verzögerung der Adreßumschaltung gilt dabei analog zur Umschaltung in Phase Q3, daß sich dadurch keine Verletzung der Adreß-Setup-Zeit in Phase Q1 des nächsten Instruktionszyklus ergeben darf, wobei die Kriterien in beiden Fällen die gleichen sind.

Da beim Mikrocontrollerkern alle 4 Taktphasen gleich lang sind, bedeutet diese Betrachtung der Mindestdauern der Phasen, daß sich die minimale Dauer aller Taktphasen und damit die maximal mögliche Frequenz des Systemtakts direkt aus der Taktphase mit der größten Mindestdauer ableitet, d.h. die maximal mögliche Taktfrequenz hängt einerseits von den einzelnen Anforderungen des RAM und andererseits von der maximalen Signallaufzeit auf dem kritischen Pfad des Mikrocontrollerkerns ab. Wie man aber an den angegebenen Beispielwerten für ein 256 x 16 Bit RAM erkennen kann, sind die Mindestdauern der einzelnen Taktphasen annähernd gleich, wobei in der vierten Taktphase noch die Signallaufzeit im Kern mit berücksichtigt ist. Da aber die Taktfrequenz maximiert wird, wenn die Mindestdauern der Taktphasen gleich verteilt und damit minimal sind, folgt daraus, daß sich die gewählte Unterteilung in vier Phasen mit dem entsprechenden Timing als optimal erweist.

Eine deutliche Entspannung der Anforderungen an einzelne Timingkriterien tritt bei Einsatz eines dual-ported RAM ein. In diesem Fall kann nämlich, wie in Abbildung 2.11 dargestellt, das RAM so an den Prozessormakro angeschaltet werden, daß jeweils ein Port des RAMs als reiner Schreib- bzw. Leseport fungiert. Dadurch entfallen einerseits die Umschaltung zwischen Quell- und Zieladresse auf dem Adreßbus und andererseits die zusätzliche externe Zwischenschaltung von Tristate-Treibern auf Prozessorseite zur Realisierung des sonst nötigen bidirektionalen Datenbusses zwischen RAM und Prozessormakro. Die ansonsten für die Umschaltungen nötigen Lauf- und Zugriffszeiten fallen somit weg. Es ergibt sich daraus also sowohl eine deutlich einfachere Beschaltung der Datenspeicherschnittstelle als auch ein größerer Spielraum im Zeitverhalten beim Zugriff auf den Datenspeicher.

2.6 Spezialregister

Der Mikrocontrollerkern enthält eine Reihe von Spezialregistern, die bereits direkt im Kern implementiert sind, da sie wichtige Aufgaben erfüllen und für die Funktion des Prozessormakros der unmittelbare Zugriff auf die diversen Register an den verschiedenen Stellen notwendig ist. Diese Register enthalten sowohl Informationen über den Status des

Prozessorkerns als auch Daten zur Steuerung und Konfiguration der Abläufe im Mikrocontroller. Um den auf dem Mikrocontrollerkern ablaufenden Programmen einen einfachen Zugriff auf die Spezialregister zu ermöglichen, sind diejenigen Register, auf die der Programmierer zugreifen kann bzw. muß, im Datenspeicherbereich ab Adresse 00h eingeblendet, so daß sie, analog zu den sonstigen normalen Registern bzw. Zellen im Datenspeicherbereich, zusammen mit jedem beliebigen Befehl der ein Registerargument erwartet angegeben werden können.

Adr	Name	Bedeutung (Bit 7 | ... | Bit 0)	Reset
00	INDF0	Indirekte Adressierung mit Zeiger 0	00000000
01	FSR0	Zeiger 0 für indirekte Adressierung (File Select Register 0)	xxxxxxxx
02	PCL	Low-Byte des Programmzählers (Program Counter)	00000000
03	PCLATH	Zwischenspeicherlatch für High-Byte des PC (PCH Latch)	xxxxxxxx
04	ALUSTA	ALU Statusregister: File-Select Modus, Overflow, Zero, Digit C, Carry FS3 | FS2 | FS1 | FS0 | OV | Z | DC | C	1111xxxx
05	RTCSTA	Ereigniszeitzähler (RTCC) Statusregister: Edges, Clock, Prescaler INTEDG | RTEDG | T/C | RTPS3 | RTSP2 | RTPS1 | RTPS0 | -	00000000
06	CPUSTA	CPU Statusregister: Stack Available, Global Interrupt Disable ´0´ | ´0´ | STAVL | GLINTD | ´1´ | ´1´ | ´0´ | ´0´	00111100
07	INTSTA	Interruptstatusregister: Interrupt Request, Interrupt Enable PEIR | INT1IR | RTCIR | INT0IR | PEIE | INT1IE | RTCIE | INT0IE	00000000
08	INDF1	Indirekte Adressierung mit Zeiger 1	00000000
09	FSR1	Zeiger 1 für indirekte Adressierung (File Select Register 1)	xxxxxxxx
0A	W	W-Register, Arbeitsregister (Working Register)	xxxxxxxx
0B	RTCCL	Low-Byte des Ereigniszählers (Real Time Clock Counter)	xxxxxxxx
0C	RTCCH	High-Byte des Ereigniszählers (Real Time Clock Counter)	xxxxxxxx
0D	TBLPTRL	Low-Byte des Programmspeicherzeigers (Table-Pointer)	xxxxxxxx
0E	TBLPTRH	High-Byte des Programmspeicherzeigers (Table-Pointer)	xxxxxxxx
0F	BSR	Bank Select Register	00000000

x: nicht definiert

Tabelle 2.4: Spezialregister.

Tabelle 2.4 gibt einen Überblick über alle sechzehn eingeblendeten Spezialregister im Datenspeicherbereich von Adresse 00h bis 0Fh. In der letzten Spalte der Tabelle sind die Werte angegeben, die die einzelnen Register bei einem Reset einnehmen. Zusätzlich zu diesen Registern enthält der Mikrocontrollerkern noch eine Reihe weiterer interner Register, die aber nicht in den Datenspeicherbereich eingeblendet sind und auf die deshalb auch nicht beliebig zugegriffen werden kann. Auf die zusätzlichen Register wird aber in einigen oder allen Fällen implizit bei der Ausführung einzelner Befehle zugegriffen, und die Register beeinflussen gegebenenfalls die Befehlsausführung, so daß diese Register hier kurz beschrieben werden sollen, um das Verständnis der verschiedenen Befehle und der Programmierung zu erleichtern:

- **Register IR (engl. *Instruction Register*)**
  Das Instruktionsregister ist 16 Bit breit und enthält den binär kodierten Befehl (Opcode) der momentan ausgeführt wird. Im allgemeinen wird in jedem Instruktionszyklus ein neuer 16 Bit breiter Befehl aus dem Programmspeicher in das Register IR geladen und dann im nächsten Zyklus dekodiert und ausgeführt. Der

ebenfalls 16 Bit breite Programmzähler enthält dabei die Adresse des zu lesenden Befehls im Programmspeicher (siehe auch Abschnitt 2.8).

- **Register TBLAT (engl. *Table Latch*)**
  Das Table-Latch kommt bei den Instruktionen `TABLRD` und `TABLWT`, mit denen Daten zwischen Programm- und Datenspeicher ausgetauscht werden, zum Einsatz. Es enthält den Wert, der aus dem Programmspeicher gelesen bzw. in den Programmspeicher geschrieben wird. Entsprechen der Programmspeicherdatenbreite ist das Register 16 Bit breit, aufgeteilt in eine untere Hälfte, TBLATL, und eine obere Hälfte TBLATH, jeweils mit 8 Bit. Bei jedem Lese- oder Schreibzugriff auf den Programmspeicher mit den Befehlen `TABLRD` bzw. `TABLWT` wird also ein komplettes 16 Bit vom Programmspeicher Wort in das Register gelesen bzw. vom Register in den Programmspeicher geschrieben. Die beiden Hälften des Registers sind nicht in den Datenspeicherbereich eingeblendet und können deshalb auch nicht mit beliebigen Befehlen geschrieben oder gelesen werden. Anwendungsprogramme können aber mit den Instruktionen `TLRD` und `TLWT` gezielt Daten zwischen einem beliebigen Register des Datenspeicherbereichs und den beiden Table-Latch Hälften austauschen.

Im folgenden werden die einzelnen eingeblendeten Spezialregister des Datenspeicherbereichs 00h bis 0Fh näher beschrieben. Soweit nicht anders angegeben, können alle Register normal mit den verschiedenen zur Verfügung stehenden Befehlen sowohl geschrieben als auch gelesen werden.

2.6.1 Register INDF0 (Adresse 00h)

Das Register INDF0 bildet mit dem Register FSR0 zusammen das erste Paar Register zur indirekten Adressierung. Es ist nicht physikalisch vorhanden und rein logisch implementiert. Zugriffe auf das Register führen auf einen Zugriff auf das Register, dessen Adresse dem momentanen Inhalt von FSR0 entspricht, d.h. durch Verwendung dieses Registers in einem Befehl wird eine indirekte Adressierung über den Zeiger FSR0 realisiert (siehe dazu auch Abschnitt 2.7.1 über indirekte Adressierung).

Verweist eine indirekte Adressierung über INDF0 oder INDF1 selbst wieder auf INDF0, d.h. enthält FSR0 bzw. FSR1 den Wert null und es wird dann auf INDF0 bzw. INDF1 zugegriffen, so wird bei einem entsprechenden Lesezugriff der Wert 00h gelesen und ein entsprechender Schreibzugriff wird nicht ausgeführt.

2.6.2 Register FSR0 (Adresse 01h)

Wie im vorangegangenen Abschnitt bereits beschrieben, ist das Register FSR0 (engl. *File Select Register 0*) der Zeiger in den Datenspeicherbereich für die indirekte Adressierung über INDF0, d.h. es enthält die Datenspeicheradresse für den indirekten Zugriff. Entsprechend der Adreßbusbreite des Datenspeichers ist das Register 8 Bit breit und es läßt sich somit der gesamte Datenspeicherbereich mit einer Größe von 256 Registern, inklusive der Spezialregister, per indirekter Adressierung ansprechen.

Der Mikrocontrollerkern läßt sich so konfigurieren, daß nach einem indirekten Zugriff auf eine Datenspeicherzelle über FSR0 und INDF0 der Zeiger FSR0 automatisch inkrementiert oder dekrementiert wird, was den fortlaufenden Zugriff auf einen zusammenhängenden Bereich im Datenspeicher, z.B. für Blockverschiebungen, erheblich vereinfachen kann. Die Einstellung des File-Select-Modus für FSR0 erfolgt dazu über die Bits FS0 (Bit 4) und FS1 (Bit 5) des Registers ALUSTA an Adresse 04h (siehe Tabelle 2.5 und Abschnitt 2.6.5). Zu beachten ist, daß das Register *nach* dem Zugriff verändert wird, d.h. bei der Ausführung eines Befehls mit indirekter Adressierung werden erst alle Zugriffe (lesen und/oder schreiben) mit dem momentanen Inhalt von FSR0 durchgeführt und dann, am Ende der Befehlsausführung, gegebenenfalls FSR0 noch um eins erhöht oder erniedrigt.

ALUSTA FS1 (Bit 5)	ALUSTA FS0 (Bit 4)	FSR0 Modus
´0´	´0´	Auto-Dekrement
´0´	´1´	Auto-Inkrement
´1´	´0´	Keine Änderung
´1´	´1´	Keine Änderung

Tabelle 2.5: Konfiguration des Modus für FSR0.

2.6.3 Register PCL (Adresse 02h)

Das Register PCL (engl. *Program Counter Low*) ist die untere Hälfte des 16 Bit breiten Programmzählers des Mikrocontrollerkerns und kann wie jedes andere Register auch normal gelesen und geschrieben werden. Durch Lesen des Registers ist es Anwendungsprogrammen damit auf der einen Seite möglich, den momentanen Inhalt des Programmzählers auszulesen und zu sichern. Auf der anderen Seite kann durch einen Schreibzugriff auf PCL der Programmzähler für die Durchführung eines Sprunges an eine vorher berechnete Adresse mit einem Wert geladen werden. Im Gegensatz zu den direkten Sprungbefehlen, wie z.B. `GOTO`, bei denen die Sprungadresse fest im Programm kodiert ist, lassen sich so Sprünge an zur Laufzeit variable Adressen verwirklichen.

Da mit einem Befehl jeweils nur ein 8 Bit Wert geschrieben oder gelesen werden kann, der Programmzähler aber entsprechend der Programmspeicheradreßbusbreite 16 Bit breit ist, und da aus Gründen der Datenkonsistenz stets nur der komplette Inhalt des Programmzählers in einem Befehl gelesen oder geschrieben werden darf, existiert das Register PCLATH an Adresse 03h (siehe nächsten Abschnitt) als Zwischenspeicher für die obere Hälfte des Programmzählers bei Zugriffen auf das Register PCL und damit auf den Programmzähler selbst. Die oberen 8 Bit des Programmzählers sind nicht direkt ansprechbar, da das Register PCH nicht in den Datenspeicher eingeblendet ist, d.h. ein Zugriff auf die obere Hälfte des Programmzählers muß über das Register PCLATH und Zugriff auf PCL erfolgen. Bei Zugriffen auf das Register PCL werden folgende Operationen durchgeführt:

- **PCL Lesen (PCL ist Quelloperand der Instruktion)**
  PCL → Operand: Die untere Hälfte des Programmzählers wird auf den Datenbus gelegt und als Operand für den Befehl benutzt.
  PCH → PCLATH: Die obere Hälfte des Programmzählers wird in das Register PCLATH gesichert.
- **PCL Schreiben (PCL ist Ziel des Ergebnisses der Instruktion)**
  Ergebnis → PCL: Das Ergebnis der Befehlsausführung wird in die untere Hälfte des Programmzählers geschrieben.
  PCLATH → PCH: Die obere Hälfte des Programmzählers wird mit dem Inhalt des Registers PCLATH geladen.
- **PCL Lesen, Bearbeiten, Schreiben (PCL ist Quelle und Ziel der Instruktion)**
  PCL → Operand: Die untere Hälfte des Programmzählers wird auf den Datenbus gelegt und als Operand für den Befehl benutzt.
  Ergebnis → PCL: Das Ergebnis der Befehlsausführung wird in die untere Hälfte des Programmzählers geschrieben.

PCLATH → PCH: Die obere Hälfte des Programmzählers wird mit dem Inhalt des Registers PCLATH geladen.

Um einen Sprung an eine berechnete, 16 Bit breite Adresse durchzuführen, muß also die obere Hälfte der Adresse in das Register PCLATH geladen und dann PCL mit der unteren Hälfte der Zieladresse beschrieben werden. Alternativ ist es z.B. auch möglich, mit Hilfe einer Additionsinstruktion einen Wert zu PCL hinzuzuaddieren und so einen Sprung mit einem bestimmten Offset von der momentanen Position aus durchzuführen, wobei beachtet werden muß, daß vorher PCLATH mit einem passenden Wert geladen wird.

Beim Auslesen des aktuellen Programmzählerinhalts über Lesezugriffe auf das Register PCL ist zu berücksichtigen, daß aufgrund des Pipelinings (Abschnitt 2.8) nicht die Adresse der gerade ausgeführten Instruktion, die den Lesezugriff selbst beinhaltet, als Operand (und ggf. in das Register PCLATH) gelesen wird, sondern daß der Programmzähler beim Auslesen desselben bereits die Adresse der nachfolgenden Instruktion beinhaltet, die schon im voraus, parallel zur Ausführung der aktuellen Instruktion, geladen wird.

2.6.4 Register PCLATH (Adresse 03h)

Im obigen Abschnitt wurde die Funktion des Registers PCLATH (engl. *Program Counter Latch High*) als Zwischenspeicher für die oberen 8 Bit des Programmzählers bei Zugriffen auf das Register PCL schon ausführlich beschrieben. Wie bereits erwähnt, sind die oberen 8 Bit des PC (Register PCH) nicht in den Speicherbereich eingeblendet und somit auch nicht direkt ansprechbar. Das Register PCLATH dient deshalb dazu eine Kopie der oberen Hälfte des Programmzählers zu halten.

2.6.5 Register ALUSTA (Adresse 04h)

Tabelle 2.6 zeigt eine Aufschlüsselung der Belegung der einzelnen Bits des Registers ALUSTA (engl. *Arithmetic Logical Unit Status*). Zum einen enthält das Register die vier Bits FS0 bis FS3 zur Konfiguration der beiden File-Select Modi, d.h. zur Festlegung, ob die Register FSR0 bzw. FSR1 nach einem Zugriff mit indirekter Adressierung über INDF0 bzw. INDF1 inkrementiert oder dekrementiert werden sollen (siehe Abschnitt 2.6.2 und Abschnitt 2.6.10). Die Einstellung des Modus für die indirekte Adressierung über das Registerpaar INDF0 und FSR0 erfolgt über die Bits FS0 und FS1 (Bits 4 und 5 des Registers), für das Paar INDF1 und FSR1 über die Bits FS2 und FS3 (Bits 6 und 7).

Auf der anderen Seite beinhaltet das ALU Statusregister die Flags OV (engl. *Overflow*), Z (engl. *Zero*), DC (engl. *Digit Carry*) und C (engl. *Carry*), die den eigentlichen Status der ALU wiedergeben. Hauptsächlich bei den verschiedenen arithmetischen und logischen Instruktionen werden einige oder alle dieser Statusflags, je nach Art der Instruktion, entsprechend den bei Ausführung der Operation auftretenden Bedingungen gesetzt. Zusätzlich beeinflußt der Inhalt der einzelnen Flags die Ausführung verschiedener Instruktionen, z.B. zur Steuerung des Kontrollflusses, oder geht als Wert in die Berechnungsoperationen ein. Für eine Beschreibung derjenigen Flags, die bei den einzelnen Instruktionen tatsächlich gesetzt bzw. verwendet werden gilt der Verweis auf Abschnitt 2.7.2.

Die vier Statusflags können auch aus den Anwendungsprogrammen heraus durch Schreibzugriff auf das Register ALUSTA gesetzt werden, z.B. um einzelne Flags zurückzusetzen. Zu beachten ist, daß es dabei eventuell zu Konflikten zwischen implizitem Setzen der Flags durch den Befehl und explizitem Schreiben des Ergebnisses in das Register ALUSTA kommen kann, z.B. wenn das Ergebnis einer Addition nach ALUSTA geschrieben wird. In diesem Fall hat das implizite Setzen der Statusflags entsprechend des Operationsstatus Priorität, d.h. im Prinzip verhält sich der Mikrocontrollerkern so, als ob zuerst das Ergebnis vom Datenbus in das Register geschrieben wird und dann die einzelnen Statusflags je nach Befehl und Bedingungen gesetzt werden, wobei ggf. die vorher geschriebenen originalen Bits des Ergebnisses überschrieben werden.

Bit	Wert	Bedeutung	Reset
7	FS3	FSR1 Modus nach ind. Zugriff über INDF1: (´FS3 FS2´) ´0 0´ Automatisch Dekrementieren. ´0 1´ Automatisch Inkrementieren. ´1 0´ Keine Änderung. ´1 1´ Keine Änderung.	´1´
6	FS2		´1´
5	FS1	FSR0 Modus nach ind. Zugriff über INDF0: (´FS1 FS0´) ´0 0´ Automatisch Dekrementieren. ´0 1´ Automatisch Inkrementieren. ´1 0´ Keine Änderung. ´1 1´ Keine Änderung.	´1´
4	FS0		´1´
3	OV	Overflow Flag: Wird nach arith. Operationen auf ´1´ gesetzt, wenn ein Überlauf (vom Betrag in das Vorzeichenbit) aufgetreten ist, sonst auf ´0´ (kein Überlauf).	x
2	Z	Zero Flag: Wird nach arith. und logischen Operationen auf ´1´ gesetzt, wenn das Ergebnis Null war, sonst auf ´0´ (Ergebnis ungleich Null).	x
1	DC	Digit Carry Flag: Wird bei arith. Operationen auf den Wert des Übertrags von Bit 3 nach Bit 4 (von unterer nach oberer Byte-Hälfte) gesetzt.	x
0	C	Carry Flag: Wird bei arith. und Schiebeoperationen auf den Wert des Übertrags von Bit 7 zum nächsten Byte bzw. des herausgeschobenen Bits gesetzt.	x

x: nicht definiert

Tabelle 2.6: Belegung des Registers ALUSTA.

2.6.6 Register RTCSTA (Adresse 05h)

Das Register RTCSTA (engl. *Real Time Clock Counter Status*, *RTCC Status*) dient im wesentlichen zur Konfiguration des RTCC-Moduls (siehe Abschnitt 2.11). Eine Übersicht über die einzelnen Bits des Register ist in Tabelle 2.7 dargestellt. Bit 0 des Registers ist nicht mit einer speziellen Funktion belegt. Es ist aber trotzdem im Mikrocontrollerkern implementiert, kann also beliebig vom Anwendungsprogramm gesetzt und gelesen, d.h. allgemein verwendet werden.

Die Konfigurationsbits des Registers dienen erstens zur Festlegung des Faktors des Vorteilers (engl. *Prescaler*) im RTCC-Modul. Darüber kann in gewissen Stufen eingestellt werden, nach wieviel eintretenden Ereignissen jeweils der Ereigniszähler (implementiert durch die Register RTCCL und RTCCH, siehe Abschnitt 2.6.12) inkrementiert werden soll. Zweitens kann über ein Flag des Registers festgelegt werden, ob Ereignisse durch Flanken am externen RTCC-Takteingang (Pin RTCC) oder durch den internen Systemtakt, der ja einem durch vier geteilten externen Systemtakt am Pin CLK entspricht, ausgelöst werden sollen. Schließlich kann mit den Bits INTEDG (engl. *Interrupt Edge*) und RTEDG (engl. *RTCC Edge*) selektiert werden, ob externe Interrupts und externe RTCC-Ereignisse (falls eingestellt ist, daß RTCC-Ereignisse extern und nicht intern ausgelöst werden, s.o.) durch fallende oder steigende Flanken an den entsprechenden Pins ausgelöst werden sollen. Bit 6 legt dabei die Flankentriggerung sowohl für Interrupt 1 als auch für externe RTCC-Ereignisse, d.h. für die Pins INT1 und RTCC fest. Beim Mikrocontrollerkern existieren im Gegensatz zum originalen PIC17 zwei getrennte Pins für den zweiten Interrupt und für den RTCC-Eingang an der externen Schnittstelle. Beim PIC17 gibt es nur einen Pin, der als RTCC-Eingang und / oder als sog. RT-Interrupteingang verwendet werden kann, weshalb dort auch nur ein Bit zur Konfiguration der Flanke am entsprechenden Pin vorgesehen ist. Aus Kompatibilitätsgründen wurde dies hier beim Mikrocontrollerkern beibehalten und kein zusätzliches Konfigurationsbit eingeführt, obwohl aus Gründen der zusätzlichen Flexibilität ein weiterer externer Pin zu Trennung von RTCC-Eingang und Interrupteingang hinzugefügt wurde (siehe auch Abschnitt 2.10 zum Thema Interrupts).

Bit	Wert	Bedeutung	Reset
7	INTEDG	Flankenselektion für Interrupt 0: ´0´ Interrupt ausgelöst bei fallender Flanke. (Pin INT0) ´1´ Interrupt ausgelöst bei steigender Flanke.	´0´
6	RTEDG	Flanke für Interrupt 1 / RTCC: ´0´ Interrupt / RTCC-Takt bei fallender Flanke. (Pin INT1 / Pin RTCC) ´1´ Interrupt / RTCC-Takt bei steigender Flanke.	´0´
5	T/$\overline{C}$	RTCC Taktmodus: ´0´ Zähler Modus. Externer Takt (Pin RTCC). (Timer / Counter) ´1´ Timer Modus. Interner Takt (externer Systemtakt / 4).	´0´
4	RTPS3	RTCC-Prescaler: ´0000´ Vorteiler 1:1 ´1000´ Vorteiler 1:256 (´RTPS3...RTPS0´) ´0001´ Vorteiler 1:2 : ´0010´ Vorteiler 1:4 : ´0011´ Vorteiler 1:8 : ´0100´ Vorteiler 1:16 : ´0101´ Vorteiler 1:32 : ´0110´ Vorteiler 1:64 : ´0111´ Vorteiler 1:128 ´1111´ Vorteiler 1:256	´0´
3	RTPS2		´0´
2	RTPS1		´0´
1	RTPS0		´0´
0	-	Unbenutzt.	´0´

Tabelle 2.7: Belegung des Registers RTCSTA.

2.6.7 Register CPUSTA (Adresse 06h)

Im Register CPUSTA (engl. *Central Processing Unit Status*) sind nur zwei der acht möglichen Bits tatsächlich belegt: STKAVL (Bit 5) und GLINTD (Bit 4), siehe Tabelle 2.8. Die restlichen Bits sind nicht physikalisch im Mikrocontrollerkern vorhanden und können deshalb auch keine Werte speichern. Daraus folgt, daß beim Lesen des Registers CPUSTA in diesen Bits stets der selbe feste Wert gelesen wird und bei Schreibzugriffen auf CPUSTA werden die entsprechenden Bits ignoriert. Aus Kompatibilitätsgründen zum originalen PIC17 haben die nicht implementierten Bits größtenteils den festen Wert ´0´, mit Ausnahme der Bits 3 und 4. Beim PIC17 enthalten diese Bits die Flags für den Power-Down Status und den Timeout Status im Zusammenhang mit verschiedenen Reset-Modi und dem Watch-Dog Timer. Da beim Mikrocontrollerkern beide Funktionalitäten nicht integriert sind, entfallen diese beiden Statusflags und die beiden Bits haben den festen Wert ´1´, der dem Status eines PIC17 nach normalem Power-On Reset entspricht.

Mit dem Flag GLINTD (engl. *Global Interrupt Disable*) des Registers CPUSTA können Anwendungsprogramme global festlegen, ob Interrupts prinzipiell erlaubt sind oder nicht. Einzelne Interrupts können zusätzlich noch über entsprechende Flags im Register INTSTA (Abschnitt 2.6.8) einzeln maskiert werden. Weiterhin wird das Flag GLINTD vom Mikrocontrollerkern automatisch auf ´1´ gesetzt, wenn ein Interrupt ausgelöst und in die Interruptroutine verzweigt wird. Dadurch werden standardmäßig verschachtelte Interrupts vermieden, können aber vom Programmierer explizit durch Rücksetzen von GLINTD in der Interruptroutine wieder erlaubt werden. Über die Instruktion `RETFIE` kann schließlich implizit beim Rücksprung aus der Interruptroutine GLINTD auf jeden Fall wieder zurückgesetzt werden. Eine genauere Beschreibung der Interruptlogik findet sich in Abschnitt 2.10.

Das Flag STKAVL (engl. *Stack Available*) im Register CPUSTA gibt den Status des Stacks wieder. Es wird ausschließlich vom Mikrocontroller selbst intern verwaltet und kann vom Anwendungsprogramm nicht gesetzt, sondern nur ausgelesen werden, um den Stackstatus zu erfragen. Das Flag ist im normalen Betrieb ´0´ und wird vom Prozessor dann auf ´1´ gesetzt, wenn der Stack voll ist. Läuft der Stack über, bleibt das Flag ständig gesetzt und kann erst wieder durch einen Reset zurückgesetzt werden. Abschnitt 2.9 enthält eine detaillierte Beschreibung des Stacks und des STKAVL Flags.

Bit	Wert	Modus	Bedeutung	Reset
7	´0´	RO	Nicht implementiert, immer ´0´.	´0´
6	´0´	RO	Nicht implementiert, immer ´0´.	´0´
5	STKAVL	RO	Stack Available Flag: ´0´ Stack normal. ´1´ Stack voll oder war / ist übergelaufen.	´1´
4	GLINTD	R/W	Global Interrupt Disable Flag: ´0´ Interrupts prinzipiell erlaubt. ´1´ Alle Interrupts gesperrt.	´1´
3	´1´	RO	Nicht implementiert, immer ´1´.	´1´
2	´1´	RO	Nicht implementiert, immer ´1´.	´1´
1	´0´	RO	Nicht implementiert, immer ´0´.	´0´
0	´0´	RO	Nicht implementiert, immer ´0´.	´0´

RO: Read-Only. Kann vom Programm nur gelesen, nicht aber gesetzt werden (Setzen wird ignoriert).
R/W: Read / Write. Kann vom Programm gelesen und gesetzt werden.

Tabelle 2.8: Belegung des Registers CPUSTA.

2.6.8 Register INTSTA (Adresse 07h)

Das Register INTSTA (engl. *Interrupt Status*) setzt sich aus zwei Teilen zusammen: auf der einen Seite Informationen über den Status des Interruptmoduls und auf der anderen Seite Konfigurationsflags zur Maskierung der vier möglichen Interruptquellen im Mikrocontrollerkern (Tabelle 2.9).

Die untere Hälfte des Registers enthält die vier Bits INT0IE, INT1IE, RTCIE und PEIE (engl. *INT0*, *INT1*, *RTCC* und *Peripheral Interrupt Enable*) zur Maskierung der vier Interruptquellen, d.h. über diese Bits kann gesteuert werden, ob bei Anliegen einer Anforderung für einen entsprechenden Interrupt dieser tatsächlich auch ausgelöst werden soll. Unabhängig davon kann über das Bit GLINTD im Register CPUSTA (siehe Abschnitt 2.6.7) global festgelegt werden, ob überhaupt Interrupts ausgelöst werden können oder ob alle Interrupts gesperrt sind. Die obere Hälfte des Registers INTSTA umfaßt den eigentlichen Status der anliegenden Interruptanforderungen. Die Bits INT0IR, INT1IR und RTCIR (engl. *INT0*, *INT1* und *RTCC Interrupt Request*) werden unabhängig vom Zustand der Maskierungsbits oder des Flags GLINTD vom Mikrocontroller auf ´1´ gesetzt, wenn eine fallende oder steigende Flanke (Konfigurierbar über die Bits INTEDG und RTEDG im Register RTCSTA, Abschnitt 2.6.6) an den Pins INT0 oder INT1 erkannt wurde bzw. wenn ein Überlauf des Ereigniszählers (gebildet durch die Register RTCCL und RTCCH, Abschnitt 2.6.12; siehe auch Abschnitt 2.11) aufgetreten ist. Eine Ausnahme bildet das Bit PEIR (engl. *Peripheral Interrupt Request*), das nicht vom Mikrocontroller selbst beeinflußt wird, sondern nur den abgetasteten Wert am gleichnamigen externen Pin widerspiegelt. Ist eines oder sind mehrere der Anforderungsbits gesetzt und sind die entsprechenden Interrupts weder global noch einzeln gesperrt, so wird vom Mikrocontrollerkern unter Berücksichtigung der Prioritäten ein Interrupt ausgelöst und zur entsprechenden Routine verzweigt. Das Anforderungsbit INT0IR, INT1IR oder RTCIR wird dabei jeweils beim Sprung in die entsprechende Interruptroutine automatisch vom Prozessor wieder zurück auf ´0´ gesetzt, um die Anforderung zu löschen und so zu verhindern, daß der Interrupt erneut ausgelöst wird. In den Interruptroutinen muß das Anforderungsbit also nicht explizit rückgesetzt werden, um anzuzeigen daß der Interrupt abgearbeitet wurde. Bei einem Peripherieinterrupt hingegen muß die Interruptroutine dafür sorgen, daß das Bit PEIR, d.h. der Wert am Eingang PEIR, zurückgesetzt wird. Ist ein Anforderungsbit gesetzt, der entsprechende Interrupt aber maskiert,

sind global alle Interrupts gesperrt oder wird ein Interrupt mit höherer Priorität zuerst abgearbeitet (was standardmäßig global alle Interrupts über GLINTD sperrt), so bleibt die Anforderung solange gespeichert, bis nach Aufhebung der Sperre bzw. der Maskierung der Interrupt dann schließlich ausgelöst werden kann.

Das Anwendungsprogramm hat vollen Zugriff auf das gesamte Register und kann sowohl die Maskierung der einzelnen Interruptquellen festlegen als auch eigenständig Interruptanforderungen setzen oder rücksetzen und so z.B. einen Interrupt per Software auslösen oder eine gespeicherte Anforderung löschen.

Im Abschnitt 2.10 dieses Kapitels ist die gesamte Interruptlogik und damit auch die Funktion des Register INTSTA nochmals ausführlicher beschrieben.

Bit	Wert	Bedeutung	Reset
7	PEIR	Peripheral Interrupt Request: ´1´, wenn eine Anforderung für einen Peripherieinterrupt vorliegt. Entspricht Wert am Eingangspin PEIR.	´0´
6	INT1IR	INT1 Interrupt Request: ´1´, wenn eine Anforderung für einen Interrupt 1 vorliegt. Wird gesetzt durch Flanke am Pin INT1. Wird automatisch zurückgesetzt.	´0´
5	RTCIR	RTCC Interrupt Request: ´1´, wenn eine Anforderung für einen RTCC-Interrupt vorliegt. Wird gesetzt durch Überlauf des Ereigniszählers. Automatisch zurückgesetzt.	´0´
4	INT0IR	INT0 Interrupt Request: ´1´, wenn eine Anforderung für einen Interrupt 0 vorliegt. Wird gesetzt durch Flanke am Pin INT0. Wird automatisch zurückgesetzt.	´0´
3	PEIE	Peripheral Interrupt Enable: ´0´ Peripherieinterrupts gesperrt. ´1´ Peripherieinterrupts möglich.	´0´
2	INT1IE	INT1 Interrupt Enable: ´0´ Interrupt 1 (Pin INT1) gesperrt. ´1´ Interrupt 1 (Pin INT1) möglich.	´0´
1	RTCIE	RTCC Interrupt Enable: ´0´ RTCC Interrupt gesperrt. ´1´ RTCC Interrupt möglich.	´0´
0	INT0IE	INT0 Interrupt Enable: ´0´ Interrupt 0 (Pin INT0) gesperrt. ´1´ Interrupt 0 (Pin INT0) möglich.	´0´

Tabelle 2.9: Belegung des Registers INTSTA.

2.6.9 Register INDF1 (Adresse 08h)

Analog zum Registerpaar INDF0 und FSR0 (siehe Abschnitt 2.6.1 und Abschnitt 2.6.2) bilden die Register INDF1 und FSR1 das zweite Paar zur indirekten Adressierung. INDF1 ist nicht physikalisch implementiert und Zugriffe auf das Register entsprechen einem Zugriff mit indirekter Adressierung über FSR1 als Zeiger in den Datenspeicher.

Zugriffe mit indirekter Adressierung, die selbst wieder auf INDF1 verweisen (FSR0 oder FSR1 enthält den Wert 08h und es wird ein Zugriff auf INDF0 bzw. INDF1 gemacht) resultieren gleichermaßen im Lesen des Wertes 00h oder im Abbruch des Schreibzugriffs.

2.6.10 Register FSR1 (Adresse 09h)

Wie bereits für das Register FSR0 (Abschnitt 2.6.2) zusammen mit INDF0 beschrieben und wie im obigen Abschnitt bereits erwähnt, enthält FSR1 (engl. *File Select Register 1*) die Adresse der Zielspeicherzelle eines indirekten Zugriffs über INDF0 und es gilt das dort gesagte entsprechend.

Die Konfiguration des Modus von FSR1, d.h. die Einstellung, ob nach einem indirekten Zugriff über INDF1 und FSR1 das Register FSR1 automatisch inkrementiert oder dekrementiert werden soll, erfolgt in diesem Fall über die Bits FS2 (Bit 6) und FS3 (Bit 7) des Registers ALUSTA an Adresse 04h (siehe Tabelle 2.10 und Abschnitt 2.6.5).

ALUSTA FS3 (Bit 7)	ALUSTA FS2 (Bit 6)	FSR1 Modus
´0´	´0´	Auto-Dekrement
´0´	´1´	Auto-Inkrement
´1´	´0´	Keine Änderung
´1´	´1´	Keine Änderung

Tabelle 2.10: Konfiguration des Modus für FSR1.

2.6.11 Register W (Adresse 0Ah)

Das zentrale Register und der Akkumulator des Mikrocontrollerkerns ist das W-Register (engl. *Working Register*). Bei den meisten Instruktionen wird der Inhalt des W-Registers implizit, d.h. ohne extra angegeben werden zu müssen, als einer der Operanden für den Befehl verwendet, und es kann in fast allen Fällen über einen zusätzlichen Parameter bei den Befehlen angegeben werden, ob das Ergebnis der Operation in das als zweiten Operanden angegebene allgemeine Register, in das W-Register oder in beide geschrieben werden soll (Abschnitt 2.7.2).

2.6.12 Register RTCCL (Adresse 0Bh) und RTCCH (Adresse 0Ch)

Die Register RTCCL und RTCCH (engl. *Real Time Clock Counter Low* und *Real Time Clock Counter High*) stellen zusammen den 16 Bit breiten Ereigniszähler des Mikrocontrollerkerns dar (siehe Abschnitt 2.11). Dabei enthält RTCCL die untere Hälfte des Zählers und RTCCH die obere Hälfte. Durch Schreiben der Register kann der Zähler mit beliebigen Werten initialisiert werden. Das RTCC-Modul läßt sich auf vielfältige Weise so konfigurieren, daß z.B. durch externe Ereignisse oder durch den internen Takt der Zähler zu bestimmten Zeitpunkten inkrementiert wird. Gleichzeitig kann ein Anwendungsprogramm durch Auslesen der beiden Register den momentanen Stand des Zählers abfragen. Weiterhin kann bei einem Überlauf des Zählers von FFFFh auf 0000h ein Interrupt ausgelöst werden.

Zu beachten ist bei Zugriffen auf die Zählerregister, daß sich der Zählerstand zwischen zwei Befehlen, die einerseits auf RTCCL und andererseits auf RTCCH zugreifen, ändern kann. Beim Schreiben eines der beiden Register wird automatisch vom Mikrocontrollerkern verhindert, daß das geschriebene Register im nächsten Instruktionszyklus inkrementiert werden kann. Durch Schreiben beider Register direkt hintereinander ist es deshalb möglich, den kompletten Zählers sicher und konsistent neu zu setzen.

2.6.13 Register TBLPTRL (Adresse 0Dh) und TBLPTRH (Adresse 0Eh)

Die beiden Register TBLPTRL und TBLPTRH (engl. *Table Pointer Low* und *Table Pointer High*) bilden zusammen einen 16 Bit breiten Zeiger in den Programmspeicherbereich, der zum Austausch von Daten zwischen Programm- und Datenspeicher mittels der Befehle `TABLRD` und `TABLWT` Verwendung findet. TBLPTRL enthält die unteren 8 Bit und TBLPTRH die oberen 8 Bit der, entsprechend der Programmspeicheradreßbusbreite, 16 Bit breiten Adresse für den Schreib- bzw. Lesezugriff auf ein Wort im Programmspeicher. Somit läßt sich mit dem aus beiden Registern zusammengesetzten Table Pointer der gesamte Programmspeicherbereich des Prozessors adressieren.

Zum Lesen oder Schreiben von Daten im Programmspeicher müssen also zuerst die beiden Register mit der oberen und unteren Hälfte der Programmspeicheradresse des zu lesenden bzw.

zu schreibenden Wortes geladen und dann die Befehle `TABLRD` bzw. `TABLWT` verwendet werden. Bei den beiden Instruktionen besteht zusätzlich noch die Möglichkeit durch einen Parameter anzugeben, daß der durch TBLPTRL und TBLPTRH gebildete 16 Bit breite Zeiger nach dem Programmspeicherzugriff automatisch inkrementiert wird.

2.6.14 Register BSR (Adresse 0Fh)

Das Register BSR (engl. *Bank Select Register*) hat keine interne Funktion im Mikrocontrollerkern und wird lediglich über eine Schnittstelle nach außen geführt, um dort über den Inhalt des Registers und entsprechende Dekodierung eine Auswahl verschiedener Bänke im Speicherbereich und somit eine Vergrößerung des adressierbaren Speichers zu ermöglichen (siehe dazu auch Abschnitt 2.4.2 und Abschnitt 2.5). Anwendungsprogramme können das Register normal lesen und schreiben und bei entsprechender externer Beschaltung darüber eine Auswahl der gerade aktiven Speicherbank machen.

Die Breite der Implementierung des Registers ist variabel und kann bei der Synthese des Mikrocontrollerkerns konfiguriert werden. Der Standardwert für die Breite des Register ist 4 Bit, analog zum originalen PIC17 Mikrocontroller. Beim Auslesen des Registers über eine Instruktion enthalten die nicht implementierten Bits des gelesenen 8 Bit breiten Worts den Wert ´0´. Beim Schreiben des Registers werden die nicht implementierten Bits des zu schreibenden Worts ignoriert.

2.7 Befehlssatzarchitektur

Im folgenden soll der Aufbau des Befehlssatz des Mikrocontrollerkerns beschrieben werden. Spezielles Augenmerk wird dabei auf die Unterschiede, die in diesem Bereich zwischen dem Original PIC17 und dem Mikrocontrollerkern bestehen, gelegt. Da bei der Entwicklung des Prozessormakros großer Wert auf Kompatibilität zum Original gelegt wurde, bestehen allerdings nur sehr wenige Differenzen, und Programme, die für den PIC17 entwickelt wurden, sollten in den allermeisten Fällen ohne Änderungen auf dem Mikrocontrollerkern ablaufen. Weitere Hinweise zum Befehlssatz und zur Programmierung können deshalb den entsprechenden Spezifikationen und Datenbüchern zum PIC17 entnommen werden [8,10]. Soweit hier nicht anders angegeben, treffen die dort gemachten Aussagen auch auf den Mikrocontrollerkern zu.

2.7.1 Adressierungsarten

Der Mikrocontrollerkern unterstützt generell nur drei verschiedene Adressierungsarten, um Operanden oder Adressen von Sprungzielen als Parameter der verschiedensten Befehle festzulegen. Bezugnehmend auf die normalerweise verwendete Klassifizierung [6], können die im Befehlssatz verwendeten Adressierungsmöglichkeiten folgendermaßen eingeteilt werden:

Direkte Adressierung (engl. *Direct Adressing*): Der Normalfall im Befehlssatz des Mikrocontrollerkerns ist die direkte Adressierung. In diesem Fall wird bei entsprechenden Instruktionen die Adresse des Operanden oder die Adresse der Quelle bzw. des Ziels einer Datentransferaktion bei der Programmierung direkt angegeben und fest im Befehl kodiert. Bei der Ausführung der Operation wird dann der Inhalt des Registers an der übergebenen Adresse ausgelesen und als Operand verwendet, und / oder es wird das Ergebnis der Operation in das Register an dieser Adresse geschrieben.

Beim Mikrocontrollerkern wird die direkte Adressierung ausschließlich von Instruktionen unterstützt, die im Datenspeicherbereich operieren und bei denen sich die angegebene Adresse dann auch auf den Datenspeicherbereich bezieht, d.h. z.B. bei arithmetischen, logischen und Transferoperationen. Mögliche Operanden, deren Wert

dem Inhalt des adressierten Registers entspricht, sind z.B. Summanden, Multiplikatoren oder zu verschiebende Daten. Im allgemeinen stellen bei solchen Instruktionen die unteren 8 Bit des Befehlsworts die benötigte Adresse dar. Daraus folgt, daß in diesen Fällen alle 256 Register des Datenbereichs direkt adressierbar sind und als Operanden verwendet werden können. Eine Ausnahme bilden nur Datentransferoperationen mit zwei Operanden, bei denen, zusätzlich zur Adresse des ersten Operanden in der unteren Worthälfte, die Adresse des zweiten Operanden in den Bits 8 bis 12 des Befehlswortes kodiert ist. Dementsprechend lassen sich damit per direkter Adressierung nur Werte von oder zu Registern im Bereich von 00h bis 1Fh verschieben.

Sprungbefehle, die als Operand die Adresse des Sprungziels benötigen, unterstützen die Übergabe der Zieladresse per direkter Adressierung nicht, d.h. es ist nicht möglich anzugeben, daß die Adresse des Sprungziels einem Register entnommen wird, dessen Adresse wiederum selbst in der Instruktion kodiert ist. In solchen Fällen kann die Zieladresse und damit der Operand des Sprungbefehls nur als Konstante kodiert werden (s.u.). Anzumerken bleibt, daß Sprünge aber auch realisiert werden können, indem das Ergebnis einer Operation in das im Datenbereich eingeblendete Register PCL geschrieben und so der Programmzähler neu geladen wird (siehe Abschnitt 2.6.3). Damit lassen sich mit beliebigen Instruktionen, die die direkte Adressierung unterstützen und so erlauben ihr Ergebnis in das direkt adressierte Register PCL zu schreiben, Sprünge verwirklichen.

Adressierung über Konstante (engl. *Literal* oder *Immediate Adressing*): Bei dieser Art der Adressierung wird der Wert des zu verwendenden Operanden selbst fest in der Instruktion als Konstante kodiert. Anstatt den benötigten Datenwert aus einem Register zu laden, verarbeiten die arithmetischen, logischen, Verschiebeoperationen, usw., direkt die angegebene Konstante, z.B. als Summand, Multiplikator oder zu verschiebenden Wert. Logischerweise wird ein anfallendes Ergebnis aber trotzdem in ein (implizit angegebenes) Register geschrieben werden, da es nicht im Befehl selbst abgelegt werden kann.

Die beiden Adressierungsarten sind für jeden Befehl einzeln verbindlich festgelegt, d.h. jeder Befehl unterstützt entweder die direkte Adressierung oder die Adressierung per Konstante, nicht aber beide. Aus diesem Grund existieren gegebenenfalls für die selbe Funktionalität verschiedene Befehle, die sich nur in der Art der Adressierung des Operanden unterscheiden. Allerdings enthält der Befehlssatz nicht immer Varianten mit beiden Adressierungsarten und es wird dann nur eine der beiden Adressierungsarten für die Durchführung einer Operation unterstützt. Analog zur direkten Adressierung, wird bei Befehlen mit Konstantenadressierung anstatt der Operandenadresse die 8 Bit breite Konstante in der unteren Hälfte des Befehlswortes abgelegt.

Bei den Sprung- und Verzweigeinstruktionen zur Programmablaufsteuerung ist die Angabe einer Konstante die einzige Möglichkeit die Adresse des Sprungziels, sofern benötigt, festzulegen. In den nur 16 Bit breiten Instruktionsworten kann aber keine vollständige, 16 Bit breite Programmspeicheradresse des Sprungziels abgelegt werden. Aus diesem Grund werden je nach Befehl entweder nur die unteren 8 oder 12 Bit der Zieladresse in den unteren 8 bzw. 12 Bit des Befehlsworts kodiert. Somit lassen sich prinzipiell mit diesen Instruktionen nur Sprünge realisieren, die einen gewissen Bereich um die momentane Programmspeicheradresse herum abdecken. Über Umwege werden aber trotzdem beliebige Sprünge über den gesamten Programmspeicherbereich hinweg unterstützt. Dazu wird die 16 Bit Zieladresse der entsprechenden Instruktion aus dem Inhalt eines impliziten, 8 Bit breiten Register und einer angegebenen, 8 Bit breiten Konstante zusammengesetzt.

Indirekte Adressierung (engl. *Indirect Adressing*): Der Mikrocontrollerkern implementiert die indirekte Adressierung über zwei speziell dafür vorgesehene Registerpaare: INDF0 und FSR0 einerseits und INDF1 und FSR1 andererseits. Die beiden Register INDF0 und INDF1 sind nicht physikalisch im Kern implementiert und nur als logische Register für den indirekten Zugriff vorhanden, d.h. in diesen Registern können keine Werte gespeichert werden. Die Register FSR0 und FSR1 bilden die 8 Bit breiten Zeiger in den Datenspeicherbereich für die indirekte Adressierung. Entsprechend können alle 256 Register des Datenspeicherbereichs über die indirekte Adressierung erreicht werden. Eine indirekte Adressierung wird verwirklicht, indem bei einem beliebigen Befehl, der direkte Adressierung unterstützt, eines der beiden Register INDF0 und INDF1 als Operand angegeben wird. Bei der Ausführung der Operation greift der Mikrocontrollerkern dann nicht auf diese Register zu, sondern er entnimmt die Adresse des Operanden dem Register FSR0, wenn als Operand das Register INDF0 angegeben wurde, bzw. dem Register FSR1, wenn INDF1 angegeben wurde. Daraus folgt, daß die indirekte Adressierung mit jedem Befehl anwendbar ist, der auch die direkte Adressierung unterstützt. Allerdings unterstützt der Mikrocontrollerkern die indirekte Adressierung nicht in vollem Umfang für jedes beliebige Register, da die (indirekte) Adresse des Operanden nur den Registern FSR0 und FSR1 entnommen werden kann. Wird also z.B. als Adresse des Operanden die Adresse des Registers INDF0 bzw. des Registers INDF1 (Adresse 00h bzw. Adresse 08h) in der Instruktion kodiert, dann wird der Inhalt des Registers, dessen Adresse im Register FSR0 bzw. FSR1 steht als Wert des Operanden ausgelesen, und / oder es wird das Ergebnis der Operation in das Register geschrieben, dessen Adresse im Register FSR0 bzw. FSR1 steht.

Über die indirekte Adressierung lassen sich somit Zugriffe kodieren, bei denen die Operandenadresse durch das Programm erst zur Laufzeit und nicht schon während der Übersetzung durch den Programmierer festgelegt wird. Weiterhin erlaubt es die indirekte Adressierung bei Datentransferoperationen, bei denen ja die Adresse des zweiten Operanden nur 5 Bit breit ist, als zweiten Operanden eines der Register INDF0 bzw. INDF1 anzugeben und so über indirekte Adressierung des zweiten Operanden Datentransfers zwischen Registern, die beliebig innerhalb des gesamten Datenspeicherbereich verteilt liegen können, durchzuführen.

Als zusätzliche Funktionalität im Zusammenhang mit der indirekten Adressierung besteht die Möglichkeit den Mikrocontrollerkern durch Setzen der Bits 4 bis 7 des Registers ALUSTA (Abschnitt 2.6.5) auch während der Laufzeit jederzeit so zu konfigurieren, daß die Register FSR0 bzw. FSR1 nach Ausführung eines indirekten Zugriffs vom Prozessor automatisch inkrementiert oder dekrementiert werden. Dabei wird zuerst der Befehl vollständig ausgeführt, d.h. alle Lese- und / oder Schreibzugriffe auf das indirekt adressierte Register durchgeführt, und danach erst das entsprechende File-Select Register entweder erhöht, erniedrigt oder ggf. auch nicht verändert. Diese Funktion kann ausgenutzt werden, um z.B. bei Blockverschiebung im Datenspeicherbereich den Zeiger in den Datenspeicherbereich bei aufeinanderfolgenden Zugriffen automatisch auf das nächste zu bearbeitende Register zu setzen.

Zu beachten ist bei der indirekten Adressierung, daß indirekte Zugriffe, die selbst wieder auf eines der Register INDF0 oder INDF1 verweisen, beim Lesen zur Folge haben, daß der Wert 00h gelesen wird und beim Schreiben ignoriert werden. Dies ist dann der Fall, wenn die Register FSR0 bzw. FSR1 einen der Werte 00h oder 08h beinhalten und dann ein Befehl ausgeführt wird, bei dem auf INDF0 bzw. INDF1 zugegriffen wird. Es ist also nicht möglich einen doppelt indirekten Zugriff, z.B. über Zugriff auf INDF0, wobei FSR0 die Adresse von INDF1 (08h) enthält und schließlich die Operandenadresse in FSR1 stehen soll, durchzuführen.

Zusätzlich zu diesen drei grundsätzlichen Adressierungsarten existieren noch einige Sonderfälle, die sich nicht direkt in eine dieser drei Kategorien einordnen lassen. Sie stehen im Zusammenhang mit Zugriffen auf das Register PCL und damit auf den Programmzähler (Abschnitt 2.6.3). Wie bereits erwähnt, können beliebige Instruktionen, die die direkte und damit auch indirekte Adressierung eines Registers unterstützen, das Register PCL als Operand angeben und so durch Schreiben eines Wertes in das Register PCL den Programmzähler neu setzen, d.h. einen Sprung durchführen. Im Endeffekt lassen sich zwei Fälle bezüglich der daraus resultierenden Adressierungsarten für den Operanden des Sprungs, d.h. für die Sprungzieladresse, prinzipiell unterscheiden. Einerseits läßt sich das Register PCL z.B. durch eine Transferaktion mit dem Inhalt eines anderen Registers oder einer Konstanten laden. Dies entspricht einer direkten oder einer Konstantendressierung, da sich die Sprungzieladresse aus dem Inhalt des in der Instruktion adressierten Registers bzw. aus der angegebenen Konstanten ergibt. Andererseits läßt sich die Zieladresse aber auch, z.B. durch eine Addition einer Konstanten oder eines Registerinhalts auf PCL, relativ zur aktuellen Programmadresse berechnen und so ein Sprung relativ zur momentanen Position durchführen. Dies entspricht keiner der bisherigen Adressierungsarten, da die Zieladresse und damit der Operand selbst erst berechnet werden. Im allgemeinen bezeichnet man diese spezielle Art der Adressierung als relative Adressierung (relativ zum Programmzähler).

2.7.2 Befehlssatz

Der Befehlssatz des Mikrocontrollerkerns ist in vier Gruppen aufgeteilt: Datentransferbefehle, Befehle zur Programmablaufsteuerung (Sprungbefehle), Bitmanipulationen und schließlich die Hauptgruppe mit arithmetischen und logischen Instruktionen. Innerhalb dieser Gruppen können die Befehle weiter anhand der Adressierungsart unterschieden werden. Schließlich existiert eine Gruppe von sonstigen Befehlen, die sich in keine der vier Gruppen einordnen lassen.

Der Mikrocontrollerkern ist eine sog. 1-Adreß Maschine mit Akkumulator, d.h. daß im allgemeinen bei Instruktionen, die zwei Operanden zu einem Ergebnis verknüpfen, nur ein Quelloperand explizit im Befehl angegeben wird, wohingegen für den zweiten Operanden und für das Ergebnis vom Befehl implizit ein dediziertes Register, der Akkumulator, verwendet wird. Beim Mikrocontrollerkern stellt das W-Register den Akkumulator dar. Standardmäßig wird der Wert des Akkumulators mit dem Wert des angegeben Register verknüpft und das Ergebnis wieder im Akkumulator abgelegt. Zusätzlich kann aber beim Mikrocontrollerkern im allgemeinen bei normalen arithmetischen und logischen Operationen auch angegeben werden, daß das Ergebnis alternativ nicht im W-Register, sondern im als Operanden übergebenen allgemeinen Datenregister abgelegt wird. Im Prinzip bedeutet dies, daß das angegebene Register zum Akkumulator wird und die Funktionalitäten von W-Register und angegebenem Register vertauscht sind. Bei einigen Befehle wird aber statt dessen auch angegeben, ob das Ergebnis alternativ nicht nur im W-Register, sondern sowohl im angegebenen Datenregister als auch im W-Register abgelegt wird.

Im folgenden werden alle vom Mikrocontrollerkern unterstützten Befehle der verschiedenen Gruppen kurz mit ihren Eigenschaften und Funktionen beschrieben. Die Übersicht enthält auch die Opcodes, d.h. die binäre Kodierung der Befehle und gibt an, welche Statusbits bei den einzelnen Befehlen, entsprechend den Bedingungen und Ergebnissen bei der Ausführung des Befehls, gesetzt werden. Die Bedeutung der einzelnen Statusbits, die sich in den Bits 0, 1 ,2 und 3 des Registers ALUSTA befinden, ist in Abschnitt 2.6.5 beschrieben. Soweit nicht anders angegeben, dauert die Ausführung der einzelnen Instruktionen jeweils genau einen Instruktionszyklus. Eine Ausnahme hiervon bilden hauptsächlich die Sprung- und Verzweigungsbefehle zur Programmablaufsteuerung, die zwei Instruktionszyklen benötigen. Da aber ein Sprung auch durch Schreiben des Registers PCL in einer nahezu beliebigen Instruktion durchgeführt werden kann (siehe Abschnitt 2.6.3), dauert die Ausführung solcher allgemeiner Instruktionen, bei denen durch Schreiben des Ergebnisses in das Register PCL ein

Sprung durchgeführt wird, auch zwei Zyklen. Nicht belegte Opcodes, d.h. Befehlswörter, die keinem der hier wiedergegebenen binär kodierten Befehl entsprechen, werden wie die Instruktion `NOP` behandelt und ausgeführt.

