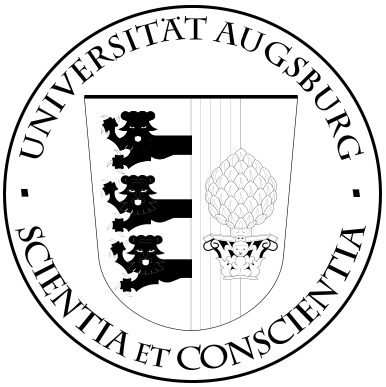
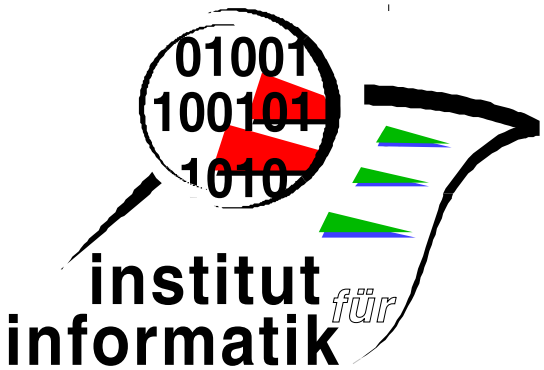
Universität Augsburg



Der Komodo Mikrocontroller

Sascha Uhrig

Report 2003-3 Juli 2003



Institut für Informatik
D-86135 Augsburg

Inhaltsverzeichnis

1 Einleitung

Im Rahmen des im Jahre 2001 begonnen und von der Deutschen Forschungs Gemeinschaft (DFG) geförderten Komodo-Projekts ist ein echtzeitfähiger mehrfädiger Java Mikrocontroller und die dazugehörige Java Virtual Machine (JVM) und das Application Programming Interface (API) entwickelt worden. Der Mikrocontroller ist sowohl in Form eines abstrakten Software Simulators (geschrieben in Java), als auch als VHDL-Beschreibung verfügbar. Die VHDL-Version wurde für die Implementierung in Form eines FPGA-Prototyps verwendet. Die momentan realisierte Version realisiert 6 Hardware Thread Slots bei insgesamt 4 verschiedenen Scheduling Algorithmen. Neben dem Prozessorkern sind ergänzende periphere Einheiten zur Datenübertragung und Signalbehandlung auf dem Prototypen integriert. Ebenfalls wurde eine informelle Synthese mit einer 0.18 micron ASIC-Technologie und 16 Thread Slots durchgeführt.

Über die Projektlaufzeit hinweg wurden verschiedene Themen von unterschiedlichen Personen bearbeitet, die größtenteils nicht mehr im Projekt tätig sind. Im Folgenden sei ein Aufzählung der am Komodo-Projekt beteiligten Personen gegeben, die aber keinen Anspruch auf Vollständigkeit erhebt:

Prof. Dr. Theo Ungerer - Projektleitung
Prof. Dr. Uwe Brinkschulte - Projektleitung
Dr. Jochen Kreuzinger - Softwaresimulator, VHDL Entwurf, JVM; Promotion
Dipl.-Inform. Matthias Pfeffer - JVM, API; Promotion
Dipl.-Inform. Sascha Uhrig - VHDL Entwurf; Diplomarbeit
Dipl.-Inform. Alexander Schulz - Scheduling Algorithmen in VHDL; Diplomarbeit
Dipl.-Inform. Robert Zulauf - VHDL Entwurf; Diplomarbeit
cand. Inform. Stephan Fuhrmann - Speicherbereinigung; Diplomarbeit
cand. Inform. Alexander Lange - Dribbler-Techniken; Studienarbeit
Dipl.-Inform. Christian Krakowski - Middleware
DEA d'Informatique Etienne Schneider - Middleware, Rekonfiguration
Dipl.-Inform. Florentin Picioroaga - Middleware, Dienstkonzept
cand. Inform. Mirko Schur - API; student. Hilfskraft
Dipl.-Ing. Christof Liemke - ASIC Hardware-Synthese; Diplomarbeit

2 Überblick über den Prozessorkern des Komodo-Mikrocontrollers

Der Mikrocontroller besteht aus einer 5 stufigen, mehrfädigen Pipeline und diversen peripheren Komponenten. Die Pipeline besitzt zwei separate Bussysteme; eines für den Zugriff auf den extern angeordneten Programm- und Datenspeicher (von-Neumann Architektur) und ein anderes Bussystem für die im Chip integrierte Peripherie. Die Mehrfädigkeit ist durch die gesamte Pipeline geführt. Ein Befehl wird durch die Pipeline geleitet, wobei er stets von einem Thread-Tag begleitet wird. Unter bestimmten Umständen kann dieses Thread-Tag innerhalb der Pipeline wechseln (s. Abschnitt 4.2). Ein Bruch in dieser Thread-Tag Kette besteht zusätzlich zwischen der ersten und der zweiten Pipelinestufe. Im Folgenden werden die einzelnen Pipelinestufen im Detail erklärt.

2.1 Instruction Fetch Unit (Fetch)

Die Instruction Fetch Unit ist dafür verantwortlich, Befehle aus dem Programmspeicher in die in der nächsten Pipelinestufe (Instruction Decode and Window Unit, IWDU) angeordneten Befehlsfenster zu transportieren. Dazu wird der Füllstand aller Befehlsfenster für die Entscheidung herangezogen, für welchen Thread ein Befehl geholt werden soll. Wurde ein Thread gefunden, dessen Befehlsfenster nicht voll ist, so wird ein Signal an die IWDU gegeben, das den Beginn eines Fetch-Vorgangs bekannt gibt. Für diesen Thread wird anschließend der Befehlszeiger (PC) an die Speicher-Schnittstelle gegeben. Falls das Memory Interface diesen Befehl tatsächlich aus dem Speicher holt, wird dieser an der Fetch-Unit vorbei an die IWDU gereicht. Zusätzlich bekommt die