2.7.2.1 Datentransferoperationen

Befehl		Beschreibung (D[x], P[x]: Daten-/Programmspeicherzelle x)		Statusbits
`MOVFP`	*f,p*	*Move f to p* D[f] → D[p] Schreibt den Inhalt des Registers *f* in das Register *p*.	´011*p pppp ffff ffff*´	-
`MOVLB`	*k*	*Move literal to BSR* k → BSR Schreibt den Wert *k* in das Register BSR.	´1011 1000 *kkkk kkkk*´	-
`MOVPF`	*p,f*	*Move p to f* D[p] → D[f] Schreibt den Inhalt des Registers *p* in das Register *f*.	´010*p pppp ffff ffff*´	Z
`MOVWF`	*f*	*Move W to f* W → D[f] Schreibt den Inhalt des Registers W in das Register *f*.	´0000 0001 *ffff ffff*´	-
`TABLRD`	*t,i,f*	*Table Read* t = 0: TBLATL t = 1: TBLATH } → D[f] P[TBLPTR] → TBLAT i = 1: TBLPTR + 1 → TBLPTR Schreibt zuerst in Abhängigkeit von *t* entweder die untere Hälfte (*t* = ´0´) oder die obere Hälfte (*t* = ´1´) des Table-Latches in das Register *f*. Liest dann das Programmwort, dessen Adresse durch den Inhalt des Registers TBLPTR gegeben ist, aus dem Programmspeicher und schreibt es in das Table-Latch. Ist schließlich *i* gesetzt, dann wird das Register TBLPTR am Ende inkrementiert. Mit dieser Instruktion kann ein Wort aus dem Programmspeicher in das Register TBLAT gelesen werden. Das Register TBLPTR ist dabei der Zeiger in den Programmspeicher für den Zugriff. Mit der Instruktion `TLRD` kann der Inhalt des Register TBLAT und damit das gelesene Programmspeicherwort anschließend in den Datenspeicher transferiert werden. Insgesamt läßt sich dadurch ein 16 Bit breiter Befehl aus dem Programmspeicher in zwei 8 Bit breite Register des Datenspeichers schreiben und somit Daten zwischen Programm- und Datenspeicher austauschen. Durch das automatische Sichern einer Table-Latch-Hälfte vor dem Lesezugriff und durch die Möglichkeit einer automatischen Inkrementierung des Programmspeicherzeigers lassen sich aufeinanderfolgende Zugriffe auf fortlaufende Bereich, z.B. zum Lesen eines kompletten Programmspeicherblocks, einfacher realisieren. Aufgrund des zusätzlichen Programmspeicherzugriffs, der einen weiteren Instruktionszyklus beansprucht, umfaßt die	´1010 10*ti ffff ffff*´	-

Ausführung dieser Instruktion insgesamt zwei Instruktionszyklen. Wird allerdings das Register PCL als Register *f* angegeben, d.h. wenn der Inhalt von TBLATL bzw. TBLATH in das Register PCL geschrieben werden soll, benötigt die Instruktion zur Ausführung des dadurch initiierten Sprungs (siehe Abschnitt 2.6.3) insgesamt drei Instruktionszyklen, da zusätzlich noch ein Zyklus benötigt wird, um die nächste Instruktion an der Sprungzieladresse zu lesen, bevor die Ausführung fortfahren kann.

`TABLWT` *t,i,f* *Table Write* `´1010 11ti ffff ffff´` -

$$D[f] \rightarrow \begin{cases} \text{TBLATL} & t = 0 \\ \text{TBLATH} & t = 1 \end{cases}$$

$$\text{TBLATL} \rightarrow P[\text{TBLPTR}]$$

$$i = 1: \quad \text{TBLPTR} + 1 \rightarrow \text{TBLPTR}$$

Schreibt zuerst das Register *f* in Abhängigkeit von *t* entweder in die untere Hälfte (*t* = ´0´) oder die obere Hälfte (*t* = ´1´) des Table-Latches und schreibt dann den Inhalt des Table-Latches in das Programmwort im Programmspeicher, dessen Adresse durch den Inhalt des Registers TBLPTR gegeben ist. Ist schließlich *i* gesetzt, dann wird das Register TBLPTR am Ende inkrementiert.

Mit dieser Instruktion kann ein Wort aus dem Register TBLAT in den Programmspeicher geschrieben werden. Das Register TBLPTR ist dabei der Zeiger in den Programmspeicher für den Zugriff. Mit der Instruktion `TLWT` können vorher die Inhalte von Datenspeicherregistern in die beiden Hälften des Register TBLAT transferiert werden. Insgesamt lassen sich dadurch zwei 8 Bit breite Register des Datenspeichers in ein 16 Bit breites Wort im Programmspeicher schreiben und somit Daten zwischen Programm- und Datenspeicher austauschen. Durch das automatische Setzen einer Table-Latch-Hälfte vor dem Schreibzugriff und durch die Möglichkeit einer automatischen Inkrementierung des Programmspeicherzeigers lassen sich aufeinanderfolgende Zugriffe auf fortlaufende Bereich, z.B. zum Schreiben eines kompletten Programmspeicherblocks, einfacher realisieren.

Beim Original PIC17 kann mit dieser Instruktion gegebenenfalls auch das auf dem Chip integrierte Programmspeicher-EPROM programmiert werden, sofern der Programmspeicherzeiger auf eine Zelle im internen EPROM zeigt. Diese Möglichkeit wird vom Mikrocontrollerkern nicht unterstützt, da er kein integriertes EPROM enthält und der Programmspeicher komplett extern angeschaltet werden muß.

Aufgrund des zusätzlichen Programmspeicherzugriffs, der einen weiteren Instruktionszyklus beansprucht, umfaßt die Ausführung dieser Instruktion insgesamt zwei Instruktionszyklen.

`TLRD` *t,f* *Table Latch Read* `´1010 00tx ffff ffff´` -

$$\left.\begin{array}{l} t = 0: \quad \text{TBLATL} \\ t = 1: \quad \text{TBLATH} \end{array}\right\} \rightarrow D[f]$$

x: beliebig

Schreibt in Abhängigkeit von *t* entweder die untere Hälfte (*t* = ´0´) oder die obere Hälfte (*t* = ´1´) des Table-Latches in das Register *f*.

Damit können Daten, die vorher mit der Instruktion `TABLRD` aus dem Programmspeicher in das Register TBLAT geladen wurden von dort weiter in ein Register des Datenbereichs transferiert werden, um den Austausch von Daten vom Programm- zum Datenspeicher zu vervollständigen.

`TLWT` *t,f* *Table Latch Write* `´1010 01tx ffff ffff´` -

$$D[f] \rightarrow \begin{cases} \text{TBLATL} & t = 0 \\ \text{TBLATH} & t = 1 \end{cases}$$

x: beliebig

Schreibt das Register *f* in Abhängigkeit von *t* entweder in die untere Hälfte (*t* = ´0´) oder die obere Hälfte (*t* = ´1´) des Table-Latches.

Damit können Daten, die vorher mit der Instruktion `TABLRD` aus dem Programmspeicher in das Register TBLAT geladen wurden von dort weiter in ein Register des Datenbereichs transferiert werden, um den Austausch von Daten vom Programm- zum Datenspeicher zu vervollständigen.

2.7.2.2 Arithmetische und logische Operationen

Befehl		Beschreibung (D[x], P[x]: Daten-/Programmspeicherzelle x)		Statusbits
`ADDLW`	*k*	*Add literal to W*	`´1011 0001 kkkk kkkk´`	OV C DC Z

$W + k \rightarrow W$

Addiert den Wert *k* zum Inhalt des Registers W, und schreibt das Ergebnis in das Register W.

`ADDWF`	*f,d*	*Add W to f*	`´0000 111d ffff ffff´`	OV C DC Z

$$W + D[f] \rightarrow \begin{cases} W & d = 0 \\ D[f] & d = 1 \end{cases}$$

Addiert den Inhalt des Registers *f* und den Inhalt des Registers W, und schreibt das Ergebnis, in Abhängigkeit von *d*, entweder in das Register W (*d* = ´0´) oder in das Register *f* (*d* = ´1´).

`ADDWFC`	*f,d*	*Add W and Carry to f*	`´0001 000d ffff ffff´`	OV C DC Z

$$W + D[f] + C \rightarrow \begin{cases} W & d = 0 \\ D[f] & d = 1 \end{cases}$$

Addiert den Inhalt des Registers *f,* den Inhalt des Registers W und den Wert des Carry-Flags, und schreibt das Ergebnis, in Abhängigkeit von *d*, entweder in das Register W (*d* = ´0´) oder in das Register *f* (*d* = ´1´).

Mit Hilfe dieser Instruktion können zwei größere Zahlen, die mehrere Bytes umfassen, byteweise, unter Berücksichtigung der Überträge von einem Byte zum nächsten, addiert werden.

`ANDLW` *k* — *AND literal and W* — `´1011 0101 kkkk kkkk´` — Z

$W \wedge k \rightarrow W$

Macht eine logische Und-Verknüpfung zwischen dem Inhalt des Registers W und dem Wert *k*, und schreibt das Ergebnis in das Register W.

`ANDWF` *f,d* — *AND W with f* — `´0000 101d ffff ffff´` — Z

$$W \wedge D[f] \rightarrow \begin{cases} W & d = 0 \\ D[f] & d = 1 \end{cases}$$

Macht eine logische Und-Verknüpfung zwischen dem Inhalt des Registers W und dem Inhalt des Registers *f*, und schreibt das Ergebnis, in Abhängigkeit von *d*, entweder in das Register W (*d* = ´0´) oder in das Register *f* (*d* = ´1´).

`CLRF` *f,d* — *Clear f and d* — `´0010 100d ffff ffff´` — -

$$'00000000' \rightarrow \begin{cases} W, D[f] & d = 0 \\ D[f] & d = 1 \end{cases}$$

Schreibt den Wert 00h in das Register *f* und, in Abhängigkeit von *d*, zusätzlich in das Register W (*d* = ´0´).

`COMF` *f,d* — *Complement f* — `´0001 001d ffff ffff´` — Z

$$\overline{D[f]} \rightarrow \begin{cases} W & d = 0 \\ D[f] & d = 1 \end{cases}$$

Bildet das logische Komplement (Einerkomplement) des Inhalts des Registers *f* und schreibt es, in Abhängigkeit von *d*, entweder in das Register W (*d* = ´0´) oder in das Register *f* (*d* = ´1´).

`DAW` *f,d* — *Decimal Adjust W* — `´0010 111d ffff ffff´` — C

$$\left.\begin{matrix} W \\ W + 06h \\ W + 66h \end{matrix}\right\} \rightarrow \begin{cases} W, D[f] & d = 0 \\ D[f] & d = 1 \end{cases}$$

Der Inhalt des Registers W wird, unter der Annahme, daß er das Ergebnis einer vorherigen normalen, binären Addition zweier Zahlen im BCD-Format ist, so korrigiert, daß er das korrekte Ergebnis der Addition im BCD-Format darstellt. Der korrigierte Wert wird in das Register *f* und, in Abhängigkeit von *d*, zusätzlich in das Register W (*d* = ´1´) geschrieben, und das Carry-Bit wird passend gesetzt, um das Additionsergebnis korrekt wiederzugeben.

Zahlen im BCD-Format (engl. *Binary Coded Decimal*) sind so kodiert, daß jede Byte-Hälfte genau einer Dezimalstelle einer Zahl entspricht. Aufgrund des Wertebereichs von 0-9 einer Dezimalstelle sind zu ihrer Kodierung 4 Bit, d.h. eine Byte-Hälfte, nötig und somit können mit einem Byte im BCD-Format Werte im Bereich von 0 bis 99 dezimal dargestellt werden. Dezimalstellen einer BCD-Zahl belegen entsprechend ihres Wertes die Codes ´0000´ (0h) bis ´1001´ (9h) der zugehörigen Byte-Hälfte. Da mit einer 4 Bit breiten Hälfte aber bis zu 16 Werte kodiert werden können, werden bei BCD-

Zahlen die restlichen 6 Codes von ´1010´ (Ah) bis ´1111´ (Fh) nicht verwendet und dürfen deshalb bei BCD-kodierten Zahlen nicht auftreten. Die Dezimalzahl 47 z.B. ergibt BCD-kodiert den hexadezimalen Wert 47h, der wiederum dem dezimalen Wert 71 im Dualsystem entsprechen würde. Auf der anderen Seite ist z.B. der hexadezimale Wert 0Ah (entsprechen dezimal 10 im Dualsystem) eine ungültige BCD-Zahl.

Werden nun zwei BCD-kodierte Zahlen mit Hilfe einer normalen binäre Additions- oder Subtraktionsoperation (Instruktionen `ADDLW`, `ADDWF` und `ADDWFC` bzw. `SUBLW`, `SUBWF` und `SUBWFB` entsprechend einer Addition des Zweierkomplements) addiert, so kann es durch Überläufe aus dem BCD-Wertebereich hinaus vorkommen, daß das Ergebnis keine korrekte BCD-Zahl oder eine gültige BCD-Zahl aber mit falschem Wert enthält. Werden z.B. die Ziffern 5 und 6 einer Byte-Hälfte binär addiert, so enthält die Byte-Hälfte des Ergebnisses den binären Wert 11 (´1011´). Das korrekte Ergebnis einer BCD-Addition wäre aber der Wert 1 für diese Byte-Hälfte, also für diese Dezimalstelle, und ein Übertrag in die nächste Byte-Hälfte, also in die nächste Dezimalstelle.

Ausgehend vom binären Additionsergebnis im Register W und in Abhängigkeit vom Status der binären Addition, wiedergegeben durch die Statusbits DC und C, korrigiert die Instruktion DAW nun den Inhalt des Registers W und die Statusbits so, daß beide das korrekte Ergebnis der BCD-Addition zweier BCD-Zahlen wiedergeben. Dazu wird gegebenenfalls der Wert 6 zu den einzelnen Bytehälften hinzuaddiert, um so eine außerhalb des BCD-Wertebereiches liegende, d.h. übergelaufene Dezimalstelle, auf den passenden Wert zu bringen und einen Überlauf in die nächste Stelle hinein zu erzeugen. Im obigen Beispiel wird dann aus dem Wert ´1011´ durch Addition von 6 der Wert ´0001´ mit einem Carry in das nächste Bit. In diesem Fall wurde eine Korrektur vorgenommen, weil der Wert der Byte-Hälfte größer als neun und damit ungültig war. In anderen Fällen kann aber durchaus der Wert kleiner als neun sein, die Zahl aber trotzdem falsch sein, z.B. wenn die Ziffern 9 und 8 zu einem binären Ergebnis von 17 (´1 0001´) entsprechend einer Ziffer 1 in der Byte-Hälfte plus Übertrag addiert werden. In diesem Fall wird von der Instruktion anhand der Statusbits erkannt, daß bei der binären Addition ein Übertrag aus der Byte-Hälfte hinaus stattgefunden hat und deshalb die Ziffer durch Addition des Wertes 6 ebenfalls auf den korrekten Wert gebracht. Gleichzeitig wird in der Instruktion aber auch verarbeitet, daß bei der Korrektur der unteren Byte-Hälfte ein Übertrag in die obere Byte-Hälfte erzeugt werden kann, der dort bei der Korrektur berücksichtigt werden muß. Zusätzlich wird das Carry-Flag passend korrigiert, so daß dieses im Falle der Addition größerer BCD-Zahlen bei der binären Addition des nächsten Bytes der BCD-Zahl den korrekten Wert entsprechend der Verhältnisse enthält und dort mit einbezogen werden kann.

`DECF` *f,d* *Decrement f* `´0000 011d ffff ffff´` OV C DC Z

$$D[f]-1 \rightarrow \begin{cases} W & d = 0 \\ D[f] & d = 1 \end{cases}$$

Dekrementiert (durch Addition des Zweierkomplements von eins, FFh) den Inhalt des Registers *f* und schreibt das Ergebnis, in Abhängigkeit von *d*, entweder in das Register W (*d* = ´0´) oder in das Register *f* (*d* = ´1´).

`INCF` *f,d* *Increment f* `´0001 010d ffff ffff´` OV C DC Z

$$D[f]+1 \rightarrow \begin{cases} W & d = 0 \\ D[f] & d = 1 \end{cases}$$

Inkrementiert den Inhalt des Registers *f* und schreibt das Ergebnis, in Abhängigkeit von *d*, entweder in das Register W (*d* = ´0´) oder in das Register *f* (*d* = ´1´).

`IORLW` *k* *Inclusive OR literal with W* `´1011 0011 kkkk kkkk´` Z

$$W \vee k \rightarrow W$$

Macht eine logische Oder-Verknüpfung zwischen dem Inhalt des Registers W und dem Wert *k*, und schreibt das Ergebnis in das Register W.

`IORWF` *f,d* *Inclusive OR W with f* `´0000 100d ffff ffff´` Z

$$W \vee D[f] \rightarrow \begin{cases} W & d = 0 \\ D[f] & d = 1 \end{cases}$$

Macht eine logische Oder-Verknüpfung zwischen dem Inhalt des Registers W und dem Inhalt des Registers *f*, und schreibt das Ergebnis, in Abhängigkeit von *d*, entweder in das Register W (*d* = ´0´) oder in das Register *f* (*d* = ´1´).

`NEGW` *f,d* *Negate W* `´0010 110d ffff ffff´` OV C DC Z

$$\overline{W}+1 \rightarrow \begin{cases} W, D[f] & d = 0 \\ D[f] & d = 1 \end{cases}$$

Bildet das Zweierkomplement des Inhalts des Registers W und schreibt es in das Register *f* und, in Abhängigkeit von *d*, zusätzlich in das Register W (*d* = ´0´).

`RLCF` *f,d* *Rotate Left f through Carry* `´0001 101d ffff ffff´` C

$$D[f]\big|_{\text{Bit 0}}^{\text{Bit 6}} \rightarrow \begin{cases} W\big|_{\text{Bit 1}}^{\text{Bit 7}} & d = 0 \\ D[f]\big|_{\text{Bit 1}}^{\text{Bit 7}} & d = 1 \end{cases}$$

$$C \rightarrow \begin{cases} W\big|_{\text{Bit 0}} & d = 0 \\ D[f]\big|_{\text{Bit 0}} & d = 1 \end{cases}$$

$$D[f]\big|_{\text{Bit 7}} \rightarrow C$$

Rotiert den Inhalt des Registers *f* unter Berücksichtigung des Carry-Flags um eins nach links und schreibt das Ergebnis, in Abhängigkeit von *d*, entweder in das Register W (*d* = ´0´) oder in das Register *f* (*d* = ´1´).

`RLNCF` *f,d* *Rotate Left f (no Carry)* `´0010 001d ffff ffff´` -

$$D[f]\big|_{\text{Bit 0}}^{\text{Bit 6}} \rightarrow \begin{cases} W\big|_{\text{Bit 1}}^{\text{Bit 7}} & d = 0 \\ D[f]\big|_{\text{Bit 1}}^{\text{Bit 7}} & d = 1 \end{cases}$$

$$D[f]\big|_{\text{Bit 7}} \rightarrow \begin{cases} W\big|_{\text{Bit 0}} & d = 0 \\ D[f]\big|_{\text{Bit 0}} & d = 1 \end{cases}$$

Rotiert den Inhalt des Registers *f* um eins nach links und schreibt das Ergebnis, in Abhängigkeit von *d*, entweder in das Register W (*d* = ´0´) oder in das Register *f* (*d* = ´1´).

`RRCF` *f,d* *Rotate Right f through Carry* `´0001 100d ffff ffff´` C

$$D[f]\big|_{\text{Bit 1}}^{\text{Bit 7}} \rightarrow \begin{cases} W\big|_{\text{Bit 0}}^{\text{Bit 6}} & d = 0 \\ D[f]\big|_{\text{Bit 0}}^{\text{Bit 6}} & d = 1 \end{cases}$$

$$C \rightarrow \begin{cases} W\big|_{\text{Bit 7}} & d = 0 \\ D[f]\big|_{\text{Bit 7}} & d = 1 \end{cases}$$

$$D[f]\big|_{\text{Bit 0}} \rightarrow C$$

Rotiert den Inhalt des Registers *f* unter Berücksichtigung des Carry-Flags um eins nach rechts und schreibt das Ergebnis, in Abhängigkeit von *d*, entweder in das Register W (*d* = ´0´) oder in das Register *f* (*d* = ´1´).

`RRNCF` *f,d* *Rotate Right f (no Carry)* `´0010 000d ffff ffff´` -

$$D[f]\big|_{\text{Bit 1}}^{\text{Bit 7}} \rightarrow \begin{cases} W\big|_{\text{Bit 0}}^{\text{Bit 6}} & d = 0 \\ D[f]\big|_{\text{Bit 0}}^{\text{Bit 6}} & d = 1 \end{cases}$$

$$D[f]\big|_{\text{Bit 0}} \rightarrow \begin{cases} W\big|_{\text{Bit 7}} & d = 0 \\ D[f]\big|_{\text{Bit 7}} & d = 1 \end{cases}$$

Rotiert den Inhalt des Registers *f* um eins nach rechts und schreibt das Ergebnis, in Abhängigkeit von *d*, entweder in das Register W (*d* = ´0´) oder in das Register *f* (*d* = ´1´).

`SETF` *f,d* *Set f and d* `´0010 101d ffff ffff´` -

$$'11111111' \rightarrow \begin{cases} W, D[f] & d = 0 \\ D[f] & d = 1 \end{cases}$$

Schreibt den Wert FFh in das Register *f* und, in Abhängigkeit von *d*, zusätzlich in das Register W (*d* = ´0´).

`SUBLW` *k* *Subtract W from Literal* `´1011 0010 kkkk kkkk´` OV C DC Z

$$k - W = k + \overline{W} + 1 \rightarrow W$$

Zieht den Inhalt des Registers W vom Wert *k* ab und schreibt das Ergebnis in das Register W.

Die Subtraktion wird durch Addition des Werts *k* und des Zweierkomplements von W durchgeführt. Das Carry-Bit wird gemäß dem nichtinvertierten Wert des aus Bit 7 der Addition heraus entstandenen Überlaufs gesetzt.

`SUBWF` *f,d* *Subtract W from f* `´0000 010d ffff ffff´` OV C DC Z

$$D[f] - W = D[f] + \overline{W} + 1 \rightarrow \begin{cases} W & d = 0 \\ D[f] & d = 1 \end{cases}$$

Zieht den Inhalt des Registers W vom Inhalt des Registers *f* ab, und schreibt das Ergebnis, in Abhängigkeit von *d*, entweder in das Register W (*d* = ´0´) oder in das Register *f* (*d* = ´1´).

Die Subtraktion wird durch Addition des Registers *f* und des Zweierkomplements von W durchgeführt. Das Carry-Bit wird gemäß dem nichtinvertierten Wert des aus Bit 7 der Addition heraus entstandenen Überlaufs gesetzt. Nachfolgende Subtraktionen höherwertiger Bytes einer zusammengesetzten Zahl mit Hilfe von `SUBWFB` berücksichtigen dies entsprechend.

`SUBWFB` *f,d* *Subtract W from f with Borrow* `´0001 001d ffff ffff´` OV C DC Z

$$D[f] - W - \overline{C} = D[f] + \overline{W} + C \rightarrow \begin{cases} W & d = 0 \\ D[f] & d = 1 \end{cases}$$

Zieht den Inhalt des Registers W und Wert des sog. *Borrow*-Flags vom Inhalt des Registers *f* ab, und schreibt das Ergebnis, in Abhängigkeit von *d*, entweder in das Register W (*d* = ´0´) oder in das Register *f* (*d* = ´1´). Das Borrow-Flag ist dabei der invertierte Wert des Carry-Flags einer vorangegangenen Subtraktion, denn wenn bei einer vorangegangenen Addition eines Zweierkomplements ein Überlauf (Carry) aufgetreten ist, dann war das Ergebnis positiv und es muß bei der Subtraktion des nächsthöheren Bytes nichts abgezogen werden. Ist hingegen kein Überlauf (Carry) aufgetreten, so war dort das Ergebnis negativ und bei der darauffolgenden Subtraktion muß zusätzlich noch der Wert 1 abgezogen werden.

Die Subtraktion wird durch Addition des Registers *f,* des logischen (Einer-)Komplements von W und des Carry-Flags durchgeführt. Dies führt zum selben Ergebnis wie eine Addition des Registers *f*, des Zweierkomplements von W ($-W = \overline{W} + 1$) und des 8 Bit breiten Zweierkomplements des invertierten Carry-Flags ($-\overline{C} = FFh + C = -1 + C$, denn es gilt $-'...0x' = \overline{'...0x'} + 1 = '...11' + \overline{x}$). Das Carry-Bit wird für nachfolgende Subtraktionen weiterer Bytes einer größeren Zahl gemäß dem nichtinvertierten Wert des aus Bit 7 der Addition heraus entstandenen Überlaufs gesetzt.

Mit Hilfe dieser Instruktion können zwei größere Zahlen, die mehrere Bytes umfassen, byteweise, unter Berücksichtigung der Überträge von einem Byte zum nächsten, voneinander abgezogen werden. Dabei wird die Subtraktion der beiden großen Zahlen auf eine entsprechende Addition des Zweierkomplements über die gesamte Zahlenbreite hinweg zurückgeführt ($Z = X - Y = X + \overline{Y} + 1$ über alle Bytes, d.h. für das erste Byte gilt $z_0 = x_0 + \overline{y_0} + 1$, für alle weiteren $z_n = x_n + \overline{y_n} + C$, wobei C bei jedem Byte entsprechend dem Überlauf bei der Addition gesetzt wird).

`SWAPF` *f,d* *Swap f* `´0001 110d ffff ffff´` -

$$D[f]\big|_{\text{Bit 4}}^{\text{Bit 7}} \rightarrow \begin{cases} W\big|_{\text{Bit 0}}^{\text{Bit 3}} & d = 0 \\ D[f]\big|_{\text{Bit 0}}^{\text{Bit 3}} & d = 1 \end{cases}$$

$$D[f]\big|_{\text{Bit 0}}^{\text{Bit 3}} \rightarrow \begin{cases} W\big|_{\text{Bit 4}}^{\text{Bit 7}} & d = 0 \\ D[f]\big|_{\text{Bit 4}}^{\text{Bit 7}} & d = 1 \end{cases}$$

Vertauscht die obere und untere Byte-Hälfte des Inhalts des Registers *f* und schreibt das Ergebnis, in Abhängigkeit von *d*, entweder in das Register W (*d* = ´0´) oder in das Register *f* (*d* = ´1´).

`XORLW` *k* *Exclusive OR literal with W* `´1011 0100 kkkk kkkk´` Z

$W \oplus k \rightarrow W$

Macht eine logische Exklusiv-Oder-Verknüpfung zwischen dem Inhalt des Registers W und dem Wert *k*, und schreibt das Ergebnis in das Register W.

`XORWF` *f,d* *Exclusive OR W with f* `´0000 110d ffff ffff´` Z

$$W \oplus D[f] \rightarrow \begin{cases} W & d = 0 \\ D[f] & d = 1 \end{cases}$$

Macht eine logische Exklusiv-Oder-Verknüpfung zwischen dem Inhalt des Registers W und dem Inhalt des Registers *f*, und schreibt das Ergebnis, in Abhängigkeit von *d*, entweder in das Register W (*d* = ´0´) oder in das Register *f* (*d* = ´1´).

2.7.2.3 Operationen zur Steuerung des Programmablaufs

Befehl		Beschreibung		Statusbits
`CALL`	*k*	*Subroutine Call*	`´111k kkkk kkkk kkkk´`	-

$PC \rightarrow \text{Stack}$

$$k \rightarrow PC\big|_{\text{Bit 0}}^{\text{Bit 12}}$$

$PCH \rightarrow PCLATH$

Unterroutinenaufruf. Schiebt zuerst den aktuellen Inhalt des Programmzählers als Rücksprungadresse auf den Stack, schreibt dann den Wert *k* in die unteren 13 Bits des Programmzählers und schreibt schließlich den neuen Inhalt der oberen Hälfte des Programmzählers in das Register PCLATH.

Die Instruktion verzweigt die Programmausführung zu der Unterroutine, deren Adresse durch *k* gegeben ist und bereitet gleichzeitig einen späteren Rücksprung mittels `RETURN` aus der Unterroutine heraus zum Aufrufer zurück vor. Gleichzeitig werden die oberen 8 Bit der Ziel- bzw. Unterroutinenadresse im Register PCLATH gesichert. Da *k* nur 13 Bit breit ist und dementsprechend nur die unteren 13 Bits des Programmzählers neu gesetzt werden, können mit dieser Instruktion nur Routinen aufgerufen werden, deren Startadresse sich innerhalb des selben, 8k Worte umfassenden Bereichs befindet, in dem auch die momentane Position ist.

Da nach dem Setzen des Programmzählers im nächsten

Instruktionszyklus erst der erste Befehl der Unterroutine an der neuen Adresse gelesen werden muß bevor der Programmablauf fortfährt, benötigt die Instruktion zwei Instruktionszyklen zur Ausführung.

`CPFSEQ f` *Compare f with W, Skip if Equal* `´0011 0001 ffff ffff´` -

$D[f] = W : \text{"NOP"} \rightarrow IR$

Überspringt die nächste Instruktion, wenn der Inhalt des Registers *f* und der Inhalt des Registers W gleich sind.

Die Instruktion vergleicht die Inhalte der Register W und *f* durch Subtraktion voneinander (Zweierkomplementaddition) und übergeht ggf. die Ausführung des nachfolgenden Befehls, indem der Mikrocontrollerkern veranlaßt wird im nächsten Instruktionszyklus den dann anstehenden Befehl nicht auszuführen und statt dessen ein `NOP` einzuschieben.

`CPFSGT f` *Compare f with W, Skip if Greater* `´0011 0010 ffff ffff´` -

$D[f] > W : \text{"NOP"} \rightarrow IR$

Überspringt die nächste Instruktion, wenn der Inhalt des Registers *f* größer als der Inhalt des Registers W ist.

Die Instruktion vergleicht die Inhalte der Register W und *f* durch Subtraktion voneinander (Zweierkomplementaddition) und übergeht ggf. die Ausführung des nachfolgenden Befehls, indem der Mikrocontrollerkern dazu veranlaßt wird, im nächsten Instruktionszyklus den dann anstehenden Befehl nicht auszuführen und statt dessen ein `NOP` einzuschieben.

`CPFSLT f` *Compare f with W, Skip if Less* `´0011 0000 ffff ffff´` -

$D[f] < W : \text{"NOP"} \rightarrow IR$

Überspringt die nächste Instruktion, wenn der Inhalt des Registers *f* kleiner als der Inhalt des Registers W ist.

Die Instruktion vergleicht die Inhalte der Register W und *f* durch Subtraktion voneinander (Zweierkomplementaddition) und übergeht ggf. die Ausführung des nachfolgenden Befehls, indem der Mikrocontrollerkern dazu veranlaßt wird, im nächsten Instruktionszyklus den dann anstehenden Befehl nicht auszuführen und statt dessen ein `NOP` einzuschieben.

`DECFSZ f,d` *Decrement f, Skip if Zero* `´0001 011d ffff ffff´` -

$D[f] - 1 = \text{'00000000'} : \text{"NOP"} \rightarrow IR$

$$D[f] - 1 \rightarrow \begin{cases} W & d = 0 \\ D[f] & d = 1 \end{cases}$$

Dekrementiert (durch Addition des Zweierkomplements von eins, FFh) den Inhalt des Registers *f*, schreibt das Ergebnis, in Abhängigkeit von *d*, entweder in das Register W (*d* = ´0´) oder in das Register *f* (*d* = ´1´), und überspringt die nächste Instruktion, wenn das Ergebnis null war.

Die Instruktion testet das Ergebnis auf null und übergeht ggf. die Ausführung des nachfolgenden Befehls, indem der Mikrocontrollerkern dazu veranlaßt wird, im nächsten Instruktionszyklus den dann anstehenden Befehl nicht auszuführen und statt dessen ein `NOP` einzuschieben.

`DCFSNZ` *f,d* *Decrement f, Skip if Not Zero* `´0010 011d ffff ffff´` -

$D[f]-1 \neq \text{'00000000'}: \text{"NOP"} \rightarrow IR$

$$D[f]-1 \rightarrow \begin{cases} W & d = 0 \\ D[f] & d = 1 \end{cases}$$

Dekrementiert (durch Addition des Zweierkomplements von eins, FFh) den Inhalt des Registers *f*, schreibt das Ergebnis, in Abhängigkeit von *d*, entweder in das Register W (*d* = ´0´) oder in das Register *f* (*d* = ´1´), und überspringt die nächste Instruktion, wenn das Ergebnis nicht null war.

Die Instruktion testet das Ergebnis auf null und übergeht ggf. die Ausführung des nachfolgenden Befehls, indem der Mikrocontrollerkern dazu veranlaßt wird, im nächsten Instruktionszyklus den dann anstehenden Befehl nicht auszuführen und statt dessen ein `NOP` einzuschieben.

`GOTO` *k* *Unconditional Branch* `´110k kkkk kkkk kkkk´` -

$$k \rightarrow PC\Big|_{\text{Bit 0}}^{\text{Bit 12}}$$

$PCH \rightarrow PCLATH$

Sprung an eine andere Programmstelle. Schreibt den Wert *k* in die unteren 13 Bits des Programmzählers und anschließend den neuen Inhalt der oberen Hälfte des Programmzählers in das Register PCLATH.

Die Instruktion verzweigt die Programmausführung zu der Stelle im Programmspeicher, deren Adresse durch *k* gegeben ist. Gleichzeitig werden die oberen 8 Bit der Sprungzieladresse im Register PCLATH gesichert. Da *k* nur 13 Bit breit ist und dementsprechend nur die unteren 13 Bits des Programmzählers neu gesetzt werden, kann mit dieser Instruktion nur an Stellen gesprungen werden, die sich innerhalb des selben, 8k Worte umfassenden Bereichs befinden, in dem auch die momentane Position ist.

Da nach dem Setzen des Programmzählers im nächsten Instruktionszyklus erst der nächste Befehl an der neuen Adresse gelesen werden muß bevor der Programmablauf fortfährt, benötigt die Instruktion zwei Instruktionszyklen zur Ausführung.

`INCFSZ` *f,d* *Increment f, Skip if Zero* `´0001 111d ffff ffff´` -

$D[f]+1 = \text{'00000000'}: \text{"NOP"} \rightarrow IR$

$$D[f]+1 \rightarrow \begin{cases} W & d = 0 \\ D[f] & d = 1 \end{cases}$$

Inkrementiert den Inhalt des Registers *f*, schreibt das Ergebnis, in Abhängigkeit von *d*, entweder in das Register W (*d* = ´0´) oder in das Register *f* (*d* = ´1´), und überspringt die nächste Instruktion, wenn das Ergebnis null war.

Die Instruktion testet das Ergebnis auf null und übergeht ggf. die Ausführung des nachfolgenden Befehls, indem der Mikrocontrollerkern dazu veranlaßt wird, im nächsten Instruktionszyklus den dann anstehenden Befehl nicht auszuführen und statt dessen ein `NOP` einzuschieben.

`INFSNZ` *f,d* *Increment f, Skip if Not Zero* `´0010 010d ffff ffff´` -

$D[f]+1 \neq \text{'00000000'} : \text{"NOP"} \rightarrow IR$

$$D[f]+1 \rightarrow \begin{cases} W & d = 0 \\ D[f] & d = 1 \end{cases}$$

Inkrementiert den Inhalt des Registers *f*, schreibt das Ergebnis, in Abhängigkeit von *d*, entweder in das Register W (*d* = ´0´) oder in das Register *f* (*d* = ´1´), und überspringt die nächste Instruktion, wenn das Ergebnis nicht null war.

Die Instruktion testet das Ergebnis auf null und übergeht ggf. die Ausführung des nachfolgenden Befehls, indem der Mikrocontrollerkern dazu veranlaßt wird, im nächsten Instruktionszyklus den dann anstehenden Befehl nicht auszuführen und statt dessen ein `NOP` einzuschieben.

`LCALL` *k* *Long Call* `´1011 0111 kkkk kkkk´` -

$PC \rightarrow \text{Stack}$

$k \rightarrow PCL$

$PCLATH \rightarrow PCH$

Unterroutinenaufruf. Schiebt zuerst den aktuellen Inhalt des Programmzählers als Rücksprungadresse auf den Stack, und schreibt dann den Wert *k* in die unteren Hälfte des Programmzählers und den Inhalt des Registers PCLATH in die oberen Hälfte des Programmzählers.

Die Instruktion verzweigt die Programmausführung zu der Unterroutine, deren Adresse durch die Kombination von PCLATH und *k* gegeben ist und bereitet gleichzeitig einen späteren Rücksprung mittels `RETURN` aus der Unterroutine heraus zum Aufrufer zurück vor. Da mit der aus dem 8 Bit breiten Register PCLATH und dem 8 Bit breiten Wert *k* zusammengesetzten Zieladresse alle 16 Bits des Programmzählers neu gesetzt werden, können mit dieser Instruktion Routinen aufgerufen werden, die sich an einer beliebigen Stelle im gesamten Programmspeicherbereich befinden. Vor Aufruf der Instruktion muß das Anwendungsprogramm das Register PCLATH entsprechend laden.

Da nach dem Setzen des Programmzählers im nächsten Instruktionszyklus erst der erste Befehl der Unterroutine an der neuen Adresse gelesen werden muß bevor der Programmablauf fortfährt, benötigt die Instruktion zwei Instruktionszyklen zur Ausführung.

`RETFIE` *Return From Interrupt* `´0000 0000 0000 0101´` GLINTD

$\text{Stack} \rightarrow PC$

$\text{'0} \rightarrow GLINTD$

Rücksprung aus einer Interruptroutine. Holt den obersten Wert als Rücksprungadresse vom Stack, schreibt diesen in den Programmzähler und setzt das Flag GLINTD zurück auf ´0´.

Die Instruktion verzweigt die Programmausführung zurück zum Aufrufer der Unterroutine, dessen Adresse an oberster

Stelle des Stacks liegt, und setzt so die Programmausführung nach der Stelle fort, an der die Unterroutine (i.a. durch Auslösen eines Interrupts) aufgerufen wurde. Die Instruktion ist hauptsächlich zur Rückkehr aus Interruptroutinen und nicht zur Rückkehr aus per `CALL` oder `LCALL` aufgerufenen Unterroutinen gedacht. Beim Auslösen eines Interrupts wird vom Mikrocontrollerkern in die entsprechende Interruptroutine verzweigt und gleichzeitig das Flag GLINTD gesetzt, um durch globales Sperren aller Interrupts automatisch sicherzustellen, daß während der Abarbeitung des Interrupts keine weiteren, verschachtelten Interrupts auftreten können. Mit Hilfe dieser Instruktion wird am Ende einer Interruptroutine dann automatisch bei der Rückkehr aus dem Interrupt das Flag GLINTD wieder gelöscht und so Interrupts wieder allgemein freigeschaltet.

Da nach dem Setzen des Programmzählers im nächsten Instruktionszyklus erst der erste Befehl des Aufrufers an der neuen Adresse gelesen werden muß bevor der Programmablauf fortfährt, benötigt die Instruktion zwei Instruktionszyklen zur Ausführung.

`RETLW` *k* — *Return Literal to W* — `´1011 0110 kkkk kkkk´` — -

Stack → PC

k → W

Rücksprung aus einer Unterroutine mit Setzen von W. Holt den obersten Wert als Rücksprungadresse vom Stack, schreibt diesen in den Programmzähler und schreibt schließlich noch den Wert *k* in das Register W.

Die Instruktion verzweigt die Programmausführung zurück zum Aufrufer der Unterroutine, dessen Adresse an oberster Stelle des Stacks liegt, und setzt so die Programmausführung nach der Stelle fort, an der die Unterroutine durch einen `CALL` oder einen `LCALL` aufgerufen wurde. Das Programm fährt also die Ausführung mit der Instruktion, die direkt auf die aufrufende Instruktion (`CALL` oder `LCALL`) folgt, fort. Zusätzlich wird beim Rücksprung das Register W mit dem angegebenen Wert geladen, um z.B. einen Wert an den Aufrufer zurückzuliefern.

Da nach dem Setzen des Programmzählers im nächsten Instruktionszyklus erst der erste Befehl des Aufrufers an der neuen Adresse gelesen werden muß bevor der Programmablauf fortfährt, benötigt die Instruktion zwei Instruktionszyklen zur Ausführung.

`RETURN` — *Return from subroutine* — `´0000 0000 0000 0010´` — -

Stack → PC

Rücksprung aus einer Unterroutine. Holt den obersten Wert als Rücksprungadresse vom Stack, und schreibt diesen in den Programmzähler.

Die Instruktion verzweigt die Programmausführung zurück zum Aufrufer der Unterroutine, dessen Adresse an oberster Stelle des Stacks liegt, und setzt so die Programmausführung

nach der Stelle fort, an der die Unterroutine durch einen `CALL` oder einen `LCALL` aufgerufen wurde. Das Programm fährt also die Ausführung mit der Instruktion, die direkt auf die aufrufende Instruktion (`CALL` oder `LCALL`) folgt, fort.

Da nach dem Setzen des Programmzählers im nächsten Instruktionszyklus erst der erste Befehl des Aufrufers an der neuen Adresse gelesen werden muß bevor der Programmablauf fortfährt, benötigt die Instruktion zwei Instruktionszyklen zur Ausführung.

2.7.2.4 Operationen zur Bitmanipulation

Befehl		Beschreibung (D[x], P[x]: Daten-/Programmspeicherzelle x)		Statusbits
`BCF`	*f,b*	*Bit Clear f* $'0' \rightarrow D[f]\big\vert_{Bit\ b}$ Setzt das Bit *b* im Register *f* auf ´0´.	´1000 1*bbb ffff ffff*´	-
`BSF`	*f,b*	*Bit Setr f* $'1' \rightarrow D[f]\big\vert_{Bit\ b}$ Setzt das Bit *b* im Register *f* auf ´1´.	´1000 0*bbb ffff ffff*´	-
`BTFSC`	*f,b*	*Bit Test f, Skip if Clear* $D[f]\big\vert_{Bit\ b} = '0' : "NOP" \rightarrow IR$ Überspringt die nächste Instruktion, wenn das Bit *b* des Registers *f* gleich ´0´ ist. Die Instruktion testet, ob das angegebene Bit im Register *f* gelöscht ist und übergeht ggf. die Ausführung des nachfolgenden Befehls, indem der Mikrocontrollerkern dazu veranlaßt wird, im nächsten Instruktionszyklus den dann anstehenden Befehl nicht auszuführen und statt dessen ein `NOP` einzuschieben.	´1001 1*bbb ffff ffff*´	-
`BTFSS`	*f,b*	*Bit Test f, Skip if Set* $D[f]\big\vert_{Bit\ b} = '1' : "NOP" \rightarrow IR$ Überspringt die nächste Instruktion, wenn das Bit *b* des Registers *f* gleich ´1´ ist. Die Instruktion testet, ob das angegebene Bit im Register *f* gesetzt ist und übergeht ggf. die Ausführung des nachfolgenden Befehls, indem der Mikrocontrollerkern dazu veranlaßt wird, im nächsten Instruktionszyklus den dann anstehenden Befehl nicht auszuführen und statt dessen ein `NOP` einzuschieben.	´1000 1*bbb ffff ffff*´	-
`BTF`	*f,b*	*Bit Toggle f* $\overline{D[f]\big\vert_{Bit\ b}} \rightarrow D[f]\big\vert_{Bit\ b}$ Invertiert das Bit *b* im Register *f*.	´0011 1*bbb ffff ffff*´	-

2.7.2.5 Sonstige Operationen

Befehl	Beschreibung		Statusbits
`NOP`	*No Operation* - Keine Operation.	´0000 0000 0000 0000´	-

Die beim originalen PIC17 Mikrocontroller zusätzlich vorhandenen Instruktionen `CLRWDT` und `SLEEP` zur Steuerung des Watch-Dog Timers und zum Umschalten des Prozessors in den „Sleep“-Modus werden vom Prozessormakro nicht unterstützt, da im Mikrocontrollerkern weder ein Watch-Dog Timer noch der Sleep-Modus implementiert ist. Wie alle anderen nicht belegten Opcodes, resultiert auch die Verwendung der Befehlswörter für `CLRWDT` und `SLEEP` in der Ausführung eines `NOP`.

2.8 Befehlsausführung

Der Mikrocontrollerkern implementiert ein zweistufiges sog. *Pipelining*, aufgeteilt in eine Stufe zum Laden eines Befehls aus dem Programmspeicher (engl. *Instruction Fetch*) und eine Stufe zum Ausführen des zuvor geladenen Befehls (engl. *Instruction Execute*). Durch die gleichzeitige Abarbeitung beider Stufen für zwei aufeinanderfolgende Befehle wird ein maximaler Durchsatz von einem Befehl pro Instruktionszyklus (entsprechend vier externer Taktzyklen) erreicht. Im folgenden wird das Prinzip und die Umsetzung der Pipeline des Mikrocontrollerkerns zusammen mit den sich daraus ergebenden zeitlichen Abläufen dargelegt. Insbesondere werden die daraus resultierenden speziellen Bedingungen und Besonderheiten im Zusammenhang mit dem Ablauf der Befehlsausführung, vorwiegend für spezielle Befehlsgruppen, dargestellt.

2.8.1 Allgemein

Pipelining ist eine Technik zur Beschleunigung der Befehlsabarbeitung und zur Erhöhung des Befehlsdurchsatzes, die die Basis der Architektur moderner, schneller (RISC-)Prozessoren bildet. Beim Pipelining wird durch Überlappen der Abarbeitung mehrerer Befehle eine Geschwindigkeitssteigerung erzielt. Die Idee ist vergleichbar mit dem Prinzip des Fließbands in der industriellen Fertigungstechnik, z.B. im Automobilbau. Der normale, sequentielle Befehlsablauf wird aufgeteilt in mehrere Stufen möglichst gleichen Umfangs, die aneinandergereiht eine Befehlsabarbeitungslinie ergeben. Jeder Befehl durchläuft nacheinander alle Stufen der Pipeline. Die eigentliche Beschleunigung wird erst erzielt, wenn sich, aufgrund der Feststellung, daß die einzelnen Stufen mehrerer Befehle unabhängig voneinander parallel abgearbeitet werden können, im normalen Programmablauf in jeder Stufe der Pipeline gleichzeitig ein Befehl aus einer Reihe aufeinanderfolgender Befehle befindet und dort jeweils der entsprechende Teil der Befehle abgearbeitet wird. Durch das Pipelining wird so der implizit in den Befehlen enthaltene Parallelismus ausgenützt. Im Takt der Pipeline (normalerweise ein Instruktionszyklus) betritt ein neuer Befehl die erste Stufe der Pipeline, die Befehle in der Pipeline rücken eine Stufe weiter und ein Befehl verläßt die Pipeline am Ende. Daraus folgt, daß pro Zyklus ein Befehl die Pipeline fertig abgearbeitet verläßt und sich so ein Durchsatz von einem Befehl pro Zyklus ergibt. Theoretisch wird die Ausführung der Befehle bei einer n-stufigen Pipeline im Vergleich zur sequentiellen Ausführung also um den Faktor n beschleunigt, da bei der sequentiellen Abarbeitung die n Stufen der einzelnen Befehle hintereinander abgearbeitet werden und sich dort ein Durchsatz von einem Befehl pro n Zyklen ergibt. Voraussetzung zum Erreichen dieser Beschleunigung ist aber, daß die Befehlsabarbeitung optimal in n Stufen genau gleicher Dauer eingeteilt wird, da der Takt der Pipeline durch die Dauer der langsamsten Stufe vorgegeben wird und deshalb bei nicht optimaler Unterteilung ggf. die Taktfrequenz im Vergleich zur sequentiellen Ausführung heruntergesetzt werden muß. Weiterhin ergibt sich eine optimale Ausnutzung der Pipeline auch nur, wenn diese stets in allen Stufen besetzt und vollkommen gefüllt ist. Aufgrund von diversen Effekten kann es aber dazu kommen, daß Unterbrechungen der Pipeline auftreten, z.B. bei Abhängigkeiten zwischen aufeinanderfolgenden Instruktionen, oder daß ein Teil der Pipeline komplett verworfen werden und die Pipeline neu aufgesetzt und geladen werden muß, z.B. bei Sprüngen. Speziell Sprünge unterbrechen den normalen Programmfluß und haben zur Folge,

daß alle bereits nachgerückten Befehle verworfen werden müssen, so daß der aus Sprüngen entstehende Nachteil um so größer ist, je tiefer die Pipeline ist, d.h. je mehr Stufen die Pipeline hat.

	Instruktionszyklus				
Stufe	**n-2**	**n-1**	**n**	**n+1**	**n+2**
Laden	**Befehl i-2**	**Befehl i-1**	**Befehl i**	**Befehl i+1**	**...**
Ausführen	**...**	**Befehl i-2**	**Befehl i-1**	**Befehl i**	**Befehl i+1**

	Instruktionszyklus				
	n-2	**n-1**	**n**	**n+1**	**n+2**
Befehl i-2	**Laden**	**Ausführen**			
Befehl i-1		**Laden**	**Ausführen**		
Befehl i			**Laden**	**Ausführen**	
Befehl i+1				**Laden**	**Ausführen**

Abbildung 2.12: Prinzip des 2-stufigen Pipelinings.

In Abbildung 2.12 wird das Prinzip des Pipelinings am Beispiel der 2-stufigen Pipeline des Mikrocontrollerkerns graphisch veranschaulicht. Beim Mikrocontrollerkern wird das, aufgrund der langsamen Zugriffszeiten des Programmspeicher-ROMs relativ zeitaufwendige Laden eines Befehls aus dem Programmspeicher mit der eigentlichen Ausführung des Befehls überlappt. Die beiden Pipeline- bzw. Befehlsstufen umfassen jeweils die Dauer eines Instruktionszyklus mit den vier Taktphasen Q1 bis Q4. Zu beachten ist, daß das Pipelining des Mikrocontrollerkerns mit der Unterteilung in zwei Stufen, die beide parallel in einem Instruktionszyklus abgearbeitet werden, nicht verwechselt wird mit der Unterteilung der Instruktionszyklen in vier Taktphasen, die nicht in irgendeiner Weise parallel ausgeführt werden, sondern nur dazu dienen das Timing von Steuersignalen leichter abzuleiten. Pro Instruktionszyklus wird im Mikrocontrollerkern ein Befehl fertig abgearbeitet, d.h. im Endeffekt wird im allgemeinen in einem Instruktionszyklus ein Befehl ausgeführt. Eine Ausnahme hiervon bilden Sprungbefehle und die Table-Operationen, mit denen Daten von und zum Programmspeicher transferiert werden. Aufgrund der besonderen Bedingungen und Funktionalitäten dieser Befehle kann bei ihrer Abarbeitung das normale Pipelining nicht aufrechterhalten werden und muß unterbrochen bzw. neu aufgesetzt werden, so daß i.a. zwei Zyklen für die Ausführung dieser Befehle zu veranschlagen sind.

Die genaue Umsetzung des 2-stufigen Pipelinings im Mikrocontrollerkern und das sich daraus ergebende Timing zeigt Abbildung 2.13 für den Fall der Ausführung normaler Befehle, die keine Eingriffe in diesen normalen Programmablauf zur Folge haben. Die Schnittstelle zwischen den beiden Stufen der Pipeline bildet beim Mikrocontrollerkern das Register IR, d.h. alle von der zweiten Stufe der Pipeline benötigten Daten werden dieser von der ersten Stufe über das Register IR zwischen den Instruktionszyklen übergeben. Im Detail passiert bei der Abarbeitung der einzelnen Stufen in einem Instruktionszyklus folgendes:

1. In der ersten Stufe der Pipeline wird während des Instruktionszyklus ein Befehlswort aus dem Programmspeicher in das Register IR geladen. Der Inhalt des Programmzählers (PC) gibt die Adresse des zu ladenden Befehls an und wird für den Lesezugriff auf den Instruktionsdatenbus gegeben. Die Kontrolleitungen der

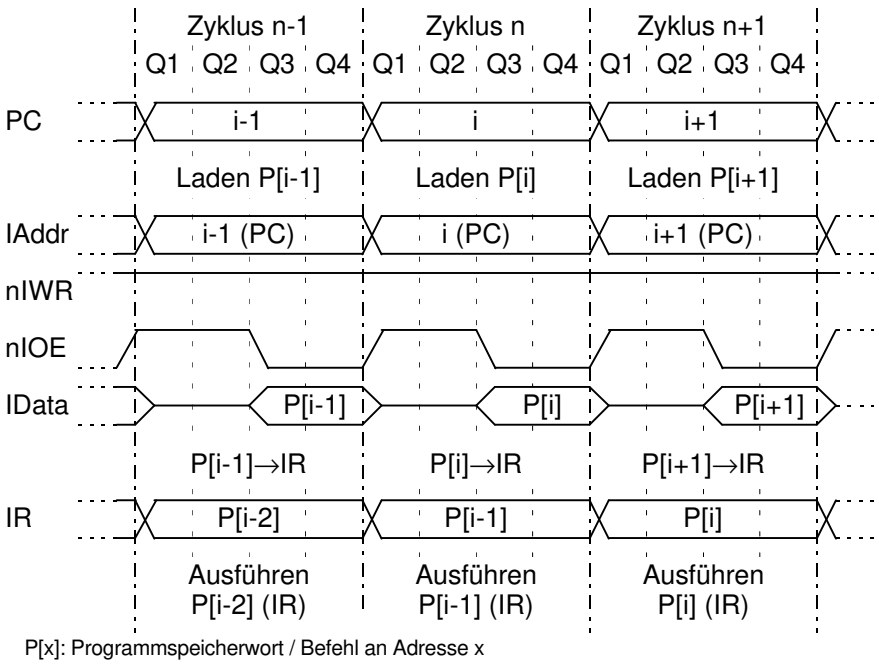


Abbildung 2.13: Normaler Befehlsablauf.

Schnittstelle zum Programmspeicher werden entsprechend dem Timing eines Lesezugriffs angesteuert (siehe auch Abschnitt 2.5.2). Das adressierte Programmspeicherwort wird daraufhin vom Programmspeicher auf den Instruktionsdatenbus gelegt und von dort vom Mikrocontrollerkern am Ende des Zyklus zur Übergabe an die nächste Stufe direkt in das Register IR übernommen. Mit Ausnahme von Sprungbefehlen, Interruptverzweigungen und sonstigen Operationen, die den Programmzähler direkt mit einem neuen Wert laden, wird schließlich in jedem Instruktionszyklus der Wert des Programmzählers um eins erhöht und am Ende des Zyklus in das Register geschrieben, so daß im nächsten Zyklus der nächste Befehl an der darauffolgenden Adresse gelesen wird. Der Inhalt des Registers IR und des Programmzählers ändert sich also zu Beginn jedes Instruktionszyklus und bleibt ansonsten für die Dauer des Zyklus konstant.