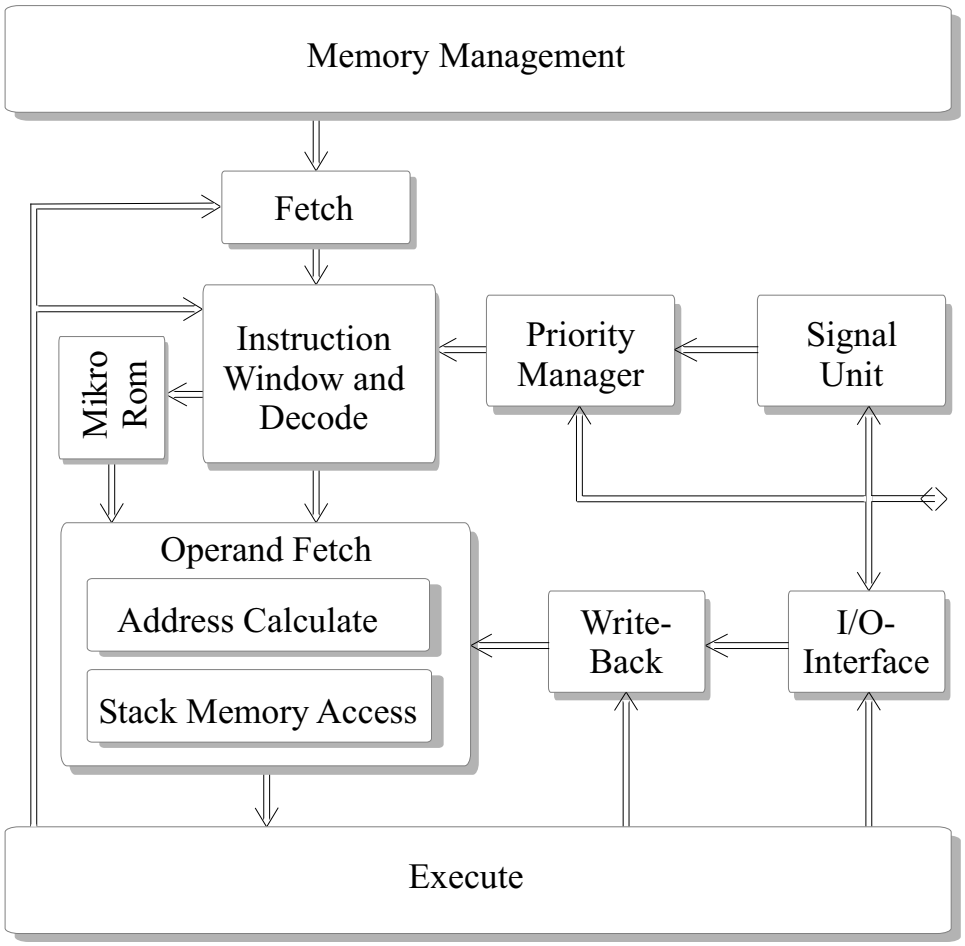


Abbildung 1: Die Komodo-Pipeline

Fetch-Unit das Signal, dass dieser Befehls-Hole-Vorgang erfolgreich abgeschlossen wurde. Bei der Auswahl des nächsten Thread muss der letzte durchgeführte Fetch-Zugriff berücksichtigt werden, da die Informationen über den aktuellen Füllstand einen Takt verzögert bei der IFU ankommen. Wird ein Sprung eines Thread über die entsprechenden Signale angezeigt, so wird der PC dieses Thread neu gesetzt und der neue PC für das Holen eines Befehls genutzt. In der Fetch-Unit wird je Thread ein separater PC geführt. Bei diesem werden allerdings die unteren 2 Bit ignoriert, da diese für die Adressierung nicht notwendig sind; alle Zugriffe erfolgen 4-Byte aligned.

2.2 Instruction Window and Decode Unit (Window & Decode)

Die Instruction Window and Decode Unit hat viele unterschiedliche Aufgaben. Zu Beginn wird das evtl. empfangene Befehlswort in dem entsprechenden Befehlsfenster gespeichert. Dies wird allerdings nur dann durchgeführt, wenn sowohl das Signalisierungs-Bit von der IFU sowie das der BMIU gesetzt sind und nicht ein Sprung des selben Thread angezeigt wird. Andernfalls entfällt dieser Vorgang vollständig. Im Falle eines Sprungs wird stattdessen das entsprechende Befehlsfenster geleert und der interne PC des Thread neu gesetzt. Im nächsten Schritt wird mittels des Prioritäten Managers aus den vorhandenen Threads einer für die weitere Befehlsverarbeitung ausgewählt. Dieser Auswahlvorgang wird in 4.1 näher beschrieben. Anschließend wird geprüft, ob der ausgewählte Thread bereits ein Mikroprogramm ausführt, andernfalls wird, sofern keine Interrupt-Service-Routinen Anforderung ansteht, eine "normale" Befehlsdekodierung durchgeführt. Dazu wird der

aktuelle Befehl mit allen in Hardware oder als Mikrocode realisierten Befehlscodes verglichen. Bei Befehlen, die direkt in Hardware implementiert sind, werden Steuersignale für die restliche Pipeline erzeugt, sowie ein Selektor gelöscht, der der nachfolgenden Operand-Fetch-Stufe anzeigt, dass die aktuellen Steuersignale von der IWDU zu beziehen sind. Ist der aktuelle Befehl als Mikrocode realisiert, wird dieser Selektor gesetzt, um anzuzeigen, dass jetzt Mikrocode ausgeführt werden muss. Außerdem wird der thread-interne Mikro-PC auf den Start des entsprechenden Mikroprogramms gesetzt und der interne Zustand dieses Thread auf Mikrocode-Ausführung gesetzt. Wird der Befehl nicht als hardware- oder Mikrocode-implementierter Befehl erkannt, wird das Mikroprogramm zum Aufruf einer Trap-Routine gestartet. Steht die Anforderung einer Interrupt-Service-Routine an, so wird ebenfalls der Mikrocode zum Aufruf einer Trap-Routine genutzt; allerdings kommt hierbei ein Zeiger auf eine thread-spezifische Interrupt-Service-Routine zum tragen. Durch die Reihenfolge, erst zu prüfen, ob momentan Mikrocode ausgeführt werden muss und anschließend eine evtl. anstehende Interrupt-Service-Routinen-Anforderung zu bearbeiten, wird sichergestellt, dass eine Interrupt Behandlung nur dann eingeleitet wird, wenn gerade kein Mikroprogramm ausgeführt wird.

2.3 Priority Manager

Der Prioritäten Manager entscheidet in jedem Takt, von welchem Thread der nächste Befehl dekodiert werden soll. Ausgewählt wird aus allen Threads, die aktiv sind, nicht auf ein AtomLock oder auf eine Latenz warten müssen und deren Befehlsfenster mindestens 3 Byte beinhaltet. Unter all diesen Threads wird der ausgewählt, dessen Gesamtpriorität am niedersten ist. Wie dieser Auswahlvorgang vonstatten geht, wird später genauer erläutert.

2.4 Operand Fetch Unit (Operand Fetch)

Die Operand-Fetch Unit belegt zwei Takte der Pipeline. Im ersten Takt werden die für die aktuelle Operation benötigten Zeiger, Adressen oder globale Register gelesen bzw. berechnet. Diese werden ggf. noch im selben Takt entsprechend der Operation modifiziert (erhöht bzw. erniedrigt). Im darauffolgenden Takt werden, falls notwendig, ein oder zwei Lesezugriffe auf den allen Threads gemeinsamen Stackspeicher durchgeführt. Ebenso wird in diesem Takt das Ergebnis der vorangegangenen Instruktion in den Speicher geschrieben. Ggf. werden die Daten dieses Schreibvorgangs direkt an die Execute Unit weitergeleitet (Data-Forwarding).

2.5 Execute Unit (Execute)

Die eigentliche Operation, d.h. die Ausführung des Befehls, erfolgt in der Execute Unit. Hier werden die übergebenen Operanden addiert, multipliziert oder verglichen. Ebenso werden von hier aus bedingte und unbedingte Sprünge eingeleitet. Werden Zugriffe auf den Speicher oder die peripheren Bausteine durchgeführt, so führt die Execute Unit eine NOP-Instruktion aus; lediglich der nachfolgende Schreibzugriff in den Stackspeicher wird ggf. angestoßen.

2.6 Speicher-Schnittstelle (Memory Management)

Alle Zugriffe auf Daten oder Code, die den Komodo-Mikrocontroller verlassen, werden von der Speicher-Schnittstelle koordiniert und durchgeführt. Dabei wird überprüft, ob ein gültiger Schreib- oder Lesezugriff der Pipeline (d.h. die Durchführung eines load/store-Befehls) ausgeführt werden sollen. Ist dies nicht der Fall, so kann ein Instruktion-Fetch-Vorgang, angestoßen von der Fetch-Unit, durchgeführt werden. Fetchzugriffe werden ebenfalls wie Lesezugriffe prinzipiell immer mit einer Breite von 32-Bit durchgeführt; bei Schreibzugriffen werden entsprechende Datenverschiebungen und Signalaktivierungen durchgeführt, um auch 8- und 16-Bit Schreibzugriffe zu erlauben. Allerdings sind alle Zugriffe entsprechend ihrer Breite nur aligned möglich.

2.7 Peripherie-Schnittstelle (Chip Select)

Die Peripherie-Schnittstelle ist für Zugriffe der Pipeline auf die peripheren Bausteine zuständig. Dabei erzeugt sie die Chip-Select-Signale und ist für die Trennung des Peripherie-Busses und der Pipeline verantwortlich. Dies ist notwendig, da auf Seiten der Peripherie die seriellen Schnittstellen selbständig Vorgänge auf dem Bus initiieren können, d.h. Busmastertätigkeiten übernehmen können.

2.8 Write-Back-Multiplexer (Write Back)

Aufgabe des Write-Back-Multiplexers ist die Selektion der Schreib-Daten. Dabei werden je nach Befehl die Daten für den Schreibvorgang in den Stackspeicher ausgewählt. In Frage kommen dabei Daten von der Speicher-Schnittstelle, Daten von der Chip-Select Unit oder das Resultat einer Operation in der Execute-Unit. Dieser Multiplexer stellt keine eigene Pipelinestufe dar, da er seine Arbeit asynchron verrichtet.

3 Peripherie

Um die Kommunikation des Komodo Prozessorkerns mit seiner Umwelt zu ermöglichen, wurde er um einige Komponenten erweitert. Hier seien alle Komponenten kurz erwähnt, ihre Programmierung wird später genau erläutert. Abbildung 2 zeigt den Aufbau des Komodo Mikrocontrollers als Blockschaltbild.

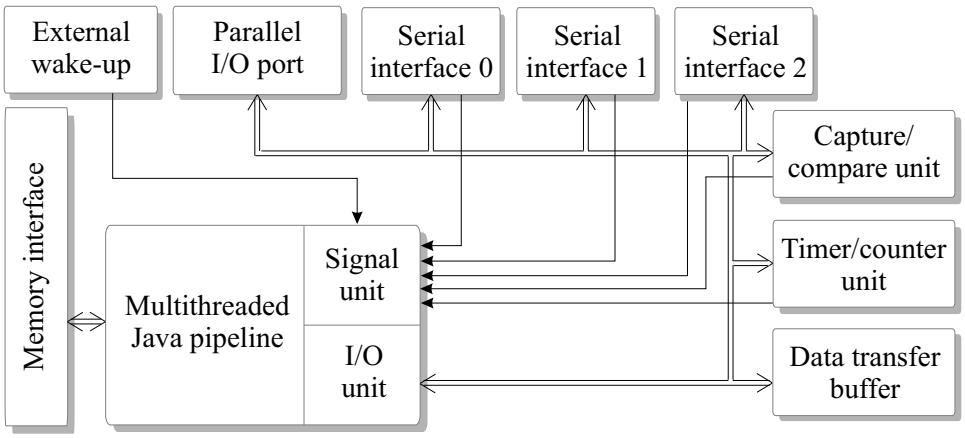


Abbildung 2: Komodo Mikrocontroller

3.1 Zähler/Zeitgeber

Die Zähler/Zeitgeber Einheit verfügt über 4 separate 16-Bit Zähler/Zeitgeber. Diese können auf Grundlage des internen oder eines externen Taktes in programmierbaren Zeitintervallen Signalisierungen an den Prozessorkern senden. Außerdem ist es hiermit möglich, ein externes Ereignissignal zu zählen. Um einen Zähler/Zeitgeber über einen größeren Bereich als die vorhandenen 16-Bit zählen zu lassen, ist es möglich, mehrere Zähler zu kaskadieren.

3.2 Capture/Compare

Mittels der Capture/Compare Einheit lassen sich 4 unterschiedliche Puls-Weiten modulierte Signale erzeugen bzw. ausmessen. Dazu verfügt diese Einheit über 4 separate Kanäle mit jeweils

eigenem Zähler, Reload-Register und Capture/Compare Register. Jeder Kanal kann einmalige oder fortlaufende Messungen durchführen bzw. Signale erzeugen.