2. In der zweiten Stufe der Pipeline wird der im Register IR stehende Befehl während der Dauer des Instruktionszyklus dekodiert und ausgeführt. Im Normalfall läßt sich die Befehlsausführung folgendermaßen grob auf die vier Taktphasen des Zyklus aufteilen (Abbildung 2.14): In der ersten Taktphase Q1 wird der Inhalt des Registers IR dekodiert und die internen Steuersignale werden vom Instruktionsdekoder für die weitere Ausführung gemäß dem in IR enthaltenen Befehl geschaltet. Die Adressen des Quell- und Zielregisters, d.h. des Operanden- und Ergebnisregisters, werden auf die Datenadreßausgänge (*DAddrSrc* und *DAddrDest*) gegeben, wobei ggf. vorher noch die indirekte Adressierung aufgelöst werden muß. Der Adreßbus des RAM wird schließlich auf die Adresse des Quellregisters geschaltet (gleichzeitig wird in Q1 im übrigen auch ein Precharge des RAMs durchgeführt, siehe Abschnitt 2.5.3). In der zweiten Phase wird dann das RAM so angesteuert, daß der adressierte Wert des Operanden gelesen und vom RAM auf den Datenbus gegeben wird. Mit Ende der Phase Q2 übernimmt der Mikrocontrollerkern diesen Wert intern. Die eigentliche Durchführung der Operation, i.a. durch die ALU, findet in Phase Q3 unter Berücksichtigung des aus dem RAM gelesenen Operanden und ggf. eines zweiten impliziten Operanden, der direkt intern aus dem Kern entnommen wird, statt. Falls sich Quell- und Zielregister des Befehls unterscheiden, wird zu Beginn der Phase der

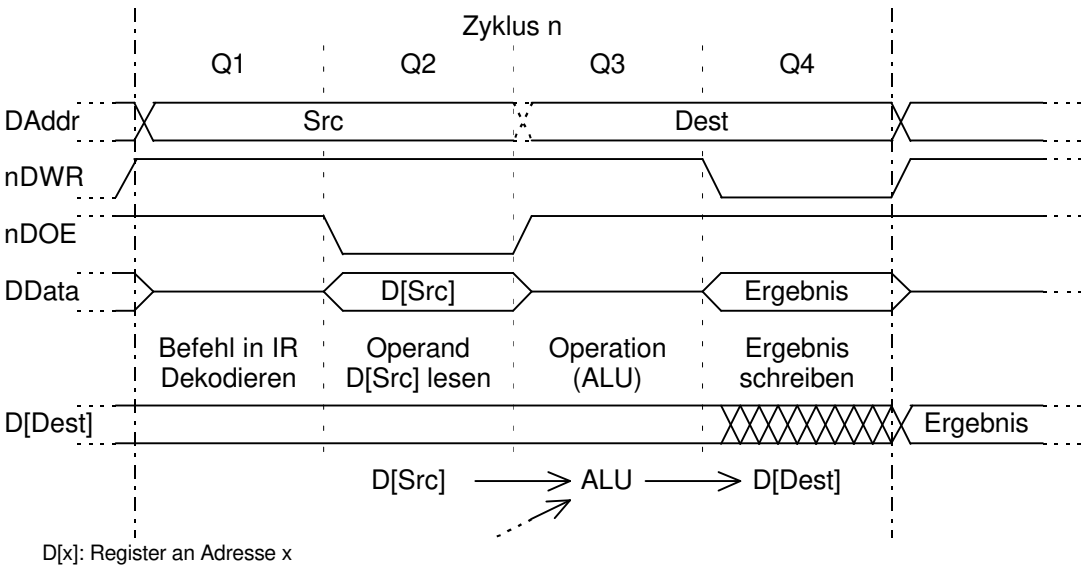


Abbildung 2.14: Allgemeine Befehlsausführung.

Adreßbus des RAM entsprechend umgeschaltet (wiederum Precharge des RAM in Q3). In der letzten Phase Q4 wird schließlich das Ergebnis der Operation durch entsprechendes Ansteuern des RAM und durch Ausgabe des Ergebnisses auf dem Datenausgangsbus in das Zielregister geschrieben. Ab Beginn des nächsten Instruktionszyklus steht das Ergebnis stabil im Zielregister und kann dort bereits von der nächsten Instruktion in der Phase Q2 gelesen werden.

Zu beachten ist, daß diese Einteilung einerseits aufgrund der Vielzahl der befehlsbedingten Konfigurationen sehr variieren kann und andererseits die Phasen aufgrund der unterschiedlichen Signallaufzeiten fließend ineinander übergehen. So kann auf der einen Seite z.B. die Ausführung der Operation in der ALU gegebenenfalls, je nach Zugriffszeit des RAM, schon vor Ende der Phase Q2 beginnen, sobald der vom RAM gelesene Wert stabil anliegt. Auf der anderen Seite wird z.B., je nach Befehl und Parametern, gegebenenfalls kein Ergebnis geschrieben (z.B. Vergleichsbefehle), kein externer Operand vom RAM gelesen (z.B. `CLRF`), weder ein externer Operand gelesen noch ein Ergebnis geschrieben (Sprünge), Operand und / oder Ergebnis internen Spezialregistern entnommen bzw. zugeführt, usw. An der Schnittstelle zum RAM ändert sich dabei aber nur die Ansteuerung des RAMs in Phase Q4. Wenn kein Wert in das RAM geschrieben wird, z.B. wenn kein Ergebnis anfällt oder das Ergebnis ausschließlich in ein internes Register geschrieben wird, dann wird in Phase Q4 auch kein Schreibzugriff auf das RAM durchgeführt und Zieladresse und Datenausgangsbus nehmen zufällige Werte an. Die Ansteuerung des RAMs in den anderen drei Phasen Q1 bis Q3 bleibt hingegen stets gleich. Wird kein Operand aus dem RAM benötigt, so wird trotzdem ein Lesezugriff an einer zufälligen Adresse durchgeführt, der gelesene (beliebige) Wert aber nicht vom Mikrocontrollerkern übernommen.

Alles in allem gilt der hier beschriebene Ablauf der überlappten Ausführung aufeinanderfolgender Instruktionen für den Großteil der vom Mikrocontrollerkern unterstützen Befehle, speziell für alle arithmetischen und logischen Operationen. Wie bereits besprochen, ergibt sich daraus im Normalfall für entsprechende Programmabschnitte eine Befehlsausführungsdauer von einem Instruktionszyklus pro Befehl, der Programmzähler wird in jedem Zyklus um eins erhöht und in jedem Zyklus schreiten die Lade- und Ausführungsstufen einen Befehl weiter. Eine Reihe von Besonderheiten und Spezialfällen im

Befehlssatz des Mikrocontrollerkerns macht es aber nötig, bei der Ausführung solcher spezieller Instruktionen diesen normalen Ablauf zu unterbrechen bzw. auszusetzen. Die ersten Ausnahmen bilden die sog. *Table*-Operationen `TABLRD` und `TABLWT` zum Transfer von Daten von und zum Programmspeicher, bei denen auf der Schnittstelle zum Programmspeicher ein Zyklus für den entsprechenden Zugriff auf die Daten eingefügt werden muß. Befehle zur Programmablaufsteuerung haben hingegen explizit die Aufgabe den normalen Programmablauf des Ladens und Ausführens aufeinanderfolgender Instruktionen abzubrechen und an einer anderen Stelle neu aufzusetzen. Schließlich kommt es bei der Auslösung eines Interrupts im Mikrocontrollerkern implizit zur Unterbrechung des normalen Programmflusses, da der Kern automatisch in eine entsprechende Interruptroutine zur Abarbeitung des Interrupts verzweigt. In den folgenden Abschnitten werden die Bedingungen bei der Abarbeitung von Table-Operationen, Sprungbefehlen (genauer von Befehlen zur Programmablaufsteuerung) und von diversen Spezialfällen der Kombination beider Operationen näher dargestellt. Die bei der Auslösung von Interrupts auftretenden vielfältigen Konfigurationen werden in Abschnitt 2.10 beschrieben.

2.8.2 Table-Operationen

Mit Hilfe der beiden Table-Operationen `TABLRD` und `TABLWT` kann aus einem Anwendungsprogramm heraus ein über das Register TBLPTR adressiertes Wort vom Programmspeicher in das interne Register TBLAT gelesen bzw. vom Register TBLAT in den Programmspeicher geschrieben werden. Da bei der Ausführung der Operationen, zusätzlich zur stets nötigen Dekodierung des Befehls im Register IR, bereits ein ganzer Instruktionszyklus für den Lese- bzw. Schreibzugriff auf das Programmspeicherwort (siehe Abschnitt 2.5.2) benötigt wird, können diese beiden Befehle, im Gegensatz zum oben beschriebenen allgemeinen Fall, nicht in einem einzigen Instruktionszyklus ausgeführt werden. Aus diesem Grund umfaßt die Ausführungsphase dieser Instruktionen insgesamt zwei Instruktionszyklen, d.h. das normale Pipelining wird für einen Zyklus unterbrochen, angehalten und nach Ausführung des zusätzlichen Zyklus fortgesetzt.

In Abbildung 2.15 und Abbildung 2.16 wird das genaue Timing bei der Ausführung der beiden Befehle dargestellt und aufgezeigt, wie die Unterbrechung der Pipeline und der zusätzliche Zyklus durchgeführt werden. Während der Ausführung des direkt vorangegangenen Befehls wird die Table-Operation normal geladen, das entsprechende Befehlswort direkt im Register IR abgelegt und der Programmzähler inkrementiert. Im darauffolgenden Instruktionszyklus, dem ersten Zyklus der Ausführung der Table-Operation, wird die Instruktion zuallererst normal dekodiert. Die Vorarbeiten für den Datentransfer werden ausgeführt, indem Daten zwischen dem angegebenen Register im Datenspeicher und der angegebenen Hälfte des Table-Latches ausgetauscht werden. Gleichzeitig wird die auf die Table-Operation folgende Instruktion normal von der durch den Programmzähler angegebenen Adresse in das Register IR geladen. Zusätzlich wird aber jetzt im ersten Ausführungszyklus der Table-Operation der Programmzähler angehalten und das automatische Inkrementieren unterbunden.

Im zweiten Zyklus der Ausführung der Table-Operation wird dann, unabhängig vom Inhalt des Registers IR, der eigentliche Datentransfer zwischen Programmspeicher und Table-Latch durchgeführt. Dazu werden die Steuerleitungen der Programmspeicherschnittstelle entsprechend einem Lese- bzw. Schreibzyklus angesteuert und entweder die zu transferierenden Daten vom Datenbus in das Table-Latch übernommen oder der Inhalt des Table-Latch auf den Datenbus gelegt und von dort in den Programmspeicher geschrieben. In beiden Fällen wird für den Zugriff nicht der Inhalt des Programmzählers, sondern der Inhalt des Registers TBLPTR auf den Adreßbus gegeben. Je nach Befehlsparametern wird das Register TBLPTR um eins erhöht und am Ende des Zyklus mit dem neuen Wert geladen. Da die Programmspeicherschnittstelle mit dem Datentransfer belegt ist, wird in dem zweiten

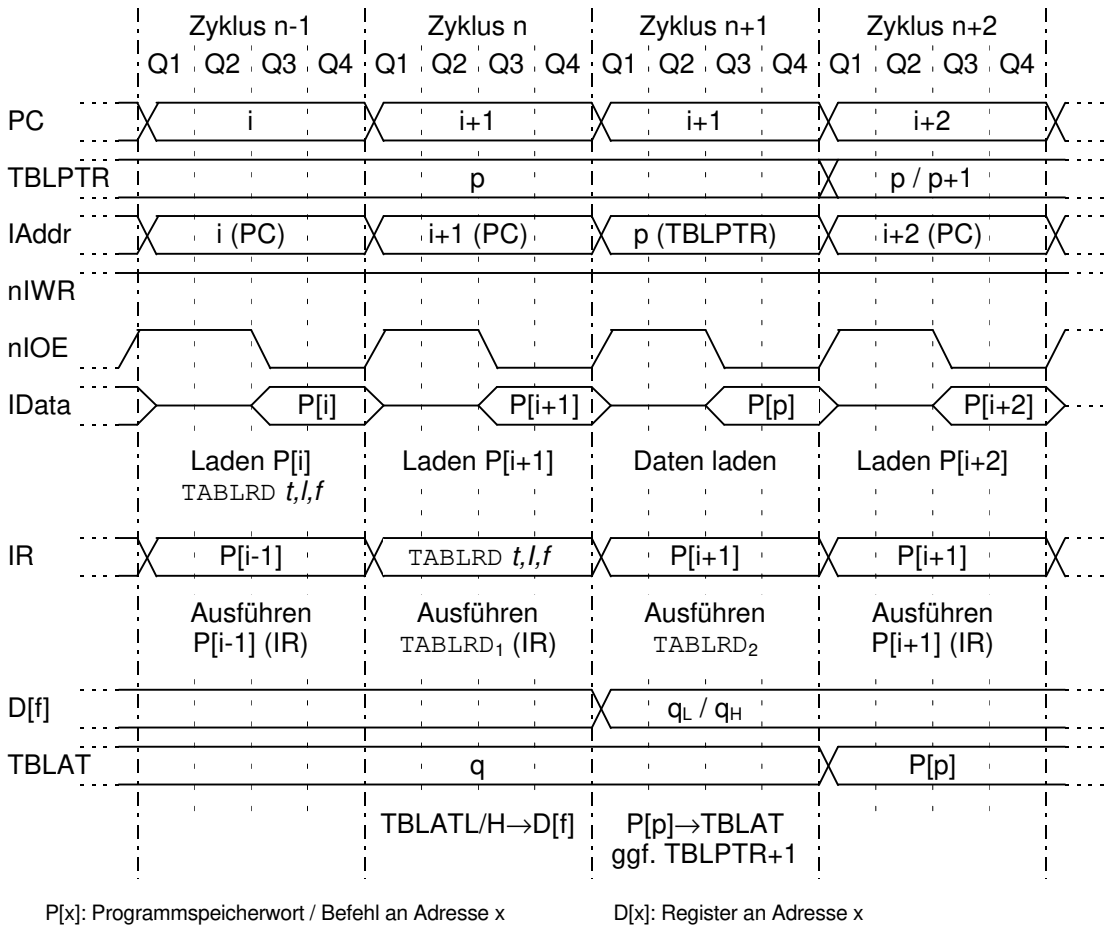


Abbildung 2.15: Ausführung des Befehls `TABLRD`.

Ausführungszyklus kein Befehl geladen und das Register IR behält seinen Wert. Das Register IR enthält also weiterhin den nachfolgenden Befehl, der direkt im Anschluß ausgeführt werden muß. Der Programmzähler wird schließlich in diesem Zyklus wieder normal inkrementiert. Damit ist die Ausführung der Table-Operation beendet und der Mikrocontrollerkern befindet sich in einem Zustand, in dem die Programmausführung und das Pipelining im nächsten Instruktionszyklus wieder normal fortgefahren werden können. Im nächsten Zyklus fährt also der Programmablauf fort, der weiterhin im Register IR enthaltene direkt nachfolgende Befehl wird dekodiert und ausgeführt, der Inhalt des Programmzählers wird wieder auf den Adreßbus gegeben, der übernächste Befehl wird gleichzeitig geladen, usw.

Wie leicht zu erkennen ist, wird bei der Ausführung der Table-Operationen das Pipelining unterbrochen, indem einen Zyklus lang einerseits der Programmzähler angehalten wird und andererseits das Register IR seinen Wert behält und nicht neu geladen wird. Die Ausführung des nächsten Befehls in IR und das gleichzeitige Laden des übernächsten Befehls wird also um einen Zyklus verschoben und statt dessen der zusätzliche Zyklus der Table-Operation durchgeführt. Anstatt das Erhöhen des Programmzählers im ersten Ausführungszyklus der Table-Operation zu unterbrechen und ihn im zweiten Zyklus normal zu inkrementieren, wäre es ohne Auswirkungen auf den korrekten Ablauf der Ausführung prinzipiell auch möglich den Programmzähler im ersten Ausführungszyklus normal zu inkrementieren und im zweiten Zyklus anzuhalten. Aus Gründen der Interruptverarbeitung scheidet die zweite Version allerdings aus, und die Ausführung muß so ablaufen, wie sie hier dargestellt wurde.

Wird nämlich der Programmzähler schon im ersten Ausführungszyklus inkrementiert und im zweiten Zyklus tritt ein Interrupt auf, so würde bei der dann durchgeführten Verzweigung in die Interruptroutine fälschlicherweise die Adresse der übernächsten Instruktion, gemäß dem

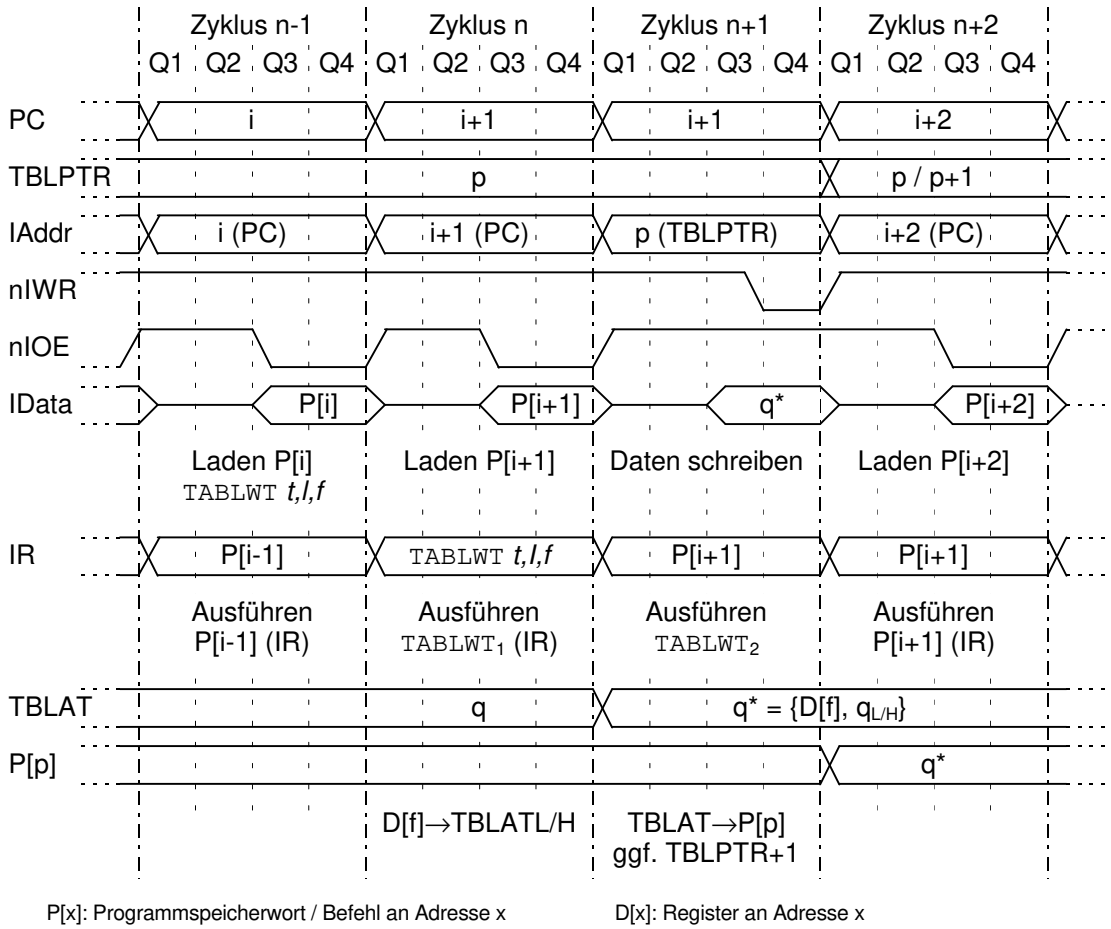


Abbildung 2.16: Ausführung des Befehls `TABLWT`.

aktuellen Inhalt des Programmzählers, als Rücksprungadresse auf dem Stack abgelegt. Damit der Programmzähler beim Sprung in eine Interruptroutine während der Ausführung der Table-Operation stets die Adresse der direkt nachfolgenden Instruktion enthält, die dann auf dem Stack abgelegt wird, darf er erst ganz zum Schluß inkrementiert werden. So wird sichergestellt, daß nach Abarbeitung der Interruptroutine die Programmausführung an die nachfolgende Instruktion zurückkehrt und dort korrekt fortgesetzt wird. Andernfalls würde die Programmausführung mit der übernächsten Instruktion fortgesetzt und deshalb beim Auftreten eines Interrupts im zweiten Zyklus im Endeffekt durch den Fehler beim Interrupthandling die nachfolgende Instruktion übersprungen bzw. ausgelassen (da beim Auslösen eines Interrupts die gerade laufende Instruktion fertig ausgeführt und nicht unterbrochen wird, wird bei Auftreten eines Interrupt während der Ausführung einer Table-Operation die Operation bereits vor dem Sprung in die Interruptroutine vollständig fertig abgearbeitet, und das Programm muß nach dem Interrupt mit der direkt nachfolgenden Instruktion fortgefahren werden; siehe Abschnitt 2.10 über Interrupts).

2.8.3 Befehle zur Programmablaufsteuerung

Mit den Befehlen des Mikrocontrollerkerns zur Programmablaufsteuerung wird dem Anwendungsprogrammierer eine Möglichkeit gegeben, den normalen sequentiellen Programmablauf, gegebenenfalls in Abhängigkeit von diversen, zur Laufzeit ausgewerteten Bedingungen, an einer Stelle zu unterbrechen und an einer anderen Stelle im Programm fortzusetzen. Daraus folgt, daß die Befehle vom Prinzip und von der Aufgabe her schon die normale Programmausführung und das Pipelining unterbrechen müssen. Allen Befehlen ist gemeinsam, daß sie in enger Interaktion mit dem Programmzähler und den sonstigen Teilen des

PC-Moduls rund um den Programmzähler stehen. Um die gewünschten Sprünge o.ä. zu verwirklichen, wird i.a. der Programmzähler von den Instruktionen mit einem festgelegten Wert geladen, anstatt ihn am Ende der Instruktion einfach nur um eins zu erhöhen. Je nach Instruktion werden im Laufe der Abarbeitung des Befehls zusätzlich ggf. auch das Register PCLATH und der Stack ausgewertet bzw. neu geschrieben. In Abhängigkeit von den bei der Abarbeitung der einzelnen Befehle auftretenden Zuständen und Abläufen, lassen sich die Befehle zur Programmablaufsteuerung in drei prinzipielle Klassen einteilen, die im folgenden näher beschrieben werden. Im Prinzip müssen auch Interrupts und der zu ihrer Abarbeitung notwendige Sprung in eine Interruptroutine zu den Sprungbefehlen gezählt werden. Da Interrupts aber nicht direkt von einzelnen Befehlen des Programms ausgelöst werden und, parallel zur normalen Programmausführung, direkt in den Ablauf der eigentlichen Befehle eingreifen, werden die bei der Abarbeitung von Interrupanforderungen auftretenden Abläufe in einem eigenen Abschnitt beschrieben (Abschnitt 2.10).

2.8.3.1 Skips

Es existieren eine Reihe von Befehlen, bei denen, in Abhängigkeit von bei der Ausführung des Befehls auftretenden Bedingungen und Zuständen, ein sog. *Skip* ausgeführt werden kann. Ein Skip bedeutet, daß bei Eintreten der entsprechenden Bedingung der nächste Befehl, der direkt auf den Skip-Befehl folgt, nicht ausgeführt, d.h. übersprungen wird. Tritt bei der Ausführung des Skip-Befehls die entsprechende Bedingung nicht ein, so wird der nächste Befehl normal ausgeführt und der normale, allgemeine Programmablauf mit Pipelining und PC-Inkrement ändert sich nicht.

Die Verwirklichung der Ausführung eines Skips, bei Eintreten der entsprechenden Bedingung, wird in Abbildung 2.17 exemplarisch dargestellt. Da durch das Überlappen der Ausführung des Skip-Befehls und des Ladens des nächsten Befehls bereits der nachfolgende Befehl geladen und der Programmzähler erhöht ist, wenn die Bedingung überhaupt erst ausgewertet wird, muß im Prinzip das normale Pipelining nicht direkt unterbrochen und der Programmzähler nicht mit einem Wert geladen werden. Bei Ausführung eines Skips läuft die Pipeline normal weiter, d.h. in jedem Zyklus wird der Programmzähler normal inkrementiert und die nächste Instruktion in der Befehlssequenz geladen. Der eigentliche Skip wird verwirklicht, indem die Ausführung der auf den Skip-Befehl folgenden Instruktion im darauffolgenden Zyklus unterbunden wird, d.h. es wird statt dessen ein Leerzyklus in der Ausführungsstufe der Pipeline durchgeführt. Beim Mikrocontrollerkern wird dies dadurch

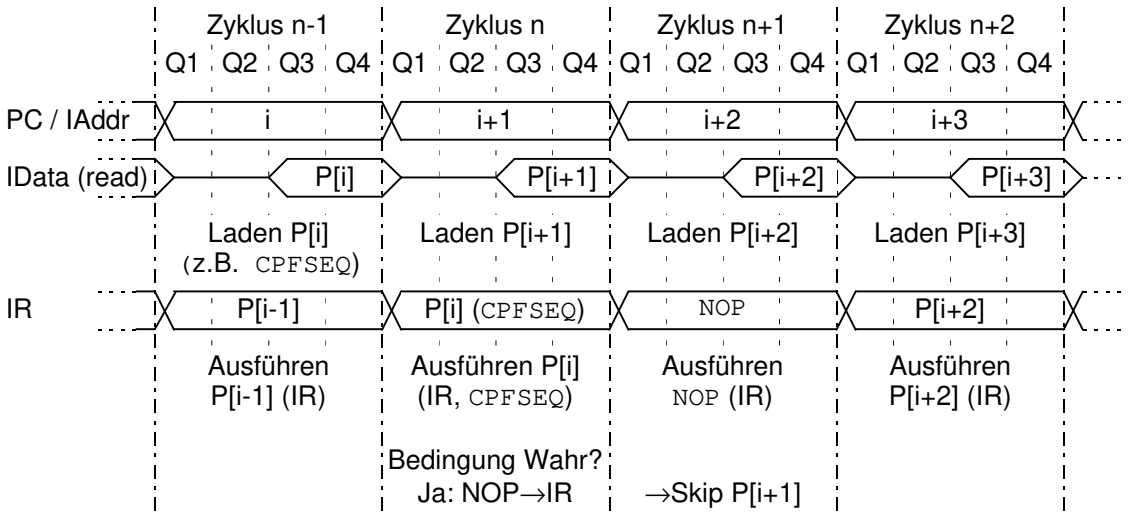


Abbildung 2.17: Ausführung eines Skips am Beispiel des Befehls `CPFSEQ`.

erreicht, daß, bei Eintreten der Skip-Bedingung während der Ausführung des Skip-Befehls, der gleichzeitig bereits geladene nachfolgende Befehl am Ende des entsprechenden Zyklus nicht in das Register IR übernommen und dieses statt dessen mit dem Befehl `NOP` geladen (d.h. auf null gesetzt) wird. Im nächsten Zyklus wird so statt des nachfolgenden Befehls ein `NOP` ausgeführt, das keinerlei Operation durchführt. Speziell im Hinblick auf die für die Ausführung des Skips zu veranschlagenden Taktzyklen bedeutet dies, daß im Endeffekt der nachfolgende Befehl stets ausgeführt wird, ggf. aber eben als `NOP` (bzw. bei einem Skip wird der nachfolgende Befehl übersprungen und der Skip selbst benötigt zwei Taktzyklen zur Ausführung).

Alternativ wäre es auch möglich gewesen, den nachfolgenden Befehl normal in das Register IR zu laden und dann der Ausführungsstufe der Pipeline durch ein Flag im nächsten Zyklus mitzuteilen, den Befehl in IR nicht auszuführen. Die Verwirklichung von Skips durch Ausführen eines NOP hat aber den Vorteil, daß dadurch eine klare Trennung und Unabhängigkeit zwischen den beiden Zyklen besteht und, speziell im Hinblick auf das Auftreten von Interrupts, so, ohne spezielle Zustände, im Prinzip ein Programm ausgeführt wird, bei dem die auf den Skip-Befehl folgende Instruktion im Programm selbst eben ein `NOP` ist.

2.8.3.2 Sprünge und Unterprogrammaufrufe

Bei Sprungbefehlen und Unterprogrammaufrufen wird die normale Befehlssequenz unterbrochen und die Programmausführung verzweigt zu einer anderen Stelle im Programm. Außer den vom Mikrocontrollerkern speziell für diese Zwecke zur Verfügung gestellten Instruktionen, bei denen der Programmzähler implizit modifiziert und geladen wird, können Sprünge aber auch mit beliebigen Instruktionen durch einen expliziten Schreibzugriff auf das im Datenspeicherbereich eingeblendete Register PCL des Programmzählers durchgeführt werden (siehe Abschnitt 2.6.3). Im Endeffekt unterscheiden sich die durch Schreiben des Registers PCL durchgeführten Sprünge in ihrem Ablauf nicht von der Ausführung spezieller Sprungbefehle und werden deshalb hier mit berücksichtigt. Bei der Ausführung aller dieser Instruktionen wird zur Durchführung des Sprungs bzw. des Routinenaufrufs das normale Pipelining unterbrochen, der Programmzähler mit der Sprungzieladresse geladen und der Programmablauf dort neu aufgesetzt. Je nach Art des Sprungs wird entweder eine 13 Bit breite Zieladresse direkt als Konstante in der Instruktion selbst kodiert (`CALL`, `GOTO`; die oberen 8 Bit der kombinierten Zieladresse werden zusätzlich im Register PCLATH gesichert) oder es wird eine 16 Bit breite Zieladresse aus der Kombination des Inhalts des Registers PCLATH mit einem von der Instruktion selbst gelieferten 8 Bit breiten Wert gebildet (`LCALL`, Schreibzugriffe auf PCL). Bei Unterprogrammaufrufen (`CALL`, `LCALL`) wird vor dem Laden der Zieladresse in den Programmzähler die Adresse der auf den Unterprogrammaufruf folgenden Instruktion als Zieladresse für einen späteren Rücksprung aus dem Programmzähler (s.u.) auf den Stack geschrieben.

Abbildung 2.18 zeigt exemplarisch die Ausführung und Umsetzung der verschiedenen Funktionalitäten von Sprüngen und Unterprogrammaufrufen. Die Abarbeitung von Sprungbefehlen läuft folgendermaßen ab:

- Während die vorhergehende Instruktion ausgeführt wird, zeigt der Programmzähler auf den Sprungbefehl, der gleichzeitig geladen wird.
- Der Programmzähler wird um eins erhöht, und im nächsten Zyklus wird parallel der Sprungbefehl ausgeführt und der nachfolgende Befehl noch normal geladen.
- Zur Ausführung des Sprungbefehls wird ggf. die Sprungzieladresse berechnet und zusammengesetzt. Bei Unterprogrammaufrufen wird der aktuelle Inhalt des Programmzählers, der ja bereits auf die nachfolgende Instruktion zeigt, als Rücksprungadresse auf den Stack geschrieben. Anstatt den Programmzähler am Ende des Zyklus normal zu inkrementieren, wird dieser mit der Sprungzieladresse geladen.

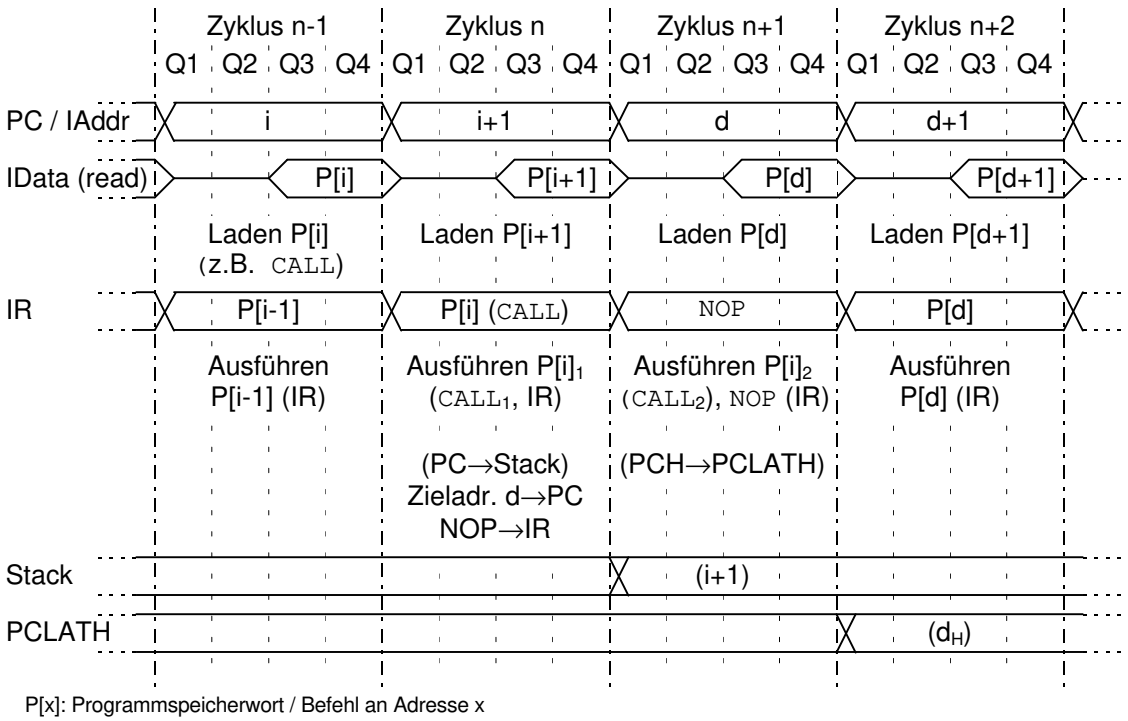


Abbildung 2.18: Ausführung eines Sprungs bzw. eines Routinenaufrufs am Beispiel des Befehls `CALL`.

- Die während der Ausführung des Sprungbefehls parallel geladene, nachfolgende Instruktion wird am Ende des Zyklus nicht in das Register IR übernommen, da sie nicht mehr ausgeführt werden soll. Analog zur Ausführung eines Skips wird statt dessen ein `NOP` in das Register IR geschrieben.
- Im nächsten Zyklus, dem zweiten Zyklus des Sprungbefehls, wird dann die Instruktion an der durch den neuen Inhalt des Programmzählers angegebenen Zieladresse geladen. Während die nächste auszuführende Instruktion erst geladen werden muß, kann prinzipiell keine Operation durchgeführt werden, und es wird deshalb das in IR enthaltene `NOP` ausgeführt. Je nach Art des Sprungbefehls wird in diesem zweiten Zyklus ggf. noch die obere Hälfte des neuen Programmzählerinhalts in das Register PCLATH gesichert.
- Im darauffolgenden Zyklus ist die Ausführung des Sprungbefehls schließlich beendet, und der Programmablauf wird mit der Ausführung der Instruktion an der Zieladresse normal fortgesetzt.

Alles in allem benötigen Sprungbefehle also im Prinzip zwei Instruktionszyklen zu ihrer Ausführung, da nach dem Laden des Programmzählers in einem zweiten Zyklus die Instruktion an der Zieladresse erst geladen werden muß, bevor das Pipelining und der normale Programmablauf mit ihrer Ausführung fortgesetzt werden können.

2.8.3.3 Rücksprünge

Rücksprungbefehle bilden das Pendant zu Unterprogrammaufrufen, um nach Ende der Abarbeitung der Unterroutine zum Aufrufer zurückzukehren und die Ausführung dort fortzusetzen, wo die Unterroutine aufgerufen wurde. Im Prinzip verhalten sich Rücksprünge ganz analog zu den oben behandelten normale Sprungbefehlen, mit der Ausnahme, daß die Zieladresse des Sprungs, d.h. die Rücksprungadresse, vom Stack geholt wird. Zusätzlich zu

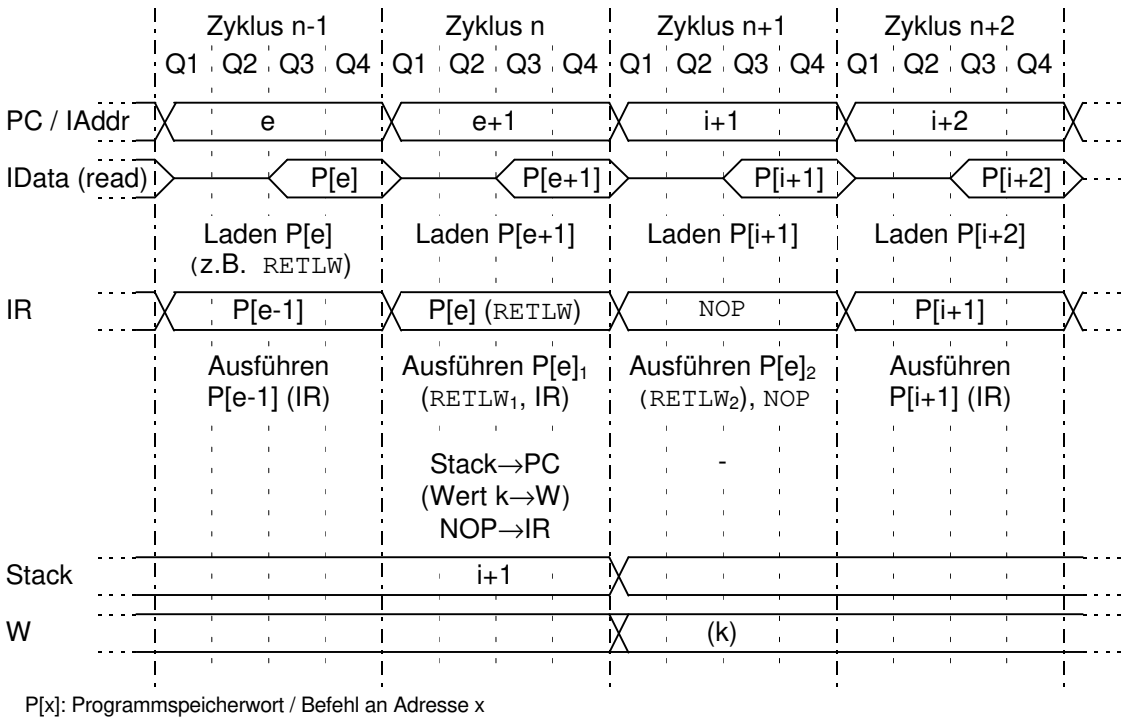


Abbildung 2.19: Ausführung einer Rückkehr aus einer Routine am Beispiel des Befehls `RETLW`.

dem normalen Rücksprung (`RETURN`) existieren noch zwei spezielle Rücksprungbefehle, mit denen einerseits eine in der Instruktion kodierte Konstante als Ergebnis der Unterroutine in das Register W geschrieben werden kann (`RETLW`) oder andererseits, bei einem Rücksprung aus einer Interruptroutine, das Flag GLINTD zurückgesetzt werden kann, um so die während der Abarbeitung der Interruptroutine gesperrten Interrupts wieder freizugeben (`RETFIE`).

In Abbildung 2.19 ist das Timing für einen Rücksprung exemplarisch dargestellt. Wie bereits erwähnt, wird ein Rücksprung prinzipiell genau gleich wie ein normaler Sprung ausgeführt (s.o.), d.h. es gilt das dort gesagte analog.

2.8.4 Spezialfälle

Ein spezieller Fall der Ausführung eines Befehls leitet sich aus der Kombination der Table-Operation `TABLRD` mit dem Register PCL als Parameter ab. In Abschnitt 2.8.2 wurde die Ausführung der Instruktion `TABLRD` ausführlich dargestellt. Im Zuge der Ausführung dieses Befehls wird eine Hälfte des Table-Latches in ein in der Instruktion angegebenes Register geschrieben. Wird nun das Register PCL als Ziel der Schreiboperation angegeben, so wird, wie bereits mehrfach dargelegt, durch die Table-Operation gleichzeitig auch ein Sprung verwirklicht, wie er in Abschnitt 2.8.3.2 dargestellt wurde. Bei der Ausführung einer `TABLRD`-Instruktion mit Register PCL wird also einerseits ein Datenwort aus dem durch den Inhalt des Register TBLPTR adressierten Programmspeicher in das Table-Latch geladen und andererseits, durch Laden des Programmzählers mit einer aus PCLATH und Table-Latch gebildeten Zieladresse, ein Sprung an eine andere Programmstelle durchgeführt, an der der Programmablauf nach Ausführung der kompletten Table-Operation fortgesetzt wird.

Das sich aus der Kombination der Ausführung einer Table-Operation und eines Sprungs ableitende Timing für die Ausführung dieser speziellen Instruktion zeigt Abbildung 2.20. Da sich aus der Kombination ergibt, daß jeweils ein zusätzlicher Zyklus für den Programmspeicherzugriff und für das Laden der Instruktion an der Zieladresse benötigt wird, bedeutet dies, daß für die Ausführung des Befehls insgesamt drei Instruktionszyklen benötigt

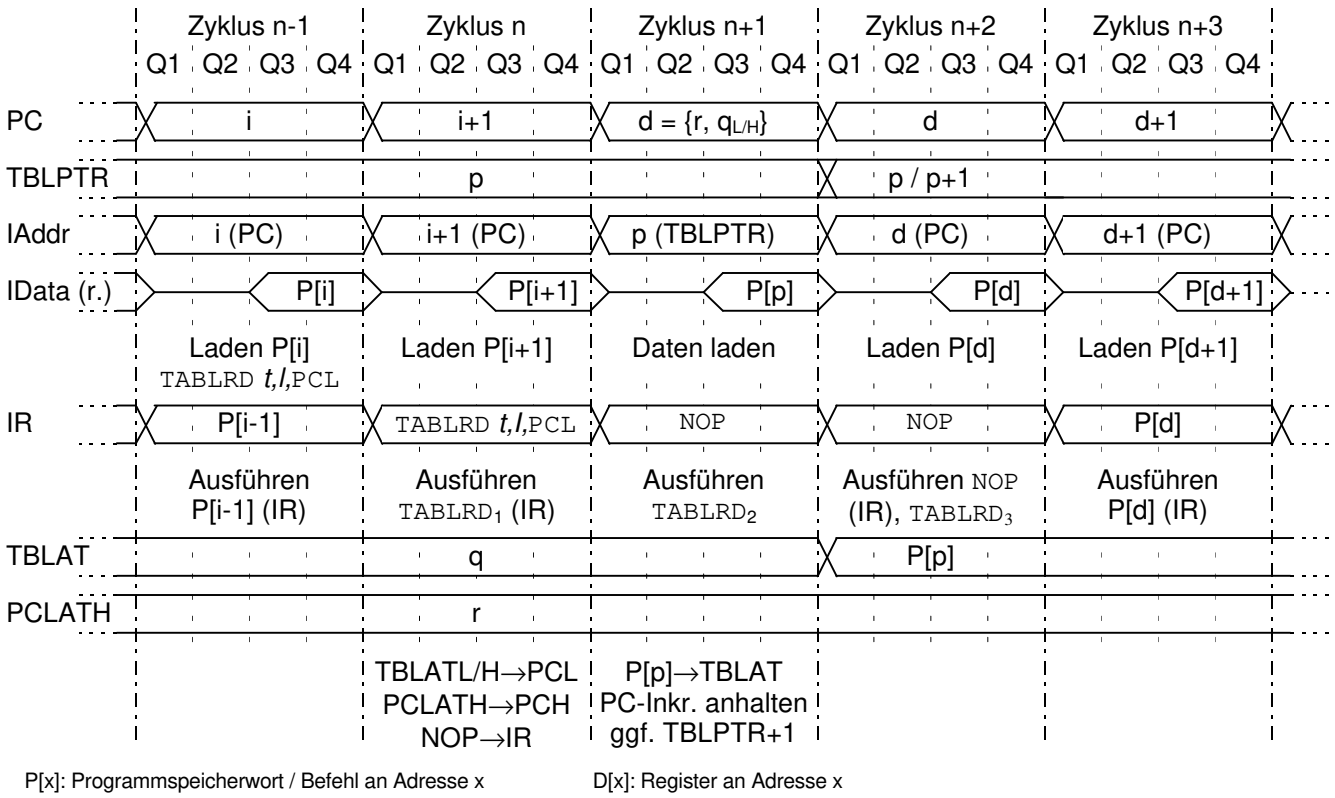


Abbildung 2.20: Ausführung des Befehls `TABLRD` mit Register PCL.

werden. Für die Ausführung der beiden zusätzlichen Teilzyklen gilt im Prinzip das in den beiden relevanten vorangegangenen Abschnitten dargelegte Verhalten. Dieser Spezialfall ist die einzige Instruktion, deren Ausführung mehr als zwei Zyklen umfaßt.

Im ersten Ausführungszyklus wird die im Register IR enthaltene Spezialinstruktion dekodiert und analog zu einem normalen `TABLRD` die selektierte Hälfte des Table-Latches in das angegebene Datenregister geschrieben. Da es sich in diesem Fall aber um einen Schreibzugriff auf das Datenregister PCL und damit den Programmzähler handelt, hat dies zur Folge, daß der Programmzähler mit der 16 Bit breiten Zieladresse, gebildet aus der Table-Latch-Hälfte und dem Register PCLATH, geladen wird, anstatt normal inkrementiert zu werden. Zusätzlich wird, analog zur Ausführung eines normalen Sprungs, statt des während dieses ersten Ausführungszyklus geladenen nachfolgenden Befehls die Instruktion `NOP` in das Register IR geladen. Der zweite Ausführungszyklus entspricht dem zweiten Zyklus der entsprechenden normalen Table-Operation (Abschnitt 2.8.2): der Inhalt des Registers TBLPTR wird als Adresse für den Lesezugriff auf den Instruktionsadreßbus gegeben, das so adressierte Programmspeicherwort geladen und im Register TBLAT abgelegt, der Inhalt des Register IR ignoriert, d.h. nicht dekodiert und ausgeführt, und ggf. das Register TBLPTR um eins erhöht. Allerdings darf in diesem Fall, im Gegensatz zur normalen Table-Operation, der Programmzähler am Ende des zweiten Zyklus nicht inkrementiert werden und muß seinen Inhalt, d.h. die Sprungzieladresse, weiterhin beibehalten! Im dritten Ausführungszyklus wird schließlich der zweite Zyklus des Sprungs ausgeführt (Abschnitt 2.8.3.2), d.h. die Zielinstruktion an der durch den neuen Inhalt des Programmzählers angegebenen Zieladresse wird geladen, während zeitgleich, zur Überbrückung, der im Register IR enthaltene `NOP`-Befehl ausgeführt wird. Somit kann schließlich die so neu aufgesetzte Pipeline im nächsten Instruktionszyklus an der neuen Programmstelle normal fortgesetzt werden.

Die Reihenfolge der Zyklen zwei und drei, d.h. zuerst der Programmspeicherzugriff, dann das Laden der Zielinstruktion und nicht umgekehrt, wurde so festgelegt, da die für den

Programmspeicherzugriffszyklus notwendigen, zusätzlichen internen Status- und Zustandsinformationen somit nur einen Zyklus lang gespeichert werden müssen. Für den zusätzlichen Ladezyklus an der Zieladresse werden hingegen keine intern gespeicherten Zustandsinformationen benötigt, da in diesem Instruktionszyklus im Prinzip ganz normal der Befehl im Register IR ausgeführt und der durch den Programmzählerinhalt adressierte Befehl geladen wird.

2.9 Stack

Der Mikrocontrollerkern enthält intern einen an das PC-Modul angekoppelten Stack zur Speicherung von Rücksprung- bzw. Aufrufadressen bei Unterroutinenaufrufen. Entsprechend der Breite der Programmspeicheradressen werden auf dem Stack 16 Bit breite Adreßworte abgelegt. Beim originalen PIC17 Mikrocontroller kann der Stack 16 Rücksprungadressen aufnehmen, hat also insgesamt dort eine Größe von 16 x 16 Bit. Bei dem in dieser Arbeit entwickelten Mikrocontrollerkern kann die gewünschte Tiefe des Stacks über einen VHDL-Generic bei der Synthese beliebig festgelegt werden. In Abhängigkeit von der maximal benötigten Unterprogrammverschachtelungstiefe kann so, je nach Applikation, die tatsächlich realisierte Größe des Stacks (*n* x 16 Bit) flexibel angepaßt werden.

Der Stack des Mikrocontrollerkerns wird implizit im Befehlssatz des Prozessor verwendet. Auf der einen Seite wird bei Unterprogrammaufrufen (Befehle `CALL` und `LCALL`) bzw. bei Verzweigungen in eine Interruptroutine (Auslösen einer Interruptanforderung) der aktuelle Inhalt des Programmzählers auf der Spitze des Stacks abgelegt. Auf der anderen Seite wird bei Rücksprüngen aus so aufgerufenen Unterroutinen (Befehle `RETURN`, `RETLW` bzw. `RETFIE`) die Aufrufadresse von der Spitze des Stacks geholt und in den Programmzähler geschrieben. Andersweitig kann der Stack nicht angesprochen werden. Insbesondere ist es nicht möglich, explizit durch sonstige Befehle auf Teile des Stacks zuzugreifen, um z.B., wie oft üblich, Parameter für Unterprogrammaufrufe über den Stack zu übergeben.

Um die aufgrund der begrenzten Stacktiefe möglichen Stacküberläufe erkennen bzw. gar verhindern zu können, besitzt der Mikrocontrollerkern einen Mechanismus, der einerseits bei Erreichen der Stackobergrenze rechtzeitig vor einem drohenden Überlauf warnt und andererseits einen bereits erfolgten Überlauf dauerhaft anzeigt. Dieser Mechanismus wird durch das Bit STKAVL im Register CPUSTA (Bit 5, Adresse 06h) verwirklicht. Im normalen Betrieb, d.h. wenn weder ein Überlauf droht noch der Stack bereits übergelaufen ist, beinhaltet das Bit den Wert ´1´, um anzuzeigen, daß der Stack uneingeschränkt zur Verfügung steht. Wird durch einen Unterroutinenaufruf (entweder durch einen entsprechenden Befehl oder durch Auslösen eines Interrupts) der letzte Platz des Stacks mit einer Aufrufadresse belegt, so wird im Zuge dieser Aktion das Bit STKAVL auf ´0´ gesetzt. Dadurch wird signalisiert, daß der Stack voll ist und vorläufig bis zur Rückkehr aus der momentanen Routine keine weiteren Unterroutinenaufrufe erfolgen dürfen, wenn ein Überlauf vermieden werden soll. Am Ende der Unterroutine wird dann durch den entsprechenden Rücksprungbefehl die oberste Rücksprungadresse wieder vom Stack genommen und, da daraufhin wieder ein Platz auf dem Stack zur Verfügung steht, das STKAVL-Bit wieder auf ´1´ gesetzt. Erfolgt allerdings ein weiterer Unterroutinenaufruf während der Stack voll ist, d.h. während das Bit STKAVL auf ´0´ gesetzt ist, dann kommt es zu einem Stacküberlauf. Die Rücksprungadresse des entsprechenden Aufrufs wird an die unterste Position im Stack geschrieben und überschreibt die dort stehende Rücksprungadresse des ersten Aufrufs unwiderruflich. Fortlaufende Routinenaufrufe überschreiben dann weitere Rücksprungadressen, d.h. der Stack wird nach einem Überlauf wieder von vorne gefüllt. Gleichzeitig wird bei einem Überlauf des Stacks das STKAVL-Bit permanent auf ´0´ gesetzt, d.h. das Bit wird auch bei Rückkehr aus den überlaufenden Routinen zum normalen Betrieb nicht wieder auf ´1´ zurückgesetzt und kann nur durch einen Reset wieder in den normalen Zustand versetzt werden.

Durch Abfragen des STKAVL-Bits können so Stacküberläufe erkannt bzw. verhindert werden. Um drohende Überläufe des Stacks rechtzeitig zu erkennen und ggf. zu verhindern, kann das Bit zu Beginn jeder Unter- und Interruptroutine auf ´0´ getestet werden. Unter der Voraussetzung, daß bisher kein Überlauf aufgetreten ist, zeigt ein Wert ´0´ des STKAVL-Bits an, daß der Stack mit dem Aufruf dieser Routine voll ist und daß innerhalb dieser Routine keine weiteren Unterroutinen aufgerufen werden dürfen, um einen Überlauf zu vermeiden. Das bedeutet auch, daß alle Interrupts innerhalb der Routine gesperrt werden müssen.

Zu beachten ist, daß, entsprechend einem Überlauf am oberen Ende des Stacks mit Fortsetzung am unteren Ende des Stacks, auch ein „Unterlauf“ am unteren Ende des Stacks auftreten kann. Einerseits tritt dieser Fall dann ein, wenn nach einem bereits erfolgten Überlauf bei der Rückkehr aus den entsprechenden Unterroutinen die Rücksprungadressen vom Stack geholt werden und beim Leeren des Stacks die untere Grenze wieder erreicht wird. Nach der Entnahme des untersten Werts im Stack wird der Stackzeiger wieder auf die obere Grenze gesetzt, d.h. als nächstes wird wieder der oberste Wert vom Ende des Stacks geholt. Im Prinzip entspricht der Stack also einem Ringpuffer, der nach dem LIFO-Prinzip arbeitet (engl. *Last In First Out*).

Ein Unterlauf kann allerdings auch auftreten, wenn der Stack leer ist und dann durch Ausführung eines Rücksprungbefehls ein Wert vom Stack geholt wird. Da sich der Stack durch diesen Unterlauf daraufhin an der oberen Grenze befindet und deshalb intern wie ein voller Stack gewertet wird, wird im Zuge der Operation auch das STKAVL-Bit auf ´0´ gesetzt. Wird im Anschluß durch einen entsprechenden Unterroutinenaufruf wieder ein Wert auf den Stack geschrieben, d.h. der Stack wieder in den normalen Zustand versetzt, so entspricht dies intern einem normalen Überlauf und dementsprechend bleibt dadurch das STKAVL-Bit dauerhaft auf ´0´ gesetzt.

2.10 Interrupts

Gemäß dem Vorbild PIC17 unterstützt der Mikrocontrollerkern Interrupts zur schnellen Reaktion auf (externe) Ereignisse, unabhängig vom normalen Programmablauf. Es werden vier verschiedene Interruptquellen mit jeweils unterschiedlichen Prioritäten und eigenen Interruptroutinen behandelt: Zustandsänderungen an den externen Interrupteingängen INT0 und INT1, Überlauf des Ereigniszählers im internen RTCC-Modul und Peripherieinterrupts zur Reaktion auf evtl. Ereignisse direkt angeschlossener Peripheriemodule auf dem selben Chip.

Beim PIC17 tragen die beiden externen Interrupteingänge und damit auch alle zugehörigen Flags und Konfigurationsbits die Namen INT und RT. Aufgrund der begrenzten Anzahl Pins ist der Interrupteingang RT dort mit dem externen Takteingang für das RTCC-Modul zusammengefaßt. Da diese Beschränkung beim Mikrocontrollerkern so nicht besteht, sind die beiden Eingänge hier getrennt und der Pin RT dient ausschließlich als externer Takteingang für das RTCC-Modul. Beim Mikrocontrollerkern existieren deshalb zwei externe Interrupteingänge mit den Namen INT0 und INT1. Entsprechend wurden die zugehörigen Flags und Bits im Register INTSTA hier umbenannt. Gegebenenfalls können die Pins RT und INT1 aber jederzeit extern zusammengeschaltet und gemeinsam aus dem Chip heraus geführt werden, um z.B. vollständige Kompatibilität zum PIC17 sicherzustellen.

Im Gegensatz zum PIC17 enthält der Mikrocontrollerkern selbst keine Peripheriemodule. Aus diesem Grund werden die Peripherieinterrupts beim Mikrocontrollerkern nicht weiter unterteilt. Beim PIC17 können Peripherieinterrupts durch acht unterschiedliche Quellen ausgelöst werden, wobei anhand von zwei zusätzlichen Statusregistern, die in den Datenspeicherbereich der Peripherie eingeblendet sind, letztendlich festgelegt wird, welche Interrupts erlaubt sind und welche der acht Quellen tatsächlich den Interrupt ausgelöst hat. Beim Mikrocontrollerkern werden hingegen anstehende Peripherieinterrupts dem Kern über einen separaten Eingang, den Pin PEIR, signalisiert, und der Kern selbst enthält keine zusätzliche Logik zur weiteren Differenzierung von Peripherieinterrupts. Je nach Anzahl und

Art der tatsächlich mit auf dem Chip realisierten Peripherie muß deshalb durch die externe Beschaltung, in Zusammenarbeit mit der zugehörigen Interruptroutine, die gewünschte Logik für Interruptanforderungen am Pin PEIR realisiert werden.

Zur Steuerung und Konfiguration einerseits und zur Speicherung von Statusinformationen andererseits enthält das Interruptmodul des Mikrocontrollerkerns das zentrale Register INTSTA, das an Adresse 07h in den Datenspeicherbereich eingeblendet ist (siehe Abschnitt 2.6.8). Das Register enthält sowohl die Flags zur Speicherung anstehender Interruptanforderungen als auch Kontrollbits zur Maskierung einzelner Interruptquellen. Weiterhin kann mit Hilfe der Bits INTEDG und RTEDG im Register RTCSTA an Adresse 05h (siehe Abschnitt 2.6.6) die Art der zu erkennenden Flanken an den externen Interrupteingängen INT0 und INT1 konfiguriert werden. Schließlich enthält das Register CPUSTA an Adresse 06h noch das Konfigurationsbit GLINTD (Bit 4), über das global alle Interrupts gesperrt werden können.

In den folgenden Abschnitten werden das Zusammenwirken der einzelnen Teile, die Bedeutung der verschiedenen Konfigurationseinstellungen und die genauen Abläufe bei der Erkennung, Verarbeitung und Auslösung von Interrupts im Detail beschrieben.