3.3 IO-Ports

Um einfache Ausgaben bzw. Eingaben durchführen zu können, ist ein IO-Port mit 8 Eingängen und 8 Ausgängen vorhanden. Dabei wird jeder Pin über eine separate Adresse angesprochen

3.4 Serielle Schnittstelle

Hierbei handelt es sich um eine einfache asynchrone Schnittstelle. Die Paketlänge kann zwischen 2 und 14 Bit eingestellt werden. Die Übertragung erfolgt mit einem Startbit, ein oder zwei Stoppbits und einer optionalen Parität. Um auch eine Übertragung an einem Bussystem erreichen zu können, ist ein Steuersignal für einen externen Leitungstreiber vorhanden. Die Schnittstelle kann im Blockmodus oder im Einzelschrittmodus betrieben werden, d.h. nach jedem empfangenen oder gesendeten Zeichen wird eine Signalisierung an den Prozessorkern gesendet. Im Blockmodus erfolgt der interne Datentransfer in oder aus dem internen Pufferspeicher; ist die Übertragung der gewünschten Anzahl an Zeichen abgeschlossen, so erfolgt eine Signalisierung.

3.5 Interner Pufferspeicher

Der interne Pufferspeicher ist für die in 3.4 genannte Datenblockübertragung zuständig. Er stellt lediglich einen internen Speicherblock dar, auf den die seriellen Schnittstellen zugreifen können. Ebenso kann er vom Prozessorkern beschrieben oder ausgelesen werden.

4 Arbeitsweise der Pipeline

Im Folgenden wird auf alle programmtechnisch relevanten Eigenschaften und Möglichkeiten des Komodo Prozessorkerns eingegangen. Dabei wird jede Funktion der Pipelinestufe zugeordnet, in der diese realisiert ist. Da nicht alle Pipelinestufen eine für den Programmierer sichtbare Funktion erfüllen, sind hier nicht alle Pipelinestufen aufgeführt. Dennoch werden auch wichtige Details der Realisierung beschrieben, auf die der Programmierer nicht direkt Einfluss nehmen kann, die aber zum besseren Verständnis des Gesamtsystems dienen.

4.1 Die Instruction-Window and Decode Unit

In der IWDU werden die Instruction-Windows (IW) verwaltet, die Scheduling-Entscheidung gefällt und anschließend der entsprechende Befehl dekodiert. Die IWs können pro Thread jeweils 8 Byte Instruktionen aufnehmen. Dabei wird der Füll-Vorgang in die Windows vorgenommen, bevor diese für das Scheduling und die Dekodierung herangezogen werden.

Für die verschiedenen Aufgaben sind diverse Konfigurationsregister nötig, die zur Steuerung des Schedulers und des Dekodierers dienen. Dies sind zum einen ein globales Konfigurationsregister, mit dessen Hilfe die fest programmierten Mikroprogramme deaktiviert und durch entsprechende Trap-Routinen ersetzt werden können. Dieses Register ist unter der Adresse 0x7F im I/O-Adressraum mittels der Befehle *io_write* und *io_read* anzusprechen. Jedes Bit in diesem Register entspricht einem Mikroprogramm. Ist ein Bit gesetzt, so wird der entsprechende Befehl mittels seiner Trap-Routine bearbeitet. Tabelle 1 zeigt die Belegung des Registers.

Zusätzlich zu dem globalen Konfigurationsregister sind je Thread noch 4 weitere Register vorhanden. Diese werden im nächsten Abschnitt genauer beschrieben. Dort ist je Thread auch ein Aktivierungs-Bit für die Garbage-Collection (GC) vorhanden. Ist dieses gesetzt, so werden für diesen Thread die GC relevanten Mikroprogramme durch ihre Trap-Routinen ersetzt. In diesem Fall ist der Zustand der entsprechenden Konfigurationsbits irrelevant.

Zusätzlich sind je Thread noch 3 Aktivitätsbits vorhanden. Diese sind

Bit	Bytecode	Befehl
0	0x2E	iaload
1	0x32	aaload
2	0x33	baload
3	0x34	caload
4	0x35	saload
5	0x4F	iastore
6	0x53	aastore
7	0x54	bastore
8	0x55	castore
9	0x56	sastore
10	0x5A	dupx1
11	0x5B	dupx2
12	0x5C	dup2
13	0x5D	dup2x1
14	0x5E	dup2x2
15	0x5F	swap
16	0x6C	idiv
17	0x70	irem
18	0xAC	ireturn
19	0xB0	areturn
20	0xb1	return
21	0xBE	arraylength
22	0xCB	ldc_quick

Tabelle 1: Belegung des globalen Konfigurationsregisters

- das Software-Aktiv (SW-aktiv)
- das Hardware-Aktiv (HW-aktiv)
- das Wait-for-Atomic-Lock (WAL)
- und das Interrupt-Service-Aktiv (ISR-aktiv) Bit.

Hinzu kommt noch ein globales Atomic-Lock-Bit (AL). Mit Ausnahme des ISR-aktiv Bits können all diese Zustandbits über spezielle Befehle beeinflusst werden. Ein Thread gilt als aktiv und nicht wartend, wenn HW-aktiv und SW-aktiv gesetzt sind sowie WAL gelöscht ist. Als wartend gilt ein aktiver Thread, wenn HW-aktiv, SW-aktiv und WAL gesetzt sind. Zusätzlich kann ein Thread auch eine herkömmliche Interrupt-Service-Routine ausführen. Wird ein ISR-Request festgestellt, so wird dieser behandelt, sobald der entsprechende Thread keinen Mikrocode ausführt und das AL-Bit gelöscht ist. Andernfalls wird die Ausführung entsprechend verzögert. Innerhalb einer ISR sollten die Befehle zur Manipulation der HW-aktiv und SW-aktiv Bits nicht benutzt werden! Das ISR-aktiv Bit wird mittels des Befehls ISReturn wieder auf 0 gesetzt.

Der Prioritäten-Manager

In der aktuellen Version des Prioritäten-Managers (PM) sind vier unterschiedliche Scheduling-Algorithmen implementiert, die auch in unterschiedlichen Kombinationen eingesetzt werden können. Genauer sind folgende Algorithmen implementiert:

Fixed Priority Preemptive (FPP) Es stehen genau so viele Prioritätsebenen zur Verfügung, wie Hardware-Threads unterstützt werden.

Earliest Deadline First (EDF) Jedem Hardware-Thread muss eine 32-Bit Deadline zugeordnet werden. Dabei entspricht eine Einheit genau einem Takt-Zyklus.