2.10.1 Allgemein

In Tabelle 2.11 sind alle vier Interruptquellen mit ihren zugehörigen Flags, Maskierungsbits und Vektoren in einer Übersicht dargestellt. Über die IE-Maskierungsbits (engl. *Interrupt Enable*) im Register INTSTA kann jede Interruptquelle einzeln maskiert werden. Hat beim Eintreffen einer Interruptanforderung das zugeordnete Maskierungsbit den Wert ´0´, so wird kein Interrupt ausgelöst und die Anforderung solange gespeichert, bis die Interrupts dieser Quelle durch Setzen des Maskierungsbits auf ´1´ wieder freigegeben werden. Unabhängig davon können aber auch alle Interruptquellen global über das Bit GLINTD im Register CPUSTA maskiert werden. Hat dieses Bit den Wert ´1´, so wird in keinem Fall ein Interrupt ausgelöst und alle eintreffenden Anforderungen werden, unabhängig vom Zustand der einzelnen Maskierungsbits, im Register INTSTA gespeichert. Daraus folgt, daß ein Interrupt nur dann ausgelöst wird, wenn sowohl das Bit GLINTD den Wert ´0´ als auch das zugehörige Maskierungsbit in INTSTA den Wert ´1´ beinhaltet.

Jeder Interruptquelle ist im Register INTSTA ein sog. IR-Flag (engl. *Interrupt Request*) zugeordnet. Wird vom Mikrocontrollerkern ein Interrupt einer entsprechenden Quelle erkannt, so wird das zugehörige Interruptflag vom Mikrocontrollerkern automatisch auf ´1´ gesetzt. Dies zeigt an, daß eine Anforderung für diesen Interrupt vorliegt, d.h. daß ein entsprechendes Ereignis eingetroffen ist. Bei Eintreffen einer Anforderung wird das zugehörige Flag in jedem Fall, unabhängig vom Zustand der Maskierungsbits oder des GLINTD-Bits, auf ´1´ gesetzt.

Priorität	Interruptquelle	Interruptflag	Maskierungsbit	Vektor
Höchste	Flanke am Pin INT0	INT0IR (INTSTA, Bit 4)	INT0IE (INTSTA, Bit 0)	0008h
↑	Ereigniszählerüberlauf	RTCIR (INTSTA, Bit 5)	RTCIE (INTSTA, Bit 1)	0010h
	Flanke am Pin INT1	INT1IR (INTSTA, Bit 6)	INT1IE (INTSTA, Bit 2)	0018h
Niederste	Eingangspin PEIR	PEIR (INTSTA, Bit 7)	PEIE (INTSTA, Bit 3)	0020h

Tabelle 2.11: Interrupts des Mikrocontrollerkerns.

Ist der Interrupt entweder global gesperrt oder einzeln maskiert, so findet keine weitere Aktion statt, es wird kein Interrupt ausgelöst und die Interruptanforderung bleibt in dem Interruptflag gespeichert. Das Interruptflag wird erst zurückgesetzt, wenn die Anforderung durch Auslösen des zugehörigen Interrupts abgearbeitet wurde. Hat das Interruptflag bei

Eintreffen einer neuen Interruptanforderung schon den Wert ´1´, weil zuvor bei gesperrtem Interrupt schon ein Interrupt der selben Quelle eingetroffen ist, der bisher noch nicht abgearbeitet wurde, so geht eine der Anforderungen verloren.

Ist der Interrupt nicht gesperrt, dann wird, unter Berücksichtigung der Prioritäten bei gleichzeitigem Eintreffen mehrerer Interrupts verschiedener Quellen, der entsprechende Interrupt ausgelöst, der aktuelle Programmablauf unterbrochen, der aktuelle Inhalt des Programmzählers als Rücksprungadresse auf dem Stack abgelegt und in die dem Interrupt fest zugeordnete Interruptroutine verzweigt. Jeder Interruptquelle ist fest ein Interruptvektor zugeordnet. Der Interruptvektor gibt dabei die Startadresse der zugehörigen Interruptroutine an, d.h. bei Auslösen eines Interrupts verzweigt die Programmausführung an die angegebene Stelle im Programmspeicher, indem der entsprechende Interruptvektor in den Programmzähler geladen wird. Die Verzweigung in eine Interruptroutine bei Eintreffen eines Interrupts entspricht einer normalen Verzweigung in ein Unterprogramm mittels einer Call-Operation aus dem normalen Programm heraus, so daß am Ende der Interruptroutine mit einer normalen Return-Operation zum Programm zurückgekehrt werden kann bzw. muß.

Bei der Verzweigung in eine Interruptroutine wird vom Mikrocontrollerkern automatisch das Bit GLINTD auf ´1´ gesetzt. Dadurch werden standardmäßig weitere, verschachtelte Interrupts innerhalb der Abarbeitung einer Interruptroutine vermieden. Zur Rückkehr aus der Interruptroutine existiert deshalb der spezielle Befehl `RETFIE`, bei dessen Ausführung im Zuge des Rücksprungs automatisch auch gleichzeitig wieder alle Interrupts durch Rücksetzen des Bits GLINTD auf ´0´ freigegeben werden. Insgesamt werden also normalerweise während der Bearbeitung eines Interrupts alle weiteren Interrupts gesperrt. Es ist aber durchaus möglich, auch verschachtelte Interrupts zuzulassen und zu unterstützen, indem zu Beginn der Interruptroutine durch entsprechende Befehle das Bit GLINTD explizit auf ´0´ gesetzt wird. Zu beachten ist allerdings, daß aufgrund des nichtdeterministischen Eintreffens von Interruptereignissen die Gefahr eines Stacküberlaufs besteht.

Je nach Interruptquelle wird außerdem vom Mikrocontrollerkern bei der Verzweigung in eine Interruptroutine automatisch das zugehörige Interruptflag wieder zurück auf ´0´ gesetzt und damit die anstehende Interruptanforderung gelöscht, um die Tatsache widerzuspiegeln, daß der Interrupt damit abgearbeitet ist. Dies ist der Fall bei den Interruptflags INT0IR, RTCIR und INT1IR. Eine Ausnahme bildet das Flag PEIR, das nicht automatisch zurückgesetzt wird, da es nur direkt den Zustand am externen Pin PEIR widerspiegelt. In diesem Fall ist also ggf. die für den Peripherieinterrupt zuständige Interruptroutine dafür verantwortlich, daß bei ihrem Aufruf der entsprechende Interrupt abgearbeitet und die am Pin PEIR anliegende Anforderung zurückgesetzt wird.

Das Register INTSTA und das Bit GLINTD im Register CPUSTA können beliebig als Quelle oder Ziel jedes einzelnen Befehls eingesetzt werden, d.h. die verschiedenen Flags und Bits können ohne Einschränkungen geschrieben und gelesen werden. Eine Ausnahme bildet das Interruptflag PEIR, das nicht beschrieben werden kann und stets den Wert am Pin PEIR widerspiegelt. Auf der einen Seite läßt sich so zu jedem Zeitpunkt der aktuelle Status der Interruptlogik des Mikrocontrollerkerns auslesen. Auf der anderen Seite können durch Schreibzugriffe auf die im Datenspeicherbereich eingeblendeten Register die einzelnen Bits beliebig gesetzt werden. Durch Manipulation der Maskierungsbits und des Bits GLINTD wird es dadurch überhaupt erst möglich, aus dem Anwendungsprogramm heraus einzelne oder alle Interrupts freizuschalten oder zu sperren. Werden zuvor gesperrte Interrupts freigegeben und liegen in den Interruptflags gespeicherte Anforderungen vor, so wird durch den Schreibzugriff auf das Register und die Freigabe der Interrupts die Abarbeitung der wartenden Interrupts in der Reihenfolge ihrer Prioritäten ausgelöst. Andererseits können auch die Interruptflags selbst (mit Ausnahme von PEIR) durch Schreibzugriffe auf INTSTA direkt gesetzt oder zurückgesetzt werden, um z.B. Interrupts ohne eigentliches Ereignis auszulösen oder um anstehende Anforderungen ohne Auslösen eines Interrupts zu löschen.

2.10.2 Interruptquellen

In der Übersicht der Tabelle 2.11 sind die vier Interruptquellen des Mikrocontrollerkerns dargestellt. In der Reihenfolge ihrer Prioritäten haben die einzelnen Quellen dabei folgende Eigenschaften:

1. Die höchste Priorität haben Interrupts, die durch Flanken am externen Interrupteingang INT0 ausgelöst werden. Dies entspricht Interrupts am Pin INT beim PIC17. Interruptanforderungen können bei fallenden oder bei steigenden Flanke am Pin INT0 getriggert werden. Die entsprechende Konfiguration der Flankendetektion des Mikrocontrollerkerns erfolgt über das Bit INTEDG im Register RTCSTA (Adresse 05h) und kann jederzeit aus dem Anwendungsprogramm heraus eingestellt und geändert werden (siehe Tabelle 2.12). Zu beachten ist, daß bei einer Änderung der Flankeneinstellung vom Mikrocontrollerkern unter Umständen eine Flanke erkannt und ein (falscher) Interrupt ausgelöst werden kann. Aus diesem Grund sollte die Umstellung nur bei gesperrtem Interrupt erfolgen und anschließend das evtl. dadurch gesetzte Interruptflag gelöscht werden.

INTEDG (RTCSTA, Bit 7)	Interrupt INT0
'0'	Interrupt bei fallender Flanke am Pin INT0.
'1'	Interrupt bei steigender Flanke am Pin INT0.

Tabelle 2.12: Flankenselektion für Interrupt am Pin INT0.

Dem Interrupt ist das Interruptflag INT0IR (Bit 4, entspricht INTIR beim PIC17) und das Maskierungsbit INT0IE (Bit 0, entspricht INTIE beim PIC17) im Register INTSTA zugeordnet. Die zugehörige Interruptroutine beginnt an Adresse 0008h im Programmspeicher. Bei der Verzweigung in die Interruptroutine wird das Interruptflag INT0IR automatisch zurückgesetzt.

2. Ein Überlauf des Ereigniszählers des RTCC-Moduls (siehe Abschnitt 2.11) löst eine Interruptanforderung mit der zweithöchsten Priorität aus. Das RTCC-Modul implementiert einen 16 Bit breiten Zähler mit einstellbarem Vorteiler und externem oder internem Takteingang. Bei einem Überlauf des RTCC-Zählers von FFFFh nach 0000h wird das Interruptflag RTCIR (Bit 5) im Register INTSTA gesetzt. Das zugehörige Maskierungsbit im Register INTSTA ist das Bit RTCIE (Bit 1). Die zugehörige Interruptroutine befindet sich an Adresse 0010h im Programmspeicher. Das Interruptflag RTCIR wird bei Auslösen des Interrupts automatisch zurückgesetzt.

   Mit Hilfe des RTCC-Moduls und des Interrupts können Aufgaben, die in regelmäßigen Abständen durchgeführt werden müssen, parallel zum normalen Programmablauf durchgeführt werden. Dazu wird das System so konfiguriert, daß alle n Taktzyklen ein RTCC-Interrupt ausgelöst wird, der dann die entsprechenden Aktionen triggert.

3. An dritter Stelle stehen Interrupts, die durch Flanken am externen Interrupteingang INT1 ausgelöst werden. Dies entspricht Interrupts am Pin RT beim PIC17, wobei beim PIC17 der Pin RT gleichzeitig auch als externer Takteingang für das RTCC-Modul dient, beim Mikrocontrollerkern aber der Interrupteingang getrennt vom RTCC-Takteingang realisiert ist. Interruptanforderungen können bei fallenden oder bei steigenden Flanke am Pin INT1 getriggert werden. Die entsprechende Konfiguration der Flankendetektion des Mikrocontrollerkerns erfolgt über das Bit RTEDG im

Register RTCSTA (Adresse 05h) und kann jederzeit aus dem Anwendungsprogramm heraus eingestellt und geändert werden (siehe Tabelle 2.13). Da beim PIC17 Interrupteingang und RTCC-Takteingang zusammengefaßt sind, dient das Bit RTEDG gleichzeitig auch zur Flankenselektion für den RTCC-Takteingang. Aus Kompatibilitätsgründen wurde dies beim Mikrocontrollerkern beibehalten und trotz Trennung der Pins kein zusätzliches Konfigurationsbit eingeführt. Zu beachten ist, daß bei einer Änderung der Flankeneinstellung vom Mikrocontrollerkern unter Umständen eine Flanke erkannt und ein (falscher) Interrupt ausgelöst werden kann. Aus diesem Grund sollte die Umstellung nur bei gesperrtem Interrupt erfolgen und anschließend das evtl. dadurch gesetzte Interruptflag gelöscht werden.

RTEDG (RTCSTA, Bit 6)	Interrupt INT1
'0'	Interrupt bei fallender Flanke am Pin INT1.
'1'	Interrupt bei steigender Flanke am Pin INT1.

Tabelle 2.13: Flankenselektion für Interrupt am Pin INT1.

Dem Interrupt ist das Interruptflag INT1IR (Bit 6, entspricht RTXIR beim PIC17) und das Maskierungsbit INT1IE (Bit 2, entspricht RTXIE beim PIC17) im Register INTSTA zugeordnet. Die zugehörige Interruptroutine beginnt an Adresse 0018h im Programmspeicher. Bei der Verzweigung in die Interruptroutine wird das Interruptflag INT1IR automatisch zurückgesetzt

4. Peripherieinterrupts haben die niedrigste Priorität. Diese Interruptquelle ist vorgesehen für Interrupts, die von der zusammen mit dem Mikrocontrollerkern auf dem Chip integrierten Peripherie ausgelöst werden können. Zu diesem Zweck besitzt der Mikrocontrollerkern mit dem Pin PEIR einen externen Eingang für entsprechende Anforderungen. Der Wert am Pin PEIR wird dabei vom Mikrocontrollerkern in jedem Taktzyklus direkt und unverändert in das Interruptflag PEIR (Bit 7) im Register INTSTA übernommen. Durch entsprechende Beschaltung des Mikrocontrollerkerns können externe Module anliegende Interruptanforderungen signalisieren, indem der Pin PEIR auf logisch ´1´ gesetzt wird. Eine andere Manipulation des Interruptflags PEIR im Register INTSTA ist nicht möglich, insbesondere kann das Bit vom Anwendungsprogramm nicht geschrieben werden und wird auch nicht vom Mikrocontrollerkern automatisch bei Verzweigung in die Interruptroutine zurückgesetzt. Eine Peripherieinterruptanforderung besteht, solange der Pin PEIR den Wert ´1´ hat. Die zugehörige Interruptroutine, beginnend an Adresse 0020h im Programmspeicher, muß dafür sorgen, daß dem auslösenden externen Modul signalisiert wird, daß der Interrupt abgearbeitet wird. Das externe Modul muß daraufhin den Pin PEIR wieder zurücksetzen, um zu vermeiden, daß bei Freigabe der Interrupts nach Rückkehr aus der Interruptroutine erneut ein Interrupt ausgelöst wird.

   Beim PIC17, bei dem die entsprechenden Peripheriemodule bereits mit integriert sind, wird das Interruptflag PEIR direkt aus der Oder-Verknüpfung der Interruptflags der Peripheriemodule gebildet. Das Verhalten des Mikrocontrollerkerns entspricht weitestgehend dem des PIC17, wenn der Mikrocontrollerkern so mit einem externen Register für Peripherieinterruptflags beschaltet wird, daß die einzelnen Flags über ein Oder-Gatter auf den Pin PEIR geführt werden. Wie beim PIC17 müssen die Prioritäten der einzelnen Peripherieinterrupts dann von der Interruptroutine festgelegt werden.

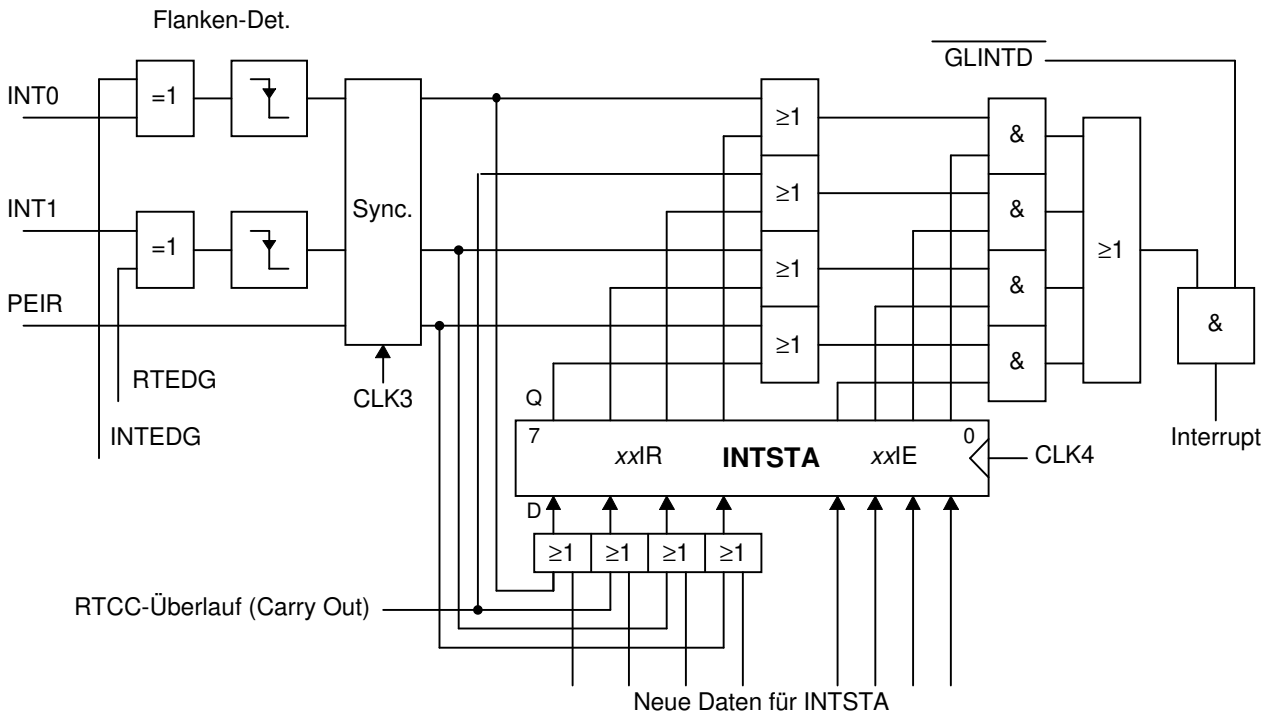


Abbildung 2.21: Blockdiagramm der Interruptlogik des Mikrocontrollerkerns.

2.10.3 Interruptlogik

In Abbildung 2.21 ist das auf der bisher beschriebenen Funktionsweise basierende Blockdiagramm der Interruptlogik des Mikrocontrollerkerns schematisch dargestellt. Zentraler Punkt der Interruptlogik ist das Register INTSTA, in dem anliegende Interruptanforderungen und die Maskierung einzelner Interrupts gespeichert werden. Das Register wird mit CLK4 getaktet und übernimmt immer am Ende der Taktphase Q4, d.h. am Ende eines Instruktionszyklus, mit der fallenden Flanke von CLK4 einen neuen Wert. Wird das Register mittels eines Befehls beschrieben, so wird der neue Wert vom Datenergebnisbus des Mikrocontrollerkerns übernommen. Andernfalls wird der bisherige Inhalt des Register wieder als neuer Wert in das Register geschrieben, d.h. das Register behält seinen Wert. In beiden Fällen überschreiben aber eingehende Interruptanforderungen ggf. den Wert des zugehörigen Interruptflags und setzen das Flag, unabhängig vom zu schreibenden Wert des Operationsergebnisses, auf ´1´.

Wie bereits mehrfach erwähnt, können Interruptanforderungen vier verschiedenen Quellen entspringen. Zur Erkennung von Anforderungen an den externen Interruptpins INT0 und INT1 besitzt der Mikrocontrollerkern einen Flankendetektor, der entsprechende Zustandswechsel an den Pins erkennt. Die Selektion der zu erkennenden Flankenart wird dabei über der eigentlichen Detektion vorgeschaltete Exklusiv-Oder-Gatter bewerkstelligt. Die Eingangssignale an den externen Pins werden über diese Gatter mit den Konfigurationsbits INTEDG bzw. RTEDG verknüpft und im Anschluß stets auf fallende Flanken untersucht. Haben die entsprechenden Konfigurationsbits den Wert ´1´, so werden die der Detektionsschaltung zugeführten Signale von den XOR-Gattern invertiert und die von der Schaltung erkannten fallenden Flanken entsprechen steigenden Flanken an den Pins, so daß insgesamt steigende Flanken von der Logik erkannt werden. Daraus folgt aber, daß bei der Umschaltung der Flankenkonfiguration fallende Flanken erzeugt werden können, die von der Detektionsschaltung irrtümlich als Interruptanforderungen ausgewertet werden. Dies ist genau dann der Fall wenn einerseits das Signal am Pin logisch ´1´ ist und dann das Konfigurationsbit

von ´0´ auf ´1´ umgeschaltet wird oder wenn andererseits der Pin den Wert ´0´ hat und die Konfiguration von ´1´ auf ´0´ geändert wird.

Das Ausgangssignal des Flankendetektors wird schließlich mit der fallenden Flanke von CLK3 synchronisiert und so das eigentliche Interruptsignal der entsprechenden Quelle erzeugt. Ebenfalls mit der fallenden Flanke von CLK3 wird der Wert am Eingangspin PEIR in den Mikrocontrollerkern übernommen und als Peripherieinterruptsignal intern in die Logik eingespeist. Als viertes Interruptsignal dient das Carry-Out Signal des Ereigniszählers, das einen Überlauf des RTCC-Zählers anzeigt. Da dieses Signal bereits intern erzeugt wurde muß es nicht weiter synchronisiert werden.

Die vier Interruptsignale werden in einem ersten Schritt dazu benutzt, die zugehörigen Interruptflags im Register INTSTA am Ende des selben Instruktionszyklus zu setzen. Auf der anderen Seite werden die Signale direkt weiterverarbeitet, so daß ggf. noch in dem Zyklus, in dem ein Interruptereignis erkannt wurde, ein Interrupt ausgelöst werden kann. Durch entsprechende Oder-Verschaltung werden Interruptanfoderungen entweder direkt aus den Interruptsignalen oder aus den im Register gespeicherten Anforderungen der Interruptflags erzeugt. Eine Interruptanforderung beginnt also mit der Erkennung und Synchronisation (Taktphase Q3) des zugehörigen Interruptereignisses und bleibt bestehen bis das zugehörige Interruptflag bei Abarbeitung des Interrupts zurückgesetzt wird.

Die anstehenden Interruptanforderungen werden anschließend mit den zugeordneten Konfigurationsbits in der unteren Hälfte des Register INTSTA maskiert, um die tatsächlich aktiven Interruptquellsignale zu erzeugen. Anhand dieser vier Quellsignale wird bei Auslösung eines Interrupts, unter Berücksichtigung der Prioritäten, schließlich entschieden, welcher Interrupt ausgelöst, d.h. welcher Interruptvektor in den Programmzähler geladen wird. Damit überhaupt ein Interrupt ausgelöst wird, muß einerseits mindestens eines der Quellsignale aktiv sein und es dürfen andererseits die Interrupts nicht global gesperrt sein. Intern wird ein globales Interruptsignal, das schlußendlich die Auslösung eines Interrupts triggert, aus der entsprechenden Verknüpfung der Quellsignale und des Werts des Konfigurationsbits GLINTD gebildet.

In Abbildung 2.22 ist das Timing der diversen Signale bei der Auslösung eines Interrupts, als Reaktion auf eine externe Signalflanke am Interruptpin, nochmals zusammengefaßt. Als Reaktion auf die fallende Signalflanke wird das Ausgangssignal des Flankendetektors aktiviert. Das Flankendetektorsignal wird von der Synchronisationsschaltung in jedem Instruktionszyklus

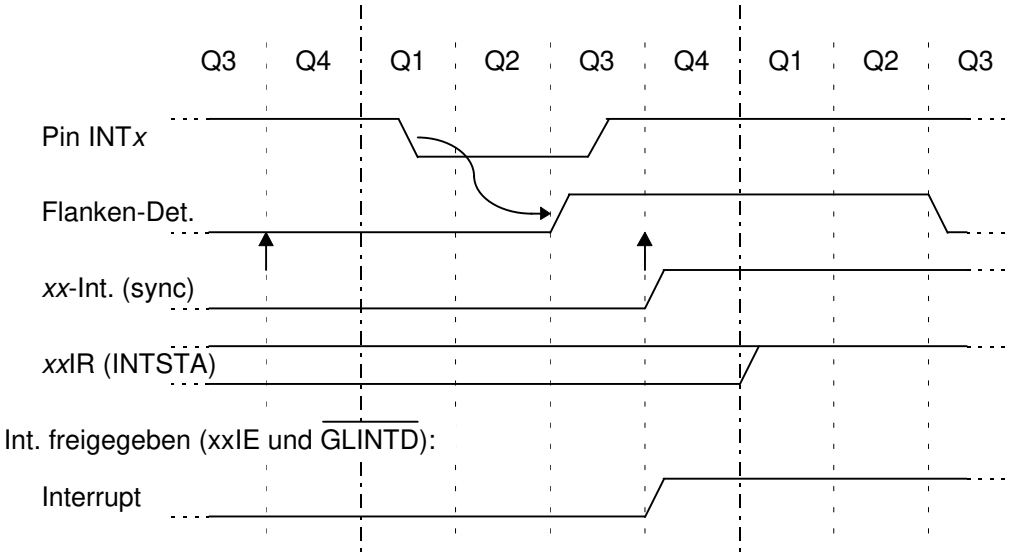


Abbildung 2.22: Auslösen eines Interrupts durch Verarbeitung eines Interruptereignisses.

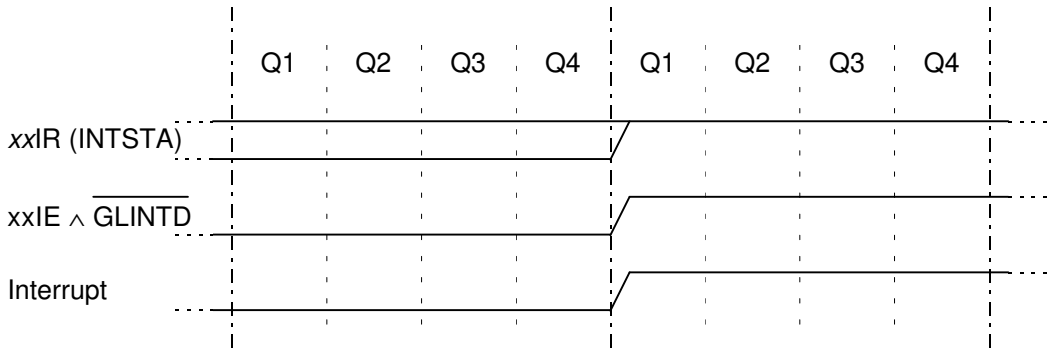


Abbildung 2.23: Auslösen eines Interrupts durch Freigabe einer anstehenden Anforderung.

am Ende der Taktphase Q3 abgetastet und übernommen. Der Flankendetektor berücksichtigt dies und hält sein Ausgangssignal solange aktiv, bis es einmal abgetastet wurde. Am Ende des Instruktionszyklus wird das synchronisierte Signal in das zugehörige Interruptflag übernommen und dort auf jeden Fall abgespeichert. Ist der entsprechende Interrupt nicht maskiert und global freigegeben, so aktiviert das synchronisierte Interruptquellsignal direkt das interne Interruptsignal und löst damit noch im selben Instruktionszyklus einen Interrupt aus.

Auf der anderen Seite kann ein Interrupt auch ausgelöst werden, indem eine bei gesperrtem Interrupt eingetroffene Interruptanforderung durch Rücksetzen der Maskierungsbits oder des Bits GLINTD freigegeben wird. Dazu wird das Register INTSTA bzw. das Bit GLINTD im Register CPUSTA direkt durch einen Schreibzugriff manipuliert. Alternativ kann bei einem Schreibzugriff auf INTSTA auch (zusätzlich) direkt ein Interruptflag gesetzt und so eine Anforderung erzeugt werden. In Abbildung 2.23 ist das entsprechende Timing dargestellt. Schreibzugriffe auf INTSTA oder CPUSTA haben zur Folge, daß am Ende des Instruktionszyklus die Register einen neuen Wert annehmen und so ein Zustand hergestellt wird, bei dem ein Interruptflag eines nicht maskierten Interrupts gesetzt ist und die Interrupts global freigegeben sind. Der Übergang in den neuen Zustand am Ende der Taktphase Q4 hat zur Folge, daß das interne Interruptsignal aktiviert und dadurch im nächsten Zyklus ein Interrupt ausgelöst wird.

2.10.4 Interrupt-Timing

Als Reaktion auf die Aktivierung des internen Interruptsignals durch die Interruptlogik wird vom Mikrocontrollerkern ein Interrupt ausgelöst. Dabei werden alle nötigen Aktionen durchgeführt und in die zugehörige Interruptroutine verzweigt. In Abhängigkeit vom Zustand der vier maskierten Interruptquellsignale wird unter den aktivierten Interruptquellen der Interrupt mit der höchsten Priorität ausgelöst.

Abbildung 2.24 zeigt das Timing aller relevanten Signale und den Ablauf bei der Auslösung eines Interrupts. Während parallel eine Instruktion ausgeführt und die nachfolgende bereits geladen wird, wird von der Interruptlogik durch Aktivierung des Interruptsignals ein Interrupt ausgelöst. Wie bereits dargelegt, wird gleichzeitig am Ende des Instruktionszyklus das zugehörige Interruptflag gesetzt. Der Mikrocontrollerkern unterbricht noch im selben Zyklus den normalen Programmablauf, indem einerseits, anstatt des in diesem Zyklus geladenen Befehls, am Ende des Zyklus ein `NOP` in das Register IR geschrieben und andererseits der Programmzähler am Ende des Zyklus angehalten und nicht inkrementiert wird. Die gerade ausgeführte Instruktion wird vollständig abgearbeitet, während die nächste Instruktion nicht mehr zur Ausführung kommen wird. Dies gilt auch für Interrupts, die bei der Ausführung mehrzykliger Instruktionen auftreten. Unabhängig davon, in welchem Ausführungszyklus des Befehls der Interrupt auftritt, wird im Zuge der Auslösung des Interrupts parallel die gerade laufende Instruktion fertig ausgeführt, der nachfolgende Befehl,

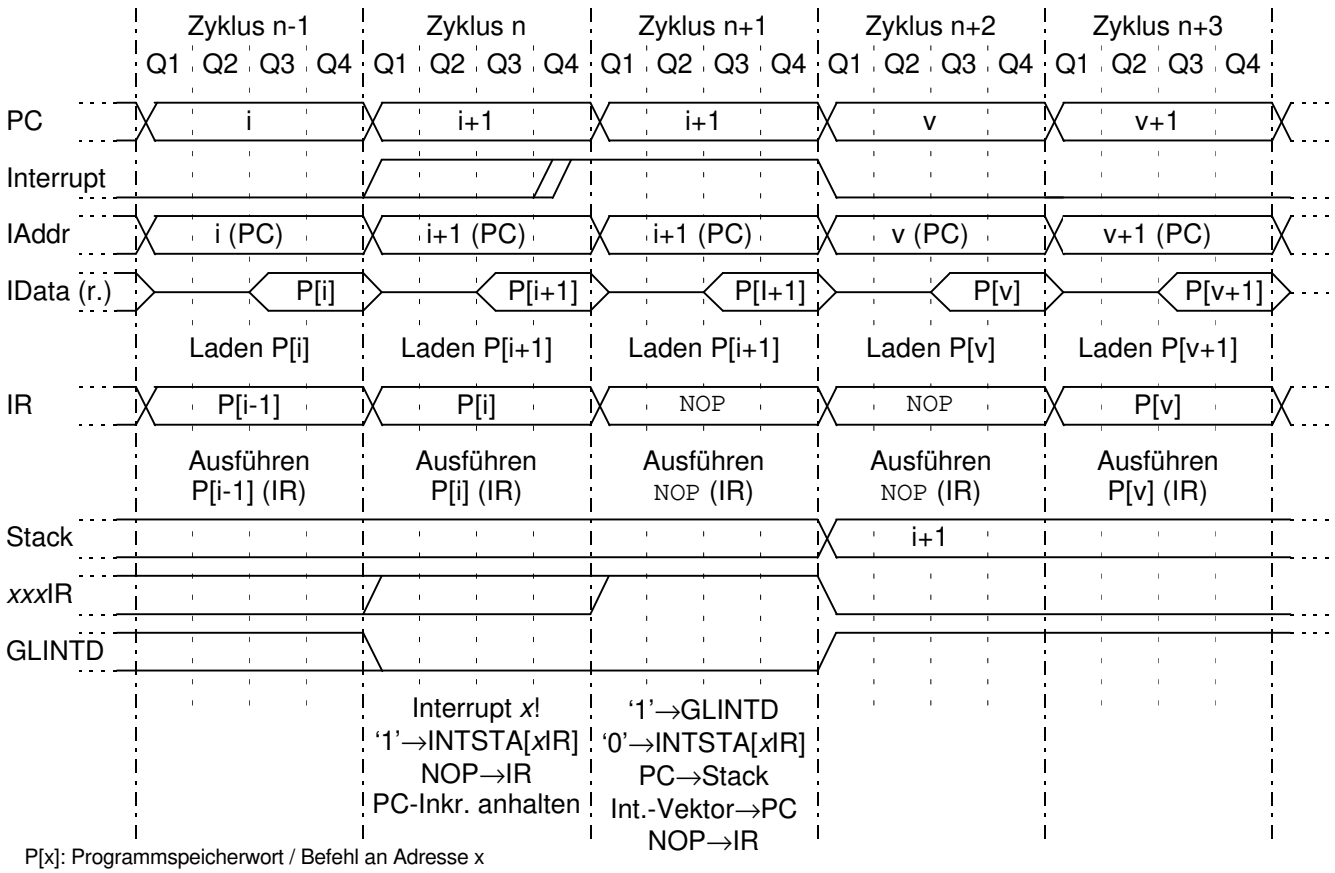


Abbildung 2.24: Interrupt-Timing.

der bereits geladen worden sein kann, aber in keinem Fall mehr abgearbeitet. Um in jedem Fall die Ausführung mehrzykliger Instruktionen abschließen zu können, wird ein zusätzlicher Zyklus eingeschoben, und es werden nicht schon im ersten Zyklus, bei Auftreten des Interrupts, alle notwendigen Aktionen zur Verzweigung in die Interruptroutine durchgeführt.

Im nächsten Instruktionszyklus wird keine Operation durchgeführt, d.h. es wird der im Register IR enthaltene Befehl `NOP` ausgeführt. Benötigt die Ausführung der letzten Instruktion mehr als einen Instruktionszyklus, so wird sie ggf. erst in diesem Zyklus zu Ende gebracht. Gleichzeitig wird die Verzweigung in die Interruptroutine durchgeführt. Der aktuelle Inhalt des Programmzählers wird auf dem Stack als Rücksprungadresse abgelegt. Da der Programmzähler im vorangegangenen Zyklus nicht inkrementiert wurde, zeigt er weiterhin auf die korrekte Rücksprungadresse, so daß die Programmausführung bei der Rückkehr aus der Interruptroutine mit der nächsten nicht ausgeführten Instruktion fortgefahren wird. Zusätzlich wird der Interruptvektor in den Programmzähler geladen, das Bit GLINTD gesetzt und ggf. das Interruptflag zurückgesetzt. In das Register IR wird wieder der Befehl `NOP` zur Ausführung im nächsten Zyklus geladen. Der während des Zyklus geladene Programmspeicherwert wird verworfen. Mit dem Setzen des Bits GLINTD sind ab diesem Zeitpunkt alle Interrupts gesperrt. Gleichzeitig wird dadurch auch das Interruptsignal zurückgesetzt und die Interruptaktivierung aufgehoben.

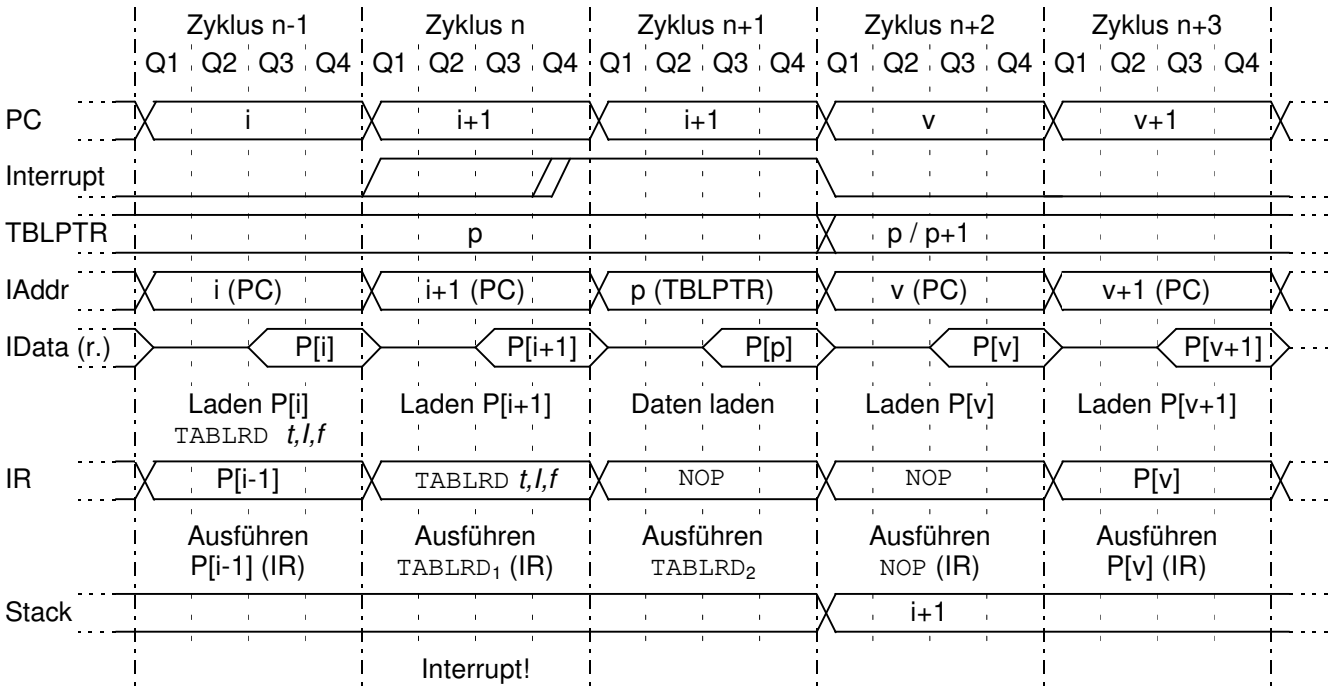
Im dritten Zyklus des Interrupts wird schließlich die Pipeline neu gefüllt und die Programmausführung fährt normal fort: der erste Befehl der Interruptroutine wird geladen, während entsprechend dem im Register IR enthaltenen `NOP` keine Operation durchgeführt wird. Mit der Ausführung des ersten Interruptroutinenbefehls im nächsten Zyklus ist dann die Verzweigung in die Interruptroutine abgeschlossen.

Das hier geschilderte Timing ist der Allgemeinfall bei der Ausführung normaler Instruktionen. Es existieren aber eine Reihe von Instruktionen, bei deren Ausführung teilweise spezielle Abläufe eintreten können. In einigen Fällen kommt es z.B. zu einem Konflikt zwischen den bei der Ausführung des Befehls durchzuführenden und den durch den Interrupt bedingten Aktionen. Bis auf wenige Ausnahmen haben im allgemeinen die Interruptaktionen in solchen Konfliktfällen Priorität gegenüber den Aktionen der Befehlsausführung. In den folgenden Abschnitten werden alle Spezialfälle beschrieben, bei denen es zu besonderen Abweichungen gegenüber dem allgemeinen Interrupttiming kommt. Alle hier nicht aufgeführten Fälle weichen, unter der Berücksichtigung der Priorität der Interruptaktionen, nicht von dem in diesem Abschnitt dargelegten allgemeinen Timing ab.

2.10.4.1 Table-Operationen

Wie in Abschnitt 2.8.2 beschrieben, umfaßt die Ausführung von Table-Operationen insgesamt zwei Instruktionszyklen, wobei im zweiten Ausführungszyklus der zusätzliche Programmspeicherzugriff durchgeführt wird. Bei der Ausführung von Table-Operationen können Interrupts sowohl im ersten als auch im zweiten Ausführungszyklus auftreten. Im folgenden werden die aus beiden Fällen resultierenden Abläufe kurz dargelegt. Sofern nicht anders dargestellt, gilt für die Ausführung und das Timing der eigentlichen Table-Operation das in Abschnitt 2.8.2 gezeigte Verhalten und für die Durchführung des Interrupts das im vorangegangenen Abschnitt beschriebene Timing.

In Abbildung 2.25 sind die Abläufe bei Auftreten eines Interrupts im ersten Ausführungszyklus einer Table-Operation am Beispiel eines `TABLRD` dargestellt. Während der erste Zyklus der Table-Operation ausgeführt wird, tritt ein Interrupt auf. Zusätzlich zur Ausführung der Table-Operation werden die ersten Interruptaktionen durchgeführt. Das Interruptflag wird normal gesetzt, in Einklang mit der Ausführung der Table-Operation wird der Programmzähler nicht inkrementiert und anstatt des geladenen Nachfolgebefehls wird ein `NOP` in das Register IR geschrieben. Im zweiten Zyklus wird dann der zweite Zyklus der Table-Operation, d.h. der Programmspeicherzugriff durchgeführt. Gleichzeitig werden alle



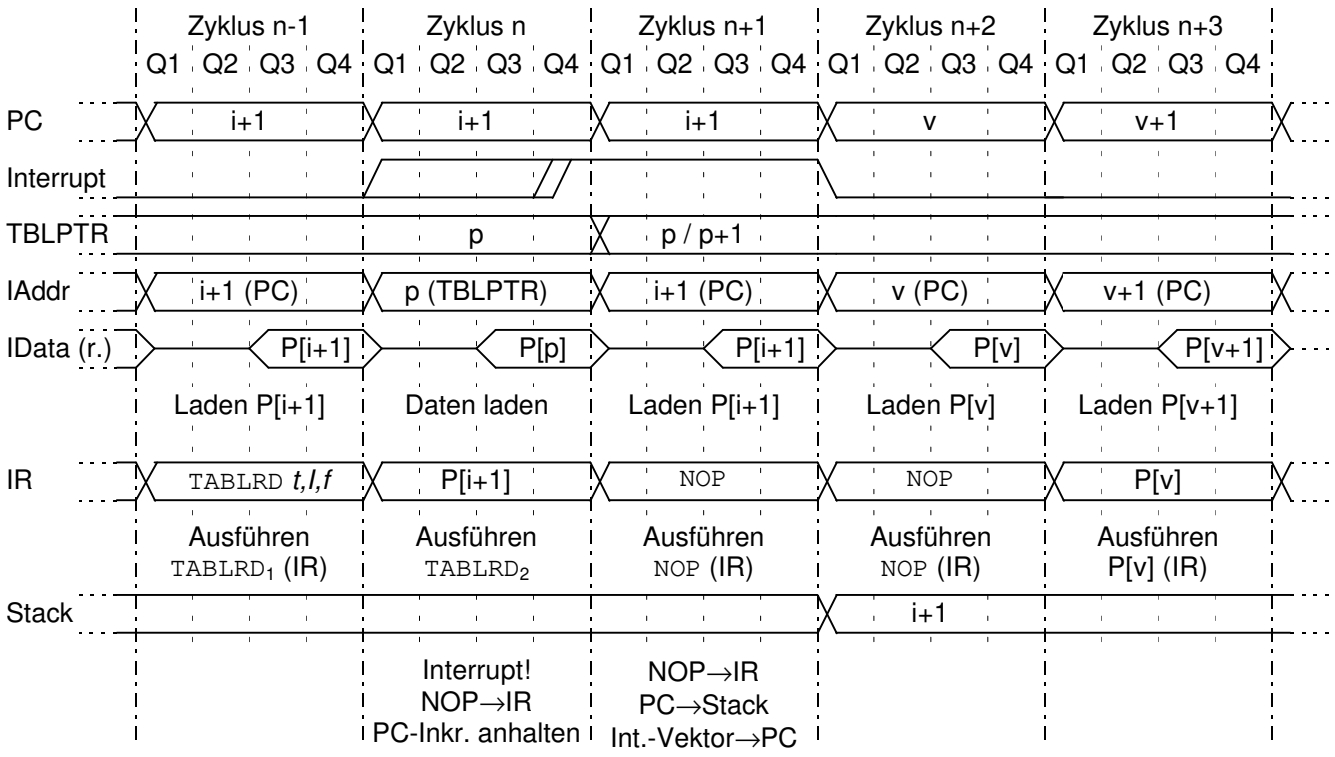
P[x]: Programmspeicherwort / Befehl an Adresse x

Abbildung 2.25: Interrupt im ersten Zyklus einer Table-Operation.

Interruptaktionen zur Verzweigung in die Interruptroutine, d.h. Sichern und Laden des Programmzählers, Setzen des Bits GLINTD, etc., ausgeführt. Hier ist zu erkennen, daß der zusätzliche Zyklus bei der Auslösung eines Interrupts notwendig ist, um im zweiten Zyklus die Table-Operation mit dem Programmspeicherzugriff fertig auszuführen. Alle weiteren Aktionen unterscheiden sich nicht von den normalen Abläufen bei der Verzweigung in die Interruptroutine.

Abbildung 2.26 spiegelt die Verhältnisse bei Auftreten eines Interrupts im zweiten Ausführungszyklus einer Table-Operation, wieder am Beispiel eines `TABLRD`, wieder. Es wird insbesondere deutlich, warum bei der Ausführung einer Table-Operation der Programmzähler auch im zweiten Zyklus noch auf die direkt nachfolgende Instruktion zeigen muß und nicht schon im ersten Ausführungszyklus der Table-Operation inkrementiert werden darf. Bei Auftreten des Interrupts im zweiten Zyklus der Table-Operation muß der Programmzähler die korrekte Rücksprungadresse der nachfolgenden Instruktion, zu der nach Ende der Interruptroutine zurückgekehrt werden soll, beinhalten. Würde der Programmzähler schon im ersten Zyklus der Table-Operation inkrementiert, so würde er bei Auftreten des Interrupts bereits auf die übernächste Instruktion zeigen und es würde vom Mikrocontrollerkern bei der Verzweigung in die Interruptroutine der entsprechende Programmzählerinhalt als Rücksprungadresse auf dem Stack abgelegt. Bei der Rückkehr aus der Interruptroutine würde dann bei der übernächsten Instruktion wieder aufgesetzt werden und die direkt nachfolgende Instruktion fälschlicherweise nicht ausgeführt bzw. übersprungen.

Bei Auftreten des Interrupts im zweiten Zyklus der Table-Operation werden zusammen mit der Ausführung des zweiten Zyklus der Table-Operation die üblichen Interruptaktionen durchgeführt.. Der Programmzähler wird dadurch, entgegen der eigentlich zu diesem Zeitpunkt durch die Table-Operation veranlaßten Inkrementierung, angehalten. Während durch die Table-Operation der Programmspeicherzugriff ausgeführt wird, veranlaßt die Interruptbedingung das Setzen des zugeordneten Interruptflags und das Laden des Befehls



P[x]: Programmspeicherwort / Befehl an Adresse x

Abbildung 2.26: Interrupt im zweiten Zyklus einer Table-Operation.

NOP in das Register IR. Der im ersten Zyklus der Table-Operation bereits in das Register IR geladene nachfolgende Befehl wird dadurch verworfen.

Im zweiten Interruptzyklus ist dann die Ausführung der Table-Operation bereits abgeschlossen, und die entsprechenden Aktionen des zweiten Interruptzyklus zur Verzweigung in die Interruptroutine werden bis zur Ausführung der ersten Instruktion der Interruptroutine normal abgearbeitet, während vom Mikrocontrollerkern keine weitere Operation durchgeführt wird.

2.10.4.2 Befehle zur Programmablaufsteuerung

In Abschnitt 2.8.3 wurde die normale Ausführung von Befehlen zur Programmablaufsteuerung ausführlich beschrieben. Es wurden dabei drei Klassen von Befehlen unterschieden: Skips, Sprünge bzw. Unterprogrammaufrufe und Rücksprünge aus Unterprogrammen. In diesem Abschnitt werden im folgenden die Abläufe von Interrupts beschrieben, die während der Ausführung von Befehlen der drei verschiedenen Klassen auftreten:

- Interrupts, die bei der Ausführung von Skip-Befehlen auftreten sind mit einer Ausnahme relativ einfach zu behandeln, da sie sich im Prinzip nicht von der Durchführung von Interrutps bei der normalen Befehlsausführung unterscheiden. Die Ausführung eines Skip-Befehls unterscheidet sich i.A., sowohl bei durchgeführtem Skip als auch bei nicht durchgeführtem Skip, kaum von der normalen sequentiellen Befehlsausführung.

  Tritt bei der Ausführung des Skip-Befehls die für den Skip notwendige Bedingung nicht ein, d.h. der Skip wird nicht durchgeführt, so ändert sich am normalen sequentiellen Befehlsablauf nichts und auftretende Interrupts werden vollkommen orthogonal wie im allgemeinen Fall behandelt.

  Tritt dagegen die für den Skip notwendige Bedingung ein, d.h. der Skip wird durchgeführt, dann wird die nachfolgende Instruktion nicht ausgeführt und statt dessen durch Laden des Register IR mit dem Befehl NOP ein Zyklus eingeschoben, in dem keine Operation durchgeführt wird. Interrupts können dabei sowohl bei der Ausführung des eigentlichen Skip-Befehls als auch bei der Ausführung des nachfolgenden NOPs auftreten.

  Abbildung 2.27 zeigt den Ablauf bei Auftreten eines Interrupt während der Ausführung eines Skip-Befehls, bei dem die notwendige Bedingung eintritt, der Skip also durchgeführt wird. Durch den Skip-Befehl wird, in Einklang mit der Interruptbehandlung, ein NOP anstatt des während des Zyklus geladenen nachfolgenden Befehls in das Register IR geladen, um zu erreichen, daß der nachfolgende Befehl übersprungen und nicht ausgeführt wird. In diesem speziellen Fall darf aber im ersten Zyklus der Interruptbehandlung die Inkrementierung des Programmzählers, im Gegensatz zum allgemeinen Fall, *nicht* unterbrochen bzw. angehalten werden, d.h. der Programmzählerinhalt wird trotz auftretendem Interrupt um eins erhöht! Wäre dem nicht so, d.h. würde der Programmzähler nicht inkrementiert, so würde im Zuge der Verzweigung in die Interruptroutine die Adresse des zu überspringenden Befehls, entsprechend dem Inhalt des nicht inkrementierten Programmzählers, als Rücksprungadresse auf dem Stack abgelegt. Bei der Rückkehr aus der Interruptroutine würde dann zu diesem Befehl zurückgekehrt und der Befehlsablauf mit der Ausführung des eigentlich zu überspringenden Befehls fortgesetzt, d.h. der Befehl würde fälschlicherweise ausgeführt, obwohl er eigentlich durch den Skip übersprungen werden sollte. Bei Auftreten eines Interrupts während der Ausführung eines Skip-Befehls, bei dem durch Laden eines NOP in das Register IR tatsächlich der Skip durchgeführt wird, darf also der Programmzählerinkrement nicht angehalten werden. Alle anderen Interruptaktionen im aktuellen und den darauffolgenden Zyklen, d.h. Setzen des

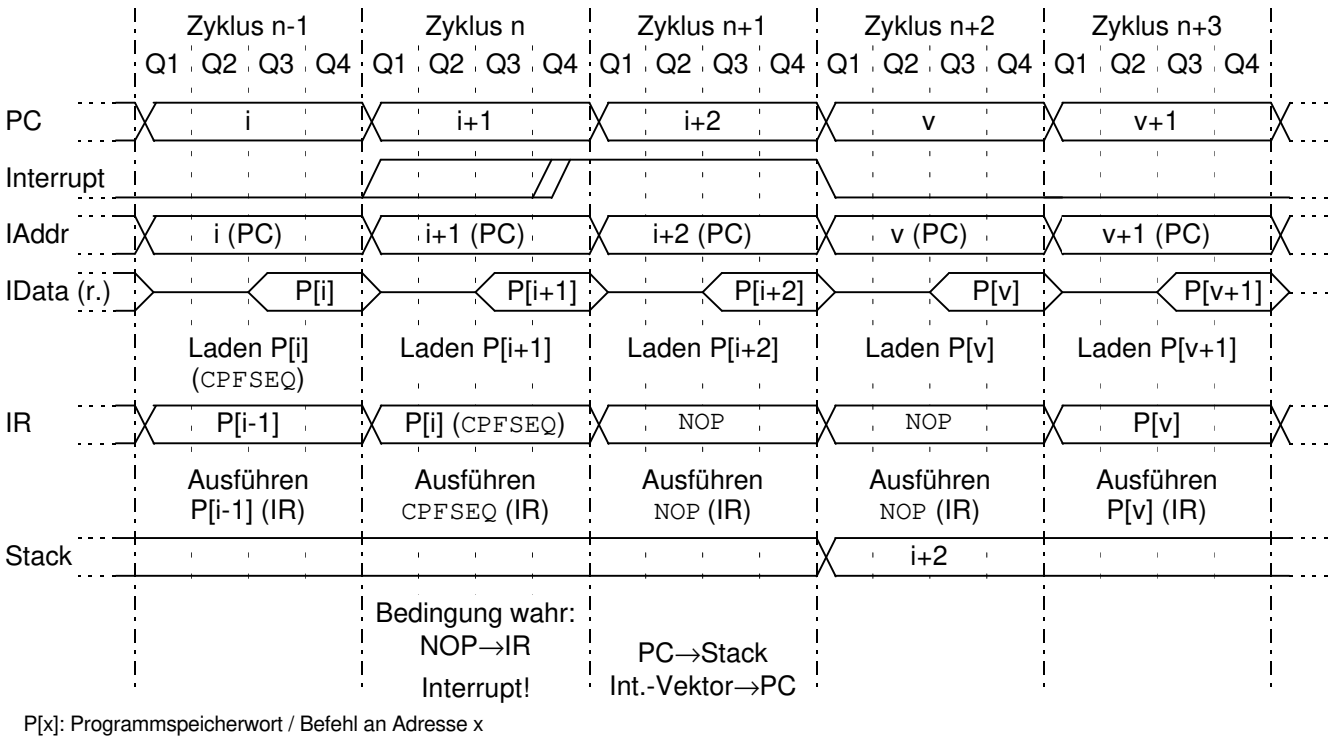


Abbildung 2.27: Interrupt im ersten Zyklus eines Skips.

Interruptflags, Verzweigen in die Interruptroutine, etc., werden dahingegen normal durchgeführt.

Ein Interrupt, der schließlich bei dem bei ausgeführtem Skip auf den Skip-Befehl folgenden `NOP` auftritt, unterscheidet sich nicht von einem Interrupt bei der normalen Ausführung eines Befehls, da durch den ausgeführten Skip ein nicht zu unterscheidender Zustand erzeugt wird, bei dem an dieser Stelle des Programms ein `NOP` anstatt des eigentlich dort auszuführenden Befehls steht und ausgeführt wird. Entsprechend wird auch ein dabei auftretender Interrupt wie im allgemeinen Fall abgearbeitet. Zu diesem Zeitpunkt enthält der Programmzähhler auch bereits die richtige Rücksprungadresse, d.h. die Adresse der auf den übersprungenen Befehl folgenden Instruktion, so daß in diesem Fall der Programmzählerinkrement im Zuge der Interruptbehandlung normal angehalten wird.

- Die Ausführung von Sprüngen und Unterprogrammaufrufen beansprucht durchweg zwei Instruktionszyklen. Dementsprechend können Interrupts bei der Ausführung solcher Befehle sowohl im ersten Zyklus als auch im zweiten Zyklus der Ausführung auftreten. Im folgenden werden die Verhältnisse exemplarisch am Beispiel von Unterprogrammaufrufen dargestellt. Die Ergebnisse sind aber durchweg auch auf Sprünge übertragbar. Der einzige grundlegende Unterschied besteht darin, daß bei Sprüngen im Gegensatz zu Unterprogrammaufrufen keine Rücksprungadresse auf dem Stack abgelegt wird.