Adresse	Bitbreite	Register
0x00 + Thread-Nr	8	Mode-Register
0x10 + Thread-Nr	7	GP-Register
0x20 + Thread-Nr	32	EDF/LLF-Register
0x30 + Thread-Nr	ld(Threads)	FPP-Register

Tabelle 2: Adressen der Register des Prioritäten-Managers

Least Laxity First (LLF) Jedem Hardware-Thread wird eine 32-Bit Laxity (Differenz zwischen der Deadline und der Laufzeit des Threads) zugeordnet.

Guaranteed Percentage (GP) Hier bekommt jeder Thread einen Prozentanteil zugewiesen. Je nach Konfiguration bekommt dieser Thread dann genau, minimal oder maximal diesen Anteil Rechenzeit.

Gemischt werden können alle diese Algorithmen außer EDF und LLF, da für diese dieselbe Logik verwendet wird. Außerdem war es schwierig, eine Reihenfolge dieser Algorithmen festzulegen, so dass im Voraus theoretische Betrachtungen eines angestrebten Systems gemacht werden können.

Der/die gewünschten Algorithmen können über folgende Register konfiguriert werden:

Mode-Register Hier können der gewünschte Scheduling-Algorithmus ausgewählt, sowie evtl. Energiesparmodi aktiviert werden.

FPP-Register Hier muss die Priorität für den FPP-Algorithmus festgelegt werden.

EDF/LLF-Register Hier wird je nach gewähltem Algorithmus die Deadline bzw. die Laxity des jeweiligen Thread eingetragen. Dieser Wert wird bei einer Thread-Aktivierung durch die Signaleinheit oder den *hw_resume*-Befehl in ein Schattenregister übertragen, wo er dann entsprechend dem Scheduling-Algorithmus behandelt wird.

GP-Register Die für die gewählte GP-Art (genau, minimal, maximal) relevante Prozentzahl wird hier angegeben.

Dieser Registersatz ist je Thread einmal vorhanden. Die entsprechenden Adressen sind in Tabelle 2 aufgeführt und über die Befehle *io_read* und *io_write* anzusprechen. Die **Mode-Register** haben den in Tabelle 3 angegebenen Aufbau.

Bit	Funktion
0	FPP aktiv
1	GP aktiv
2	EDF aktiv
3	LLF aktiv
4	GC aktiv
5	lokale Low-Power Aktivierung
6	GP-Maximum Aktivierung
7	globale Low-Power Aktivierung

Tabelle 3: Belegung des Konfigurationsregisters

Dabei dürfen die Bits 2 und 3 nie gemeinsam aktiviert sein und ebenso darf innerhalb eines Komodo-Mikrocontrollers nur einer der beiden Algorithmen verwendet werden. Ein Test hierauf wird seitens der Hardware nicht durchgeführt; das Ergebnis einer entsprechenden Aktivierung ist undefiniert.

Um eine GP-Minimum Konfiguration zu erreichen, muss das GP-Bit aktiviert sein, eine Prozentzahl angegeben, eine zusätzliche Scheduling-Methode gewählt und deren entsprechender Parameter angegeben werden. Die Rechenzeitzuteilung erfolgt dann wie im nächsten Abschnitt beschrieben.

Eine GP-Maximum Konfiguration wird in ähnlicher Weise erreicht, mit dem Unterschied, dass zusätzlich das GP-Maximum-Bit gesetzt werden muss.

Das GC-Bit hat die im vorherigen Abschnitt beschriebene Wirkung.

Die beiden Low-Power Aktivierungs-Bits haben folgende Funktion: Die lokale Low-Power Aktivierung gibt Auskunft darüber, ob in der Zeit, in der der entsprechende Thread aktiv und nicht wartend ist, der Energie-Sparmodus in Frage kommt. Das globale Low-Power Bit erlaubt die generelle Nutzung des Low-Power Modus, d.h. ist dieses Bit gelöscht (Reset-Zustand) und eines der unteren 4 Bit des Modus-Registers gesetzt, so wird die Aktivierung des Energie-Sparmodus generell verboten.

In den Energie-Sparmodus gewechselt wird dementsprechend genau dann, wenn

1. das globale LP-Aktivierungs-Bit aller Threads mit gewähltem Schedulingverfahren gesetzt ist,
2. das lokale LP-Aktivierungs-Bit aller aktiven, nicht wartenden Threads gesetzt ist
3. und die Summe der Prozente aller aktiven, nicht wartenden Threads mit GP-Scheduling nicht größer als 50/25 ist.

Ist diese Summe kleiner oder gleich 50, so wird die interne Pipeline-Frequenz und die Frequenz für das externe SRAM durch 2 geteilt. Ist die Summe nicht größer als 25, so wird durch 4 geteilt. Diese Frequenzteilung wird bei der Verkürzung der Deadline bei EDF mitberücksichtigt, d.h. je Pipelinetakt kommt die Deadline ggf. 2 bzw. 4 Einheiten näher. Die Verkürzung der Laxity wird ebenfalls entsprechend der Frequenzteilung vorgenommen; allerdings kommen hier noch andere Faktoren hinzu (s.u.). Der Zustand der Energie-Sparstufe (aus, halber Takt, viertel Takt) kann nur zur Grenze eines 100-Takte Intervalls wechseln; der Grund dafür wird im nächsten Abschnitt erläutert.

Arbeitsweise des Prioritäten-Managers

Prinzipiell wird die gesamte Laufzeit in Intervalle zu je 100 Takten eingeteilt. Zu Beginn eines solchen Intervalls wird geprüft, ob ein Low-Power Modus für dieses Intervall aktiviert werden kann. Wenn ja, werden entsprechende interne Signale gesetzt und die Länge des Intervalls auf 50 bzw. 25 Takte reduziert. Dadurch wird sichergestellt, dass dieses Intervall zu einem festen Zeitpunkt endet, unabhängig vom gewählten Energie-Sparmodus.

Innerhalb eines Intervalls existiert eine variable Grenze, die unterschiedliche Prioritätsschemata voneinander trennt. Zeitlich vor dieser Grenze gilt folgende Reihenfolge der verschiedenen Scheduling Algorithmen:

1. EDF/LLF
2. FPP
3. GP

Nach dieser Grenze wird GP nach vorne gezogen:

1. GP
2. EDF/LLF
3. FPP

Sinn und Zweck dieser Vertauschung ist es, die laufenden GP-Threads ans Ende des Intervalls zu verschieben, um diese (in diesem Intervall) möglichst lange aktiv zu halten. Im vorderen Bereich des Intervalls können sie als Latenz-Lücken-Füller anderer Threads dienen, im hinteren Bereich ist dieses umgekehrt. Dass EDF/LLF in der Priorisierung vor FPP stehen, dürfte intuitiv klar sein, da diese im Gegensatz zu FPP mit einer fixen Deadline versehen sind.