  In Abbildung 2.28 ist das Timing für einen Interrupt im ersten Zyklus eines entsprechenden Befehls am Beispiel der Ausführung eines `CALL`-Befehls dargestellt. In dem Zyklus, in dem der Interrupt auftritt, wird durch den ersten Ausführungszyklus des Unterprogrammaufrufs sowohl der aktuelle Inhalt des Programmzählers auf dem Stack als Rücksprungadresse abgelegt als auch die Zieladresse des Unterprogramms in den Programmzähler geladen. In diesem Fall kommt es zu einem Konflikt zwischen der Ausführung des Unterprogrammaufrufs und den Interruptaktionen. Durch den

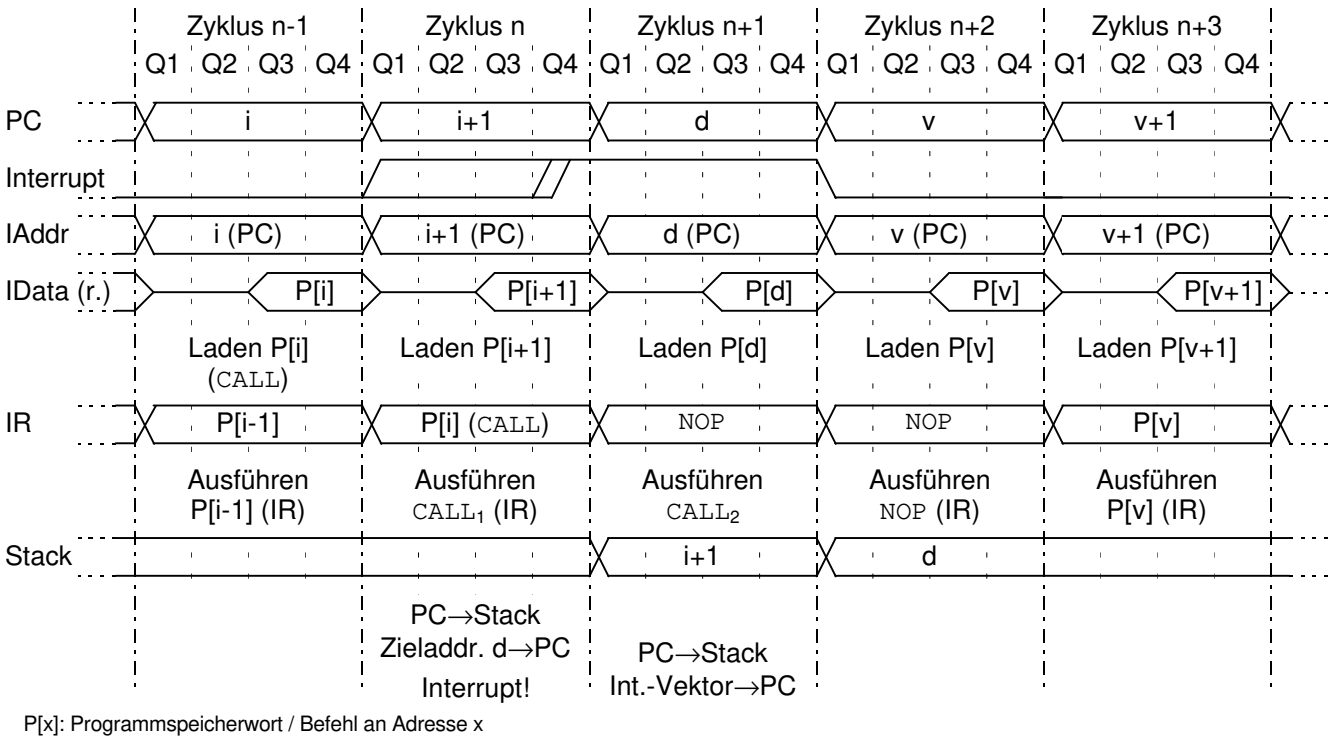


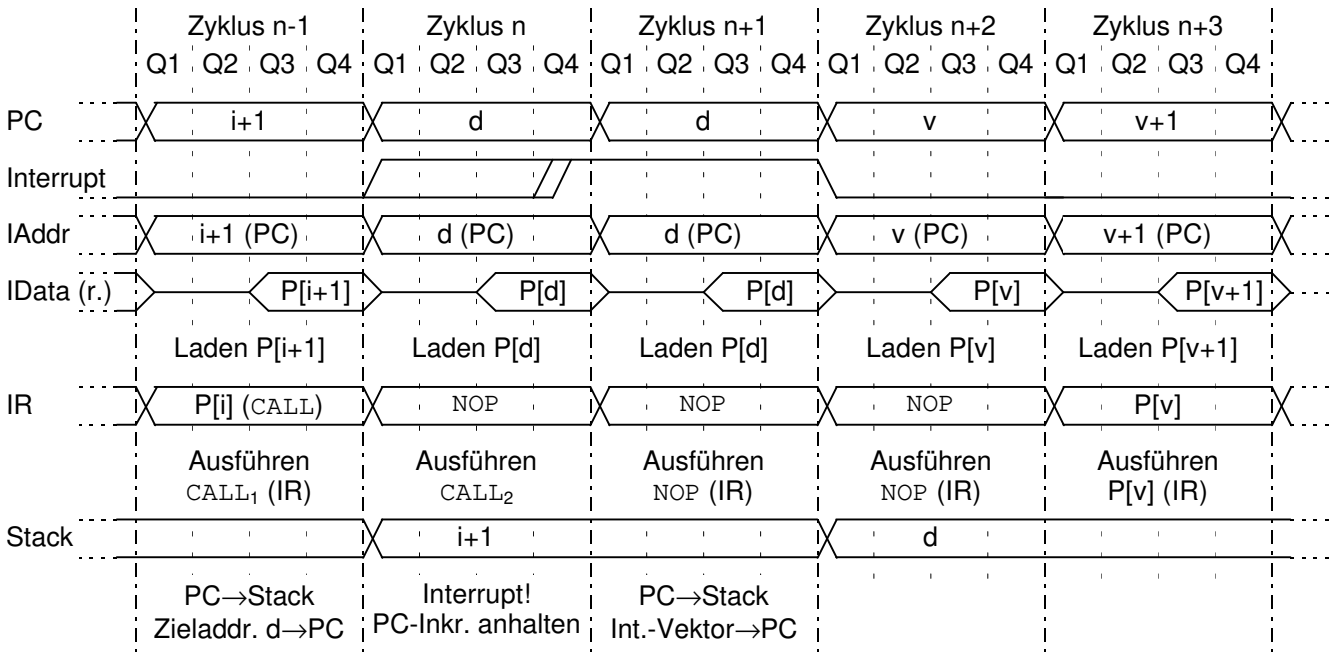
Abbildung 2.28: Interrupt im erstem Zyklus eines Unterprogrammaufrufs.

auftretenden Interrupt wird normalerweise u.a. der Programmzähler angehalten und unverändert in den nächsten Zyklus übernommen. Das Laden des Programmzählers mit einem neuen Wert durch den Unterprogrammaufruf hat aber hier in diesem speziellen Fall Priorität gegenüber dieser Interruptaktion. Im nächsten Zyklus wird dann nämlich die Verzweigung in die Interruptroutine durchgeführt und dabei der momentane Programmzählerinhalt für den späteren Rücksprung aus der Interruptroutine auf dem Stack gespeichert. Damit die Programmausführung korrekterweise bei der Rückkehr aus der Interruptroutine mit dem ersten Befehl des Unterprogramms und nicht erneut mit der Ausführung des eigentlichen Unterprogrammaufrufs fortfährt, muß der Programmzähler zu diesem Zeitpunkt bereits die Startadresse des Unterprogramms und nicht mehr die Adresse des Unterprogrammaufrufs beinhalten. Aus diesem Grund hat das Laden der Unterprogrammadresse Vorrang gegenüber dem Beibehalten des Programmzählerinhalts.

Die ansonsten ausgeführten Aktionen entsprechen der normalen Ausführung des Unterprogrammaufrufs und der Durchführung der Interruptaktionen. Im zweiten Zyklus wird neben der Verzweigung in die Interruptroutine der zweite Zyklus des Sprungs bzw. Unterprogrammaufrufs ausgeführt, d.h. z.B. je nach Instruktion die obere Hälfte des Programmzählers in das Register PCLATH gesichert.

In Abbildung 2.29 ist der Vollständigkeit halber das Timing bei Auftreten eines Interrupts im zweiten Zyklus der Ausführung eines Unterprogrammaufrufs dargestellt. Im Prinzip kommt es aber dabei zu keinen Konflikten zwischen der Befehlsausführung und den Interruptaktionen, so daß parallel beide Operationen normal und ohne Abweichungen von den bereits beschriebenen Abläufen, sowohl bei der Ausführung des Unterprogrammaufrufs als auch bei der Durchführung der Interruptaktionen, ablaufen.

- Wie in Abschnitt 2.8.3.3 bei der Beschreibung des Ablaufs von Rücksprüngen bereits dargelegt, unterscheidet sich die Ausführung von Rücksprüngen nicht prinzipiell von der Ausführung von Unterprogrammaufrufen. Bei Unterprogrammaufrufen wird der aktuelle Programmzählerinhalt auf den Stack geschrieben und die Zieladresse des



P[x]: Programmspeicherwort / Befehl an Adresse x

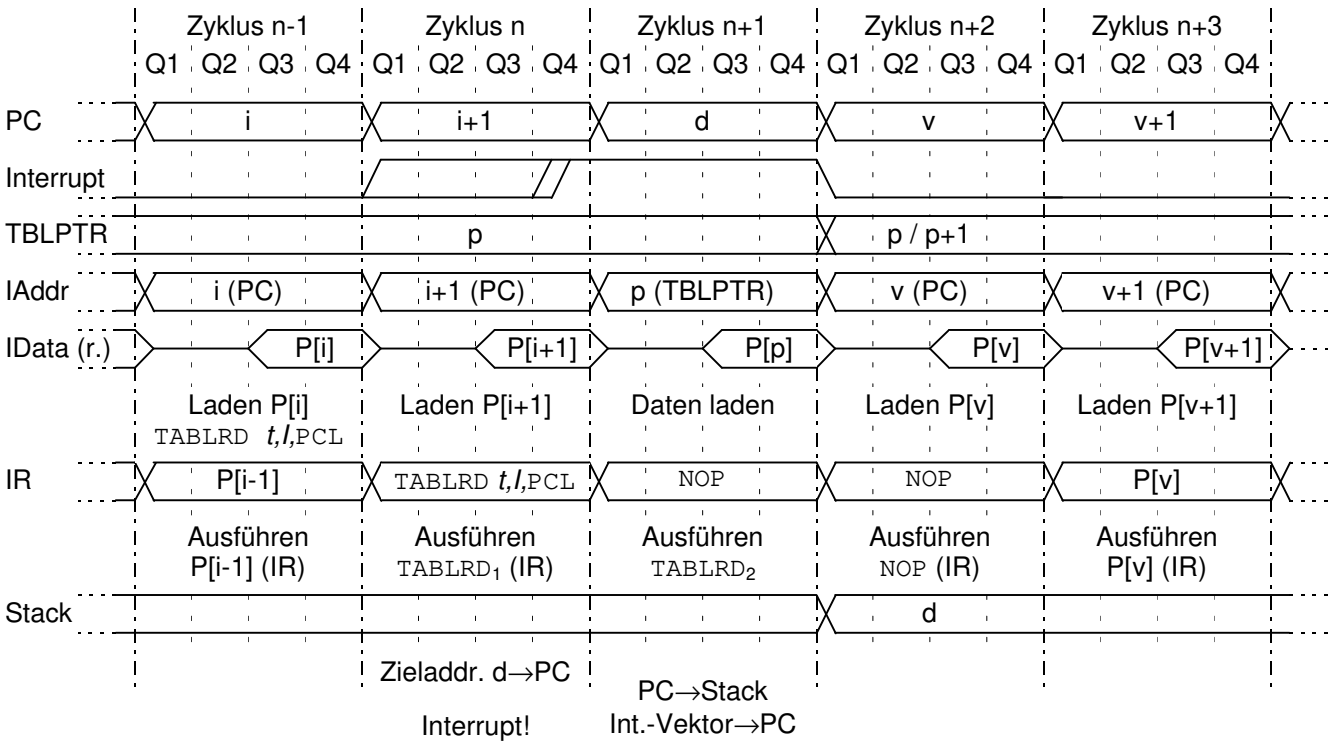
Abbildung 2.29: Interrupt im zweiten Zyklus eines Unterprogrammaufrufs.

entsprechenden Sprungs, die in den Programmzähler geladen wird, entweder direkt oder indirekt berechnet bzw. dem Befehlswort entnommen. Bei Rücksprüngen wird hingegen kein Wert auf den Stack geschrieben und statt dessen die in den Programmzähler zu schreibende Zieladresse dem Stack entnommen. Unter Berücksichtigung dieses Unterschieds entspricht ansonsten der prinzipielle Befehlsausführungsablauf von Rücksprüngen dem von Unterprogrammaufrufen. Dementsprechend sind, in Zusammenhang mit dem Auftreten von Interrupts bei der Ausführung von Rücksprüngen, die im vorangegangenen Punkt dargelegten Abläufe für Interrupts bei Sprüngen und Unterprogrammaufrufen unter Einbeziehung der oben beschriebenen Differenzen direkt übertragbar.

2.10.4.3 Spezialfälle

In Abschnitt 2.8.4 wurde im Rahmen der Beschreibung der Ausführung von Befehlen auf den Spezialfall der Kombination von Table-Operation und Sprung (Befehl `TABLRD` mit Registerargument PCL) hingewiesen, der die einzig mögliche Befehlskombination darstellt, deren Ausführung insgesamt drei Instruktionszyklen umfaßt. Treten allerdings während der Ausführung der entsprechenden Instruktion in einem der drei Zyklen Interruptanforderungen auf, so stellen deren Abarbeitung und die daraus resultierenden Abläufe keinen speziellen Sonderfall dar und können analog zu den oben beschriebenen Fällen behandelt werden. Wie leicht aus den im entsprechenden Abschnitt gemachten Darstellung ersichtlich ist, treten in keinem Fall weitere Konflikte zwischen der Ausführung der Instruktion und der Bearbeitung von dabei auftretenden Interrupts auf, so daß beide Operationen, wie bereits dargestellt, parallel und gleichzeitig durchgeführt werden können, ohne sich gegenseitig zusätzlich zu beeinflussen.

Bei Interrupts im ersten Zyklus der Instruktion (Abbildung 2.30) hat, wie bei Unterprogrammaufrufen, Sprüngen und Rücksprüngen das Laden der Zieladresse des Sprungs in den Befehlszähler Priorität gegenüber dem Beibehalten des Programmzählerinhalts im Rahmen der Interruptaktion. Auf der anderen Seite hat dann im nächsten Zyklus das Laden des Interruptvektors in den Programmzähler Vorrang gegenüber dem durch die Spezialinstruktion



P[x]: Programmspeicherwort / Befehl an Adresse x

Abbildung 2.30: Interrupt im ersten Zyklus eines `TABLRD` mit Registerargument PCL.

veranlaßten Beibehalten des Programmzählerinhalts. Im ersten und im zweiten Zyklus wird sowohl durch die Interruptaktion als auch durch den speziellen Befehl das Speichern eines `NOP` im Register IR veranlaßt. Im zweiten Zyklus wird durch die spezielle Table-Operation außerdem der Programmspeicherzugriff und ggf. eine Inkrementierung des Registers TBLPTR durchgeführt. Gleichzeitig wird durch den Interrupt die Verzweigung in die Interruptroutine vorbereitet bzw. angestoßen.

Interrupts, die während der Ausführung des zweiten Instruktionzyklus, in dem der Programmzähler bereits mit der Zieladresse geladen ist und u.a. der Programmspeicherzugriff durchgeführt wird, auftreten, sind davon ausgehend vollkommen gleich wie Interrupts im zweiten Zyklus einer normalen Table-Operation zu behandeln. Im dritten Zyklus der speziellen Table-Operation, der normalerweise nötig ist, um die Pipeline vor der Ausführung des ersten Befehls an der Zieladresse neu zu füllen, wird schließlich nur das bereits von der Table-Operation in den vorangegangenen Zyklen in das Register IR geladene `NOP` ausgeführt. Ein Interrupt in diesem Zyklus wird deshalb gemäß dem allgemeinen Fall wie ein Interrupt bei der Ausführung eines normalen Befehls abgearbeitet.

Abschließend soll noch kurz das Verhalten des Mikrocontrollerkerns bei der Rückkehr aus der Interruptroutine beschrieben werden, wenn während der Abarbeitung der Interruptroutine weitere Interrupts aufgetreten sind, die aufgrund der global gesperrten Interrupts nicht ausgelöst werden konnten. Ein Rücksprung aus einer Interruptroutine wird normalerweise mittels des Befehls `RETFIE` durchgeführt. Durch diesen Befehl werden automatisch gleichzeitig die Interrupts wieder global freigegeben, so daß wartende Interruptanforderungen, sofern die entsprechenden Quellen nicht ausmaksiert sind, als direkte Folge davon einen erneuten Interrupt auslösen.

In Abbildung 2.31 ist das zugehörige Timing für diesen Fall dargestellt. Am Ende des ersten Zyklus der Ausführung des Rücksprungs wird als direkte Folge das Bit GLINTD durch den Befehl zurückgesetzt. Dementsprechend wird das Interruptsignal bei wartenden

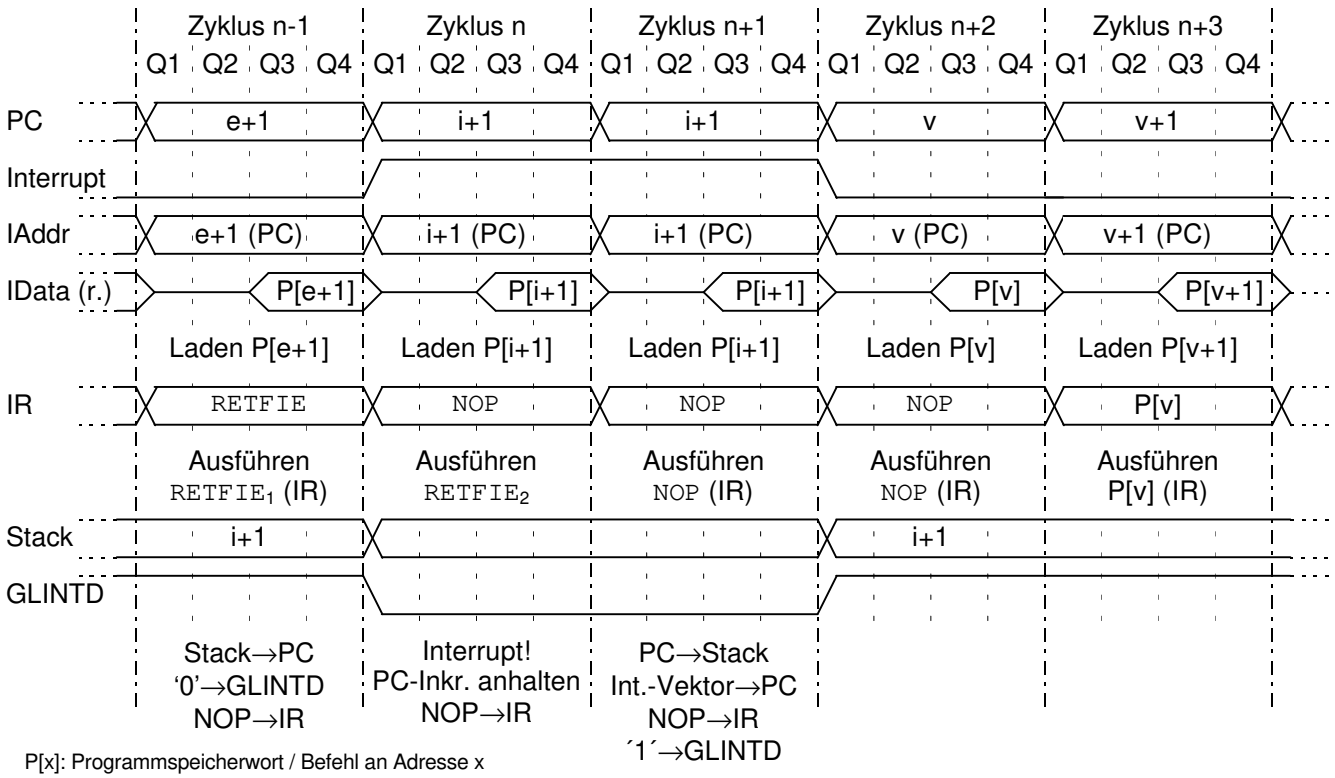


Abbildung 2.31: Interrupt durch Freigabe einer wartenden Anforderung bei Rückkehr aus Interruptroutine.

Anforderungen zu Beginn des nächsten Zyklus aktiv, d.h. im zweiten Zyklus der Ausführung des Rücksprungs wird ein Interrupt ausgelöst. Die Behandlung dieses Falles wurde schon im vorangegangenen Abschnitt besprochen. Anzumerken ist vielleicht, daß durch den erneuten Interrupt wieder direkt in die zugehörige Interruptroutine verzweigt wird und der normale Programmablauf erst gar nicht erneut aufgesetzt wird, d.h. es wird keine Instruktion des normalen, unterbrochenen Programms ausgeführt. Bei der Verzweigung in die Interruptroutine wird auf dem Stack wieder genau die selber Rücksprungadresse wie bei der vorangegangenen Interruptroutine abgelegt und die Interrupts durch Setzen des Bits GLINTD erneut global gesperrt.

2.10.5 Besonderheiten

In Zusammenhang mit dem Konfigurationsbit GLINTD und dessen Manipulation muß eine Besonderheit des Mikrocontrollerkerns beachtet werden, die so auch beim PIC17 auftritt. Im Prinzip handelt es sich dabei um ein fehlerhaftes Verhalten des Mikrocontrollerkerns im Hinblick auf das globalen Sperren von Interrupts aus einem Anwendungsprogramm heraus. Es wäre durchaus möglich gewesen, unter Einsatz von zusätzlicher Logik dieses Fehlverhalten des PIC17 bei der Entwicklung des Mikrocontrollerkerns zu umgehen. Aus Kompatibilitätsgründen und aufgrund des zusätzlichen Aufwands bei Berücksichtigung aller möglichen Befehlskombinationen wurde das Verhalten aber beim Mikrocontrollerkern unverändert übernommen.

Prinzipiell besteht aus dem Anwendungsprogramm heraus die Möglichkeit durch Setzen des Flags GLINTD Interrupts explizit zu sperren, z.B. vor Eintritt in zeitkritische Programmbereiche. Dazu wird mit einem beliebigen Befehl ein Schreibzugriff auf das Register CPUSTA durchgeführt und der neue Wert in das Register geschrieben. Tritt nun genau während der Ausführung des Befehls, an dessen Ende der Schreibzugriff auf das Register CPUSTA steht, eine Interruptbedingung ein, so wird zwar am Ende des Instruktionszyklus das

Bit GLINTD auf ´1´ gesetzt, der in diesem Zyklus begonnene Interrupt aber trotzdem durchgeführt und nicht abgebrochen. Die Auslösung des Interrupts verzweigt in die entsprechende Interruptroutine und setzt das bereits gesetzte Bit GLINTD erneut, ohne am Inhalt des Bits etwas zu ändern. Das Problem dabei ist nun, daß normalerweise bei der Rückkehr aus der Interruptroutine mittels des Befehls `RETFIE` das Bit GLINTD bei der Ausführung des Rücksprungs automatisch wieder zurückgesetzt wird. Das Anwendungsprogramm wird dann mit der Instruktion fortgesetzt, die direkt auf den Befehl folgt, der das Bit GLINTD auf ´1´ setzen sollte, wobei das Bit GLINTD aber trotzdem fälschlicherweise wieder den Wert ´0´ hat. Das bedeutet, daß das Anwendungsprogramm nach Rückkehr aus der Interruptroutine, trotz explizitem Sperren der Interrupts durch den bereits ausgeführten Befehl, mit freigegebenen Interrupts fortgeführt wird!

Der Schreibzugriff auf GLINTD und damit das Sperren der Interrupts erfolgt bei der Ausführung des entsprechenden Befehls mit der fallenden Flanke von CLK4 und wird deshalb erst zu Beginn des nächsten Zyklus wirksam. Tritt bei der Ausführung ein Interrupt ein, so sind zu diesem Zeitpunkt die Aktionen des ersten Interruptzyklus, d.h. Laden des Registers IR mit `NOP` und Anhalten des Programmzählers, als Reaktion auf die Interruptbedingung schon durchgeführt und können nicht mehr rückgängig gemacht werden. Intern ist zu diesem Zeitpunkt dann auch festgelegt, daß ein Interrupt ausgelöst wird und die entsprechenden Aktionen finden im zweiten Interruptzyklus statt. Zur hardwareseitigen Lösung des Problems müßte im zweiten Interruptzyklus vom Mikrocontrollerkern das erfolgte Sperren der Interrupts erkannt und daraufhin alle weiteren Interruptaktionen, d.h. die eigentliche Verzweigung in die Interruptroutine, abgebrochen werden. Da der nachfolgende Befehl im zweiten Interruptzyklus nochmals geladen wird, würde dies im Prinzip einem eingeschobenen zusätzlichen Zyklus entsprechen, in dem entsprechend dem im Register IR enthaltenen `NOP` keine Operation durchgeführt wird. Probleme würden in diesem Zusammenhang aber z.B. mit Table-Operationen, die auch auf das Register CPUSTA schreiben können, auftreten. Tritt ein Interrupt im ersten Zyklus eines `TABLRD`, bei dem das Register CPUSTA geschrieben wird, auf, so steht der bereits im ersten Zyklus geladene nachfolgende Befehl im zweiten Zyklus nicht mehr zur Verfügung und wird dort auch nicht nochmals geladen, da im zweiten Zyklus der Programmspeicherzugriff der Table-Operation stattfindet. Es müßte also ein dritter zusätzlicher Zyklus eingeschoben werden, in dem der nachfolgende Befehl abermals geladen wird.

Bei der hier entwickelten Version des Mikrocontrollerkerns läßt sich das Problem nur softwareseitig umgehen. Eine Möglichkeit besteht darin, Interrupts nicht über das Bit GLINTD, sondern über die vier Maskierungsbits zu sperren. Dies hat den Nachteil, daß bei der erneuten Freigabe der Interrupts ggf. bekannt sein muß, welche Interrupts freigegeben werden sollen, d.h. welche der vier Interrupts vor dem Maskieren freigegeben waren. Alternativ gibt es die Möglichkeit, nach dem Setzen des Bits GLINTD dieses erneut auszulesen und zu testen, ob das Bit tatsächlich gesetzt ist. Ist dies nicht der Fall, so muß die Aktion solange wiederholt werden, bis das Bit den Wert ´1´ hat und damit alle Interrupts gesperrt sind. Hat das ausgelesene Bit den Wert ´1´, dann ist sichergestellt, daß es nicht mehr zurückgesetzt werden kann, da dann bereits keine Interrupts mehr auftreten können.

2.11 RTCC-Modul

Das im Mikrocontrollerkern enthaltene Ereigniszählermodul (RTCC, engl. *Real Time Clock Counter*) erfüllt die Aufgaben eines Zählers bzw. eines Timers, um z.B. einerseits externe Ereignisse zu erfassen oder andererseits eine exakte Zeitbasis für zeitabhängige Funktionalitäten zur Verfügung zu stellen. Das RTCC-Modul enthält im Kern einen 16 Bit breiten Zähler, der in Abhängigkeit von den Einstellungen im Konfigurationsregister bei verschiedenen Ereignissen inkrementiert werden kann.

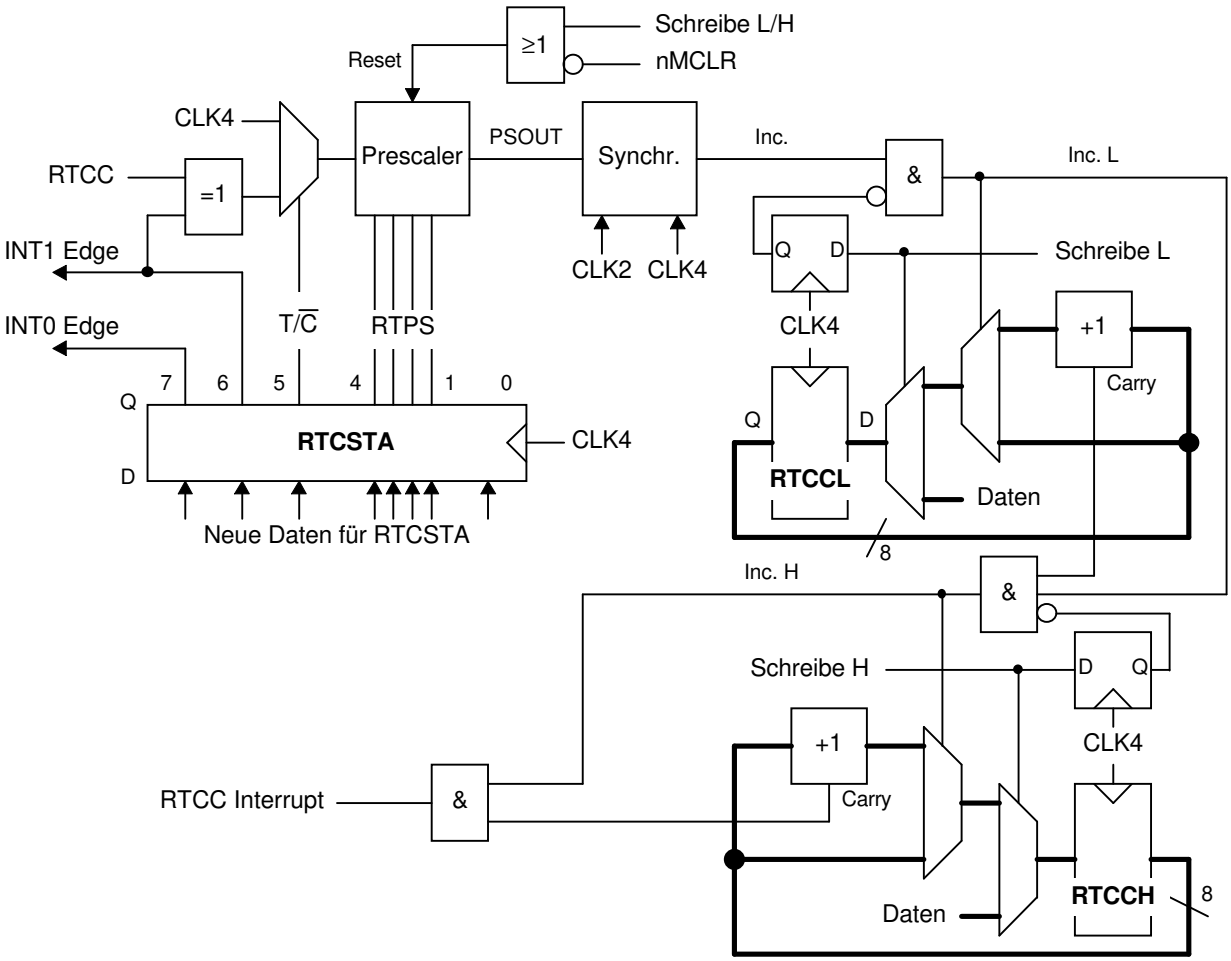


Abbildung 2.32: Blockdiagramm des RTCC-Moduls.

In Abbildung 2.32 ist in einer Komplettübersicht das Blockdiagramm des RTCC-Moduls dargestellt. Zur Konfiguration des Moduls dient das Register RTCSTA. Dieses Register ist an Adresse 13h in den Datenspeicherbereich eingeblendet und kann so beliebig aus dem Anwendungsprogramm heraus geschrieben und gelesen werden. Zusätzlich zu den Konfigurationsbits für das RTCC-Modul enthält das Register RTCSTA auch die Flankenselektionsbits für die Interruptlogik (siehe Abschnitt 2.10). Zu beachten ist, daß das Bit RTEDG aus Kompatibilitätsgründen zum originalen PIC17 sowohl zur Flankenselektion am Interruptpin INT1 als auch zur Flankenselektion für das RTCC-Modul dient.

Über die Bits 5 und 6 des Konfigurationsregisters RTCSTA wird ausgewählt, durch welches Signal Ereignisse für das RTCC-Modul erzeugt werden. Das ausgwählte Signal wird durch einen einstellbaren Vorteiler (engl. *Prescaler*) geleitet und schließlich auf die internen Taktverhältnisse synchronisiert. Wurde ein Ereignis erkannt, so wird dadurch das 16 Bit breite Zählregister inkrementiert. Ein Überlauf des Zählers von FFFFh nach 0000h erzeugt dabei eine Anforderung für einen RTCC-Interrupt, die an das Interruptmodul weitergeleitet und dort verarbeitet wird.

Die beiden 8 Bit breiten Hälften RTCCL und RTCCH des Zählregister sind normal in den Datenspeicherbereich eingeblendet und es kann beliebig auf diese beiden Zählerhälften getrennt zugegriffen werden. Eine Zählerhälfte erhält einen neuen Wert, wenn entweder der Zähler durch ein Ereignis inkrementiert oder die entsprechenden Zählerhälfte durch das Anwendungsprogramm beschrieben wird, wobei das Schreiben eines Wertes durch das Programm Vorrang gegenüber dem Inkrementieren des Zählers hat.

2.11.1 Ereignisauswahl

Das RTCC-Modul kann in zwei prinzipiellen Modi so konfiguriert werden, daß ein Ereignis entweder aufgrund einer Flanke am externen RTCC-Eingang des Mikrocontrollerkerns oder aufgrund einer internen Taktflanke, d.h. einmal pro Instruktionszyklus, ausgelöst wird. Die Einstellung des aktiven Modus erfolgt über das Bit 5 im Konfigurationsregister RTCSTA (Tabelle 2.14).

T/$\overline{C}$ (RTCSTA, Bit 5)	**Modusauswahl**
'0'	Timer Modus. Ereignisse durch internen Zyklustakt (CLK4).
'1'	Counter Modus. Ereignisse durch Flanken am externen Pin RTCC.

Tabelle 2.14: Auswahl des RTCC-Modus.

Im sog. *Timer* Modus wird ein Ereignis durch eine Flanke des internen Takts CLK4 ausgelöst, d.h. ein Ereignis wird jeweils am Ende eines Instruktionszyklus erzeugt. Im sog. *Counter* Modus wird dagegen ein Ereignis durch eine Flanke am externen Eingang RTCC des Mikrocontrollerkerns ausgelöst. Über das Bit 6 im Konfigurationsregister RTCSTA ist dabei einstellbar, ob Ereignisse bei fallenden oder steigenden Flanke am RTCC-Eingang ausgelöst werden sollen (Tabelle 2.15).

RTEDG (RTCSTA, Bit 6)	**Externe Ereignisse**
'0'	Ereignis bei fallender Flanke am Pin RTCC.
'1'	Ereignis bei steigender Flanke am Pin RTCC.

Tabelle 2.15: Flankenselektion für externe Ereignisse im Counter Modus.

Die Auswahl des RTCC-Modus und damit des Signals, das den weiteren Stufen des RTCC-Moduls zugeführt wird, erfolgt über einen Multiplexer, der, gesteuert von Bit 5 des Register RTCSTA, entweder das interne Taktsignal CLK4 oder den Eingang RTCC durchschaltet. In den weiteren Stufen (Vorteiler und Synchronisation) ist stets die fallende Flanke des so durchgeschalteten Signals die aktive und damit maßgebende Flanke. Um in Abhängigkeit vom Konfigurationsbit RTEDG ggf. steigenden Flanken am Pin RTCC als aktive Flanken festzulegen, wird über das Bit RTEDG ein vorgeschaltetes Exklusiv-Oder-Gatter angesteuert, das gegebenenfalls das RTCC-Eingangssignal invertiert. Bei Änderungen am Bit RTEDG durch Schreibzugriffe auf RTCSTA ist zu berücksichtigen, daß dadurch unter Umständen falsche Ereignisse ausgelöst werden können, wenn durch Aktivieren oder Deaktivieren der Invertierung das Ausgangssignal des XOR-Gatters bei gleichbleibendem Wert am RTCC-Eingang von ´1´ auf ´0´ wechselt.

Beim Mikrocontrollerkern sind im Gegensatz zum originalen PIC17 die Eingänge für den zweiten externen Interrupt und der RTCC-Eingang nicht in einem gemeinsamen Pin zusammengefaßt, sondern getrennt als Pins INT1 und RTCC ausgeführt. Aus Kompatibilitätsgründen wurde aber kein zusätzliches Flankenkonfigurationsbit eingeführt, so daß auch beim Mikrocontrollerkern das Bit 6 (RTEDG) im Register RTCSTA sowohl die aktive Flanke für den zweiten Interrupt (INT1) als auch für den Ereigniszähler konfiguriert.

Bei Änderungen am Konfigurationsbits muß deshalb berücksichtigt werden, daß sich sich der Wert des Flags auf beide Bereiche auswirkt.

2.11.2 Vorteiler

Nach der Auswahl des ereignisauslösenden Signals wird dieses durch einen einstellbaren Vorteiler geführt. Mit Hilfe dieses Vorteilers kann aus dem Anwendungsprogramm heraus das Verhältnis festgelegt werden, in dem fallende Flanken des Eingangssignals tatsächlich zu einem Inkrement des Ereigniszählers führen. Bei einem Teilverhältnis von 1:4 wird der Zähler z.B. im Endeffekt nur bei jeder vierten Flanke inkrementiert. Die Einstellung des Teilverhältnisses in Zweierpotenzen von 1:1 bis 1:256 erfolgt über die RTPS-Bits (Bit 1 bis Bit 4) im Register RTCSTA (Tabelle 2.16). Ist das Bit RTPS3 (Bit 4) ´1´, so ist das Teilverhältnis unabhängig vom Wert der anderen Bits mit dem Wert 1:256 maximal.

RTCSTA				Teilverhältnis
RTPS3 (Bit 4)	RTPS2 (Bit 3)	RTPS1 (Bit 2)	RTPS0 (Bit 1)	
´0´	´0´	´0´	´0´	1:1
´0´	´0´	´0´	´1´	1:2
´0´	´0´	´1´	´0´	1:4
´0´	´0´	´1´	´1´	1:8
´0´	´1´	´0´	´0´	1:16
´0´	´1´	´0´	´1´	1:32
´0´	´1´	´1´	´0´	1:64
´0´	´1´	´1´	´1´	1:128
´1´	x	x	x	1:256

Tabelle 2.16: Einstellung des Vorteilerfaktors.

Der Vorteiler ist durch einen asynchronen Zähler implementiert, der seinen Wert mit fallenden Flanken am Eingangspin inkrementiert. Das Ausgangssignal ist der Wert eines der internen Zählregisterbits, entsprechend dem eingestellten Teilverhältnis. Abbildung 2.33 zeigt das Timing zwischen Eingang und Ausgang des Vorteilers bei einem allgemeinen Teilverhältnis.

Der Vorteiler reagiert auf fallende Flanken des Eingangssignal und teilt diese im angegeben Verhältnis herunter. Im Rahmen der Periodizität des Eingangssignals ist das Ausgangssignal des Vorteilers symmetrisch. Ist das eingestellte Teilverhältnis 1:1 so wird das Eingangssignal unverändert durch den Vorteiler geleitet.

Ein Reset des Vorteilers und damit ein Rücksetzen des Wertes des internen Zählers wird bei einem Chip-Reset oder bei einem Schreibzugriff auf das Register RTCCL oder auf das Register RTCCH ausgelöst.

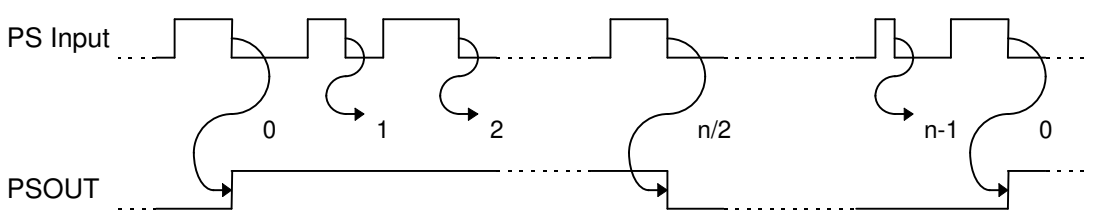


Abbildung 2.33: Arbeitsweise des Vorteilers bei einem Teilverhältnis von 1:n.

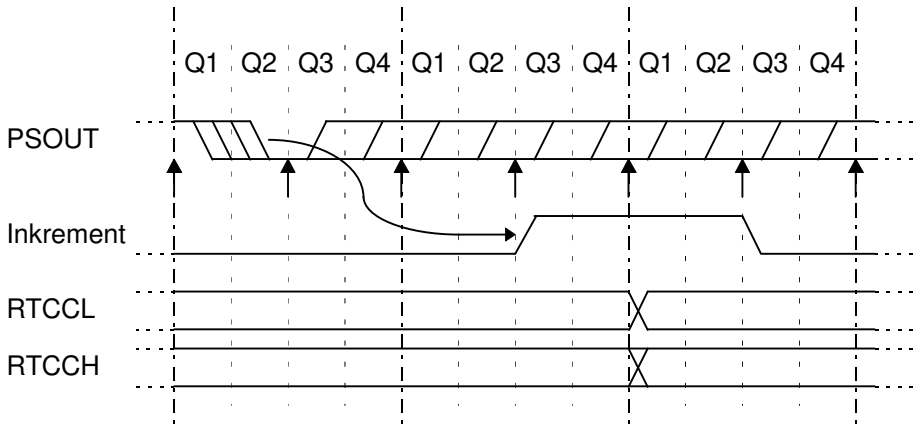


Abbildung 2.34: Timing (Synchronisation) für Flanke in der ersten Hälfte des Zyklus.

2.11.3 Synchronisation

Das asynchrone Ausgangssignal des Vorteilers PSOUT wird schließlich auf die internen Taktverhältnisse synchronisiert, und es wird durch eine Flankenerkennung das interne Steuersignal für die Inkrementierung des Ereigniszähler erzeugt. Zu diesem Zweck wird das Signal PSOUT zweimal pro Instruktionszyklus, am Ende der Taktphasen Q2 und Q4, abgetastet. Anhand der abgetasteten Werte wird eine Erkennung von fallenden Flanken des Signals durchgeführt.

In Abbildung 2.34 und Abbildung 2.35 ist das Timing für die Erkennung einer Flanke dargestellt. Das Erkennen einer abgetasteten fallenden Flanke aktiviert das interne Steuersignal und führt am Ende des zugehörigen Zyklus zu einem Inkrement des durch die Register RTCCL und RTCCH gebildeten Zählers. Das Register RTCCL wird dabei auf jeden Fall um eins erhöht, das Register RTCCH nur, wenn dabei das Register RTCCL von FFh nach 00h überläuft. Läuft gleichzeitig das Register RTCCH von FFh nach 00h über, so wird die entsprechende RTCC-Interruptleitung gekoppelt an das Inkrementsignal aktiviert.

Zu erkennen ist, daß, in Abhängigkeit vom Zeitpunkt des Auftretens der Flanke, prinzipbedingt eine mehr oder weniger große Verzögerung bis zum eigentlichen Inkrementieren des Zählerregister besteht. Aufgrund der nicht genau festgelegten Verzögerung kommt es so zu einer gewissen Ungenauigkeit beim Messen des Abstandes zweier Flanken über den Ereigniszähler. Weiterhin ist zu beachten, daß Impulse, die kürzer als ein Abtastabstand (ein halber Instruktionszyklus) sind, natürlich nicht erkannt werden können.

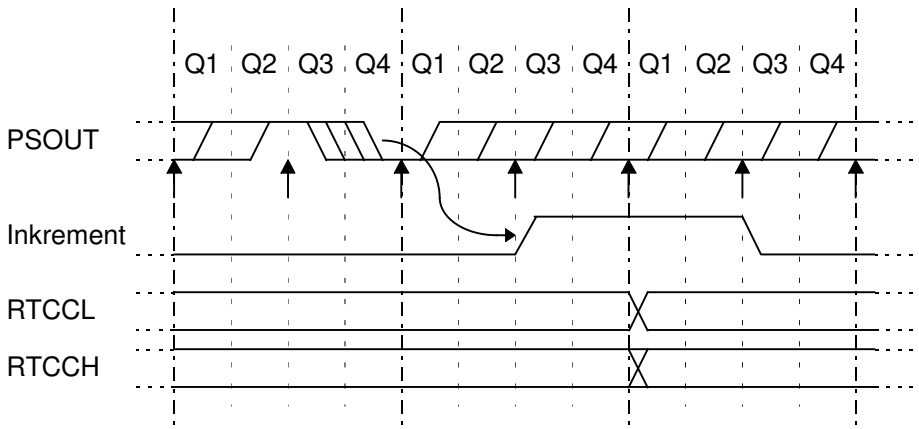


Abbildung 2.35: Timing (Synchronisation) für Flanke in der zweiten Hälfte des Zyklus.

2.11.4 Zählerregister

Wie bereits mehrfach erwähnt, bilden die beiden 8 Bit breiten Register RTCCL und RTCCH zusammen den 16 Bit breiten Ereigniszähler. Das vom Synchronisierer erzeugte Inkrement-Kontrollsignal steuert prinzipiell das Erhöhen des kompletten 16 Bit Zählers. Wird der Zähler inkrementiert, so wird das Register RTCCL um eins erhöht. Läuft dieses dabei über, so wird durch das Carry-Signal der unteren Zählerhälfte gleichzeitig auch das Register RTCCH um eins erhöht. Läuft wiederum dabei auch das Register RTCCH und damit der gesamte Zähler über, so dient das Carry-Signal der oberen Hälfte gleichzeitig dazu, dem Interruptmodul einen RTCC-Interrupt anzuzeigen.

Die beiden Zählerhälften RTCCL und RTCCH sind in den Datenspeicherbereich an Adresse 0Bh bzw. 0Ch eingeblendet, können also aus dem Anwendungsprogramm heraus parallel zur Zähloperation gelesen oder geschrieben werden. Zu beachten ist, daß Schreibzugriffe auf die beiden Register zu einem Reset des Vorteilers führen. Soll während eines Instruktionszyklus ein Register gleichzeitig sowohl inkrementiert als auch mit einem neuen Wert vom Programm geladen werden, so hat das Schreiben des neuen Registerwertes Priorität gegenüber dem Inkrementieren. Gegebenenfalls wird dabei aber trotz unterdrückter Inkrementierung ein Carry-Signal erzeugt, das wiederum zu einer Inkrementierung der oberen Hälfte bzw. zu einer Interruptanforderung führen kann.

Da jeweils nur eine 8 Bit breite Hälfte mit einem Befehl gelesen oder geschrieben werden kann, muß besonders berücksichtigt werden, daß sich der Zählerwert zwischen den Zugriffen auf die beiden Hälften verändern kann und so das Ergebnis einer Operation auf den kompletten Zähler u.U. verfälscht wird. Um beim Lesen des kompletten 16 Bit Zählers sicherzustellen, daß ein korrekter 16 Bit Wert ausgelesen wird, muß ggf. durch mehrmaliges Auslesen von RTCCL und Vergleich der gelesenen Werte überprüft werden, daß RTCCL während des Auslesens nicht von FFh nach 00h übergelaufen ist, d.h. daß die gelesenen Werte von RTCCL und RTCCH zusammenpassen [8].

Um das Schreiben eines neuen kompletten 16 Bit breiten Zählerwertes zu erleichtern, besitzen die RTCC-Register einen Mechanismus, der nach einem Schreibzugriff auf eine Zählerhälfte automatisch im darauffolgenden Instruktionszyklus das Inkrementieren der jeweiligen Hälfte verhindert. Dazu wird das Schreibkontrollsignal für einen Zyklus zwischengespeichert und dann im nächsten Zyklus dazu benutzt das Inkrement-Kontrollsignal zu maskieren. Wenn das Inkrementsignal einer Zählerhälfte so maskiert wird, dann führt das auch dazu, daß auf keinen Fall ein Carry-Signal erzeugt werden kann, so daß auch je nach Register auf keinen Fall die obere Zählerhälfte inkrementiert bzw. ein Interrupt erzeugt wird. Dieses Verhalten ist für den originalen PIC17 nicht genau spezifiziert und kann deshalb u.U. nicht genau dem Verhalten des PIC17 in solchen Fällen entsprechen. Auf jeden Fall ist so beim Mikrocontrollerkern sichergestellt, daß durch Schreiben von RTCCH und direkt danach von RTCCL in jedem Fall ein korrekter Zählerwert geschrieben wird. Das in [8] dazu vorgeschlagene Verfahren, zuerst RTCCL und dann RTCCH zu schreiben, kann zu einem falschen Wert führen, wenn in RTCCL dabei der Wert FFh geschrieben wird und wenn dann im Zyklus nach dem Schreibzugriff auf RTCCH, während das Inkrementieren von RTCCH noch maskiert ist, RTCCL durch einen Inkrement von FFh nach 00h überläuft (RTCCH behält seinen neuen Wert, während RTCCL bereits überläuft, ohne dabei RTCCH angepaßt zu inkrementieren).

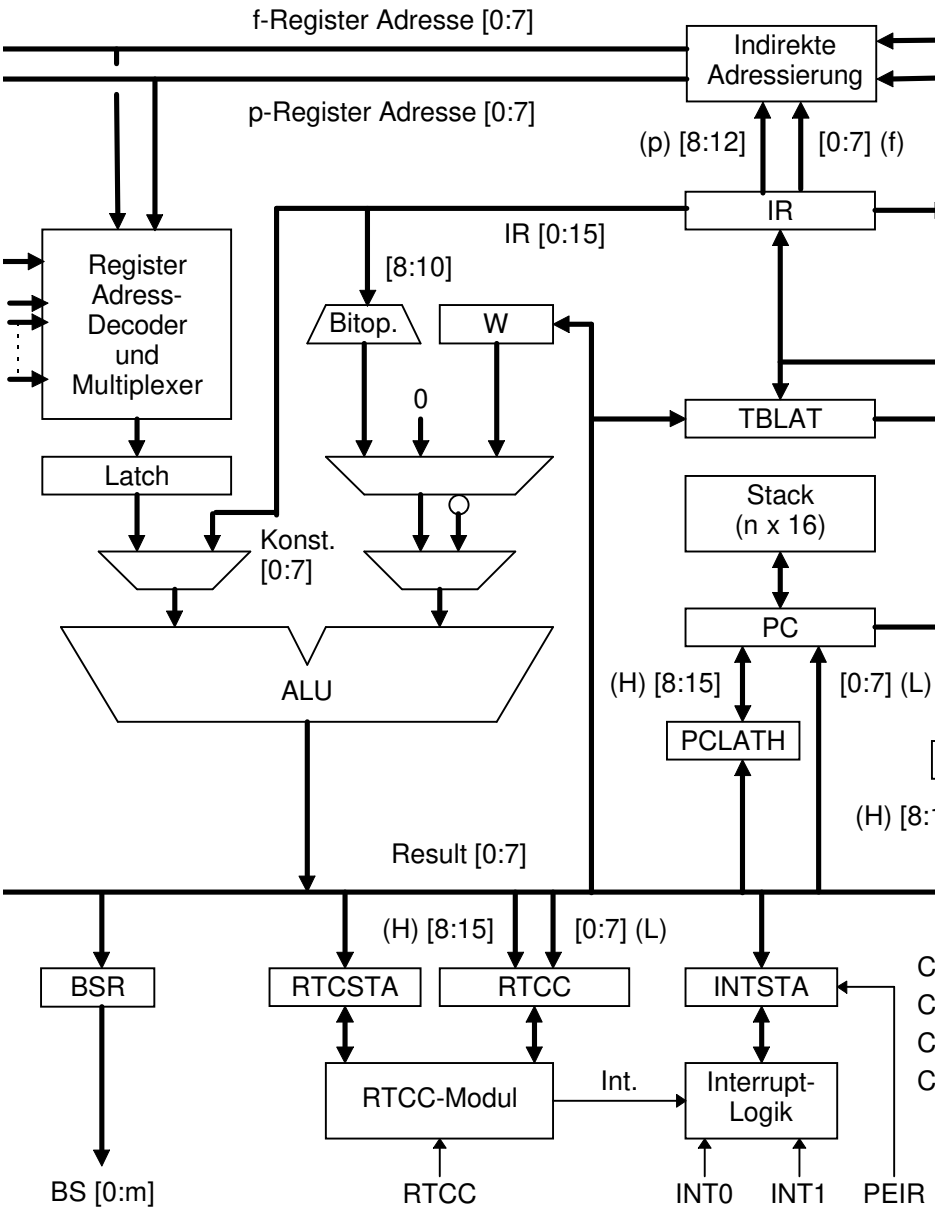


Abbildung 3.0: Blockdiagramm des Mikrocontrollerkerns

3 Architektur der RTL-Beschreibung

Dieses Kapitel beinhaltet die Beschreibung des Designs des Mikrocontrollerkerns auf RTL-Ebene, das, auf Basis der genauen Spezifikation des Mikrocontrollerkerns, die für die abschließende Synthese der eigentlichen Schaltung auf Gatterebene nötige Beschreibungsform der realen Schaltung darstellt. Dabei werden ausgehend von einer Darlegung der allgemeinen Designgrundlagen, die bei der Entwicklung des Mikrocontrollerkerns die Basis bildeten, die Struktur und der Aufbau des kompletten Designs, die Unterteilung in die einzelnen Teilblöcke und die Funktionalität der einzelnen Blöcke selbst dargelegt. Im Rahmen der RTL-Beschreibung wird der Mikrocontrollerkern ausschließlich mit Hilfe von Latches, Flip-Flops und kombinatorischer Logik aufgebaut und implementiert. Nach der Verifikation der Schaltung durch Vergleich mit der Simulation der Verhaltensbeschreibung wird die RTL-Beschreibung mit Hilfe von Synthesetools automatisch in eine Gatternetzliste umgewandelt. Die verifizierte Gatternetzliste bildet dann den Ausgangspunkt für die Plazierung und Verdrahtung der einzelnen Gatter auf dem Chip.

In Abbildung 3.0 ist das Design und der Aufbau des kompletten Mikrocontrollerkerns als Blockdiagramm in einer Gesamtübersicht dargestellt. Das Blockdiagramm ergibt sich als direkte Umsetzung der durch die Spezifikation festgelegten Funktionalität des Mikrocontrollerkerns unter Abwägung der verschiedenen möglichen Designalternativen. In den folgenden Abschnitten wird das Design genau beschrieben und die einzelnen Designentscheidungen dargelegt.

3.1 Grundlagen

In einem ersten Schritt wird bei der Entwicklung des Mikrocontrollerkerns auf RTL-Ebene die Spezifikation des Kerns in eine allgemeine Architektur umgesetzt, wobei grundlegende Designentscheidungen getroffen werden müssen. Ausgehend von dem so erhaltenen Blockdiagramm wird das Design, unter Berücksichtigung offensichtlicher interner Schnittstellen, in sinnvolle VHDL-Blöcke (Entities bzw. Components) aufgeteilt, die daraufhin der Reihe nach implementiert, getestet und schließlich in das Gesamtdesign integriert werden.

Im folgenden Abschnitt wird die grundlegende Architektur bis hin zur Einteilung in Unterblöcke erläutert. In den nachfolgenden Abschnitten werden dann die einzelnen Blöcke näher beschrieben.

3.1.1 Allgemeine Richtlinien

Bei der Entwicklung des Mikrocontrollerkerns wurden in den verschiedenen Ebenen eine Reihe von allgemeinen Designrichtlinien berücksichtigt, die im folgenden dargelegt werden sollen:

- Die im Mikrocontrollerkern vorhandenen Datenbusse werden ausschließlich über Multiplexer und nicht über Tristatetreiber angesteuert, d.h. die Auswahl des auf den Bus zu legenden Wertes erfolgt über einen Multiplexer und nicht über die Aktivierung eines ausgewählten Tristatetreibers. Die Realisierung über Multiplexer hat dabei Vorteile bezüglich der Geschwindigkeit und des Platzverbrauchs und ist unkritischer in der Implementation.
- Soweit irgend möglich wurden die internen Register aus Gründen der Platzersparnis nicht mit Flip-Flops sondern mit einfachen Latches implementiert. Dazu wurde der Datenpfad (s.u.) entsprechend ausgelegt, um Rückkopplungen und Instabilitäten zu umgehen. In vielen Fällen mußten aber die im Kern enthaltenen Register trotzdem über

Flip-Flops realisiert werden, da ihr Inhalt meist direkten Einfluß auf die Steuerung der kombinatorischen Logik des Kerns hat.

- Die Latches und Flip-Flops werden im allgemeinen für Schreibzugriffe über ein Taktsignal und einen separaten Steuereingang aktiviert. Der Enable Eingang von Latches wird dabei über die Und-Verknüpfung von Takt- und Steuereingang kontrolliert, d.h. das Taktsignal wird mit Hilfe des Steuersignals maskiert. Bei Flip-Flops besteht alternativ auch die Möglichkeit, das Taktsignal ständig anliegen zu lassen und dem Dateneingang einen über das Steuersignal aktivierten Multiplexer vorzuschalten, so daß in jedem Zyklus entweder die neu anliegenden Daten oder der bereits bisher im Register gespeicherte Wert in das Flip-Flop geschrieben wird. Die Ansteuerung der Flip-Flops über einen Multiplexer hat bei höherem Platzverbrauch den Vorteil, daß die Taktleitung nicht maskiert wird. Bei der Maskierung der Taktleitung mit der Steuerleitung kann es zu Problemen kommen, wenn bei aktivem Takt Glitches auf der Steuerleitung auftreten.

  Für die Flip-Flops im Mikrocontrollerkern wurden jeweils beide Versionen entwickelt, so daß bei der Synthese für jedes Register die eine oder die andere Realisierung gewählt werden kann. Normalerweise wird aus Platzgründen die Version mit maskierter Taktleitung (engl. *Gated Clock*) eingesetzt. Treten dabei Probleme auf, muß zur Multiplexerversion gewechselt werden.

3.1.2 Datenpfad

Aus dem Blockdiagramm des Mikrocontrollerkerns geht die Grundstruktur des im Kern realisierten Datenpfades für die allgemeine Abarbeitung eines normalen Befehls, der ein Register liest, eine Operation darauf ausführt und das Ergebnis schließlich wieder in ein Register schreibt, hervor: Ausgehend von den Registern des Datenspeichers, die einerseits direkt intern im Kern und andererseits durch externes RAM realisiert sind, wird über einen Registermultiplexer das vom Befehl adressierte Register ausgewählt (wobei ggf. vorher ein Lesezugriff auf das RAM stattgefunden hat), in der ALU verknüpft, das Ergebnis auf den Ergebnisdatenbus gelegt und von dort in das adressierte Zielregister geschrieben. Dies stellt gleichzeitig die normale Ausführung eines allgemeinen Befehls dar. In Abhängigkeit von dem tatsächlichen Befehl entfallen dabei ggf. einzelne Schritte, z.B. wenn kein Register gelesen oder kein Ergebnis geschrieben wird. Zentraler Punkt ist aber stets die rein kombinatorische ALU, durch die die Daten eines Befehls gereicht werden.

Dem Registermultiplexer ist ein Latch nachgeschaltet, das mit dem aktiven Takt CLK2 den ausgewählten Wert einlatcht und ihn für den Rest des Instruktionszyklus stabil speichert. In erster Linie wird dieses Latch zur Übernahme des in der Taktphase Q2 vom RAM gelieferten Registerwertes bei einem Lesezugriff benötigt. Gleichzeitig unterbricht das Latch aber auch eine eventuelle Rückkoppelschleife im Datenpfad bei gleichzeitigen Lese- und Schreibzugriffen eines Befehls auf ein intern im Kern als Latch realisiertes Register. Die Flip-Flops und Latches im Kern werden wie das RAM mit dem Signal CLK4 getaktet, d.h. Latches sind in Taktphase Q4 transparent und Flip-Flops übernehmen einen neuen Wert am Ende des Instruktionszyklus mit der fallenden Flanke von CLK4. Liest und schreibt ein Befehl nun das selbe, intern als Latch realisierte Register, so würde sich ohne das dem Registermultiplexer nachgeschaltete Latch der an die ALU angelegte Wert mit dem Schreibzugriff in Q4 (das Latch ist dann in Q4 transparent), in Abhängigkeit von der Laufzeit, entsprechend der Rückkopplung verändern. Gleichzeitig unterbindet die Teilung des Datenpfades in zwei getrennt getaktete Hälften mögliche Probleme in Bezug auf Taktjitter, die sonst bei der Taktsignalverteilung im Mikrocontrollerkern auftreten können, wenn Quell- und Zielregister eines durch kombinatorische Logik gebildeten Datenweges mit dem selben Signal getaktet werden.

Die restlichen Dateneingänge der ALU werden direkt oder indirekt über zusätzliche Logik aus den Registern IR oder W gespeist, weshalb diese Register als Flip-Flops implementiert sein

müssen. Prinzipiell könnte man die beiden Register auch als Latches implementieren und dem entsprechenden ALU Eingang zur Trennung ein zusätzliches Latch vorschalten. Da aber das Register IR als Basis für den Instruktionsdekoder auf jeden Fall als Flip-Flop realisiert werden muß, würde sich aus der im Endeffekt resultierenden Aufteilung des W Registers in zwei Latches kein Platzvorteil gegenüber der Realisierung als Flip-Flop ergeben.

3.1.3 Blockhierarchie

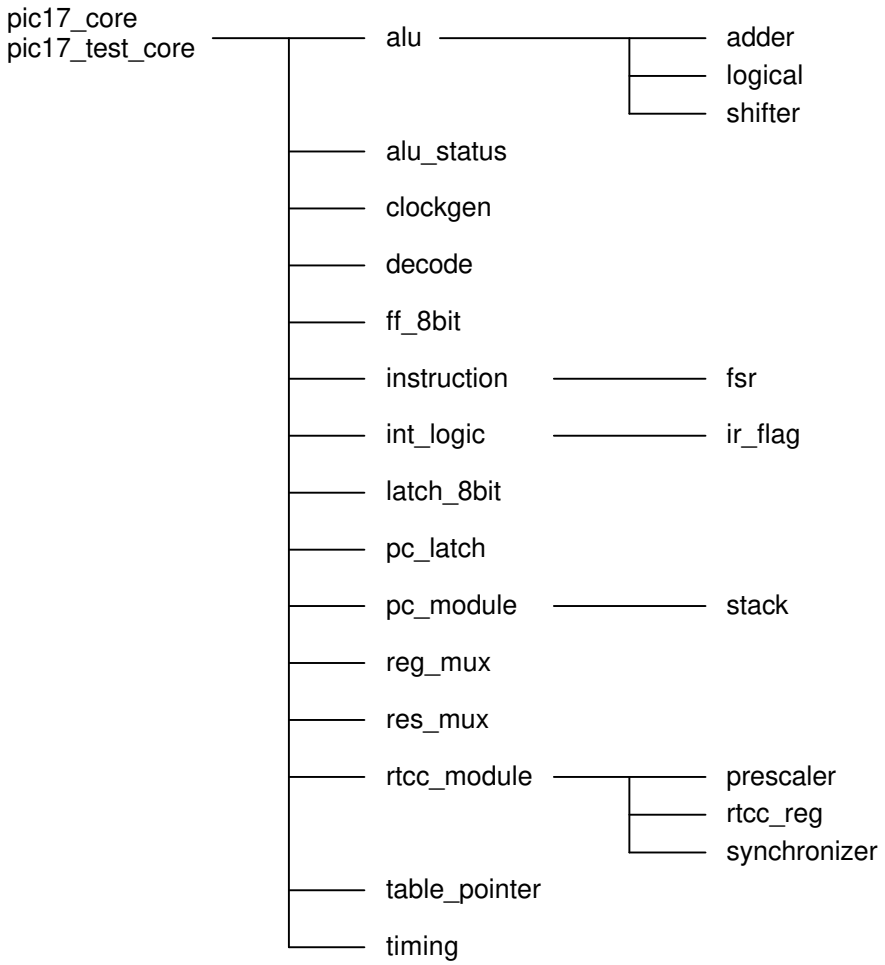


Abbildung 3.1: Blockhierarchie der RTL-Beschreibung.

In Abbildung 3.1 ist die Aufteilung des Mikrocontrollerkerndesigns in die verschiedenen Unterblöcke wiedergegeben. An oberster Stelle steht der eigentliche Mikrocontrollerkern als Block *pic17_core* bzw. als Block *pic17_test_core* mit zusätzlichen Testausgängen für die automatische Simulation. Entsprechend der Funktionalität und Zusammengehörigkeit ist der Mikrocontrollerkern vollständig in einzelne Unterblöcke aufgeteilt, die dann im obersten Block nur passend verschaltet werden. Neben der nötigen minimalen Logik zur Verschaltung der Blöcke implementiert der oberste Block nur das Register BSR und die Multiplexer an den Adreßausgängen zu Programm- und Datenspeicher.

Der Mikrocontrollerkernblock enthält an seiner Schnittstelle zwei VHDL-Generics, über die zwei Bereiche des Mikrocontrollerkerndesigns beeinflußt werden können:

- Die tatsächlich physikalisch implementierte Breite des Registers BSR kann festgelegt werden. Dementsprechend paßt sich auch die Breite des Ausgangs BS an der Schnittstelle des Mikrocontrollerkerns der Breite des Registers an. Über dieses Generic

läßt sich, je nach Applikation und Verwendung des BS Ausgangs dessen Wertebereich und damit z.B. die Anzahl zusätzlicher Bänke im Datenspeicherbereich festlegen. Beim originalen PIC17 besitzt das Register eine Breite von 4 Bit. Alternativ kann das Register ggf. auch als 8 Bit breites Allzweckregister zur Speicherung von Daten zweckentfremdet werden.

- Die Stacktiefe (beim originalen PIC17 hat der Stack eine Tiefe von 16 Speicherplätzen) kann festgelegt werden. Je nach der von der Applikation maximal benötigten Stacktiefe kann so entweder eine Platzersparnis erreicht oder ein kritisches Überlaufen des Stacks verhindert werden. Der Wert dieses Generics wird an das Programmzählermodul weitergereicht und dort ausgewertet.

Ausgehend von der hier abgebildeten Blockhierarchie werden in den nachfolgenden Abschnitten die verschiedenen einzelnen Blöcke und ihre Unterblöcke in ihrer Funktionalität und Umsetzung detailliert beschrieben.

3.2 Taktgenerator

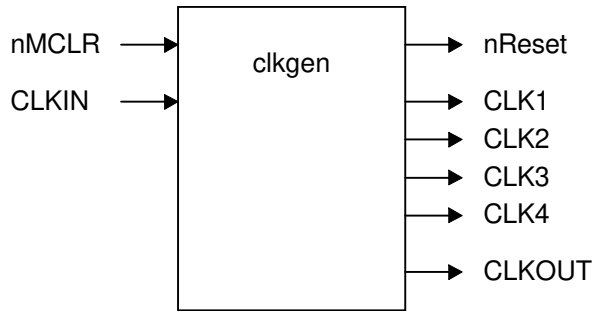


Abbildung 3.2: Taktgenerator.

Der Taktgenerator (Block *clkgen*) erfüllt die folgenden beiden Aufgaben im Design des Mikrocontrollerkerns:

- Erzeugung der vier internen Taktsignale CLK1 bis CLK4 und des Mikrocontrollerkerntaktausgangs CLKOUT aus dem an den Mikrocontrollerkern angelegten Systemtakt CLKIN. Der angelegte Systemtakt wird entsprechend heruntergeteilt. Jedem erzeugten Taktsignal ist ein eigenes Ausgangs-Flip-Flop direkt, d.h. ohne weitere Ausgangslogik, zugeordnet, um möglichst saubere Taktsignale ohne Jitter und zeitliche Verschiebungen gegeneinander zu erhalten.

- Synchronisation des externen Reset-Eingangs (Pin nMCLR) und Bereitstellung des daraus generierten internen Reset-Signals *nReset* (Aktiv low). Das Reset-Signal wird durch Abtastung so synchronisiert, daß eine steigende Flanke am Pin nMCLR, d.h. eine Auflösung eines Reset, intern stets bis zur nächsten steigenden Flanke des Systemtaktes verzögert wird. Dadurch wird verhindert, daß ein Reset genau bei einer potentiell aktiven, fallenden Flanke eines der vier internen Taktsignale inaktiv wird, was zu einer Setup- oder Hold-Verletzung an dem entsprechenden Flip-Flop führen kann.

  Fallende Flanken am Reset-Eingang, d.h. die Aktivierung des Reset-Zustands, werden hingegen direkt an das interne Reset-Signal durchgereicht und nicht weiter synchronisiert, um eine Verzögerung der Reset-Aktivierung zu vermeiden.

  Bei aktivem Reset werden die internen Taktsignale im Zustand Q1 gehalten, d.h. CLK1 ist high und alle anderen Taktsignale sind low. Nach Auflösung des internen Reset, stets synchron mit einer steigenden Flanke von CLKIN, geht dann der

Taktgenerator mit der darauffolgenden fallenden Flanke von CLKIN in den nächsten internen Taktzustand, d.h. in den Zustand Q2 über und fährt normal fort.

3.3 Schnittstellentiming

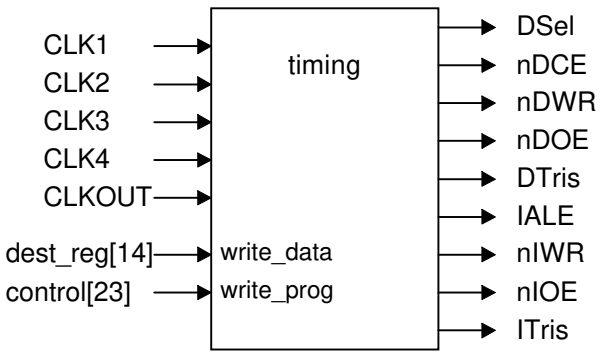


Abbildung 3.3: Schnittstellentiming.

Der Block *timing* erzeugt die Steuersignale an den externen Schnittstellen zu Programm- und Datenspeicher. Als Basis für das Timing der Steuersignale dienen die vom Taktgenerator erzeugten internen Taktsignale. In Abhängigkeit von zwei zusätzlichen Steuereingängen fährt der Block auf der jeweiligen Schnittstelle entweder einen Schreib- oder einen Lesezyklus. Im Normalfall werden auf beiden Schnittstellen nur Lesezyklen erzeugt. Durch Aktivieren der entsprechenden Steuerleitung für den Datenspeicher (angesteuert durch die vom Registermultiplexer dekodierte Steuerleitung für Schreibzugriffe auf den Datenspeicher, siehe Abschnitt 3.5) bzw. für den Programmspeicher (angesteuert durch den Instruktionsdekoder, siehe Abschnitt 3.16, im zweiten Zyklus einer `TABLWT` Instruktion) werden im aktuellen Instruktionszyklus Schreibzugriffe ausgelöst. An der Schnittstelle zum Datenspeicher führt dies dazu, daß, zusätzlich zum normalen Lesezyklus, in der zweiten Hälfte des Instruktionszyklus ein Schreibzugriff auf den Datenspeicher erzeugt wird. An der Schnittstelle zum Programmspeicher wird hingegen in diesem Fall anstatt des Lesezugriffs ein Schreibzugriff über den kompletten Instruktionszyklus gefahren.

3.4 W Register

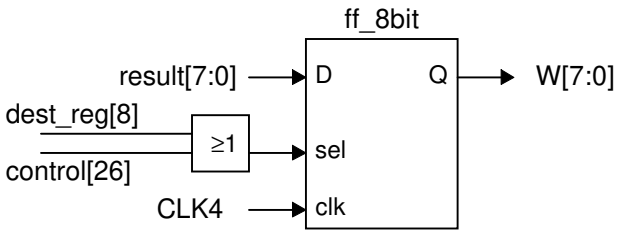


Abbildung 3.4: W Register.

Das W Register ist über ein allgemeines Master-Slave Flip-Flop mit einer Breite von 8 Bit (Block *ff_8bit*) implementiert. Die aktive Flanke des Flip-Flops ist die fallenden Flanke des internen Taktes CLK4. Neben den normalen Daten- und Takteingängen besitzt das Flip-Flop einen zusätzlichen Kontrolleingang, über den Schreibzugriffe auf das Register gesteuert werden. Normalerweise übernimmt das Register die angelegten Daten nicht bei jeder fallenden

Taktflanke. Ist hingegen das angelegte Steuersignal bei der fallenden Taktflanke aktiv, so werden die neuen Daten in das Register geschrieben.

Beim Mikrocontrollerkern kann das W Register in zwei Fällen neu beschrieben werden: Entweder durch explizite Angabe des Registers als Parameter einer entsprechenden Instruktion oder implizit durch den Befehl selbst festgelegt. Dementsprechend werden die beiden zugehörigen Steuersignale mit der vom Registermultiplexer dekodierten Zieladresse einerseits bzw. dem speziellen Steuerausgang des Instruktionsdekoders andererseits durch eine Oder-Verknüpfung auf den Steuereingang des Registers geführt.

Da der Ausgang und damit der Inhalt des W Registers direkt auf den Eingang der rein kombinatorischen ALU führt, muß der Wert während des ganzen Instruktionszyklus unverändert anliegen, weswegen das W Register als Flip-Flop und nicht als Latch realisiert werden muß.

3.5 Registermultiplexer

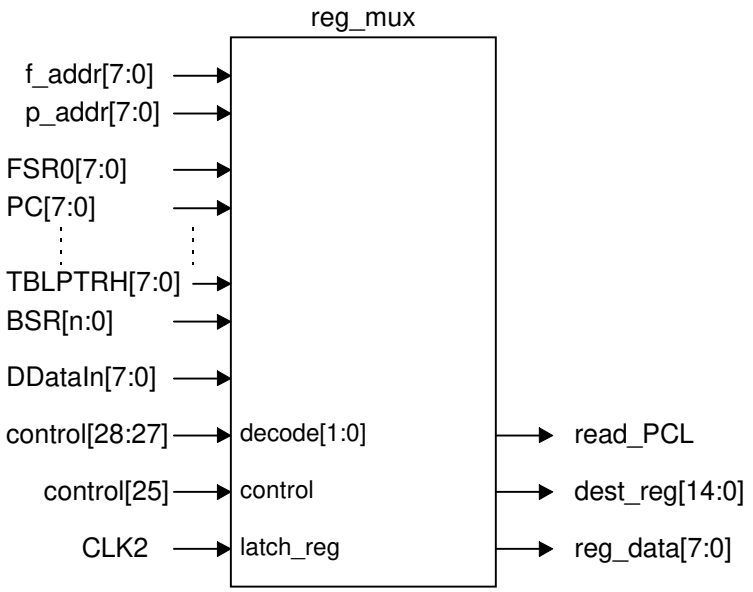


Abbildung 3.5: Registermultiplexer.

Der Registermultiplexer (Block *reg_mux*) umfaßt sowohl die Dekodierung und Auswahl der Quell- und Zielregister einer Operation als auch das Latch zum Zwischenspeichern des ausgewählten Quelloperanden. Im Kern besteht der Block einerseits aus einem Multiplexer mit nachgeschaltetem Latch für den Quelloperanden und andererseits aus einer reinen Adreßdekodierung für das Zielregister. In Abhängigkeit von den Adreß- und Steuereingängen wird für den Quelloperanden festgelegt, welcher Dateneingang (eines der Register oder der Datenspeicherwert) auf den Eingang des Latches durchgeschaltet wird. Das Latch ist in der Taktphase Q2 aktiv, übernimmt diesen Wert und speichert ihn für die restliche Zeit. Am Ausgang des Registermultiplexers erscheint der Inhalt des Latches und wird von dort auf den entsprechenden Eingang der ALU geführt. Zusätzlich erzeugt der Registermultiplexer ein Signal, das aktiv wird, wenn als Quelloperand speziell das Register PCL gewählt wird. Zusammen mit dem zugehörigen Steuersignal für Schreibzugriffe auf das Register PCL werden damit beim Programmzählermodul und beim PC Latch die besonderen Aktionen in Zusammenhang mit Zugriffen auf das Register PCL gesteuert.

Auf der anderen Seite dekodiert der Registermultiplexer die angelegte Zieladresse, so daß für jedes mögliche Ziel (Spezialregister im Kern oder Datenspeicherregister) ein Steuersignal

vorhanden ist (Steuervektor *dest_reg*), das zur Aktivierung von Schreibzugriffen auf die entsprechenden Ziele verwendet werden kann.

Eingang	Funktion	Steuerleitung
decode[0]	Selektion der Adresse des Quellregisters (reg_data). '1': f_addr '0': p_addr	control[27]
decode[1]	Selektion der Adresse des Zielregisters (dest_reg). '1': f_addr '0': p_addr	control[28]

Tabelle 3.1: Steuerleitungen des Registermultiplexers.

Zur Auswahl der Quell- und Zieloperanden einer Operation werden dem Registermultiplexer von der Instruktionsverarbeitung die entsprechenden Adressen, bei denen bereits die indirekte Adressierung aufgelöst ist, zugeführt. Zusätzlich wird dem Registermultiplexer über drei Kontrolleitungen vom Instruktionsdekoder mitgeteilt, welche der beiden Adressen die Quell- und Zieloperanden jeweils auswählen. In Tabelle 3.1 ist die Belegung der beiden Steuerleitungen wiedergegeben, über die die Zuordnung von Adreßeingängen zu Quell- und Zielregistern festgelegt wird. Zu beachten ist, daß diese beiden Kontrolleitungen des Instruktionsdekoders auch den Instruktionsverarbeitungsblock ansteuern (siehe Abschnitt 3.15) und dort deren Bedeutung noch etwas genauer definiert ist. Der dritte Steuereingang des Registermultiplexers dient schließlich dazu festzulegen, ob die Zieladresse überhaupt dekodiert werden soll. Ein Teil der Instruktionen schreibt nämlich überhaupt nicht auf ein Register und in diesem Fall darf keine der Schreibsteuerleitungen aktiviert werden.

3.6 ALU

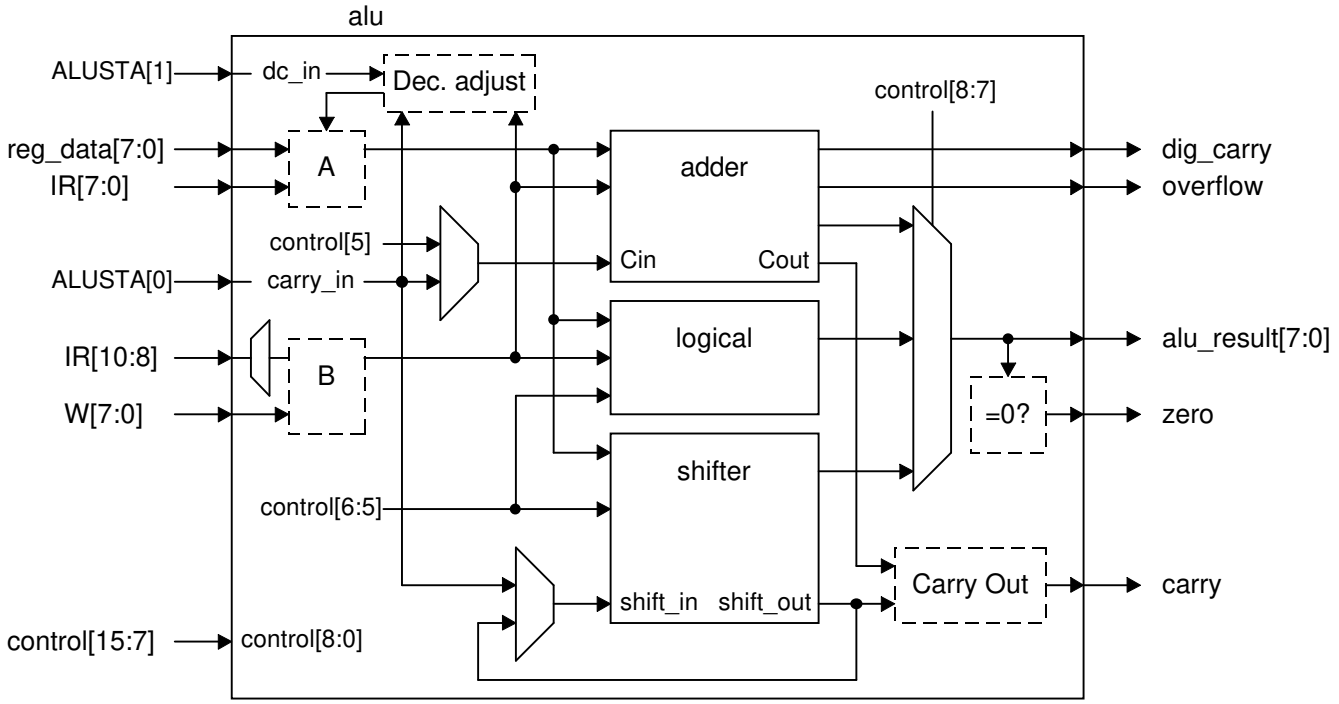


Abbildung 3.6: ALU.

Die ALU bildet das zentrale Element für die Befehlsausführung des Mikrocontrollerkerns. Bei fast allen Befehlen, inklusive der Datentransferbefehle, werden Daten in der ALU verknüpft und das an deren Ausgang erhaltene Ergebnis ausgewertet. Eine Ausnahme hiervon bilden nur die Table-Operationen, bei denen der Inhalt des Table-Latches direkt ausgewertet werden muß. Zur Vereinfachung werden die Werte der Register TBLATH und TBLATL aber nicht durch die ALU geleitet und statt dessen über einen der ALU nachgeschalteten Ergebnismultiplexer auf den Ergebnisdatenbus gegeben (siehe Abschnitt 3.8).

Die ALU ist rein aus kombinatorischer Logik aufgebaut und besteht im Kern aus drei parallelen Zweigen zur Durchführung von Additionsaufgaben, logischen Operationen und Schiebeoperationen, die der leichten Austauschbarkeit wegen als separate Unterblöcke realisiert sind (Blöcke *adder*, *logical* und *shifter*). Vor den drei Blöcken sitzt eine Logik zur Auswahl der bzw. des Operanden aus einer Reihe von möglichen Eingangsdaten in Abhängigkeit von der durchzuführenden Operation. Da die drei Blöcke parallel arbeiten, ist ihnen ein Multiplexer nachgeschaltet, der die Auswahl des eigentlichen ALU Ergebnisses aus den Ausgängen der drei Blöcke erlaubt. Des weiteren enthält die ALU zusätzliche Logik zur Selektion des Carry Eingangs für den Addierer, zur Auswahl des einzuschiebenden Wertes bei Schiebeoperationen und zur Erkennung eines ALU Ergebnisses von 00h, d.h. zur Ansteuerung des entsprechenden Statusausgangs. Gesteuert wird die Logik der einzelnen Teile der ALU, d.h. die von der ALU durchgeführte Operation, durch den Instruktionsdekoder über eine Reihe von Kontrolleitungen.

Der Addierer des Mikrocontrollerkerns ist als einfacher Ripple-Carry Addierer implementiert, der die zwei 8 Bit breiten Operanden und den Carry Eingang zu einem ebenfalls 8 Bit breiten Summanden mit Überlaufbit (Carry Ausgang) addiert. Die Auswahl eines Ripple-Carry Designs für den Addierer stellt dabei einen Kompromiß bezüglich Platzbedarf und Geschwindigkeit dar. Sollte die Geschwindigkeit des Addierers nicht ausreichen, läßt sich der Unterblock leicht durch ein komplexeres und schnelleres Design ersetzen. Zusätzlich zum Carry Ausgang steuert der Addierer die beiden Statusausgänge für Digit Carry und Overflow an, die ja nur bei Additionsoperationen von Bedeutung sind. Der Digit Carry Ausgang ist dabei der Wert des Übertrags von der unteren Bytehälfte in die obere Bytehälfte im Rahmen der Addition. Der Overflow Ausgang, der einen Überlauf bei vorzeichenbehafteter Addition anzeigt, wird durch die Exklusiv-Oder Verknüpfung der Überträge in das Vorzeichenbit hinein und aus dem Vorzeichenbit heraus gebildet.

Die Blöcke *logical* und *shifter* werden durch zwei gemeinsame Steuerleitungen kontrolliert. Der Block für logische Operationen macht, je nach Zustand der Steuerleitungen, eine von vier logischen Verknüpfungen zwischen zwei Operanden und stellt das Ergebnis am Ausgang zur Verfügung. Die Zuordnung der Steuersignale zu den einzelnen Operationen wurde dabei mit Hilfe von Karnaugh-Diagrammen so festgelegt, daß der Flächenbedarf bei entsprechender Optimierung minimal ist. Der Shifter führt, in Abhängigkeit von den Steuerleitungen, eine Schiebeoperation (Left Shift, Right Shift, Swap) auf einen Operanden durch. Am Eingang des Shifters wird das einzuschiebende Bit angelegt und am Ausgang erscheint neben dem Ergebnis das herausgeschobene Bit.

Ausgehend vom Befehlssatz des Mikrocontrollerkerns wurde das Design der ALU, d.h. speziell die Auswahl der Operanden und die von den Blöcken durchführbaren Aktionen, so ausgelegt, daß alle nötigen Operationen möglichst effektiv abgedeckt sind. Die Operation der ALU wird vom Instruktionsdekoder über einen 9 Bit breiten Kontrolleitungsvektor gesteuert. Im folgenden wird die Bedeutung der einzelnen Steuerleitungen und damit die genaue Funktionalität der ALU beschrieben.

Mit Ausnahme von Schiebeoperationen werden in der ALU im allgemeinen zwei Operanden zu einem Ergebnis verknüpft. Die beiden Operanden werden intern als Operand A und Operand B bezeichnet, wobei dem Shifter nur Operand A zugeführt wird. Jeder Operand kann über jeweils zwei Steuerleitungen aus einer von vier möglichen Quellen ausgewählt werden. Die Auswahl des Operanden A erfolgt über die ersten beiden Steuerleitungen des

Vektors, wie in Tabelle 3.2 zusammengefaßt. Je nach Operation kann zwischen dem über den Registermultiplexer vom Befehl ausgewählten Registerargument, dem Wert 00h, einer ggf. im Befehl kodierten Konstanten oder dem speziellen Wert zur Durchführung einer dezimalen Anpassung nach einer Addition zweier BCD-kodierten Zahlen (Befehl `DAW`) ausgewählt werden.

control[1] (control[8])	control[0] (control[7])	Operand A
'0'	'0'	Register (reg_data[7:0])
'0'	'1'	'00000000'
'1'	'0'	Konstante (IR[7:0])
'1'	'1'	Dec. adjust (00h / 06h / 66h)

Tabelle 3.2: Auswahl des ersten Operanden.

Die letze Möglichkeit bildet dabei einen Spezialfall. In diesem Fall richtet sich nämlich der tatsächliche Wert des Operanden A (00h, 06h oder 66h) nach dem Wert des Operanden B und den beiden Statuseingängen Carry und Digit Carry. Der Spezialfall wird für die Ausführung des Befehls `DAW` benötigt, bei dem der Inhalt des Registers W, das dabei über den Operanden B ausgewählt wird, durch Addition eines passenden Wertes korrigiert wird, wobei sich der Korrektursummand nach dem Ergebnis (Additionsergebnis und Statusbits) der potentiell vorangegangenen BCD-Addition richtet. Zu diesem Zweck besitzt die ALU einen zusätzlichen Logikblock, der den Operanden B und die beiden Statusbits auswertet und der Logik zur Auswahl des Operanden A über diverse Steuerleitungen den notwendigen Korrektursummanden mitteilt. Wird über die Steuerleitungen des Instruktionsdekoders die entsprechende Aktion ausgewählt, so liefert die Auswahllogik für den Operanden A den passenden Wert.

Das Vorgehen der Logik zur Auswahl des Korrekturwerts ist dabei wie folgt: Ausgehend von der Tatsache, daß vor der Addition jedes Nibble, d.h. jede Bytehälfte der beiden zu addierenden BCD-kodierten Zahlen einen binären Wert enthält, der im Bereich von 0h bis 9h liegt (die Werte Ah bis Fh sind nicht erlaubt), liegt das Additionsergebnis für jedes einzelne Nibble theoretisch im Bereich von 0 bis 18. In einem Nibble sind aber nur Werte von 0 bis 15 darstellbar und es kann deswegen zu Überträgen in das nächsthöhere Nibble kommen. Unter Berücksichtigung eines möglichen Übertrags in ein Nibble hinein liegt der Ergebnisbereich jedes Nibbles also sogar zwischen 0 und 19. Jedes Nibble muß nun nach einer dezimalen Addition so korrigiert werden, daß es entsprechend der Dezimalstelle des Ergebnisses wieder im Bereich von 0h bis 9h liegt und alle größeren Werte einen Übertrag in die nächste Dezimalstelle, d.h. in das nächste Nibble, auslösen. Für jedes Nibble des Ergebnisses lassen sich die folgenden vier Fälle unterscheiden:

- Ergebnis ≤ 9h, d.h. auch kein Übertrag in nächstes Nibble: Das Ergebnis stellt den korrekten BCD-Wert für diese Nibble dar und muß deshalb nicht korrigiert werden.
- Ah ≤ Ergebnis ≤ Fh und kein Übertrag: Das Ergebnis liegt im Bereich von 10 bis 15, d.h. das Nibble muß durch Addition von 6 so korrigiert werden, daß es entsprechend der niederwertigen Dezimalstellen im Bereich von 0 bis 5 liegt. Durch die Korrekturaddition findet nach der eigentlichen Addition ein Übertrag in das nächste Nibble statt.

- Übertrag und Ergebnis ≤ 2 bzw. Ergebnis ≤ 3 (bei Übertrag in das Nibble hinein): Ergebnis im Bereich von 16 bis 18 bzw. 19 (plus Übertrag) wird durch Addition von 6 auf den korrekten Bereich 6 bis 8 bzw. 9 gebracht. Ein Übertrag, der schon vorher bei der eigentlichen Addition stattgefunden hat, tritt dabei nicht auf.
- Übertrag und Ergebnis > 2 bzw. Ergebnis > 3: Bei Addition zweier korrekter BCD-Zahlen darf dieses Ergebnis nicht auftreten, d.h. die Reaktion darauf ist nicht definiert und kann beliebig festgelegt werden.

Daraus läßt sich, unter Ausnutzung des Freiheitsgrades für den letzten Fall, ein einfaches Prinzip zur Auswahl des Korrekturwertes derart ableiten, daß immer dann der Wert 6 hinzuaddiert wird, wenn das Ergebnis größer als neun ist oder wenn ein Übertrag aus dem Nibble heraus stattgefunden hat. Zu berücksichtigen ist aber, daß bei der Korrektur des vorangegangenen Nibbles ein Übertrag in ein Nibble hinein auftreten kann, durch den dann im Grenzfall erst einer Korrektur ausgelöst wird. Dieser Fall tritt ein, wenn nach der eigentlichen Addition das Nibble den Wert 9 enthält — also nicht korrigiert würde — und bei der Korrektur des niederen Nibbles ein Übertrag dazu führt, daß das Nibble den Wert 10 erhält, d.h. korrigiert werden muß. Im Prinzip muß bei der Korrektur also beginnend mit dem niederwertigsten Nibble so vorgegangen werden, daß, unter Berücksichtigung möglicher zusätzlicher Überträge in nachfolgende Nibble, der Reihe nach Nibble für Nibble korrigiert wird. Da beim Mikrocontrollerkern die Operation für das Byte aber in einem Schritt durchgeführt werden muß, wird die Auswahl des Korrekturfaktors für die obere Bytehälfte (in die untere Bytehälfte kann bei der Korrektur kein Übertrag hinein stattfinden, so daß dort keine speziellen Vorkehrungen nötig sind) noch während der Operation feststellen, ob bei der Korrektur der unteren Hälfte ein Übertrag entsteht. Prinzipiell wäre es möglich dazu den Digit Carry Ausgang des Addierers auszuwerten. Da dann aber die zweite Operandenhälfte überhaupt erst festgelegt werden kann, nachdem dieses Ausgangssignal stabil ansteht, würde sich die Maximallaufzeit auf dem kritischen Pfad durch den Addierer um die Laufzeit von Korrekturlogik und Operandenauswahl erhöhen. Dies hätte u.U. direkte Auswirkungen auf die maximale Taktfrequenz. Aus diesem Grund wird bei der Korrektur der oberen Hälfte so vorgegangen, daß im Falle eines Nibblewertes von neun ein Übertrag bei der Korrektur der unteren Hälfte vorausgesagt wird. Dieser findet dann und nur dann statt, wenn der Nibblewert der unteren Hälfte, d.h. der untere Hälfte des Operanden B, vor der Korrektur größer als neun ist. Entsprechend dem oben erwähnten Prinzip wird die untere Bytehälfte genau dann korrigiert, wenn der Wert der unteren Hälfte des Operanden B größer als neun ist oder wenn das Digit Carry Flag im Statusregister high ist, d.h. bei der vorangegangen Addition ein Übertrag aus der unteren Hälfte heraus stattgefunden hat. Die obere Hälfte wird dahingegen korrigiert, wenn die obere Hälfte des Operanden B größer als neun, der Wert des Carry Flags im Statusregister gleich ‘1’ oder die obere Hälfte von B gleich neun und die untere Hälfte größer als neun ist. Tritt bei der Korrektur ein Übertrag aus der oberen Hälfte und damit aus dem Byte heraus auf, so wird dadurch das Carry Flag gesetzt, falls dies nicht bereits schon vorher der Fall war. Bei Ausführung der Operation für dezimale Korrektur wird also der Carry Ausgang der ALU aus der Oder-Verknüpfung von Carry Eingang der ALU, d.h. dem Inhalt des Carry Statusbits, und Carry Ausgang des Addierers gebildet.

Die Auswahl des Operanden B (Tabelle 3.3) erfolgt im Prinzip in zwei Stufen. In der ersten Stufe wird über zwei Kontrolleitungen ein Multiplexer angesteuert, der es erlaubt, entweder den Wert 00h, den Inhalt des Registers W oder den aus dem Demultiplexen des im Befehlswort kodierten Bitoperanden gewonnenen Wert auszuwählen. Der Bitoperand stellt dabei einen 8 Bit breites Wort dar, bei dem genau ein Bit, dessen Wertigkeit im Befehlswort angegeben ist, den Wert ‘1’ und alle anderen Bits den Wert ‘0’ haben. In der zweiten Stufe läßt sich dann über eine weitere Kontrolleitung gesteuert der zuvor ausgewählte Wert invertieren, so daß insgesamt zwischen den Werten 00h, FFh, dem invertierten oder nicht invertierten

Inhalt des Registers W oder dem invertierten oder nicht invertierten Bitoperanden ausgewählt werden kann.

Eingang	Funktion Operand B	Steuerleitung
control[2]	Auswahl '00000000' für Operand B. '1': 00h '0': Nicht festgelegt.	control[9]
control[3]	Auswahl Wert für Operand B. '1': Bit-Operand '0': W Register	control[10]
control[4]	Invertiere Operand B. '1': Invertieren '0': Nicht invertieren.	control[11]

Tabelle 3.3: Auswahl des zweiten Operanden.

Mit Hilfe der Kontrolleitung, die auch das Invertieren des Operanden B steuert, wird parallel der Multiplexer für das einzuschiebende Bit, der dem entsprechenden Eingang des Shifters vorgeschaltet ist, wie in Tabelle 3.4 dargestellt, angesteuert. Es läßt sich so je nach gewünschter Operation entweder der Wert des Carry Statusbits im Register ALUSTA oder direkt das auf der anderen Seite hinausgeschobene Bit einschieben. Da bei einer Schiebeoperation der Operand B ohne Belang ist und da bei einer arithmetischen oder logischen Operation, bei der der Operand B ausgewertet wird, das einzuschiebende Bit keine Bedeutung hat, kann ohne Einschränkungen dieselbe Steuerleitung für beide Aufgaben verwendet werden, um so die Anzahl der Steuerleitungen und die Komplexität des Instruktionsdekoders zu minimieren.

control[4] (control[11])	shift_in
'0'	shift_out
'1'	carry_in

Tabelle 3.4: Auswahl des Shifter-Eingangs.

control[6] (control[13])	control[5] (control[12])	Cin (adder)	logical	shifter
'0'	'0'	'0'	$A \oplus B$	shift left
'0'	'1'	'1'	$A \vee B$	shift right
'1'	'0'	carry_in	$\overline{A} \wedge B$	swap
'1'	'1'	carry_in	$A \wedge B$	swap

Tabelle 3.5: Steuerung der ALU-Operationen.

Die Operation der drei parallel arbeitenden zentralen Blöcke der ALU wird über die nächsten beiden Kontrolleitungen gesteuert (Tabelle 3.5). Da die drei Blöcke parallel geschaltet sind, können die beiden Steuerleitungen für die Steuerung aller drei Blöcke

gleichzeitig verwendet werden und haben damit je nach tatsächlich ausgewählter Operation (s.u.) eine ganz unterschiedliche Bedeutung. Beim Addierer wird über die beiden Signale der dem Carry Eingang des Addierers vorgeschaltete Multiplexer angesteuert. Der Carry Multiplexer erlaubt die Auswahl des Inhalts des Carry Statusbits oder der Werte ´0´ oder ´1´. Der Block für logische Operationen wird über die Signale so gesteuert, daß er eine von vier logischen Operationen durchführt. Es handelt sich dabei um die drei logischen Grundoperationen Und, Oder und Exklusiv-Oder und um eine spezielle Operation zur Ausführung des Befehls `BTFSS`. Der Shifter schließlich führt in Abhängigkeit der Steuersignale entweder eine Schiebeoperation des Operanden nach links, nach rechts oder eine Vertauschung der oberen und unteren Bytehälften des Operanden (Befehl `SWAPF`) durch.

control[8] (control[15])	**control[7] (control[14])**	**alu_result[7:0]**	**Carry Out**
‘0’	‘0’	adder[7:0]	Cout (adder)
‘0’	‘1’	logical[7:0]	shift_out
‘1’	‘0’	adder[7:0]	Cout (adder)
‘1’	‘1’	shifter[7:0]	shift_out

Tabelle 3.6: Auswahl der ALU-Operation.

Tabelle 3.6 zeigt schließlich die Belegung der letzten beiden Steuerleitungen zur Auswahl des ALU Ergebnisses am Ausgangsmultiplexer. Es wird das Ergebnis eines der drei Blöcke auf den Ausgang des ALU Ergebnisses durchgeschaltet. In Abhängigkeit vom ausgewählten Ergebnis und damit in Abhängigkeit von der durchgeführten Operation wird auch entweder der Carry Ausgang des Addierers (bei Additionsoperationen) oder das aus dem Shifter herausgeschobene Bit (bei Schiebeoperationen) auf den Carry Ausgang der ALU gegeben. Bei logischen Operation ist der Wert des Carry Ausgangs egal, da er dort keine Bedeutung hat und nicht ausgewertet wird. Zu beachten ist, daß bei Auswahl einer Additionsoperation und bei Durchführung einer dezimalen Korrektur (abhängig von der Auswahl des Operanden A) ein spezieller Wert, wie im entsprechenden obigen Absatz beschrieben, auf den Carry Ausgang gegeben wird.

In der nachfolgenden Tabelle 3.7 sind abschließend alle Befehle des Mikrocontrollerkerns mit der dazugehörigen ALU Operation und den nötigen Werten für die ALU Steuerleitungen zusammengefaßt.

Befehl	Funktion	Operation (control[15:12])	$\overline{B}$ / Shift-In (control[11])	Operand B (control[10:9])	Operand A (control[8:7])
`MOVFP`	r + 00h	adder, A+B+’0’ (‘x0 00’)	B / - (‘0’)	00h (‘x1’)	Reg. (‘00’)
`MOVLB`	k + 00h	adder, A+B+’0’ (‘x0 00’)	B / - (‘0’)	00h (‘x1’)	Konst. (‘10’)
`MOVPF`	r + 00h	adder, A+B+’0’ (‘x0 00’)	B / - (‘0’)	00h (‘x1’)	Reg. (‘00’)
`MOVWF`	W + 00h	adder, A+B+’0’ (‘x0 00’)	B / - (‘0’)	W (‘00’)	00h (‘01’)
`TABLRD`	-	- (‘xx xx’)	- / - (‘x’)	- (‘xx’)	- (‘xx’)
`TABLWT`	r + 00h	adder, A+B+’0’ (‘x0 00’)	B / - (‘0’)	00h (‘x1’)	Reg. (‘00’)
`TLRD`	-	- (‘xx xx’)	- / - (‘x’)	- (‘xx’)	- (‘xx’)
`TABLWT`	r + 00h	adder, A+B+’0’ (‘x0 00’)	B / - (‘0’)	00h (‘x1’)	Reg. (‘00’)

ADDLW	k + W	adder, A+B+'0'	('x0 00')	B / -	('0')	W	('00')	Konst.	('10')
ADDWF	r + W	adder, A+B+'0'	('x0 00')	B / -	('0')	W	('00')	Reg.	('00')
ADDWFC	r + W + carry	adder, A+B+c	('x0 1x')	B / -	('0')	W	('00')	Reg.	('00')
ANDLW	k AND W	logical, A $\wedge$ B	('01 11')	B / -	('0')	W	('00')	Konst.	('10')
ANDWF	r AND W	logical, A $\wedge$ B	('01 11')	B / -	('0')	W	('00')	Reg.	('00')
CLRF	00h + 00h	adder, A+B+'0'	('x0 00')	B / -	('0')	00h	('x1')	00h	('01')
COMF	r XOR FFh	logical, A $\oplus$ B	('01 00')	$\overline{B}$ / -	('1')	00h	('x1')	Reg.	('00')
DAW	Dec. adjust W	adder, A+B+'0'	('x0 00')	B / -	('0')	W	('00')	Adj.	('11')
DECF	r + FFh	adder, A+B+'0'	('x0 00')	$\overline{B}$ / -	('1')	00h	('x1')	Reg.	('00')
INCF	r + 00h + '1'	adder, A+B+'1'	('x0 01')	B / -	('0')	00h	('x1')	Reg.	('00')
IORLW	k OR W	logical, A $\vee$ B	('01 01')	B / -	('0')	W	('00')	Konst.	('10')
IORWF	r OR W	logical, A $\vee$ B	('01 01')	B / -	('0')	W	('00')	Reg.	('00')
MOVLW	k + 00h	adder, A+B+'0'	('x0 00')	B / -	('0')	00h	('x1')	Konst.	('10')
NEGW	$\overline{W}$ + 00h + '1'	adder, A+B+'1'	('x0 01')	$\overline{B}$ / -	('1')	W	('00')	00h	('01')
RLCF	shift left r, carry	shifter, A<<	('11 00')	- / carry	('1')	-	('xx')	Reg.	('00')
RLNCF	shift left r	shifter, A<<	('11 00')	- / shift	('0')	-	('xx')	Reg.	('00')
RRCF	sh. right r, carry	shifter, A>>	('11 01')	- / carry	('1')	-	('xx')	Reg.	('00')
RRNCF	shift right r	shifter, A>>	('11 01')	- / shift	('0')	-	('xx')	Reg.	('00')
SETF	00h + FFh	adder, A+B+'0'	('x0 00')	$\overline{B}$ / -	('1')	00h	('x1')	00h	('01')
SUBLW	k + $\overline{W}$ + '1'	adder, A+B+'1'	('x0 01')	$\overline{B}$ / -	('1')	W	('00')	Konst.	('10')
SUBWF	r + $\overline{W}$ + '1'	adder, A+B+'1'	('x0 01')	$\overline{B}$ / -	('1')	W	('00')	Reg.	('00')
SUBWFB	r + $\overline{W}$ + carry	adder, A+B+c	('x0 1x')	$\overline{B}$ / -	('1')	W	('00')	Reg.	('00')
SWAPF	swap r	shifter, swap A	('11 1x')	- / -	('x')	-	('xx')	Reg.	('00')
XORLW	k XOR W	logical, A $\oplus$ B	('01 00')	B / -	('0')	W	('00')	Konst.	('10')
XORWF	r XOR W	logical, A $\oplus$ B	('01 00')	B / -	('0')	W	('00')	Reg.	('00')
CALL	-	-	('xx xx')	- / -	('x')	-	('xx')	-	('xx')
CPFSEQ	r+$\overline{W}$+'1' (zero?)	adder, A+B+'1'	('x0 01')	$\overline{B}$ / -	('1')	W	('00')	Reg.	('00')
CPFSGT	r+$\overline{W}$+'1' ($\overline{Z}\wedge$C?)	adder, A+B+'1'	('x0 01')	$\overline{B}$ / -	('1')	W	('00')	Reg.	('00')
CPFSLT	r+$\overline{W}$+'1' ($\overline{C}$?)	adder, A+B+'1'	('x0 01')	$\overline{B}$ / -	('1')	W	('00')	Reg.	('00')
DECFSZ	r + FFh (zero?)	adder, A+B+'0'	('x0 00')	$\overline{B}$ / -	('1')	00h	('x1')	Reg.	('00')
DCFSNZ	r + FFh ($\overline{\text{zero}}$?)	adder, A+B+'0'	('x0 00')	$\overline{B}$ / -	('1')	00h	('x1')	Reg.	('00')
GOTO	-	-	('xx xx')	- / -	('x')	-	('xx')	-	('xx')
INCFSZ	r+00h+'1' (Z?)	adder, A+B+'1'	('x0 01')	B / -	('0')	00h	('x1')	Reg.	('00')
INFSNZ	r+00h+'1' ($\overline{Z}$?)	adder, A+B+'1'	('x0 01')	B / -	('0')	00h	('x1')	Reg.	('00')
LCALL	k + 00h	adder, A+B+'0'	('x0 00')	B / -	('0')	00h	('x1')	Konst.	('10')
RETFIE	-	-	('xx xx')	- / -	('x')	-	('xx')	-	('xx')
RETLW	k + 00h	adder, A+B+'0'	('x0 00')	B / -	('0')	00h	('x1')	Konst.	('10')
RETURN	-	-	('xx xx')	- / -	('x')	-	('xx')	-	('xx')

TSTFSZ	r + 00h (zero?)	adder, A+B+'0' ('x0 00')	B / - ('0')	00h ('x1')	Reg. ('00')
BCF	r AND $\overline{b}$	logical, A $\wedge$ B ('01 11')	$\overline{B}$ / - ('1')	Bit-Op. ('10')	Reg. ('00')
BSF	r OR b	logical, A $\vee$ B ('01 01')	B / - ('0')	Bit-Op. ('10')	Reg. ('00')
BTFSC	r AND b (zero?)	logical, A $\wedge$ B ('01 11')	B / - ('0')	Bit-Op. ('10')	Reg. ('00')
BTFSS	$\overline{r}$ AND b (zero?)	logical, $\overline{A}$ $\wedge$ B ('01 10')	B / - ('0')	Bit-Op. ('10')	Reg. ('00')
BTG	r XOR b	logical, A $\oplus$ B ('01 00')	B / - ('0')	Bit-Op. ('10')	Reg. ('00')
NOP	-	- ('xx xx')	- / - ('x')	- ('xx')	- ('xx')

r: Registerwert (reg_data[7:0]) k: Konstante (IR[7:0]) b: Bitoperand (Demux IR[10:8]) 'x': don´t care

Tabelle 3.7: Zusammenfassung der ALU-Operationen für alle Befehle.

3.7 ALU Status Register

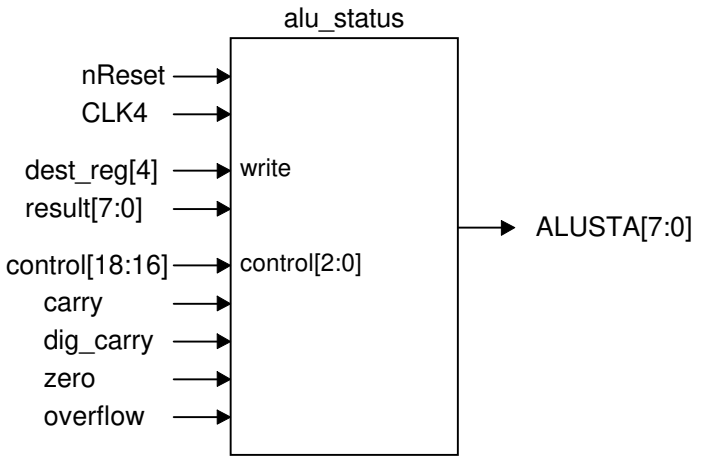


Abbildung 3.7: ALU Status Register.

Der Block *alu_status* implementiert das Register ALUSTA des Mikrocontrollerkerns. Das Register enthält die vier Statusbits und die Flags zur Steuerung der Modi der beiden File-Select Register. Da das Register in den Datenspeicherbereich eingeblendet ist, kann es einerseits normal als Quell- und Zieloperand bei Operationen verwendet werden. Dementsprechend besitzt der Block Daten- und Steuereingänge zum Speichern des Operationsergebnisses im Register. Der zugehörige Steuereingang wird vom entsprechenden Steuerausgang des Registermultiplexers geliefert.

Unabhängig davon werden die vier Statusbits bei vielen Befehlen implizit, in Abhängigkeit vom Ergebnis der Operation, neu gesetzt. Wie in Tabelle 3.8 dargestellt, wird das Modifizieren der Statusbits über drei Kontrolleitungen vom Instruktionsdekoder gesteuert. Entsprechend dem Zustand der Steuereingänge übernimmt das ALU Status Register die anliegenden Werte von den Ausgängen der ALU in die zugehörigen Statusbits. Zu beachten ist, daß das implizite Setzen der Statusbits durch einen Befehl Priorität gegenüber einem evtl. gleichzeitig stattfindenden Schreiben des Operationsergebnis bei Verwendung des Registers als Ziel der Operation hat. Ist eine der drei Steuerleitungen aktiv und wird gleichzeitig ein neuer Wert in

das gesamte Register geschrieben, so werden die entsprechenden Statusbits anhand der Statuseingänge und nicht anhand der Bits am Dateneingang gesetzt.

Eingang	Funktion	Steuerleitung
control[0]	Modifiziere Carry Flag entsprechend dem Carry Status. '1': Ändern '0': Nicht ändern	control[16]
control[1]	Modifiziere Zero Flag entsprechend dem Zero Status. '1': Ändern '0': Nicht ändern	control[17]
control[2]	Modifiziere Overflow und Digit Carry Flags. '1': Ändern '0': Nicht ändern.	control[18]

Tabelle 3.8: Steuerleitungen des ALU Status Registers.

Die acht Bits des Registers sind teilweise als Latches und teilweise als Flip-Flops realisiert. Die vier Flags für die File-Select Modi sind als Flip-Flops implementiert, da diese direkt zur Ansteuerung der File-Select Register im Instruktionsverarbeitungsblock verwendet werden und deshalb den ganzen Instruktionszyklus lang stabil bleiben müssen. Auch die Statusbits Carry und Digit Carry sind als Flip-Flops realisiert, da der Inhalt dieser Statusbits unter anderem wieder in die ALU zurückgeführt wird und dort Einfluß auf bestimmte Operationen hat, so daß die Werte ebenfalls über den gesamten Zyklus unverändert anstehen müssen. Die Statusflags Overflow und Zero werden hingegen in keiner Form kombinatorisch in kritischen Bereichen direkt im Kern ausgewertet und sind deshalb aus Platzgründen als einfache Latches implementiert.

3.8 Ergebnismultiplexer

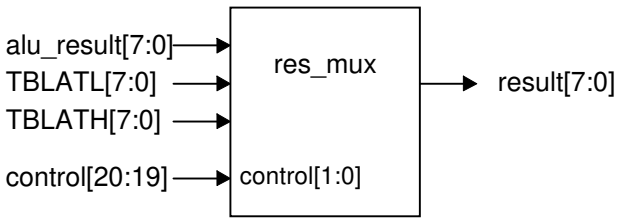


Abbildung 3.8: Ergebnismultiplexer.

Über den Ergebnismultiplexer wird ausgewählt, welcher Wert bei einer Operation auf den Ergebnisdatenbus gelegt und von dort dann schließlich am Ende der Befehlsabarbeitung in ein Register geschrieben wird. Es läßt sich entweder das Ergebnis am Ausgang der ALU, d.h. das Ergebnis der ALU Operation, oder eine der beiden Hälften des Table-Latches durchschalten. Bei den allermeisten Befehlen entspricht das Ergebnis der Operation dem von der ALU gelieferten Wert. Bei der Ausführung der Befehle `TABLRD` und `TLRD` muß hingegen, in Abhängigkeit vom angegebenen Parameter, eine der beiden Table-Latch Hälften in ein beliebig adressiertes Register geschrieben werden. Um den Aufbau der ALU und des Registermultiplexers, die beide im kritischen Pfad liegen, durch diesen Spezialfall nicht unnötig zu komplizieren, können an die Eingänge der ALU aber nur allgemein adressierbare Register angelegt werden. Da das Table-Latch aber nicht in den Datenspeicherbereich eingeblendet ist, steht es an den Eingängen der ALU nicht zur Verfügung und kann auch nicht auf den Ausgang

der ALU durchgereicht werden. Mit Hilfe des Ergebnismultiplexers lassen deshalb die Inhalte des Table-Latches für diese beiden Befehle direkt auf den Ergebnisdatenbus legen.