Die Berechnung der Grenze geschieht durch aufsummieren aller aktiven, nicht wartenden GP-Threads. Dabei werden nur die GP-Minimum und genaue GP-Threads berücksichtigt. Da diese Berechnung in jedem Takt durchgeführt wird, ist eine Verschiebung nach hinten möglich. Augenscheinlich ist hier auch, dass theoretisch durch eine Neuaktivierung eines zusätzlichen GP-Threads eine Verschiebung nach vorne möglich wäre; dies wird aber durch den PM verboten, da dadurch nicht mehr garantiert werden kann, dass alle aktiven Threads ihre gewünschte Rechenzeit bekommen.

Um eine Scheduling-Entscheidung treffen zu können, bekommt jeder Thread einen Prioritätsvektor zugeordnet. Der Aufbau dieses Vektors unterscheidet sich je nach dem, ob sich der aktuelle Takt vor oder nach der Grenze befindet. Abbildung 3 verdeutlicht den Aufbau des Prioritätsvektors.

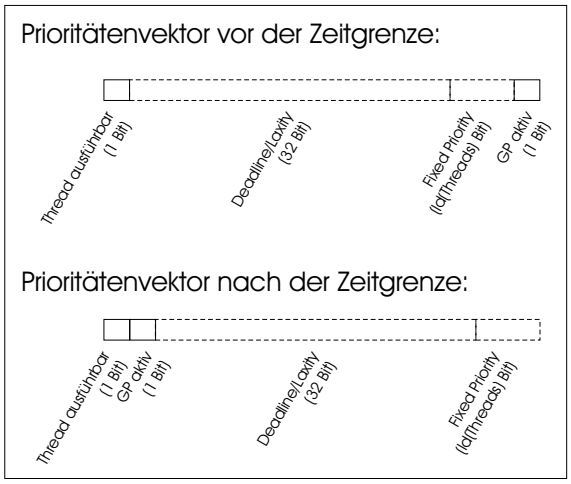


Abbildung 3: Aufbau des Prioritätsvektors vor und nach der Zeitgrenze

Das oberste Bit gibt dabei Auskunft darüber, ob der Thread überhaupt ausgeführt werden kann (SW-aktiv und HW-aktiv gesetzt oder ISR-aktiv gesetzt, keine Latenzen, Instruktion-Window enthält min. 3 Byte und min. ein Scheduling-Algorithmus ist ausgewählt). Ist ein Scheduling-Algorithmus gewählt, so wird der entsprechende Parameter in den dazugehörigen Bereich des Vektors geladen; andernfalls werden hier nur '1'en eingesetzt. Ausgeführt wird derjenige Thread, dessen Vektor den kleinsten Wert darstellt.

Hier wird nun auch deutlich, was mit der Aktivierung mehrer Algorithmen für einen Thread erreicht werden kann: So kann z.B. ein Thread, mit einer festen Deadline, der aber unbedingt ein geringes, aber gleichförmiges Fortschreiten verlangt, mit folgender Regel ausgeführt werden: Arbeite mit der Deadline 50.000 Takte, aber mit min. 10% Rechenleistung. Dies wäre eine GP-Minimum mit zusätzlichem EDF-Scheduling. Anders herum kann ein Thread mit fester Deadline aber verhältnismäßig geringer Laufzeit diese Regel nutzen: Arbeite mit Deadline 100.000 Takte, aber mit maximal 3% Rechenleistung. Dies stellt ein GP-Maximum mit zugrundeliegendem EDF-Scheduling dar.

In Abbildung 3 ist zu erkennen, dass lediglich ein Bit für das GP-Scheduling im Vektor verantwortlich ist. Dies resultiert daraus, dass für die verschiedenen GP-Threads keine Priorisierung untereinander notwendig ist, da in jedem Intervall sowieso genügend Zeit für die Ausführung aller GP-Threads reserviert wird. Somit ist die Reihenfolge der Ausführung nicht relevant; wichtig ist nur, dass alle GP-Threads ihre gewünschte Rechenleistung bekommen. Bei Auswahl des GP-Maximum-Schedulings wird dies innerhalb des Vektors nicht berücksichtig, d.h. das GP-Bit bleibt auf '1' gesetzt. Rechenzeit muss durch Auswahl eines weiteren Algorithmus erlangt werden. Bei Ausführung des Threads wird dann lediglich die zugeordnete Prozentzahl reduziert; wird dabei die 0-Prozentmarke erreicht, wird der Thread für das aktuelle Intervall suspendiert.

Arbeitet der Mikrocontroller mit EDF, so werden je nach aktueller Leistungsstufe die Werte aller Deadlines um 1,2 oder 4 reduziert. Im LLF-Betrieb werden die Laxities aller nicht ausgeführten Threads um 1,2 oder 4 reduziert. Die Laxity des ausgeführten Threads wird um 0,1 oder 3 reduziert. Je nach Energie-Sparstufe kommt dieser Thread seiner Deadline um 1 oder 3 echten Takten (nicht Pipeline-Takten) näher.

Sowohl für EDF als auch für LLF gilt: ist der Wert im Deadline/Laxity Register bei 0 angekommen, so wird ein Deadline-Miss hervorgerufen. Dies wird pro Thread auf einer separaten Signalleitung angezeigt, die dann mittels der Signaleinheit abgefragt werden kann. So kann auf einen Deadline-Miss z.B. mit der Aktivierung eines anderen Thread oder dem Aufruf einer ISR reagiert werden.

Die Befehlsdekodierung

Die ausführbaren Befehle sind in 3 Gruppen aufgeteilt:

1. Hardware-Befehle
2. Mikroprogramme
3. Trap-Routinen

Trap-Routinen werden mittels eines Mikroprogramms aufgerufen und beendet. Sowohl die Dekodierung von Hardware-Befehlen als auch die Ausführung eines Mikrocodes resultiert in der Bestimmung zweier Signalgruppen, die die Operandenquelle und Operation angeben. Das Ziel des Ergebnisses ist in der Operation impliziert. Sowohl Quelle als auch Ziel können ein Kontroll-/Steuerregister oder ein bzw. zwei (nur Quelloperanden) Stackspeicherwerte sein. Zusätzlich kann noch ein befehlsabhängiger Operationswert übergeben werden. Die Operation kann eine der in Anhang D aufgeführten Instruktionen sein.