Die Auswahl der drei möglichen Quellen des Ergebnismultiplexers erfolgt über zwei Steuerleitungen vom Instruktionsdekoder, wie in Tabelle 3.9 dargestellt. Da im allgemeinen Fall der ALU Ausgang durchgeschaltet wird, wurde der zur Verfügung stehende zusätzliche Freiheitsgrad bei der Zuordnung der Steuerleitung für die doppelte Auswahl des ALU Ergebnisses verwendet. Dies erlaubt im Instruktionsdekoder für fast alle Befehle eine eventuelle Optimierung der dortigen Logik unter Ausnutzung des beliebig wählbaren zweiten Steuerbits.

control[1] (control[20])	control[0] (control[19])	result[7:0]
'0'	'0'	TBLATL[7:0]
'0'	'1'	TBLATH[7:0]
'1'	'0'	alu_result[7:0]
'1'	'1'	alu_result[7:0]

Tabelle 3.9: Steuerleitungen des Ergebnismultiplexers.

3.9 Table Latches

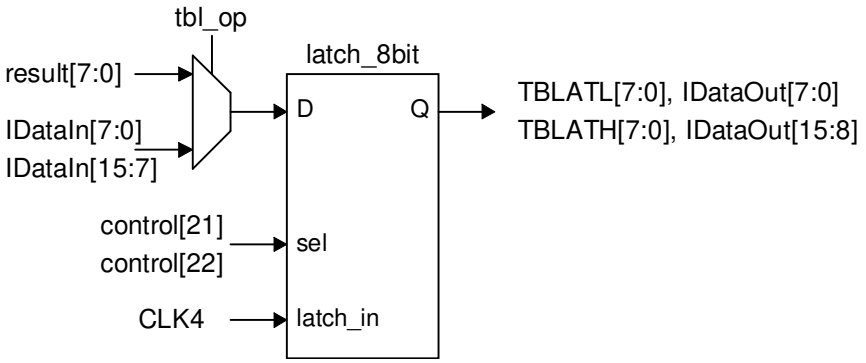


Abbildung 3.9: Table Latches.

Die beiden Hälften des Table-Latches sind jeweils über den Block *latch_8bit*, der ein allgemeines, 8 Bit breites Latch darstellt, realisiert. Neben Datenein- und ausgängen wird das Latch über zwei Kontrolleitungen gesteuert. Das Latch ist transparent, wenn die Eingänge *sel* und *latch_in* high sind. In allen anderen Fällen hält das Latch seinen gespeicherten Wert. Der Ausgang der Latches und damit der darin gespeichert Wert wird schließlich einerseits direkt auf den Ausgang des Programmspeicherdatenbus gegeben, so daß bei Schreibzugriffen auf den Programmspeicher an der adressierten Speicherstelle stets der Inhalt des Table-Latches abgespeichert wird. Andererseits können die beiden Table-Latch Hälften über den Ergebnismultiplexer ggf. auch in ein anderes Register geschrieben werden.

Die Steuerung der beiden Hälften des Table-Latches erfolgt über CLK4 und jeweils eine Kontrolleitung vom Instruktionsdekoder. Wenn der Instruktionsdekoder das Signal zum Einlatchen eines Wertes liefert, wird das Latch in der Taktphase Q4 des Instruktionszyklus

aktiv. Der Instruktionsdekoder aktiviert die Steuerleitungen dabei ausschließlich bei den diversen Table-Operationen. Je nach Art, Parameter und Zyklus der Operation wird im Table-Latch entweder ein beliebiger Registerinhalt oder ein aus dem Programmspeicher gelesener Wert abgespeichert. Dem Dateneingang der Table-Latches ist deshalb ein Multiplexer vorgeschaltet, der entweder den Wert auf dem Ergebnisdatenbus (der Registerinhalt wird über den Registermultiplexer durch die ALU und durch den Ergebnismultiplexer gereicht) oder die passende Hälfte des Programmspeicherdatenbus durchschaltet. Angesteuert wird der vorgeschaltete Multiplexer von einem speziellen Steuerausgang des Instruktionsdekoders (siehe Abschnitt 3.16), der angibt, ob sich die Ausführung im zweiten Zyklus einer Table-Operation befindet. Ist dies der Fall, so wird der Wert des Programmspeicherdatenbus auf die Table-Latches gegeben. In allen anderen Fällen liegt dort der auf dem Ergebnisdatenbus stehende Wert an. Über die normale Steuerleitung des Instruktionsdekoders wird schließlich festgelegt, ob der angelegte Wert überhaupt in das Latch geschrieben werden soll.

Wie der Name bereits andeutet, sind die Table-Latches als platzsparende Latches und nicht als Flip-Flops realisiert. Dies ist ohne weiteres möglich, da es in keinem Fall vorkommen kann, daß der Inhalt eines Table-Latches z.B. über den Ergebnismultiplexer ausgewertet wird während das Latch transparent ist.

3.10 Table Pointer

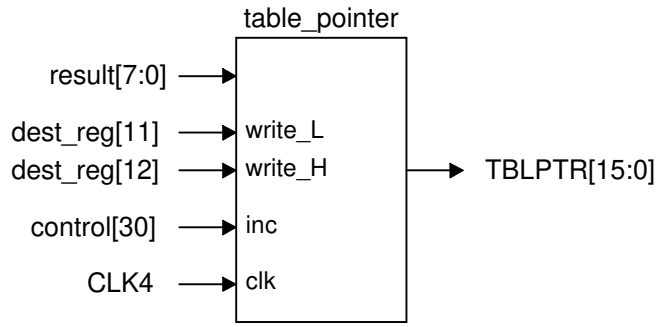


Abbildung 3.10: Table Pointer.

Der Block *table_pointer* bildet das insgesamt 16 Bit breite Register TBLPTR. Die beiden Hälften des Registers sind normal in den Datenspeicherbereich eingeblendet und können deshalb jeweils einzeln mit dem Ergebnis einer Operation neu beschrieben werden. Zusätzlich kann der Inhalt des kompletten Registers im Zuge einer Table-Operation optional um eins erhöht werden. Der Inhalt des Register kann einerseits normal über den Registermultiplexer ausgelesen werden. Andererseits dient das Register als Zeiger in den Programmspeicher beim Austausch von Daten zwischen Programmspeicher und Table-Latch im Verlauf von Table-Operationen. Dem Programmspeicheradreßbus ist ein Multiplexer vorgeschaltet, über den entweder der Inhalt des Programmzählers oder der Inhalt des Table Pointers ausgewählt werden kann. Angesteuert wird dieser Multiplexer über das vom Instruktionsdekoder gelieferte Signal *tbl_op* (siehe Abschnitt 3.16), so daß im zweiten Zyklus einer Table-Operation stets der Table Pointer die Programmspeicheradresse für den Zugriff liefert.

Neben dem Takteingang, über den das Register mit CLK4 getaktet wird, erfolgt die Steuerung der Registeroperation über drei Steuereingänge. Ist keine der Steuerleitungen aktiv, behält das Register seinen gespeichert Wert unverändert bei. Vom Registermultiplexer erhält der Table Pointer die aus der Zieladreßdekodierung gewonnenen Steuersignale zum Schreiben der angelegten Ergebnisdaten in eine der Registerhälften. Der Instruktionsdekoder liefert über eine Kontrolleitung das Signal zum Inkrementieren des Registerinhalts. Zu beachten ist, daß

maximal nur eine der Steuerleitungen aktiv sein kann bzw. darf. Andernfalls ist die Operation des Registers nicht definiert.

Das Register ist komplett aus Flip-Flops aufgebaut, die jeweils die neuen Daten mit der fallenden Flanke von CLK4 übernehmen. Das Register muß mit Flip-Flops realisiert werden, da einerseits, bei Verwendung des Registers als Zeiger in den Programmspeicher, die Programmspeicheradresse und damit der Registerinhalt über den gesamten Zyklus hinweg stabil bleiben müssen und da andererseits nur so die automatische Inkrementierung des Registers einfach und platzsparend verwirklichbar ist.

3.11 Interruptmodul

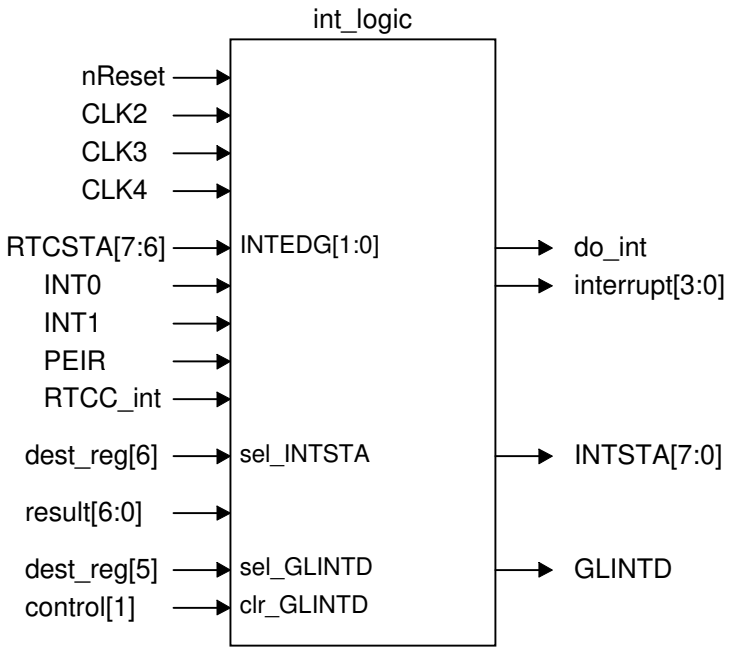


Abbildung 3.11: Interruptmodul.

Das Interruptmodul (Block *int_logic*) enthält neben der bereits im vorangegangenen Kapitel in Abschnitt 2.10 teilweise beschriebenen Interruptlogik auch das Statusbit GLINTD. Die Interruptlogik besteht aus der Erkennung von externen Interrupts, der Verarbeitung von Interruptanforderungen allgemein und dem Register INTSTA als zentralem Element. Sowohl das Register INTSTA als auch das Bit GLINTD (über das Register CPUSTA) sind in den Datenspeicherbereich eingeblendet und können daher aus Anwendungprogrammen heraus mittels normaler Befehle beschrieben werden. Zur Durchführung dieser Operationen werden an das Interruptmodul einerseits die Ergebnisdaten selbst und andererseits die vom Registermultiplexer kommenden Steuerleitungen für den Schreibzugriff angelegt. Bei Schreibzugriffen auf das Register CPUSTA wird das zugehörige Bit des Ergebnisvektors in das GLINTD-Register geschrieben. Da das Bit PEIR des Registers INTSTA (Bit 7) nicht beschreibbar ist und nur den abgetasteten Wert am zugehörigen Eingang widerspiegelt, werden am Eingang des Interruptmoduls auch nur die unteren 7 Bit der Ergebnisdaten zur Verfügung gestellt. Sowohl das Flag GLINTD als auch das Register INTSTA sind aus per fallender Flanke von CLK4 getakteten Flip-Flops aufgebaut, da sie direkten Einfluß auf die Logik des Interruptmoduls haben und sich deshalb im Laufe eines Instruktionszyklus nicht ändern dürfen.

Neben den Eingängen für Schreibzugriffe auf die Register finden sich an der Schnittstelle des Interruptmoduls noch weitere Eingänge zur Erkennung und Auswertung von Interruptbedingungen. Dazu zählen die diversen Taktsignale für Synchronisation und

Registertakt, die Statusbits des Registers RTCSTA zur Flankenauswahl und die vier Interruptquellsignale selbst. Die Erkennung von Flanken auf den beiden externen Interrupteingängen INT0 und INT1 erfolgt jeweils in einem separaten Unterblock, dessen Funktion im nachfolgenden Abschnitt näher beschrieben ist. Ein erkannter Interrupt setzt das zugehörige Anforderungsbit im Register INTSTA und führt in Verknüpfung mit den Maskierungsbits ggf. zur Auslösung eines Interrupts. Wird ein Interrupt erkannt und ausgelöst, so wird am Ende des Zyklus ein Flag gesetzt und die auszulösende Interruptquelle unter Berücksichtigung der Interruptprioritäten in einem Register gesichert. Insgesamt hat das zusätzliche interne Register eine Breite von 3 Bit. Ein Bit gibt dabei an, ob überhaupt ein Interrupt bearbeitet wird. Ist dies der Fall, so geben die anderen zwei Bits die auslösende Interruptquelle an. Am Ende des auf die Erkennung folgenden zweiten Interruptzyklus wird das Zustandsregister wieder zurückgesetzt und die Interruptlogik befindet sich wieder im Normalzustand. Die Zwischenspeicherung der Interruptdaten ist notwendig, weil der Inhalt des Registers INTSTA durch einen Anwendungsbefehl genau zwischen erstem und zweitem Interruptzyklus verändert werden kann, so daß sich ansonsten die Interruptbedingungen mitten im Interrupt ändern würden.

Am Ausgang des Interruptmoduls stehen zwei Signale zur Verfügung: Das Signal *do_int* ist für die Dauer des Interrupts, d.h. ab der Erkennung bis zum Ende des darauffolgenden Zyklus, aktiv. Diese Signal findet sowohl im Modul als auch außerhalb Verwendung, z.B. beim Setzen der Interruptflags, zum Laden des Befehls `NOP` in das Instruktionsregister oder zum Anhalten des Programmzählerinkrements. Der Vektor *interrupt* ist normalerweise Null. Erst im zweiten Zyklus des Interrupt, d.h. im Zyklus nach der Erkennung, wird über diesen Vektor die auslösende Interruptquelle ausgegeben. Über diesen Vektor wird das Programmzählermodul dazu veranlaßt, durch Laden der Startadresse in die zugehörige Interruptroutine zu verzweigen.

3.11.1 Interrupterkennung

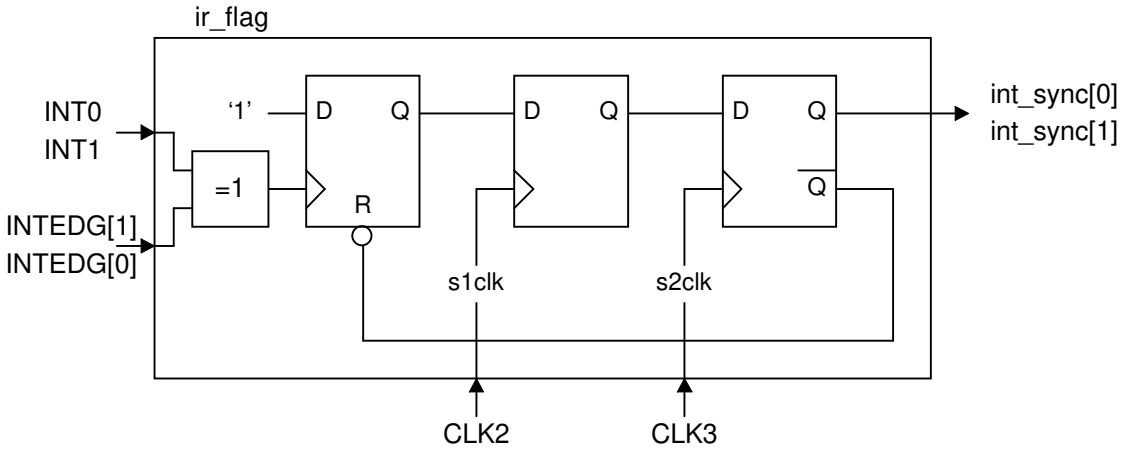


Abbildung 3.12: Prinzip der Flankendetektion.

Der Unterblock *ir_flag* wird innerhalb des Interruptmoduls jeweils zur Erkennung von Flanken, d.h. von Interrupts an den Interruptpins INT0 und INT1 eingesetzt. An erster Stelle wird das Interruptsignal über ein Exklusiv-Oder Gatter entsprechend der zu erkennenden Flankenart so aufbereitet, daß im nachhinein durch die Schaltung ausschließlich fallende Flanken erkannt werden müssen. Dieses Signal geht dann auf den Takteingang eines flankengetriggerten Flip-Flops, das bei einer fallende Flanke dazu veranlaßt wird, seinen Zustand auf '1' zu setzen. In zwei weiteren Stufen wird der Zustand des vorangegangenen Flip-Flops jeweils mit der fallenden Flanke von CLK2 bzw. CLK3 in die nächste Stufe

übernommen. Der Inhalt der letzten Stufe dient schließlich dazu das Flip-Flop der ersten Stufe wieder zurückzusetzen und stellt gleichzeitig das synchronisierte Interruptsignal dar.

Die minimale Dauer von Impulsen auf der Interruptleitung, die noch als Flanken und damit als Interrupts erkannt werden, hängt bei dieser Schaltung ausschließlich von den Eigenschaften des Flip-Flops der ersten Stufe ab und erreicht so bestmögliche Werte. Als Alternative käme nur die Abtastung der Interruptleitung in Frage. Da die Abtasttaktfrequenz aber aus dem Systemtakt abgeleitet werden müßte, wäre auch die minimale Impulsdauer abhängig von der Taktfrequenz. Um die Impulsdauer möglichst gering zu halten, müßte außerdem mit dem externen Systemtakt abgetastet werden. Aufgrund des engen Zusammenhangs und des unvermeidbaren Versatzes zwischen externem und internem Takt ist diese Lösung allerdings unpraktikabel, da es in jedem Fall an der Stelle, an der das abgetastete Signal auf einen internen Takt übernommen wird mit hoher Wahrscheinlichkeit zu einer Setup- oder Hold-Verletzung kommen muß.

3.12 RTCC Modul

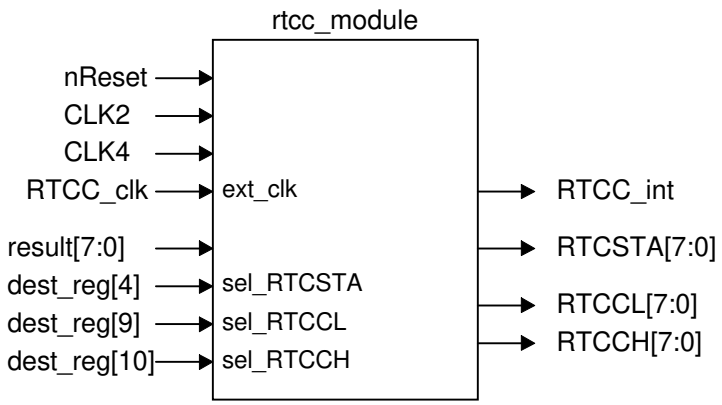


Abbildung 3.13: RTCC Modul.

Der Aufbau und die Funktionsweise des Ereigniszählermoduls wurden bereits bei des Spezifikation des Mikrocontrollerkerns in Abschnitt 2.11 des vorangegangenen Kapitels ausführlich dargestellt. Der Block *rtcc_module* enthält neben dem Statusregister RTCSTA im Prinzip nur die in Reihe geschalteten Unterblöcke für den Vorteiler, die Synchronisation und die Zählerregister (RTCCL und RTCCH). Auf der einen Seite werden am Eingang des Blocks die Daten des Ergebnisbus und die Steuerleitungen zum Schreiben der Ergebnisdaten in die drei in den Datenspeicherbereich eingeblendeten Register angelegt. Auf der anderen Seite besitzt der Block weitere Eingänge mit diversen Taktleitungen für Registertakt, Synchronisation und Ereignissignale. Am Ausgang stellt der Block die Inhalte der drei Register und die Interruptleitung für die bei Überlauf des Ereigniszählers im Interruptmodul auszulösende Unterbrechung bereit.

In den folgenden Abschnitten werden die einzelnen Unterblöcke des Moduls beschrieben. Das Statusregister ist nicht als eigener Unterblock ausgeführt. Es ist als einfaches Register implementiert, das ausschließlich mit den Werten des Operationsergebnisses beschrieben werden kann. Da die einzelnen im Statusregister enthaltenen Bits direkt zur Steuerung der Operationen im Ereigniszähler- und teilweise auch im Interruptmodul verwendet werden, ist das Register komplett aus Flip-Flops aufgebaut, getaktet mit der fallenden Flanke von CLK4.

3.12.1 Vorteiler

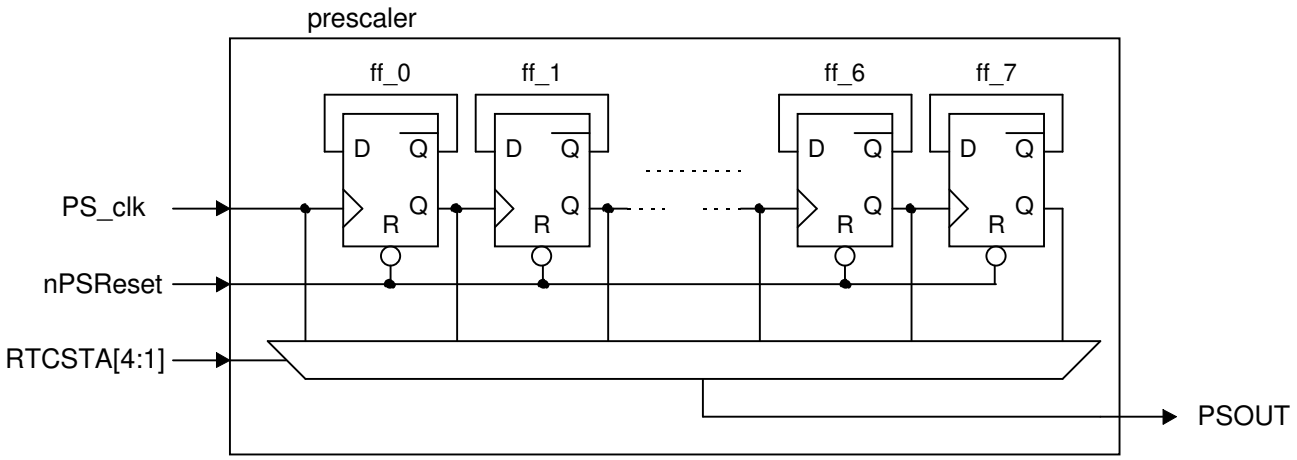


Abbildung 3.14: Vorteiler.

Der Vorteiler des RTCC Moduls (Block *prescaler*) ist als 8 Bit breiter, asynchroner Zähler aufgebaut. Am Eingang des Vorteilers wird in Abhängigkeit vom Inhalt des Statusregisters entweder der externe Ereignistakt (invertiert oder nicht invertiert) oder das interne Taktsignal CLK4 angelegt. Eine fallende Flanke auf dieser Leitung löst einen Inkrement des kompletten Zählers aus. Über die zugehörigen Bits des Statusregisters wird ein Multiplexer angesteuert, mit dem der Ausgang des entsprechenden internen Zähler-Flip-Flops auf den Ausgang *PSOUT* des Vorteilers durchgeschaltet wird. Schließlich können die Zählerregister des Vorteilers über eine Resetleitung asynchron zurückgesetzt werden. Ein Reset des Vorteilers wird im RTCC Modul bei Schreibzugriffen auf die Zählerregister oder bei einem allgemeinen externen Reset ausgelöst.

3.12.2 Synchronisation

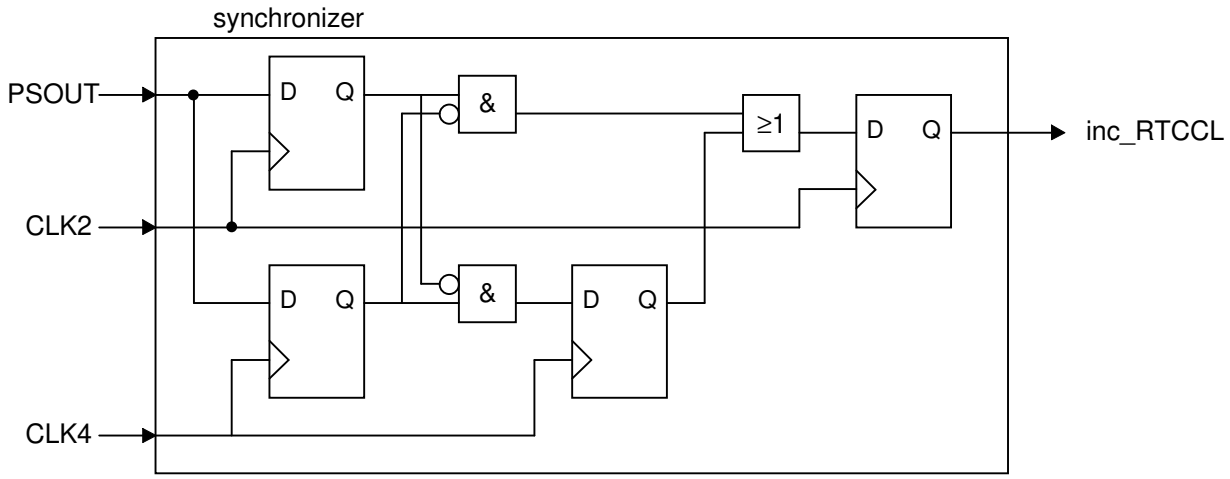


Abbildung 3.15: Synchronisation.

In der Synchronisationsschaltung (Block *synchronizer*) wird der Ausgang des Vorteilers mit den fallenden Flanken der Takte CLK2 und CLK4 abgetastet und damit auf die internen Taktverhältnisse synchronisiert. Die Schaltung erkennt dabei fallende Flanken des

Vorteilersignals und erzeugt am Ausgang in diesem Fall einen Impuls zur Inkrementierung des Ereigniszählers.

3.12.3 RTCC Register

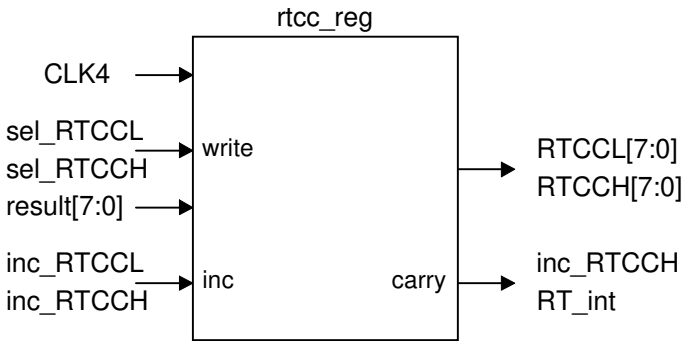


Abbildung 3.16: RTCC Register.

Die beiden Register RTCCL und RTCCH, die zusammen den eigentlichen Ereigniszähler bilden, werden jeweils durch eine Instanz des Blocks *rtcc_reg* implementiert. Aufgrund der nötigen Zählfunktion sind die Register als Flip-Flops implementiert und werden mit der fallenden Flanke von CLK4 getaktet. Die Aktivierung des Inkrement-Steuereingangs veranlaßt die Register ihren Inhalt um eins zu erhöhen. Ein dabei entstehender Überlauf wird durch ein Carry-Signal am Ausgang der Register angezeigt. Die Register sind so verschaltet, daß der Ausgang der Synchronisationsschaltung auf den Eingang des Registers RTCCL geht, dessen Carry-Ausgang dann auf den Eingang des Registers RTCCH und dessen Carry-Ausgang schließlich auf den Interruptausgang des Moduls. Wie bereits erwähnt, können gleichzeitig die angelegten Ergebnisdaten über zugehörige Steuerleitung in die Register geschrieben werden, wobei das Schreiben des Register Priorität gegenüber dem Inkrementieren hat.

Zusätzlich enthalten die Register eine Logik zum Maskieren des Inkrement-Kontrollsignals nach einem erfolgten Schreibzugriff. Wird ein Register beschrieben, so wird dies in einem zusätzlichen Flip-Flop für die Dauer des nachfolgenden Instruktionszyklus gespeichert und in Abhängigkeit vom Inhalt des Zustandsbits der Inkrementeingang gesperrt.

3.13 Programmzählermodul

Der Programmzähler, d.h. das Register PC mit umgebender Logik, der Stack und die Logik zur Verwaltung des Stacks bzw. zur Verbindung von Stack und PC sind im Block *pc_module* zusammengefaßt. Der eigentliche Stackspeicher ist aus Flexibilitätsgründen in einem extra Unterblock ausgelagert und wird in einem separaten Abschnitt behandelt. Die Tiefe des Stacks, d.h. die Größe des Stackspeichers, ist über ein VHDL-Generic beliebig einstellbar. Das Programmzählermodul stellt an seinen Ausgängen den Inhalt des Programmzählerregisters und den Wert des Bits STKAVL, das als Teil des Registers CPUSTA in den Datenspeicherbereich eingeblendet ist, zur Verfügung. Da das Bit STKAVL im Register CPUSTA aber nicht beschreibbar ist und aus Anwendungsprogrammen heraus nur ausgelesen werden kann, werden entsprechende Schreibzugriffe auf das Register im Programmzählermodul nicht ausgewertet.

Da der Programmzähler im allgemeinen die Adresse für den Programmspeicherzugriff liefert, ist das Register als Flip-Flop implementiert, das mit der fallenden Flanke von CLK4 ggf. einen neuen Wert übernimmt. Im Mikrocontrollerkern muß der Programmzähler eine Vielzahl von Operationen unterstützen, d.h. je nach Zustand der Befehlsausführung ergibt sich der neue Wert des Programmzählers im nächsten Zyklus aus den unterschiedlichsten Quellen. Am Eingang des Moduls finden sich deshalb die entsprechenden Dateneingänge und die

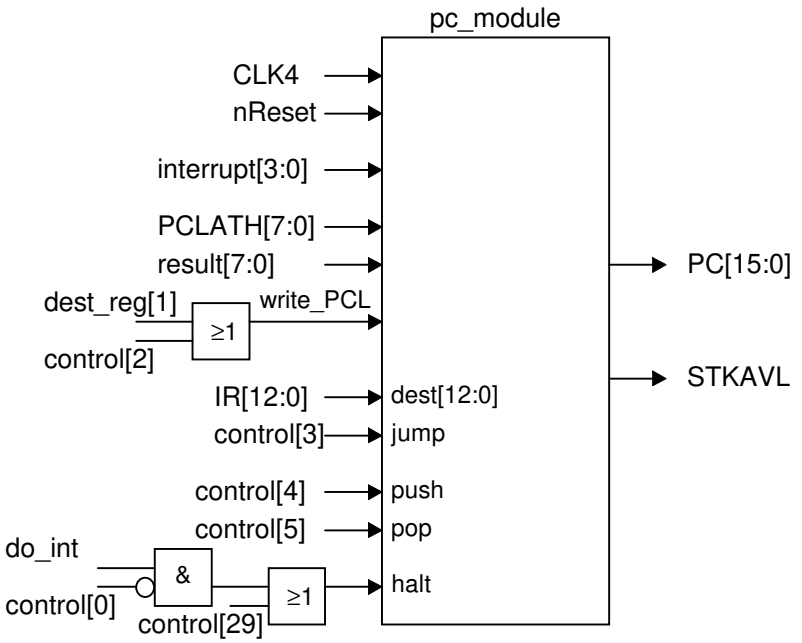


Abbildung 3.17: Programmzählermodul.

Steuerleitungen zur Kontrolle des dem Programmzähler vorgeschalteten Multiplexers. Die verschiedenen Möglichkeiten zum Laden eines neuen Wertes in das Register PC sind in Tabelle 3.10 in der Reihenfolge ihrer Prioritäten und mit zugehörigen Steuer- und Dateneingängen zusammengefaßt. Sind mehrere Steuereingänge gleichzeitig aktiv, so wird die Aktion mit der höchsten Priorität ausgeführt.

Priorität	Signal	Aktion
höchste	interrupt[0]	PC=0008h, Stack Push
↑	interrupt[1]	PC=0010h, Stack Push
	interrupt[2]	PC=0018h, Stack Push
	interrupt[3]	PC=0020h, Stack Push
	write_PCL	PC[15:8]=PCLATH[7:0], PC[7:0]=result[7:0]
	jump	PC[12:0]=dest[12:0]
	pop	PC=TOS (Top Of Stack), Stack Pop
	halt	-
niederste	-	PC=PC+1

Tabelle 3.10: Manipulation des Programmzählers.

Im normalen Betrieb des Mikrocontrollerkerns ist keine der Steuerleitungen aktiv und der Programmzähler erhöht sich am Ende eines Instruktionszyklus jeweils um eins. Ist dagegen die Steuerleitung *halt* aktiviert, so behält der Programmzähler seinen Wert unverändert bei. Das Anhalten des PC wird einerseits explizit durch den Instruktionsdekoder, z.B. bei der Ausführung von Table-Operationen, und andererseits implizit bei der Auslösung eines Interrupts, d.h. bei der Verzweigung in eine Interruptroutine, veranlaßt. Das vom Interruptmodul kommende Interruptsignal wird dazu aber noch mit der vom

Instruktionsdekoder kommenden Steuerleitung zum Laden des Befehls `NOP` in das Register IR maskiert. Dies behandelt den Spezialfall eines Interrupts, der während der Ausführung eines Skips auftritt. In diesem Fall darf nämlich der PC nicht angehalten werden und muß normal inkrementiert werden, um sicherzustellen, daß im weiteren Verlauf die richtige Rücksprungadresse auf dem Stack abgelegt wird (siehe dazu den entsprechenden Abschnitt 2.10.4.2 der Spezifikation).

Alle weiteren Aktionen stehen in der Priorität über diesen trivialen Operationen, wobei deren Prioritäten untereinander aber im Prinzip unerheblich sind, da beim Betrieb des Mikrocontrollerkerns nie gleichzeitig zwei der weiteren Steuerleitungen aktiv sein können. Alle nachfolgenden Operationen laden einen komplett neuen Wert in den Programmzähler, d.h. es wird ein Sprung im Anwendungsprogramm durchgeführt. Im einzelnen sind folgende Aktionen möglich:

- Der Programmzähler wird auf Veranlassung des Instruktionsdekoders bei der Ausführung von Rücksprungbefehlen mit dem an der Spitze des Stacks stehenden Wert der Rücksprungadresse geladen. Gleichzeitig wird in der Stacklogik des Moduls die zugehörige Aktion zur Erniedrigung des Stacks um ein Element ausgelöst, d.h. zusammengenommen wird der oberste Wert vom Stapel geholt und in den PC geschrieben.
- Im Zuge der Ausführung eines `CALL` oder `GOTO` wird die im Befehlswort kodierte Zieladresse des Sprungs durch den Instruktionsdekoder aus den unteren 12 Bits des Instruktionsregisters in die entsprechenden Bits des Programmzählers geladen.
- Die Kombination aus den auf dem Ergebnisbus liegenden Daten einer Operation und dem Inhalt des Registers PCLATH wird in den Programmzähler geschrieben. Zum einen wird diese Aktion explizit durch den Instruktionsdekoder bei der Ausführung des Befehls `LCALL` ausgelöst. Alternativ führt ein Schreibzugriff eines normalen Befehls auf das Zielregister PCL zum selben Verhalten.
- Zur Verzweigung in eine Interruptroutine wird vom Interruptmodul im zweiten Interruptzyklus eine der Interruptsteuerleitungen aktiviert, und die feste Startadresse der zugehörigen Interruptroutine wird daraufhin in den Programmzähler geladen. Von der Stacklogik des Moduls wird zuvor der aktuelle Programmzählerinhalt auf der Spitze des Stapels abgelegt.

Wie bereits erwähnt, enthält das Programmzählermodul auch den direkt an den Programmzähler angekoppelten Stack und die angrenzende Logik. Der Stack ist aus dem Anwendungsprogramm heraus nicht direkt ansprechbar und wird nur indirekt durch die Anwendung bei der Ausführung von Unterprogrammaufrufen und Rücksprüngen beeinflußt. Auf die Kombination von Stack und PC können zwei Operationen ausgeführt werden, die beide durch entsprechende Steuerleitungen vom Instruktionsdekoder angestoßen werden. Ein *pop* wird bei Rücksprüngen ausgeführt und holt den obersten Wert vom Stack, um ihn in den PC zu schreiben. Ein *push* hingegen legt den aktuellen Programmzählerinhalt auf der Spitze des Stapels ab und wird vom Instruktionsdekoder bei Unterprogrammaufrufen aktiviert. Zusätzlich wird intern im Programmzählermodul ein *push* auch implizit bei der Verzweigung in eine Interruptroutine ausgelöst, so daß im Prinzip sowohl die zugehörige externe, dedizierte Steuerleitung als auch jede der Interruptsteuerleitungen einen *push* triggern..

Neben dem Registerblock, der den eigentlichen Stackspeicher realisiert und dessen Tiefe, d.h. die Anzahl der Register, über ein VHDL-Generic des Moduls festgelegt wird, umfaßt die Stacklogik des Moduls noch einen Zustandsautomaten und einen Stapelzeiger (engl. *Stack Pointer*) zur Verwaltung des Stacks. Der Stapelzeiger adressiert im Prinzip die Spitze des Stapels und gibt stets dasjenige Register im Stack an, in das bei einem *push* der nächste Wert geschrieben wird. Die Breite des Registers für den Stapelzeiger richtet sich nach der

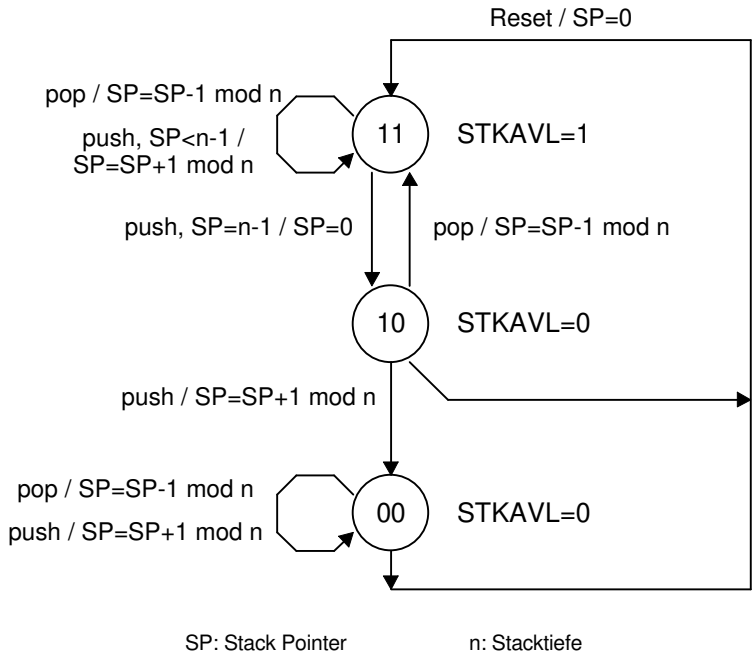


Abbildung 3.18: Zustandsdiagramm der Stacklogik.

eingestellten Stacktiefe. Um das Register für den Stapelzeiger optimal auszunützen, muß die Stacktiefe den Wert einer Zweierpotenz annehmen. Im Zuge eines *push* bzw. eines *pop* wird der Stapelzeiger jeweils um eins erhöht bzw. erniedrigt, wobei am Anfang bzw. Ende des Stackbereichs ein Überlauf zur anderen Seite stattfindet. In Abbildung 3.18 ist das Zustandsdiagramm des internen Automaten zur Verwaltung des Stackzustands dargestellt. Der Automat implementiert auch die Logik des Flags STKAVL zur Erkennung von Stacküberläufen.

3.13.1 Stack

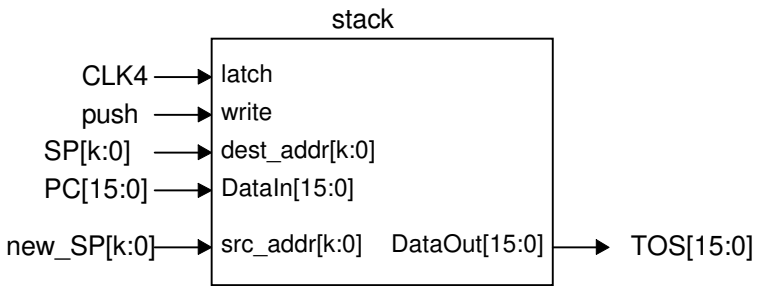


Abbildung 3.19: Stack.

Der Stapelspeicher an sich ist als separater Unterblock *stack* im Programmspeichermodul realisiert, um einen Austausch des Speichers, z.B. bei einem Wechsel der Speicherart, einfach zu ermöglichen. Die Schnittstelle zum Stack ist dabei möglichst allgemein gehalten, so daß es im Prinzip möglich sein sollte eine beliebige Speicherart einzusetzen. Beim Mikrocontrollerkern ist der Stack mit Latches implementiert, aber für zukünftige Versionen ist es durchaus in Betracht zu ziehen, den Stack aus Platzgründen mit RAM Zellen, z.B. dem am IMS zur Verfügung stehenden RAM Makro [11], zu implementieren.

Der Stackspeicher kann gelesen und geschrieben werden. Bei Lesezugriffen im Zuge eines *pop* wird dem Stack die Quelladresse, die sich aus dem neuen Inhalt des Stapelzeigers nach

dem *pop* ableitet, übergeben. In Abhängigkeit von dieser Quelladresse schaltet der Ausgangsmultiplexer die ausgewählte Speicherzelle auf den Ausgang durch, und der zugehörige Inhalt wird im Zuge des *pop* in den Programmspeicher geladen. Eine eigene Steuerleitung für den Lesezugriff existiert nicht, so daß stets das momentan adressierte Register am Ausgang des Stacks erscheint. Da der Ausgang des Stackspeichers nur bei einem *pop* ausgewertet wird, ist es ohne Belang, wenn z.B. bei einem *push*, entsprechend dem zukünftigen, neuen Wert des Stapelzeigers, eine ungültige Quelladresse angelegt wird.

Ein *push* führt durch Aktivierung der entsprechenden Steuerleitung des Stacks zu einem Schreibzugriff auf den Stackspeicher. Der aktuelle Inhalt des Stapelzeigers gibt dabei die Zieladresse für den Schreibzugriff an und der an den Dateneingang des Stacks angelegte Inhalt des Programmzählers wird in der Taktphase über CLK4 in das adressierte Register eingelatcht.

Der aktuell beim Mikrocontrollerkern eingesetzte Stack besteht insgesamt also aus einem Block von 16 Bit breiten Latches, deren Steuereingänge zeilenweise über einen Zieladreßdekodierer angesteuert werden und denen ein Ausgangsmultiplexer nachgeschaltet ist.

3.14 PC Latch

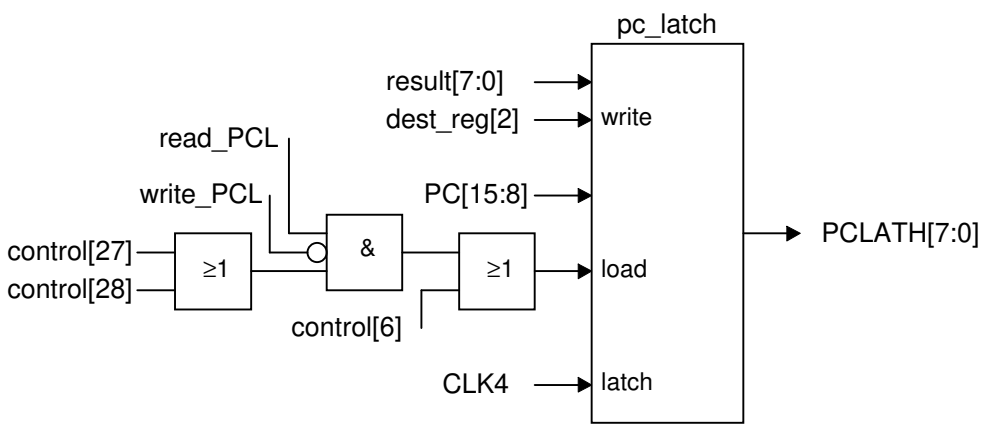


Abbildung 3.20: PC Latch.

Der Block *pc_latch* implementiert das an den Programmzähler angekoppelte Register PCLATH. Wie die anderen in den Datenspeicherbereich eingeblendeten Register des Kerns, wird das Register über eine vom Registermultiplexer kommende Steuerleitung mit den Daten auf dem Ergebnisbus neu beschrieben. Gleichzeitig dient das Register aber auch implizit bei einigen Befehlen zur Aufnahme einer Kopie der oberen Programmzählerhälfte. Wird die entsprechende Steuerleitung des Register aktiviert, so übernimmt es den angelegten Inhalt des Programmzählers. Dem Kontrolleingang ist ein Block kombinatorische Logik vorgeschaltet, der die Bedingungen zur Speicherung auswertet. Prinzipiell wird der Eingang entweder explizit durch eine vom Instruktionsdekoder kommende entsprechende Steuerleitung (bei Befehlen zur Programmablaufsteuerung, z.B. im zweiten Zyklus von `CALL` oder `GOTO`) oder implizit bei Lesezugriffen auf das Register PCL durch normale Befehle aktiviert. Der zweite Fall tritt aber nur ein, wenn es sich um reine Lesezugriffe ohne nachfolgenden Schreibzugriff auf das Register PCL handelt, d.h. die Leitung wird aktiviert, wenn das Register PCL gelesen (Steuerleitung *read_PCL*), aber nicht geschrieben (Steuerleitung *write_PCL*) wird. Das Signal *write_PCL* ist dabei das Signal, das in dieser Form auch dem Programmzählermodul zugeführt wird (s.o.). Da das Signal *read_PCL*, das vom Registermultiplexer kommt, unter Umständen auch aktiv werden kann, wenn ein Befehl, der eigentlich nicht auf ein Register zugreift, zufällig eine passende Bitkombination in seinem Befehlswort trägt, werden zusätzlich noch die

Steuerleitungen des Registerzugriffs, die bei der Instruktionsverarbeitung (siehe Abschnitt 3.15) Verwendung finden, ausgewertet.

Das Register ist vollständig aus Latches aufgebaut, die ggf. über die Taktleitung CLK4 aktiviert werden, d.h. je nach Zustand der Steuerleitungen ist das Latch in Taktphase Q4 transparent und latcht einen neuen Wert ein. Aufgrund der Funktionalität des Mikrocontrollerkerns ist sichergestellt, daß das Latch nie gleichzeitig gelesen und geschrieben wird, d.h. während das Latch transparent ist wird der Wert des Latches in keiner Form weiterverarbeitet.

3.15 Instruktionverarbeitung

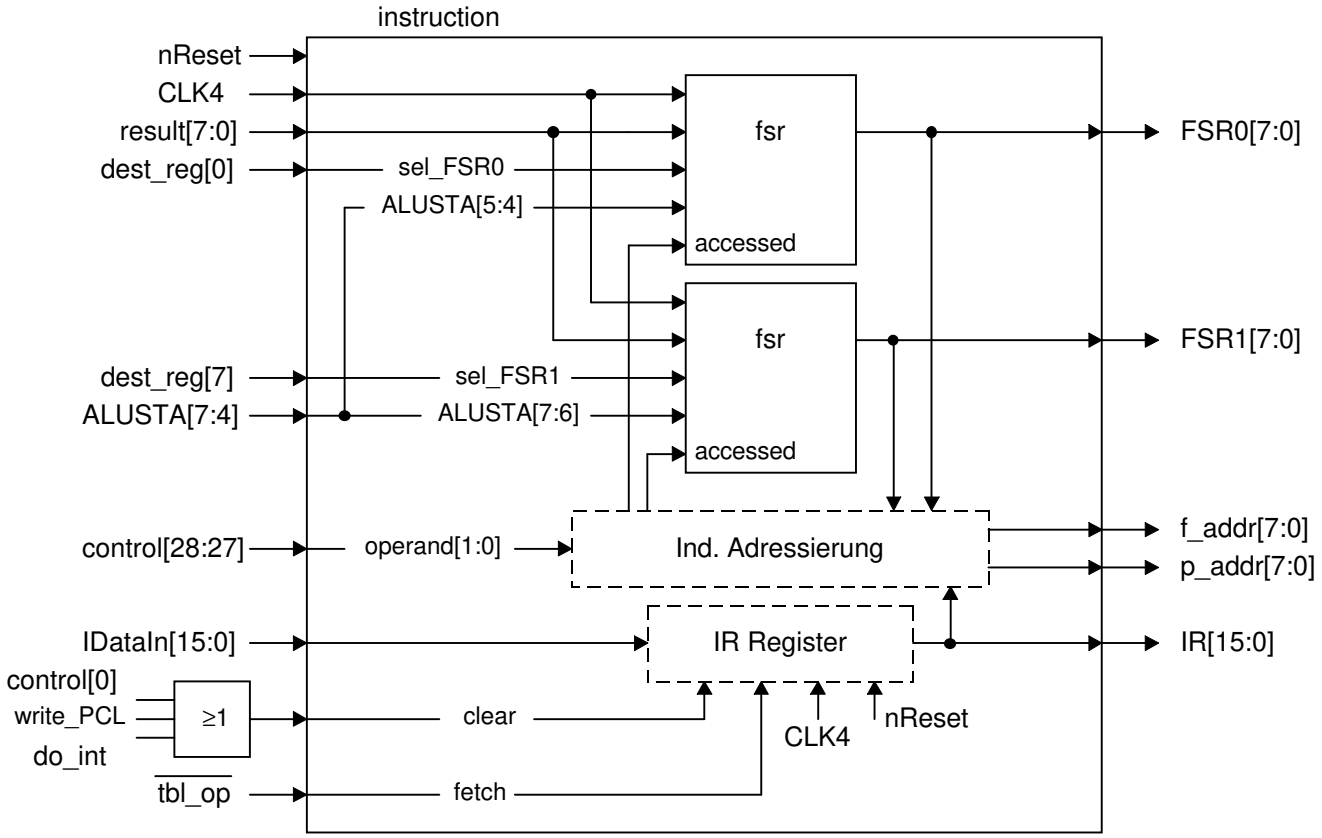


Abbildung 3.21: Instruktionsverarbeitung.

Der Block *instruction* des Mikrocontrollerkerns beinhaltet zum einen die Logik zur Auflösung der indirekten Adressierung und zum anderen die Register IR, FSR0 und FSR1, d.h. das Instruktionsregister und die beiden File-Select Register mit entsprechender umgebender Logik. Am Ausgang des Blocks erscheinen einerseits die Inhalte der drei Register und andererseits die beiden möglichen aufgelösten Operandenadressen. Bei den allermeisten Befehlen bezieht sich die Operation nur auf eine Adresse, die sog. f-Register Adresse, die in den unteren 8 Bit des Befehlsworts kodiert ist. Bei den Befehlen `MOVPF` und `MOVFP` ist hingegen zusätzlich noch eine Adresse eines zweiten Registers, die sog. p-Register Adresse, in den Bits 8 bis 12 des Befehls kodiert. Dementsprechend werden von der Instruktionsverarbeitung beide Adressen parallel aufgelöst und zur Verfügung gestellt. Für beide Adressen kann eines der Register für indirekte Adressierung angegeben werden, so daß jeweils der komplette Datenadreßraum ansprechbar ist und deshalb beide Ausgänge nach der Auflösung der indirekten Adressierung eine Adreßbreite von 8 Bit besitzen müssen.

Das Instruktionsregister ist als Flip-Flop realisiert, da sein Inhalt direkt die Basis der Steuerlogik des Instruktionsdekoders bildet. Es wird mit der fallenden Flanke von CLK4 getaktet und bei einem Reset asynchron zurückgesetzt, so daß das Register nach einem Reset den Befehl `NOP` enthält. Im ersten Zyklus nach der Auflösung des Reset wird der erste Befehl geladen, während das im Register IR enthaltene `NOP` ausgeführt wird. Im Betrieb des Mikrocontrollerkerns kann entweder der aus dem Programmspeicher gelesenen Befehl in das Instruktionsregister geladen oder das Instruktionsregister zurückgesetzt, d.h. mit dem Befehl `NOP` geladen werden. Zur Steuerung der Operation des Registers besitzt der Block, neben dem Eingang für die Daten vom Programmspeicherdatenbus, zwei Steuereingänge. Wenn keine der Steuerleitungen aktiv ist, behält das Register seinen Wert bei. Sind beide aktiv, so hat das Löschen des Registers Vorrang vor dem Laden eines neuen Befehls. Das Laden des Registers wird gesteuert durch den negierten Ausgang *tbl_op* des Instruktionsdekoders (siehe Abschnitt 3.16), d.h. außer im zweiten Zyklus einer Table-Operation wird normalerweise in jedem Zyklus ein neuer Befehl geladen. Unabhängig davon wird das Rücksetzen des Register auf Veranlassung des Instruktionsdekoders (z.B. bei Skips, Sprüngen, etc.), bei einem Schreibzugriff auf das Register PCL (entspricht einem Sprung) oder während der Verzweigung in eine Interruptroutine durchgeführt.

Der Inhalt des Registers IR bildet die Basis für die Auflösung der indirekten Adressierung. Dabei werden die f- und p-Adressen des Befehls, d.h. die Bits 0 bis 7 bzw. 8 bis 12 des Befehlsworts, ausgewertet. Sind darin die Adressen der Register INDF0 oder INDF1 angegeben, so werden stattdessen die in den File-Select Registern FSR0 bzw. FSR1 gespeicherten Werte auf die Adreßausgänge gegeben. Andernfalls werden die Bits des Befehlsworts unverändert auf die Ausgänge gegeben, wobei die p-Adresse entsprechend mit vorangestellten Nullen auf 8 Bit ergänzt wird. Zusätzlich wird ein Kontrollsignal erzeugt, daß den beiden File-Select Registern angibt, ob ein indirekter Zugriff über eines der Register stattgefunden hat, so daß die File-Select Register ggf. automatisch inkrementiert oder dekrementiert werden können. Bei der Auflösung der indirekten Adressierung werden die Befehlsadressen unabhängig davon ausgewertet, ob der Befehl überhaupt auf eines oder beide Register zugreift, d.h. ob die Befehlsbits überhaupt gültige f- oder p-Adressen darstellen. Da die Gültigkeit der Adressen aber in jedem Fall bei der Erzeugung der Steuersignale für die File-Select Register berücksichtigt werden muß, werden dem Block vom Instruktionsdekoder die entsprechenden Kontrolleitungen zugeführt, wie in Tabelle 3.11 dargestellt. Dementsprechend werden die Zugriffsleitungen der File-Select Register auch nur aktiviert, wenn der Befehl wirklich auf ein Register zugreift und eine indirekte Adressierung erkannt wird.

operand[1] (control[28])	operand[0] (control[27])	Art des Registerzugriffs durch Befehl
‘0’	‘0’	Kein Zugriff auf Register.
‘0’	‘1’	Zugriff auf f- und p-Register (IR[7:0] und IR[12:8]).
‘1’	‘0’	Zugriff auf p- und f-Register (IR[12:8] und IR[7:0]).
‘1’	‘1’	Nur Zugriff auf f-Register (IR[7:0]).

Tabelle 3.11: Steuerleitungen des Registerzugriffs.

Neben den Steuerleitungen aus der Auflösung der indirekten Adressierung werden den beiden File-Select Registern noch eine Reihe weiterer Eingänge zugeführt. Zum einen natürlich die entsprechenden Kontrollbits aus dem Register ALUSTA für den Modus der automatischen Inkrementierung bzw. Dekrementierung. Zum anderen können die Register auch aus dem Anwendungsprogramm heraus normal mit dem Ergebnis einer Operation beschrieben werden,

weshalb die entsprechenden Eingänge für die Ergebnisdaten und für die Steuerung des Schreibzugriffs durch den Registermultiplexer vorhanden sind. Das Schreiben eines neuen Registerinhalts hat Vorrang gegenüber einer eventuellen automatischen Änderung, d.h. bei einem indirekten Schreibzugriff auf eines der File-Select Register selbst, wird im Prinzip zuerst die automatische Änderung durchgeführt und dann der neue Wert geschrieben. Die beiden Register sind ebenfalls aus Flip-Flops aufgebaut, die mit der fallenden Flanke von CLK4 getaktet werden. Der Einsatz von Flip-Flops ist erforderlich, weil die Registerinhalte direkt die Basis für die Datenspeicheradressen, die über den ganzen Zyklus unverändert anliegen müssen, bilden können. Außerdem läßt sich so die Möglichkeit der automatischen Änderung einfach verwirklichen.