Einige Befehle benötigen einige Takte, bis sich ihre vollständige Wirkung im gesamten Mikrocontroller niedergeschlagen hat. Dies sind vorwiegend alle Sprungbefehle sowie einige Kontrollbefehle. Die genauen Latenzzeiten sind aus der Tabelle im Anhang ersichtlich. Darin sind auch die während der Ausführung von Mikroprogrammen auftretenden Latenzen aufgeführt.

Nachdem ein Befehl als mikroprogrammierter Befehl erkannt wurde und er nicht durch das globale Konfigurationsregister oder das GC-aktiv Bit deaktiviert wurde, wird ein separater PC, der sog. μPC, mit der Adresse des korrespondierenden Mikroprogramms geladen. Anschließend wird unverzüglich auf Mikroprogramm-Ausführung umgeschaltet. Dies gilt allerdings nur für den aktuell in der Dekodierung befindlichen Thread, alle anderen bleiben davon unberührt. Da jeder Thread-Slot über einen eigenen μPC verfügt, können auch während der Laufzeit von Mikroprogrammen Threadwechsel stattfinden. Dies ist deshalb besonders wichtig, da innerhalb von Mikroprogrammen ebenfalls Latenzen auftreten, die von anderen Threads genutzt werden können. Hier sind als Beispiele die Befehle *irem* und *idiv* zu nennen, die jeweils um die 100 Latenztakte aufweisen.

Wird während der Befehlsdekodierung ein Bytecode erkannt, der weder direkt durch die Hardware oder mittels eines Mikroprogramms ausgeführt werden kann (bzw. wenn der entsprechende Mikrocode deaktiviert ist), so wird ein Trap-Routinen Aufruf gestartet. Dazu wird eine von drei im Speicher präsenten Trap-Tabellen benutzt. Die erste Tabelle enthält 256 Einträge, jeweils einen pro Bytecode. Die zweite Tabelle enthält ebenfalls 256 Einträge, die je einem Erweiterten Bytecode entsprechen. Die letzte Tabelle enthält genau so viele Einträge, wie Threads vom Mikrocontroller unterstützt werden. Dabei ist je ein Eintrag für die Interrupt-Service-Routine eines Threads verantwortlich. Der Aufbau ist bei allen drei Tabellen gleich: jeder Eintrag ist 8 Byte lang, die ersten 4 Byte geben die Einsprungadresse und die anderen 4 Byte den Constant-Pool, falls erforderlich, an. Folgenden Tabelle fasst die Startadressen und Größen der Tabellen zusammen. Dabei fällt auf, dass die erste Tabelle an der Adresse 0 beginnt, was ebenfalls die Adresse des ersten Befehls nach einem Reset darstellt. Dies ist allerdings unproblematisch, da der erste Eintrag dem NOP-Befehl entspricht, der niemals durch eine Trap-Routine repräsentiert werden wird. Somit ist dieser Eintrag immer leer und kann für den initialen Sprungbefehl, der 5 Byte benötigt, benutzt werden.

Startadresse	Länge [Byte]	Funktion
0x0000	2048	Traptabelle für normale Bytecodes
0x0800	2048	Traptabelle für Erweiterte Bytecodes
0x1000	#Threads*8	ISR Einsprungtabelle

Tabelle 4: Basisadressen der Trap-Tabellen

Beim Aufruf von Trap-Routinen werden komplexe Befehle der JVM (z.B. *new*, *getfield*), ursprünglich nicht implementierte Befehle (z.B. *ladd*, *fmul*) und Ausnahmen erzeugende Befehle (z.B. *athrow*, *checkcast*) gleich behandelt.

4.2 Die Operand-Fetch-Unit

Die Operand-Fetch-Einheit ist für das Holen der benötigten Operanden und die Stackverwaltung zuständig. Operanden können aus dem Stackspeicher stammen oder eines der zur Stackverwaltung notwendigen Register sein. Ein Operand-Fetch Vorgang ist wiederum gepipelined in zwei Unterstufen geteilt:

1. Die Berechnung der Adresse im Stackspeicher und Aktualisierung der Verwaltungsregister
2. Der eigentliche Stackzugriff im Stackspeicher

Wird ein Operand aus dem Verwaltungsregistersatz benötigt, so ist der zweite Schritt nicht relevant. Um Datenabhängigkeiten vorzubeugen, ist im zweiten Schritt ein zweistufiges Data-Forwarding realisiert; dadurch muss nicht erst die Vollendung des Schreibzugriffs abgewartet werden, bevor ein erneuter Lesezugriff auf dieselben Daten erfolgen kann. Dieses Forwarding betrifft allerdings nur Schreibzugriffe in den Stackspeicher, Schreiboperationen auf die Stackverwaltungsregister müssen abgewartet werden und erzeugen deshalb Latenztakte.

Der Stackspeicher ist als ein gemeinsamer Speicher für alle Threads realisiert. Um diesen aufzuteilen, verfügen alle Threads über einen Stack-Beginn-Zeiger (TSB) und eine Stack-Size-Maske (TSS). Bei jeder Adressberechnung wird die gewünschte Adresse mit TSS Und-verknüpft; anschließend wird TSB hinzuaddiert. Die daraus resultierende Adresse gibt die im Stackspeicher gewünschte Stelle an. Da der Stackspeicher intern aus zwei separaten Speicherbänken aufgebaut ist, von denen eine alle geraden, die andere alle ungeraden Speicherstellen beinhaltet, sind sowohl TSS als auch TSB um ein Bit kleiner als die gesamte Stackadresse. TSS wird unten mit '1', TSB mit '0' erweitert, wodurch eine minimale Stackgröße von zwei Worten vorgegeben wird; außerdem muss die Stackgröße immer ein Vielfaches von zwei sein. Für die Standard KJM sind drei Stackspeicherzeiger und ein 32-Bit Register notwendig; für die Realisierung der Mikrocodes zwei weitere 32-Bit Register. Für die Unterstützung der Garbage-Collection und eines Dribblers sind je ein weiterer Stackspeicherzeiger mit Autoincrement bzw. -decrement Funktion und ein zusätzliches Thread-Tag für den Zugriff auf andere Threads vorhanden. In der folgenden Tabelle sind alle für jeden Thread je einmal vorhandenen Verwaltungsregister aufgeführt:

Um für die GarbageCollection und die Thread-Initialisierung Zugriffe auf die Stacks anderer Threads zu ermöglichen, wurde das CrossTag eingeführt. Dieses gibt eine Thread-ID an, die für Zugriffe auf andere Stackspeicher mittels des CrossPointers und der Befehle zum Lesen bzw. Schreiben der Verwaltungsregister als Basis dient. Tabelle 6 enthält alle Erweiterten Bytecodes, die das CrossTag als Grundlage haben.