3.16 Instruktionsdekoder

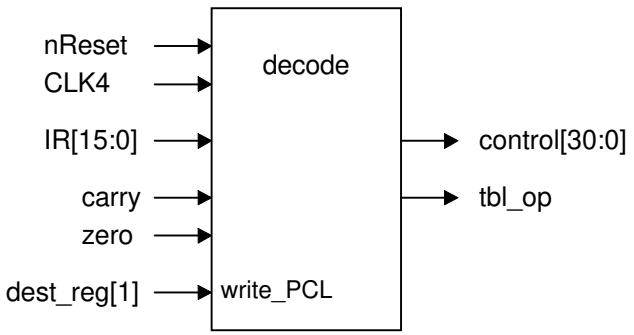


Abbildung 3.22: Instruktionsdekoder.

Der Instruktionsdekoder (Block *decode*) bildet das zentrale Steuerelement des Mikrocontrollerkerns. Er dekodiert den zuvor in das Instruktionsregister geladenen Befehl und steuert die Kontrolleitungen der anderen Blöcke des Mikrocontrollerkerns entsprechend der durchzuführenden Operation an.

Am Eingang des Instruktionsdekoders wird der Inhalt des zu dekodierenden Registers IR angelegt. Am Ausgang des Instruktionsdekoders erscheinen die angesteuerten Kontrolleitungen, bestehend aus dem allgemeinen Steuerleitungsvektor und der speziellen Steuerleitung *tbl_op*. Diese Steuerleitung wird vom Instruktionsdekoder im zweiten Zyklus einer Table-Operation (Befehle `TABLRD` oder `TABLWT`) aktiviert und steuert so den bei diesen Befehlen nötigen zusätzlichen Programmspeicherzugriff zum Transfer von Daten zwischen Programmspeicher und Table-Latch. Das Signal wird erstens dazu verwendet, anstatt des Programmzählerinhalts den Wert des Table Pointers auf den Programmspeicheradreßbus zu legen. Weiterhin steuert das Signal den Multiplexer am Dateneingang der Table-Latches so an, daß dort anstatt der Ergebnisdaten die Programmspeicherdaten eingelatcht werden können.

Die Belegung der allgemeinen Steuerleitungen ist in Tabelle 3.12 nochmals in einer Übersicht zusammengefaßt.

Zur Abfrage spezielle Bedingungen und zur Durchführung spezieller Operationen besitzt der Instruktionsdekoder eine Reihe weiterer Eingänge. Zum einen werden dem Dekoder die Statusausgänge Carry und Zero der ALU zugeführt, um bei bedingten Skip-Befehlen anhand der während der ALU Operation aufgetretenen Bedingungen zu entscheiden, ob ein Skip durchgeführt wird oder nicht. Des weiteren enthält der Dekoder einen internen Zustandsautomaten zur Zwischenspeicherung der bei mehrzykligen Operationen nötigen Informationen. Der interne Automat besitzt acht Zustände und ist dementsprechend mit drei Flip-Flops realisiert. Diese Flip-Flops werden über den Takteingang mit der fallenden Flanke von CLK4 getaktet und bei einem Reset asynchron zurückgesetzt. Im normalen Zustand

Steuerleitung	Bedeutung
control[0]	Lade IR mit `NOP` (Skip).
control[1]	GLINTD auf '0' setzen.
control[2]	PC mit Kombination (PCLATH | result) laden (`LCALL`).
control[3]	Sprung, PC mit Zieladresse in IR[12:0] laden.
control[4]	PC auf Stack schieben (push).
control[5]	PC von Stack holen (pop).
control[6]	PC[15:8] nach PCLATH schreiben.
control[7:8]	ALU: Auswahl Operand A.
control[9:10]	ALU: Auswahl Operand B.
control[11]	ALU: Invertiere B / Auswahl Shift-In.
control[12:13]	ALU: Steuerung ALU-Operationen.
control[14:15]	ALU: Auswahl ALU-Operation.
control[16]	Carry Flag wird durch Befehl gesetzt.
control[17]	Zero Flag wird durch Befehl gesetzt.
control[18]	Overflow und Digit Carry Flags werden durch Befehl gesetzt.
control[19]	Ergebnismultiplexer: Auswahl TBLATL / TBLATH.
control[20]	Ergebnismultiplexer: Auswahl TBLATx / ALU Ergebnis.
control[21]	Schreiben TBLATL.
control[22]	Schreiben TBLATH.
control[23]	Programmspeicher schreiben.
control[24]	Schreiben BSR.
control[25]	Schreiben adressiertes Zielregister.
control[26]	Schreiben W Register.
control[27:28]	Art des Registerzugriffs, Auswahl Quell- und Zielregisteradressen.
control[29]	PC nicht inkrementieren.
control[30]	TBLPTR inkrementieren.

Tabelle 3.12: Zusammenfassung der Steuerleitungen.

dekodiert der Dekoder den Inhalt von IR und führt ihn aus. Handelt es sich dabei um eine spezielle, mehrzyklige Instruktion, dann geht der interne Automat im nächsten Instruktionszyklus je nach Befehl in einen der folgenden sieben Zustände über und führt die entsprechenden speziellen Operationen durch:

- Zweiter Zyklus eines `CALL` oder `GOTO`; obere Hälfte des Programmzählers nach PCLATH sichern.
- Zweiter Zyklus eines `TABLWT`; TBLPTR inkrementieren; Programmspeicher schreiben; Signal *tbl_op* aktivieren.
- Zweiter Zyklus eines `TABLWT`; TBLPTR nicht inkrementieren; Programmspeicher schreiben; Signal *tbl_op* aktivieren.

- Zweiter Zyklus eines `TABLRD`; TBLPTR inkrementieren; Programmspeicher lesen; Signal *tbl_op* aktivieren.
- Zweiter Zyklus eines `TABLRD`; TBLPTR nicht inkrementieren; Programmspeicher lesen; Signal *tbl_op* aktivieren.
- Zweiter Zyklus eines `TABLRD` mit Registerargument PCL; TBLPTR inkrementieren; Programmspeicher lesen; Signal *tbl_op* aktivieren; PC nicht inkrementieren.
- Zweiter Zyklus eines `TABLRD` mit Registerargument PCL; TBLPTR nicht inkrementieren; Programmspeicher lesen; Signal *tbl_op* aktivieren; PC nicht inkrementieren.

Nach Durchführung der speziellen Operation im eingeschobenen zweiten Instruktionszyklus kehrt der Instruktionsdekoder wieder zum normalen Zustand zurück. Speziell zur Erkennung der letzten beiden Zustände besitzt der Instruktionsdekoder einen weiteren Steuereingang, über den der Instruktionsdekoder von der Zieladreßdekodierung des Registermultiplexers Informationen darüber erhält, ob das Register PCL als Zielregister eines Befehls beschrieben wird.

4 Synthese und Simulation

Der allgemeine Ablauf bei der Entwicklung eines ASICs und speziell bei der Entwicklung des Mikrocontrollerkerns wurde bereits in Kapitel 1 angedeutet. Im folgenden Kapitel werden die einzelnen Schritte, d.h. Simulation der Verhaltens- und RTL-Beschreibungen auf VHDL-Ebene, Synthese der RTL-Beschreibung zur Erzeugung der Gatternetzliste und abschließende Simulation auf Gatterebene, der Reihenfolge nach näher erläutert. Dabei werden neben der Vorgehensweise auch die bei den Simulationen verwendeten Testumgebungen vorgestellt. Abschließend endet das Kapitel mit einer Darlegung der aus Synthese und Simulation erhaltenen Ergebnisse der Entwicklung des Mikrocontrollerkerns, d.h. eine Übersicht und Beschreibung der wichtigsten technischen Daten des Prozessormakros.

4.1 VHDL-Simulationen

4.1.1 Überblick

Der erste Schritt bei der Entwicklung des Mikrocontrollerkerns bildet, wie bereits erwähnt, die Umsetzung der Spezifikation in eine VHDL-Verhaltensbeschreibung, anhand derer auch die Spezifikation selbst und die Umsetzbarkeit der Spezifikation überprüft werden. Da für den originalen PIC17 vom Microchip Technology weder für PCs noch für Workstations ein Simulator zur Verfügung steht, muß die Kompatibilität der Verhaltensbeschreibung zum PIC17 und die korrekte Umsetzung der Spezifikation von Hand, durch Austesten aller Funktionen und kritischer Befehls- bzw. Ereigniskombinationen, überprüft werden. Die so verifizierte Verhaltensbeschreibung bildet fortan die Referenz für alle weiteren Simulationen auf den verschiedenen Beschreibungsebenen. Im weiteren kann so mit der Verhaltensbeschreibung, die in eine entsprechende Testumgebung eingebunden wird, ein Testprogramm für den Mikrocontrollerkern entwickelt werden, dessen Ausführung auf dem Mikrocontrollerkern in unterschiedlichen Beschreibungsebenen automatisch simuliert und die Simulationsprotokolle automatisch mit der Referenzsimulation auf Verhaltensebene verglichen werden können. Eventuelle Fehler und Unstimmigkeiten beim Übergang in eine andere Beschreibungsform können so automatisch erkannt und daraufhin beseitigt werden. Bei Vorhandensein eines PIC17-Simulators könnte das Testprogramm und die Referenzsimulation noch einen Schritt früher direkt auf dem PIC17 entwickelt werden, um so, ausgehend von der Korrektheit des Simulators, alle Beschreibungen des Mikrocontrollerkerns, inklusive der Verhaltensbeschreibung, zu verifizieren.

Zur weiteren, automatischen Simulation aller VHDL-Beschreibungsformen des Mikrocontrollerkerns wurde eine VHDL-Testumgebung entwickelt. In der Testumgebung wird die für die Simulation des realen Betriebs des Prozessormakros nötige Peripherie um den Mikrocontrollerkern herum nachgebildet und an die Schnittstellen des Prozessormakros angeschaltet. Bei der angeschalteten Peripherie handelt es sich um Daten- und Programmspeicher, d.h. RAM und ROM, und um einen sog. *Tracer*, der sämtliche Ein- und Ausgänge des Mikrocontrollerkerns sowie die im Kern enthaltenen Register überwacht und so aus dem Ablauf der VHDL-Simulation eine Steuerungsdatei für die Simulation des Kerns auf Gatterebene erzeugt. RAM, ROM und Tracer der Testumgebung sind als reine VHDL-Verhaltensbeschreibungen implementiert. Da diese Blöcke nicht synthetisiert werden müssen, kann man alle Möglichkeiten von VHDL ausnützen. Mit Hilfe der von VHDL zur Verfügung gestellten hochsprachlichen Mittel lassen sich die Module entsprechend einfach und elegant nachbilden bzw. implementieren. Der Vorteil der Testumgebung liegt darin, daß bei

gleichbleibender Peripherie die verschiedenen Beschreibungsformen des Kerns durch einfachen Austausch der in der Testumgebung für die Mikrocontrollerkern-Entity eingesetzten Architektur simuliert werden können. Der genaue Aufbau der Testumgebung und die eingesetzten Periphieriemodule RAM, ROM und Tracer werden im nächsten Abschnitt 4.1.2 näher beschrieben.

Die Testumgebung erlaubt es, ein für den PIC17 entwickeltes Assemblerprogramm auf dem Mikrocontrollerkern simuliert auszuführen. Zu Beginn einer Simulation initialisiert das ROM seinen Inhalt dazu mit den in einer Datei gespeicherten Opcodes des übersetzten Programms. Das Testprogramm selbst wird mit einem beliebigen, für den PIC17 erhältlichen Assembler erzeugt, übersetzt und gelinkt. Die vom Assembler bzw. Linker erzeugte Binärdatei im sog. Intellec Hex-Format wird dann vom ROM der Testumgebung eingelesen. Da für die SUN-Workstations kein PIC17-Assembler von Microchip Technology zur Verfügung steht, wurde das Testprogramm auf einem PC entwickelt und übersetzt, und die erzeugte Hex-Datei schließlich auf die Workstation übertragen. Das Testprogramm ist so aufgebaut, daß es möglichst alle Funktionen des Mikrocontrollerkern auf ihre Korrektheit hin automatisch austestet und überprüft. Dazu zählen einerseits natürlich die korrekte Abarbeitung aller Instruktionen des Befehlssatz, die verschiedenen Adressierungsarten, Speicherzugriffe, Interrupts, die Operationen des Ereigniszählermoduls, die Funktionalität aller interner und externer Register, die Stacklogik, usw. Des weiteren führt das Testprogramm auch einige kritische Operationen bezüglich der internen Signallaufzeiten aus, so daß dadurch die internen kritischen Pfade ausgetestet und daraus die maximale Taktfrequenz abgeleitet werden kann.

Bei der eigentlichen Simulation wird die Testumgebung in den VHDL-Simulator geladen, wobei entweder die Verhaltens- oder die RTL-Beschreibung des Mikrocontrollerkerns eingebunden wird. Bei der reinen Simulation und Verifikation der verschiedenen VHDL-Beschreibungen des Mikrocontrollerkerns für den Tracer ein sog. Dummy-Tracer eingebunden, der keinerlei Funktion erfüllt und keine Stimulidatei erzeugt. Nach dem Laden des Testprogramms wird über einen Steuerungsmakro des Simulators die Simulation gestartet, und die Testumgebung führt das angegebene Testprogramm, in Zusammenarbeit mit dem Steuerungsmakro, bis zu dessen Ende aus. Die während der Simulation auftretenden internen Registerinhalte und die vom Mikrocontrollerkern erzeugten Ausgangssignale werden vom Simulator protokolliert und nach Ende der Simulation durch den Steuerungsmakro in tabellarischer Form in zwei Dateien abgespeichert. Durch Vergleich der bei der Simulation der RTL-Beschreibung erhaltenen Protokolldateien mit den Protokolldateien der Simulation der Verhaltensbeschreibung werden Fehler und Unstimmigkeiten der RTL-Beschreibung aufgedeckt und können anschließend korrigiert werden. Die Simulation ist dabei so ausgelegt, daß sich Fehler im Kern bei Ablauf des Testprogramms direkt in fehlerhaften Registerinhalten, Programmabläufen (zurückzuführen auf falschen Wert im Programmzähler) und Ausgangssignalen niederschlägt. Der Vergleich der Protokolldateien der Verhaltens- und RTL-Simulationen wird automatisch mit Hilfe zweier kleiner C-Programme, die die Protokolle parallel Zeile für Zeile einlesen und vergleichen, durchgeführt.

Ist schließlich die Korrektheit der RTL-Beschreibung durch Vergleich der VHDL-Simulationen von Verhaltens- unf RTL-Beschreibung verifiziert, so kann die RTL-Beschreibung synthetisiert werden. In einem letzten Schritt wird auf VHDL-Ebene noch die Steuerungsdatei für die Simulation auf Gatterebene erzeugt. Dazu wird ein entsprechender Tracer in die Testumgebung eingebunden und die Simulation erneut durchgeführt. Der Tracer erzeugt aus dem Simulationsablauf und den Simulationsdaten die Stimulidatei für die Gattersimulation, so daß auf Gatterebene ebenfalls der Ablauf ein und desselben Testprogramms simuliert und überwacht wird. Je nach Testumgebung auf Gatterebene ändert sich die notwendige Stimulidatei geringfügig und es muß der passende Tracer eingebunden werden.

Zur Durchführung der Simulationen auf VHDL-Ebene kam der Simulator V-System zum Einsatz [7]. Das Testprogramm wurde in Assembler entwickelt und mit dem Programmpaket ASM17 von Microchip Technology (das u.a. [10] beiliegt) übersetzt.

4.1.2 Testumgebung

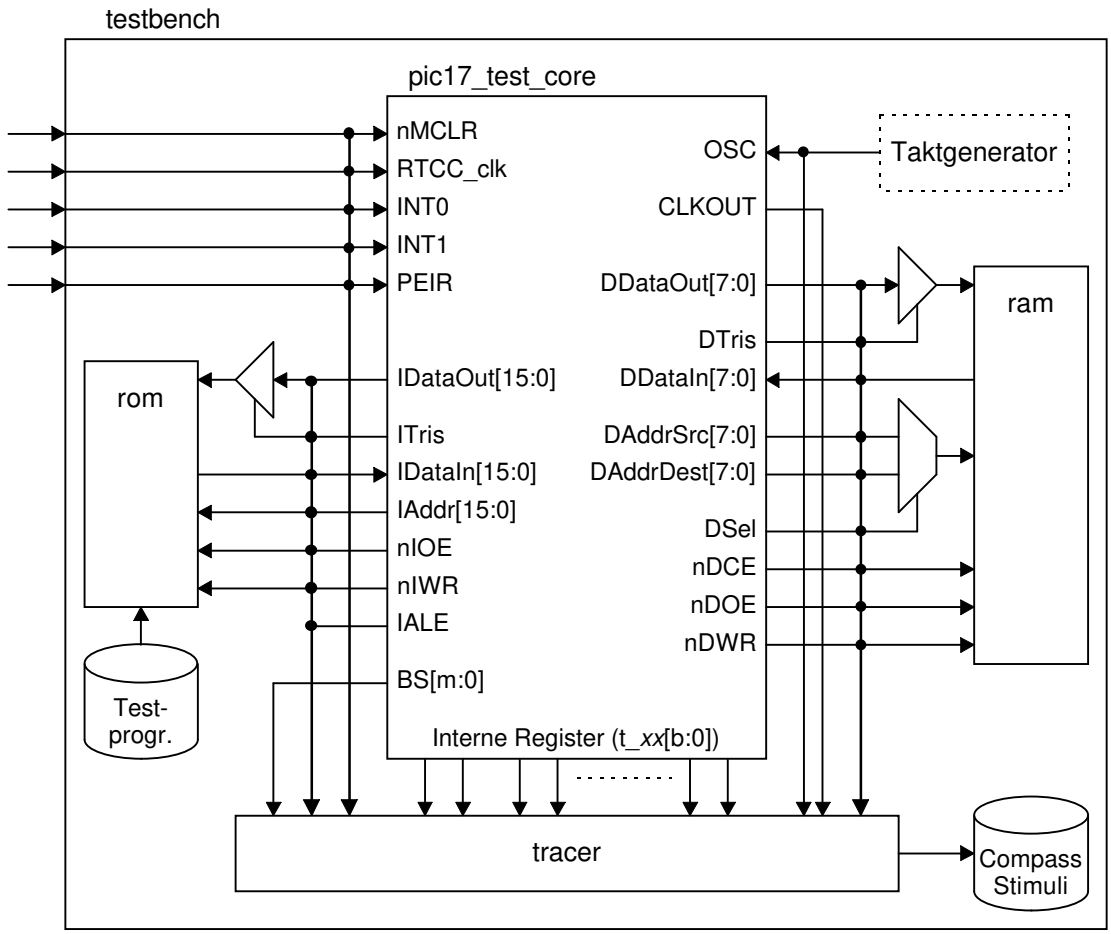


Abbildung 4.1: VHDL-Testumgebung.

Der Aufbau der Testumgebung ist in Abbildung 4.1 wiedergegeben. Zentrales Element ist die zu testende Beschreibung des Mikrocontrollerkerns. Im Gegensatz zu der für die Synthese verwendeten Entity (*pic17_core*) kommt bei der Testumgebung aber eine spezielle Version des Mikrocontrollerkerns (*pic17_test_core*) zum Einsatz. Die beiden Varianten unterscheiden sich aber nur in einem Punkt an der Schnittstelle des Mikrocontrollerkerns: die Schnittstelle der Testversion ist um zusätzliche Ausgänge erweitert, über die die Werte aller internen Register nach außen gegeben werden. Diese Ausgänge werden in der Testumgebung benötigt, damit der Tracer Zugriff auf die Registerinhalte erhält.

Neben dem Tracer enthält die Testumgebung noch zwei separate Unterblöcke, die das RAM und ROM implementieren und an die Daten- bzw. Programmspeicherschnittstellen des Mikrocontrollerkerns angeschlossen sind. Die Schnittstellen des RAM und des ROM sind an reale Module angelehnt und die beiden Blöcke werden entsprechend über Tristatetreiber für die Datenbusse und Multiplexer zur Adreßauswahl an den Kern angeschaltet. Die Testumgebung implementiert weiterhin einen kleinen Taktgenerator, der den Systemtakt erzeugt und an den Mikrocontrollerkern anlegt. Die Taktfrequenz des Systemtakts wird der Testumgebung, neben der Stacktiefe und der Breite des BS Registers, über ein VHDL-Generic übergeben. Die Reset-, Interrupt- und Ereigniszählereingänge des Mikrocontrollerkerns werden nicht von der

Testumgebung angesteuert und werden deshalb an der Schnittstelle der Testumgebung nach außen durchgereicht. Diese Eingänge müssen bei der Simulation extern, d.h. im VHDL-Simulator, wie gewünscht angesteuert werden. Im allgemeinen werden dadurch ausgelöste Ereignisse vom Mikrocontrollerkern vollkommen orthogonal zur Ausführung des Testprogramms abgearbeitet und sind deshalb im Prinzip unabhängig vom Programm selbst. Während der Simulation können die entsprechenden Reaktionen des Mikrocontrollerkerns normalerweise zu beliebigen Zeitpunkten extern ausgelöst und getestet werden, wobei sich die Ereignisse in den zugehörigen Simulationsprotokollen durch Änderungen an Registerinhalten oder im Programmablauf niederschlagen müssen. Zur Überprüfung einiger Funktionalitäten des Mikrocontrollerkerns erwartet das Testprogramm aber, daß einige Eingänge zu ganz bestimmten Zeitpunkten passend angesteuert werden, was durch den parallel zur Simulation laufenden Steuerungsmakro des Simulators sichergestellt werden muß.

In den folgenden Unterabschnitten werden die drei in der Testumgebung separat realisierten Unterblöcke RAM, ROM und Tracer nochmals detaillierter beschrieben.

4.1.2.1 RAM

Das RAM ist als reine Verhaltensbeschreibung auf hohem Abstraktionsniveau realisiert. Es simuliert einen RAM-Block der Größe 256 x 8 Bit. Neben dem Adreßeingang und den Datenein- und ausgängen werden Schreib- und Lesezugriffe auf das RAM über die Steuerleitungen nCE, nOE und nWR gesteuert. Das Verhalten des RAM selbst ist stark idealisiert und möglichst einfach gehalten. Ein Schreibzugriff auf das RAM wird durchgeführt, wenn nCE und nWR aktiv sind, so daß dann die am Dateneingang liegenden Daten bei Beendigung des Schreibvorgangs (entweder nCE oder nWR wird inaktiv) in die adressierte Zelle übernommen werden. Bei einem Lesezugriff hingegen müssen nCE und nOE aktiv sein und das RAM legt die Daten der adressierten Zelle auf den Datenausgang. Dem RAM kann dabei über ein Generic ein gewisse Zugriffszeit, um die die Daten bei einem Lesezugriff am Ausgang verzögert werden, übergeben werden. Als einfache Plausibilitätskontrolle stellt das RAM sicher, daß während eines Zugriffs die Adreßleitungen stabil anliegen.

Zu Beginn der Simulation sind alle Speicherzellen des RAM uninitialisiert und müssen erst beschrieben werden, bevor ein definierter Wert ausgelesen werden kann.

4.1.2.2 ROM

Ebenso wie das RAM ist das ROM als reine Verhaltensbeschreibung stark idealisiert und vereinfacht realisiert. Das ROM hat eine Größe von 65536 x 16 Bit und kann über einen Adreßeingang, Datenein- und Ausgänge und die Steuerleitungen nOE und nWR gelesen und geschrieben werden. Über die Aktivierung von nOE wird ein Lesezugriff gesteuert und über die Aktivierung von nWR ein Schreibzugriff. Die Zugriffszeit, d.h. die Verzögerung der Daten am Ausgang bei einem Lesezugriff, wird dem ROM über ein Generic übergeben. Während eines Zugriffs wird überprüft, daß die Adreßleitungen stabil anliegen.

Das ROM lädt zu Beginn der Simulation seinen Inhalt aus einer per Generic übergebenen Objektdatei, die mit einem Assembler und Linker aus einem Testprogramm erzeugt wurde. Beim Laden der Datei werden die in der Datei adressierten Speicherzellen mit den in der Datei angegebenen Werten initialisiert. Das ROM unterstützt folgende Dateiformate, wobei der Dateityp dem ROM ebenfalls per Generic übergeben wird:

- PICES (PIC-spezifisches Format)
- INHX8S (Split 8 Bit Intellec Hex Format)
- INHX8M (Merged 8 Bit Intellec Hex Format)
- INHX32 (16 Bit Intellec Hex Format)

Standardmäßig liest das ROM die Daten aus der Datei *testprog* im Format INHX32. Das Format INHX32 ist auch das vom Assembler ASM17 standardmäßig erzeugte Format und sollte deshalb im Normalfall verwendet werden.

4.1.2.3 Tracer

Der Tracer der Testumgebung ist ein Block, der sämtliche Ein- und Ausgänge des Mikrocontrollerkerns und die Werte aller im Kern enthaltener Register, einschließlich der nicht in den Datenspeicherbereich eingeblendeten Spezialregister (z.B. Programmzähler), überwacht und so aus dem Ablauf der VHDL-Simulation eine passende Steuerungsdatei für die Durchführung der gleichen Simulation auf Gatterebene erzeugt. Dem Tracer werden dazu einerseits sämtliche Signale an der Schnittstelle des Mikrocontrollerkerns zugeführt. Andererseits erhält der Tracer über die zusätzlichen Testausgänge der speziellen Testversion des Mikrocontrollerkerns die Werte aller im Kern enthaltener Register.

In der erzeugten Steuerungsdatei werden alle Eingänge des Mikrocontrollerkerns auf Gatterebene entsprechend dem zeitlichen Verlauf der VHDL-Simulation angesteuert. Parallel dazu erzeugt der Tracer in der Steuerungsdatei regelmäßige Testabfragen der Ausgänge und Registerinhalte des Mikrocontrollerkerns, um Fehler im Ablauf der Simulation auf Gatterebene zu erkennen. Zu Beginn der Simulation auf Gatterebene werden in der Steuerungsdatei diverse globale Einstellungen und Konfigurationen, wie z.B. die Zusammenfassung einzelner Leitungen eines Busses zu einem Vektor, vorgenommen. Insgesamt kann so auf Gatterebene die VHDL-Simulation automatisch exakt nachgebildet und verifiziert werden.

Da auf Gatterebene zwei verschiedene Testkonfigurationen (mit oder ohne angeschaltetem RAM, siehe Abschnitt 4.3) existieren, wurden zwei verschiedene Versionen des Tracers entwickelt. Die beiden Tracer erzeugen jeweils Steuerungsdateien für die Simulation auf Gatterebene mit oder ohne RAM, wobei bei der Steuerungsdatei für die Gattertestumgebung mit RAM die Ansteuerung der zugehörigen Datenspeichereingänge des Mikrocontrollerkerns entfällt. Schließlich ist noch ein weitere Tracer vorhanden, der keinerlei Operationen durchführt und keine Steuerungsdatei erzeugt. Dieser Tracer wird bei normalen VHDL-Simulationen, die zur Überprüfung der VHDL-Beschreibungen durchgeführt werden und wenn deshalb noch keine Steuerungsdatei benötigt wird, verwendet, um diese Simulationen zu beschleunigen.

4.2 Synthese

Nach der Erstellung und Überprüfung der RTL-Beschreibung des Mikrocontrollerkerns per Simulation auf VHDL-Ebene und per Vergleich mit den Simulationsergebnissen der Verhaltensbeschreibung wird die RTL-Beschreibung synthetisiert und daraus automatisch eine Gatternetzliste des Mikrocontrollerkerns erzeugt. Die Synthese wurde mit den Programmen Design Analyzer und Design Compiler der Firma Synopsys durchgeführt [12]. Zur automatischen, reproduzierbaren Synthese existiert ein Steuerungsskript für Synopsys, das alle erforderlichen Schritte zur Erzeugung der Gatternetzliste enthält. Der prinzipielle Ablauf der Synthese durch das Skript besteht aus den folgenden Schritten:

- Einlesen der VHDL-Quelldateien der RTL-Beschreibung des Mikrocontrollerkerns.
- Setzen diverser Parameter zur Steuerung des Synthesevorgangs, zur Festlegung der Optimierungsstrategien, zur Eingrenzung der Zeit-, Takt- und Platzkriterien, usw.
- Durchführen eines ersten Synthesevorgangs mit Erzeugung einer ersten Gatternetzliste.
- Entfernen der Hierarchie aus dem Design und Anpassen bzw. ggf. Neusetzen diverser Syntheseparameter.
- Inkrementelle Nachoptimierung der Gatternetzliste des flachen Designs.

Nach jedem Syntheseschritt werden neben den bisherigen Ergebnissen der Synthese die erhaltene Gatternetzliste sowohl im Synopsys-eigenen als auch im standardisierten EDIF-Format abgespeichert. Am Ende erhält man so die endgültige Gatternetzliste des Mikrocontrollerkerns.

Vor der eigentlichen Synthese müssen eine Reihe von Parameter in Synopsys gesetzt werden, um einerseits die Synthese überhaupt erfolgreich durchführen zu können und andererseits den Synthesevorgang zu optimieren. Zum einen müssen dem Syntheseprogramm die für die Optimierung nötigen Randbedingungen mitgeteilt werden. Dazu werden die einzelnen intern aus dem externen Systemtakt abgeleiteten Taktsignale und ihre gegenseitige Abhängigkeit festgelegt, damit das Syntheseprogramm die Zeitbeziehungen zwischen den einzelnen, durch verschiedene Taktsignale angesteuerten Registern korrekt ableiten kann. Bei der Synthese versucht das Programm das Design so zu optimieren, daß alle Zeitbedingungen, inklusive Gatterverzögerungen, Setup- und Hold-Zeiten, eingehalten werden. Das Syntheseprogramm selbst kann den zeitlichen Zusammenhang zwischen den Taktsignalen nicht automatisch erkennen und muß diesen deshalb vor der Synthese mitgeteilt bekommen. Für alle Ein- und Ausgänge des Designs müssen die einzuhaltenden Zeitkriterien ebenfalls explizit angegeben werden.

Auf der anderen Seite kann die Vorgehensweise des Syntheseprogramms bei der Optimierung über eine Vielzahl von Parameter beeinflußt werden. Je nach Art eines Blocks kann es von Vorteil sein, diese Parameter zu verändern. Ausgehend von ein paar wenigen Grundregeln des Synthesevorgangs kann eine optimale Lösung nur durch zeitaufwendiges Probieren gefunden werden. Im allgemeinen erhält man aber mit den Standardeinstellungen gute Ergebnisse.

Schließlich werden dem Syntheseprogramm noch Informationen über einige Besonderheiten des Mikrocontrollerkerndesigns mitgegeben, die von den allgemeinen Annahmen abweichen und vom Programm nicht selbst erkannt werden können. Dazu zählen hauptsächlich Signalpfade, die im Betrieb des Mikrocontrollerkerns nie auftreten können und die deshalb auch bei der Optimierung nicht berücksichtigt werden müssen. So wird z.B. ein Latch im Mikrocontrollerkern nie ausgelesen während es transparent ist, d.h. während es beschrieben wird, da alle nachgeschalteten Multiplexer und andere Blöcke bei transparentem Latch den Wert des Latches nicht auswerten bzw. weiterverarbeiten. Dem Syntheseprogramm muß deshalb explizit mitgeteilt werden, daß alle Pfade von einem Register durch das (potentiell) transparente Latch zu einem anderen Register nicht berücksichtigt werden müssen. Andernfalls versucht das Syntheseprogramm unnötigerweise auch solche Pfade so zu optimieren, daß die Daten nicht nur beim Latch selbst, sondern auch beim nachgeschalteten Register noch rechtzeitig ankommen.

Insgesamt wird bei der Synthese eine zweistufige Strategie verfolgt. Nach der ersten Synthese des eingelesenen Designs wird die Hierarchie aus dem Design entfernt und danach versucht, durch erneute Synthese des flachen Designs eine mögliche Verbesserung zu erreichen. Normalerweise optimiert das Syntheseprogramm jeden Block separat ohne Einbeziehung des restlichen Designs. Erst nach der Entfernung der Struktur aus dem Design ist es so ggf. möglich, alle Kombinationen bei der Optimierung zu berücksichtigen. Aus diesem Grund wird in dem zweiten Schritt versucht, entsprechende Vorteile auszuschöpfen. Am Ende des Prozesses steht dann die endgültige Gatterbeschreibung des Mikrocontrollerkerns.

4.3 Simulation auf Gatterebene

Die Simulation der synthetisierten Netzliste des Mikrocontrollerkerns auf Gatterebene stellt im Normalfall den letzten Schritt bei der Entwicklung des Designs dar. Ziel dabei ist es, die ordnungsgemäße Funktion der endgültigen, durch die automatische Synthese aus der RTL-Beschreibung gewonnenen Gatternetzliste nachzuprüfen. Bei der Simulation auf Gatterebene werden zum ersten Mal auch die Gatterlaufzeiten und die dadurch entstehenden

Signalverzögerungen berücksichtigt. Zum einen wird also durch die Gattersimulation überprüft, daß das Design bei der Synthese fehlerfrei umgesetzt wurde. Zum anderen wird untersucht, inwieweit Laufzeiteffekte Einfluß auf den fehlerlosen Betrieb des Mikrocontrollerkerns bei verschiedenen Taktfrequenzen haben. Werden dabei Unstimmigkeiten, Fehler oder Problemzonen erkannt, wird das Design auf RTL-Ebene entsprechend korrigiert und erneut synthetisiert. Erst wenn auch die Simulation auf Gatterebene erfolgreich abgeschlossen ist, kann die Gatternetzliste des Mikrocontrollerkerns zur Produktion freigegeben werden. Zu beachten ist, daß bei der hier beschriebenen Simulation auf Gatterebene die durch die Verbindungsleitungen zwischen den Gattern verursachten Verzögerungen nicht berücksichtigt werden können. Informationen darüber stehen erst nach der Plazierung und Verdrahtung der Gatter auf dem Chip zur Verfügung. Nach der Plazierung und Verdrahtung muß also der fertige Chip erneut auf Gatterebene, unter Berücksichtigung der Leitungskapazitäten simuliert und getestet werden, bevor er tatsächlich hergestellt werden kann.

Die Gattersimulation wird mit dem „Mixed Mode Simulator“ aus dem Programmpaket Compass [4] durchgeführt. Vor der Simulation auf Gatterebene muß die von Synopsys erzeugte Gatternetzlise deshalb zuallererst in das Compass-eigene Netzlistenformat umgewandelt werden. Mit Hilfe des Netzlistenkonvertierungsprogramms von Compass wird dazu die von Synopsys geschriebene Gatternetzliste im standardisierten EDIF-Format eingelesen, konvertiert und als Compass-Netzliste abgespeichert. Bei der Netzlistenkonvertierung traten aber mit verschiedenen Versionen des Konvertierungsprogramms unterschiedliche Probleme und Fehler auf. Nach mehreren Versuchen mit verschiedenen Programmversionen und EDIF-Formaten führte nur die Verwendung einer älteren Version des Compass-Programms zusammen mit einer flachen EDIF-Netzliste zum Erfolg. Hierarchische EDIF-Netzliste werden vom alten Konvertierungsprogramm nicht richtig verarbeitet, wohingegen die aktuelle Version des Konvertierungsprogramms zwar hierarchische Netzlisten korrekt einliest, aber dafür z.B. Bussignale verdreht. Um die Gatternetzlisten des Mikrocontrollerkerns weiterverarbeiten zu können, müssen deshalb bei der Synthese flache Netzlisten im EDIF-Format abgespeichert werden.

Im weiteren Verlauf wird die konvertierte Netzliste des Mikrocontrollerkerns auf Gatterebene simuliert. Als einfachste Möglichkeit wird dazu die reine Netzliste des Mikrocontrollerkerns in den Simulator geladen, wobei dann aus dem Simulator heraus alle Eingänge des Kerns angesteuert und alle vom Mikrocontrollerkern, als Reaktion auf die Stimuli der Eingangssignale angesteuerten Ausgänge abgefragt werden müssen. Zusätzlich wurde eine erweiterte Testumgebung für den Mikrocontrollerkern auf Gatterebene mit Hilfe des „Logic Assistant“ in Compass entworfen. Dabei wurde die Gatternetzliste des Mikrocontrollerkerns mit einer Beschreibung des am IMS vorhandenen Dual-Ported RAM kombiniert. Das RAM ist, entsprechend einer später real möglichen Schaltung, an die Datenspeicherschnittstelle des Mikrocontrollerkerns, wie in Abbildung 4.2 gezeigt, angeschaltet und realisiert so bei der Gattersimulation den später evtl. mit auf dem Chip implementierten Datenspeicher. Um den kritischsten Fall abzudecken, besitzt das RAM entsprechend der maximal möglichen Ausbaustufe eine Größe von 256 x 8 Bit. Die Testumgebung erlaubt so eine genauere Nachbildung und eine automatische Simulation der realen Verhältnisse an den Ein- und Ausgängen der kritischen Datenspeicherschnittstelle des Mikrocontrollerkerns. Speziell die Zugriffszeiten des RAM und die daraus resultierenden Verzögerungen werden dadurch bei der Simulation berücksichtigt. Alle anderen Ein- und Ausgänge des Mikrocontrollerkerns müssen aber weiterhin aus dem Simulator heraus angesteuert bzw. abgefragt werden.

Die eigentliche Gattersimulation ist durch Steuerungsdateien für den Simulator weitestgehend automatisiert. Die Steuerungsdateien für die Simulation der reinen Netzliste bzw. für die Simulation der Testumgebung werden bei der Simulation auf VHDL-Ebene

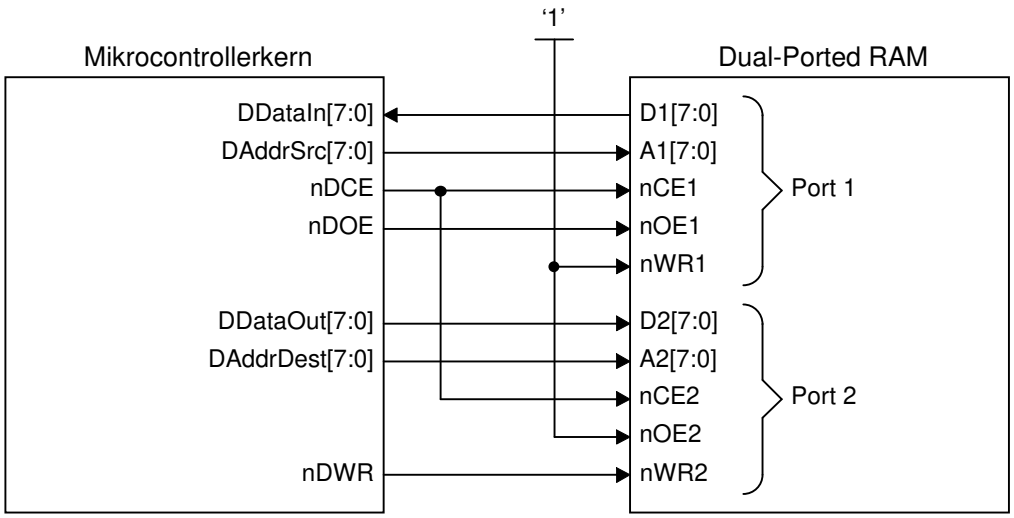


Abbildung 4.2: Testumgebung auf Gatterebene.

automatisch durch entsprechende Tracer erzeugt. Diese Stimulidateien steuern alle offenen Eingänge an und testen alle offenen Ausgänge auf die korrekten Reaktionen des Mikrocontrollerkerns. Dadurch wird die schon auf VHDL-Ebene durchgeführte Simulation, d.h. der Ablauf des Testprogramms, auf Gatterebene exakt nachgebildet. Durch Ausführen der Steuerungsdatei im Simulator werden so alle Funktionen des Mikrocontrollerkerns auf Gatterebene erneut ausgetestet. Nach dem Laden der Netzliste bzw. der Testumgebung in den Simulator wird die Steuerungsdatei geladen und damit ausgeführt. Im weiteren läuft die Simulation gesteuert durch die Datei vollkommen automatisch ab und dabei auftretende Fehler werden über den Simulator gemeldet. Gegebenenfalls können vor dem Laden der Steuerungsdatei noch diverse Simulationsparameter gesetzt oder Signaltraces aktiviert werden, um z.B. Verzögerungsfaktoren zu verändern oder Signalverläufe während der Simulation zu beobachten.

Durch verschiedene Testläufe des Mikrocontrollerkerns auf Gatterebene mit oder ohne angeschaltetem RAM lassen sich einerseits prinzipielle Fehler und Problemzonen des Designs erkennen. Andererseits können durch Variation der Umgebunsbedingung bei einzelnen Testläufen die Leistungsfähigkeit, die Grenzen und die Eigenschaften des Mikrocontrollerkerns getestet werden. Bei der reinen funktionalen Simulation wird die Taktrate relativ niedrig gewählt, um Probleme durch Laufzeiteffekte auszuschließen. Erst wenn die ordnungsgemäße Funktion des Mikrocontrollerkerns prinzipiell sichergestellt ist, wird die Taktrate bei der Simulation solange erhöht, bis Fehler, verursacht durch interne Verzögerungen, auftreten. Insgesamt läßt sich so die maximale Taktrate ermitteln, bei der der Mikrocontrollerkern noch fehlerfrei arbeitet. Die Taktrate für die Simulation auf Gatterebene wird bereits durch die bei der VHDL-Simulation eingestellte Taktrate über die Steuerungsdateien festgelegt, d.h. um die Taktrate zu ändern, muß durch erneute VHDL-Simulation mit geänderter Taktrate eine neue Steuerungsdatei erstellt werden.

4.4 Ergebnisse

In diesem Abschnitt werden die bei der Synthese und Simulation gewonnenen Ergebnisse, d.h. die Daten und Eigenschaften des Mikrocontrollerkerns, zusammengefaßt. Der Mikrocontrollerkern wurde auf Basis der neuen 0,8 µm Technologie des IMS (GFN120) synthetisiert und entwickelt, weshalb sich alle hier wiedergegebenen Ergebnisse darauf beziehen.

In Tabelle 4.1 ist die Größe der Gatterbeschreibung des Mikrocontrollerkerns, jeweils nach dem ersten Syntheseschritt und in der endgültigen Version nach der Nachoptimierung, wiedergegeben. Die hier verwendete Einheit *Sites* dient als Maßstab für die Größenabgaben bei IMS Bibliothekszellen. Die Größenangaben beziehen sich hier auf einen Prozessormakro mit einem Stack der Tiefe 16, d.h. einem Stackspeicher der Größe 16 x 16 Bit, und mit einem 4 Bit breiten BS Register. Allein der Stackspeicher nimmt dabei ungefähr eine Größe von ca. 2700 Sites ein, so daß sich bei einem kleinere Speicher die Größe des Mikrocontrollerkerns deutlich verringern läßt. In der Tabelle sind die Größen jeweils für einen Mikrocontrollerkern angegeben, bei dem alle Register mit maskierter Taktleitung bzw. über einen vorgeschalteten Multiplexer angesteuert werden. Daraus läßt sich die maximale Platzersparnis bei Verwendung von maskierten Taktleitungen bzw. der Platznachteil bei Verwendung von zusätzlichen Multiplexern ablesen. Werden nur einige Register unterschiedlich angesteuert, so resultiert eine Mikrocontrollerkerngröße im Bereich dazwischen. Für die weiteren Untersuchungen wurde stets die Version des Mikrocontrollerkerns unter Verwendung von maskierten Taktleitungen bei allen Registern eingesetzt.

Register	1. Syntheseschritt	Nachoptimierung
Gated Clock	7019 Sites	6910 Sites
Multiplexer	7402 Sites	7237 Sites

Tabelle 4.1: Größe des synthetisierten Mikrocontrollerkerns.

Bei der Simulation des Mikrocontrollerkerns auf Gatterebene sowohl mit als auch ohne angeschaltetem RAM wurde die maximale Frequenz des externen Systemtakts, bei der der Mikrocontrollerkern noch ordnungsgemäß arbeitet, untersucht. Die Ergebnisse sind in Tabelle 4.2 dargestellt. Zu beachten ist, daß der hier angegebene externe Systemtakt intern durch vier geteilt wird und deshalb der maximale Instruktionstakt ein Viertel der hier angegebenen Werte beträgt. Zum Vergleich dazu arbeitet der originale PIC17 mit externen Systemtakten von 16 MHz bzw. 25 MHz (in Planung). Bei der Simulation der reinen Gatternetzliste ohne angeschaltetes RAM wird durch die Steuerungsdatei an der Datenspeicherschnittstelle des Mikrocontrollerkerns ein ideales RAM ohne Zugriffszeiten simuliert. In der Testumgebung hingegen verhält sich das angeschaltete RAM genau wie ein später mit auf dem Chip integrierter Datenspeicher. Da das RAM bei der Simulation der Testumgebung eine maximale Ausbaustufe und damit auch maximale Zugriffszeiten besitzt, kann davon ausgegangen werden, daß bei Einsatz eines kleineren RAMs bessere Werte zu erzielen sind.

Verzögerungsfaktor	ohne RAM (ideales RAM)	mit RAM (256 x 8 Bit)
Normal (1,0)	100 MHz	62,5 MHz
Langsam (1,85)	62,5 MHz	45,5 MHz

Tabelle 4.2: Maximale externe Systemtaktfrequenz des Mikrocontrollerkerns.

Bei der Ermittlung der maximalen Taktfrequenzen wurden zusätzlich auch die Arbeitsbedingungen des Mikrocontrollerkerns entsprechend einem späteren Einsatzbereich unterschiedlich simuliert. Hierzu wird im Simulator bei den Simulationsparameter der Verzögerungsfaktor, mit dem alle internen Verzögerungszeiten der Gatter multipliziert

werden, verändert. Im Normalfall hat der Verzögerungsfaktor einen Wert von eins, entsprechend einer normalen Betriebsspannung von 5 V bei Zimmertemperatur und einem normalen Produktionsprozeß. Zur Simulation des Grenzbereichs im langsamsten Fall wird der Verzögerungsfaktor auf 1,85 gesetzt. Dies entspricht einer Betriebsspannung von 4,5 V, einer Betriebstemperatur von 85° C und einem langsamen Produktionsprozeß (d.h. größere Gatterlaufzeiten durch größere Chipkapazitäten innerhalb der Toleranz). Die maximalen Taktfrequenzen für beide Fälle sind in der Tabelle angegeben. Zusätzlich wurde noch mit einem Verzögerungsfaktor von 0,4 der untere Grenzbereich ausgetestet (5,5 V, 0° C, schneller Prozeß). Dabei wurde aber nur untersucht, ob der Mikrocontrollerkern noch ordnungsgemäß arbeitet, d.h. speziell ob es durch die verkürzten Verzögerungszeiten zu Hold-Verletzungen kommt. Im allgemeinen arbeitet der Mikrocontrollerkern auch in diesem Bereich noch fehlerfrei. Nur bei angeschaltetem RAM kommt es zu einer Hold-Verletzung am Adreßeingang des RAM, da die vom Mikrocontrollerkern gelieferten Adressen nicht lange genug anstehen. Gegebenenfalls muß dies durch eine zusätzliche externe Verzögerung der Adreßausgänge des Mikrocontrollerkerns korrigiert werden.

5 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde ein neuer Prozessorkern für den Mikrocontrollerbaukasten am IMS entwickelt. Der Mikrocontrollerkern ist befehlssatzkompatibel zum Mikrocontroller PIC17 von Microchip Technology und übernimmt dessen prinzipiellen architekturellen Merkmale und Eigenschaften. Hauptmerkmal ist eine auf RISC-Grundlagen basierende Struktur des Befehlssatzes und der darauf aufbauenden internen Architektur. Der Mikrocontrollerkern implementiert eine Harvard-Architektur mit getrennten Bussen für Instruktionen und Daten. Der Instruktionsbus hat dabei eine Breite von 16 Bit und der Datenbus eine Breite von 8 Bit. Als spezielle Eigenschaften enthält der Mikrocontrollerkern ein Ereigniszählermodul und unterstützt Interrupts aus vier verschiedenen Quellen unterschiedlicher Priorität.

Ausgehend von modernen Verfahren der ASIC-Entwicklung wurde der Prozessormakro nach der Spezifikation in die Hardwarebeschreibungssprache VHDL umgesetzt. Die VHDL-Beschreibung des Mikrocontrollerkerns wurde anschließend auf VHDL-Ebene simuliert und schließlich mit Hilfe eines Programms zur Schaltungssynthese in eine Gatternetzliste umgewandelt. Abschließend wurde die erhaltene Netzliste auf Gatterebene unter Berücksichtigung der Verzögerungszeiten der eingesetzten Gatter simuliert und endgültig verifiziert. Die Gatternetzliste des Mikrocontrollerkerns kann so als neues Modul in den Mikrocontrollerbaukasten am IMS aufgenommen werden. In zukünftigen Projekten wird der in dieser Arbeit entwickelte Prozessorkern bei der Entwicklung und Herstellung kundenspezifischer Mikrocontroller am IMS zusammen mit diversen ausgewählten Peripheriemodulen auf einem Chip integriert und eingesetzt.

Der Mikrocontrollerkern erreicht im Vergleich zu seinem Vorbild PIC17 deutlich höhere Taktraten. Weitergehende genauere Untersuchungen, die im Rahmen dieser Arbeit nicht mehr durchgeführt werden konnten, lassen sicherlich noch einigen Spielraum für Verbesserungen bezüglich Geschwindigkeit und Größe des Mikrocontrollerkerns erkennen. Im Hinblick auf die Geschwindigkeit stellt auf jeden Fall der Addierer einen Optimierungspunkt dar, da dieser im kritischen Pfad des Prozessors liegt. Der Einsatz eines verbesserten Addierers, z.B. eines sog. Carry-Lookahead Addierers, gegenüber dem momentan eingesetzten Ripple-Carry Addierer wirkt sich deshalb direkt auf die maximal erreichbare Taktrate aus. Gleichzeitig müssen aber ggf. Engpässe an anderen Stellen beseitigt werden. Bezüglich des Flächenverbrauchs läßt vor allem eine andere Realisierung des Stackspeichers, speziell wenn ein großer Stack benötigt wird, deutliche Gewinne erhoffen. Im Vergleich zu den jetzt implementierten normalen Latches wäre z.B. ein RAM deutlich kleiner realisierbar.

Bevor der Mikrocontrollerkern in realen Schaltungen auf einem Chip eingesetzt werden kann müssen auf jeden Fall weitere Überlegungen zur Testbarkeit des fertigen Chips gemacht werden, da es nach der Herstellung eines Chips unumgänglich ist, zu überprüfen, ob die Schaltung einwandfrei arbeitet oder ob einige Transistoren auf dem Chip fehlerhaft sind. Dies ist nötig, da es z.B. durch Verunreinigungen bei der Herstellung von integrierten Schaltungen immer wieder zu Fehlern kommen kann. Im allgemeinen werden zu Testzwecken festgelegte Testvektoren an den fertigen Chip angelegt und die an den Ausgängen des Chip erhaltenen Werte auf ihre Korrektheit hin überprüft. Die Testvektoren müssen so gewählt werden, daß nahezu alle möglichen internen Fehler erkannt werden können. Gegebenenfalls müssen die internen Strukturen des Chips so erweitert werden, daß auch die Werte interner Register von außen im Zuge des Tests gesetzt bzw. ausgelesen werden können. Zumindest müssen aber alle Ein- und Ausgänge des Mikrocontrollerkerns auf dem Chip von außen zugänglich gemacht werden. Aufgrund der begrenzten Anzahl Pins werden normalerweise die einzelnen Signale in

Form eines Schieberegisters verknüpft und können so durch serielles Einschreiben bzw. Auslesen des Schieberegisters über wenige Pins komplett gesetzt bzw. ausgelesen werden. Alles in allem muß der Mikrocontrollerkern im weiteren auf seine Testbarkeit hin untersucht und gegebenenfalls um entsprechende Strukturen erweitert werden. Zusätzlich müssen daraufhin passende Testvektoren für den Mikrocontrollerkern entwickelt und so optimiert werden, daß eine möglichst hohe Fehlerabdeckung, d.h. ein möglichst hoher Prozentsatz durch die Testvektoren entdeckbarer Fehler, erreicht wird. Im Endeffekt wird bei dieser Fehlersimulation durch Stimulation der Eingänge des Mikrocontrollerkerns ein spezielles Testprogramm auf dem Prozessormakro ausgeführt. Es bleibt zu untersuchen, wie groß die Fehlerabdeckung des in dieser Arbeit für den funktionalen Test entwickelten Testprogramms ist. Anhand der daraus getroffenen Aussagen über die vom Programm nicht entdeckbaren Fehler muß gegebenenfalls das Programm für die Fehlersimulation erweitert oder sogar komplett neu geschrieben werden.

Im weiteren wäre es unter Umständen interessant vor dem eigentlichen Einsatz des Mikrocontrollerkerns in realen Applikationen einen Testchip zu produzieren, um die tatsächliche Einsatzfähigkeit des Prozessormakros auf einem Chip zu testen und zu demonstrieren. Gegebenenfalls kann dabei ein RAM und/oder ein ROM mit auf dem Chip integriert werden, so daß man einen funktionsfähigen Mikrocontroller erhält. Der erste Schritt auf dem Weg zum realen Chip bildet die Plazierung und Verdrahtung der Gatter. Am IMS wird dieser Schritt automatisch durch entsprechende Werkzeuge durchgeführt, wobei je nach Bedarf von Hand nachgearbeitet wird. Mit den nach der Verdrahtung erhaltenen Informationen über die durch die Verbindungsleitungen verursachten zusätzlichen Laufzeiten wird anschließend der Testchip erneut auf Gatterebene simuliert und getestet. Erst wenn dabei keine Fehler auftreten kann der Prototyp tatsächlich hergestellt werden. Anhand des so gefertigten Testchips kann der Mikrocontrollerkern dann schließlich im praktischen Einsatz untersucht werden.

Literaturverzeichnis

[1] Asheden, P. J. *The VHDL Cookbook*. First Edition, University of Adelaide, Dept. of Computer Sience, Juli 1990.

[2] Beller, W., Bernath, E., Hainbucher, R., Schwederski, T. *IMS 1.2µm Gate Forest Library 2.0 GFD200*. Institute for Microelectronics Stuttgart, Januar 1994.

[3] Bultmann, Wolfgang. *VLSI-Entwurf eines Pulsbreitenmodulators mit integrierter Capture-Funktion*. Erste Semesterarbeit, Universität Stuttgart, Institut für Mikroelektronik Stuttgart, Institut für Automatisierungs- und Softwaretechnik, 1994.

[4] Compass Design Automation Inc. *Compass Design Automation Documentation V8R3*. Compass Design Automation Inc., 1991.

[5] Hellwig, Frank. *Vom Sand zum Chip, ASIC-Entwicklung heute*. c´t, Magazin für Computertechnik, Mai 1996.

[6] Hennessy, J. L., Patterson, D. A. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, 1990.

[7] Model Technology Inc. *V-System/Workstation, VHDL Simulator for Workstations Version 4*. Model Technology Inc., Beaverton, 1994.

[8] Microchip Technology Inc. *PIC17C42, High Performance 8-bit CMOS EPROM Microcontroller*. Microchip Data Book, 1993.

[9] Perry, Douglas L. *VHDL*. Mc-Graw Hill, 1991.

[10] Rose, Michael. *Mikroprozessor PIC17C42, Architektur und Applikation*. Hüthing Buch Verlag, Heidelberg, 1993.

[11] *Static Dual-Port RAM (DPRAM*). IMS 1.2µm Gate Forest Module Cells. Version 1.0, 2.9.1993.

[12] Synopsys Inc. *Synopsys Online Documentation 3.3a*. Synopsys Inc., Mountain View, 1994.

[13] Tietze, U., Schenk, Ch. *Halbleiter-Schaltungstechnik.* 8.Auflage, Springer Verlag, Berlin, 1986.

[14] Weste, N., Eshragian, K. *Principles of CMOS VLSI Design*. Addison-Wesley, 1988.

[15] Wahl, Volker. *Entwurf und Implementierung eines Semi-Custom Prozessor-Moduls mit RISC-Architektur*. Diplomarbeit, Universität Stuttgart, Institut für Mikroelekronik Stuttgart, Institut für Nachrichtenvermittlung und Datenverarbeitung, 1993.