Einige der Register haben Autodecrement bzw. Autoincrement Funktion. Der OPTOP muss aus Gründen der Stackarchitektur über diese Funktionen verfügen. Bei einem Schreibzugriff wird sein Wert um 1 erhöht, bei einem Lesezugriffe um 1 erniedrigt, d.h. der Stack wächst nach oben. Der OPTOP zeigt zu jeder Zeit auf den ersten freien Eintrag. Der CrossPointer ist nur mit einer Autoincrement Funktion ausgestattet; diese wird vor einem Lese- und nach einem Schreibzugriff aktiv. Diese Vorgehensweise wurde gewählt, da der CrossPointer zur Initialisierung eines neuen Threads dient, dessen Stack von der Adresse 0 an beschrieben werden muss. Weiterhin dient er

Register	Breite	Funktion	Bemerkung
OPTOP	ld(Stacksize)	KJM	Auto inc/dec
Vars	ld(Stacksize)	KJM	
Frame	ld(Stacksize)	KJM	
ConstPool	32-Bit	KJM	
Global1	32-Bit	Mikro/Trap	
Global2	32-Bit	Mikro/Trap	
CrossPointer	ld(Stacksize)	GarbageCollection	
CrossTag	ld(#Threads)	GarbageCollection	Auto inc
DribblerPointer	ld(Stacksize)	Dribbler	Auto inc/dec
TSS/TSB	je ld(HW-Stacksize)-1	Speicheraufteilung	

Tabelle 5: Threadspezifische Register

Schreibbefehle	Lesebefehle	Beschreibung
write_stack	read_stack	Zugriff auf einen fremden Stack
write_pc	-	Zugriff auf den PC (Lesen nur des eigenen PC möglich)
write_vars	read_vars	Zugriffe auf das Vars-Register
write_frame	read_frame	Zugriffe auf das Frame-Register
write_optop	read_optop	Zugriffe auf das OPTOP-Register
write_tsr	read_tsr	Zugriffe auf die TSS/TSB-Register
write_cp	read_cp	Zugriffe auf das ConstPool-Register
write_dp	read_dp	Zugriffe auf das DribblerPointer-Register
-	read_cross	Lesezugriff auf das CrossPointer-Register
write_global0	read_global0	Zugriffe auf das Global0-Register
write_global1	read_global1	Zugriffe auf das Global1-Register
-	read_sct	Lesezugriff auf das CrossTag

Tabelle 6: Befehle auf Basis des CrossTags

als Lesezeiger für die GarbageCollection, die alle Stacks von 0 bis zum aktuellen OPTOP (aufsteigend) durchsuchen muss. Der DribblerPointer wird vor einem Lesezugriff erhöht und nach einem Schreibzugriff erniedrigt. Der Dribbler ist für das Auslagern älterer Stackeinträge in den Speicher bzw. das Zurücklesen dieser verantwortlich; dabei wird er durch die Autoincrement/decrement Funktionen unterstützt.

4.3 Die Signal-Unit

Die Signal-Unit stellt den Interrupt-Controller für den mehrfädigen Prozessorkern da. Dabei ist seine Aufgabe die Zuordnung der Service-Request Signale der peripheren Module zu den entsprechenden Thread Slots. Da jeder Thread sowohl über ein Wake-Up Signal als auch ein herkömmliches Interrupt-Service-Request Signal verfügt, ist eine Zuordnung jedes Request Signals zu jedem Wake-Up und jedem ISR Signal möglich. Das Wake-Up Signal ist für die Aktivierung des HW-aktiv Bits verantwortlich, das ISR Signal für das Setzen des ISR-aktiv Bits. Zu diesem Zweck stehen pro Thread jeweils zwei Konfigurations und zwei Status Register zur Verfügung. Ein Paar (Konfigurations/Status-Register) ist dabei für die Wake-Up Signale, das andere für die ISR Signale verantwortlich. Wird ein Thread aktiviert, so kann aus dem jeweilige Status Register der Grund dafür ausgelesen werden. Dabei entspricht jedes Bit, wie im Konfigurationsregister, genau einer Signalquelle. Die Belegung ist im Anhang dargestellt. Die Adressen der Register sind aus Tabelle 7 zu entnehmen. Sie sind über die Befehle *io_write* bzw. *io_read* zu erreichen.

Über die Konfigurationsregister kann ein Thread in jeder Funktion (Wake-Up und ISR) auf ein oder mehrere Ereignisse sensibilisiert werden. Dazu müssen die zu den gewünschten Ereignissen gehörenden Bits im entsprechenden Konfigurationsregister gesetzt werden. Es werden nur Ereignisse berücksichtigt, deren Konfigurationsbit zur Zeit des Auftretens gesetzt ist. Ist das Bit gelöscht,

Adresse	Register
0x80 + Thread-Nr	Wake-Up Konfigurationsregister jedes Thread
0xA0 + Thread-Nr	ISR Konfigurationsregister jedes Thread
0xC0 + Thread-Nr	Wake-Up Statusregister jedes Thread
0xE0 + Thread-Nr	ISR Statusregister jedes Thread

Tabelle 7: Register der Signal-Unit

so geht dieses Ereignis für diesen Thread verloren. Tritt ein Ereignis auf, für das ein Bit in einem Konfigurationsregister gesetzt ist, so wird das korrespondierende Bit im entsprechenden Status Register gesetzt. Ist ein Statusregister nicht komplett 0, so wird die dazugehörige Wake-Up bzw. ISR Leitung zum Prioritäten Manager aktiviert. Dieses bleibt so lange aktiv, bis das Register von der Software ausgelesen wurde, wobei es automatisch gelöscht wird. Tritt nun ein weiteres Ereignis auf, dessen Bit im Konfigurationsregister gesetzt ist, so wird diese Aktivierung bis zum nächsten Auslesen des Status Registers gespeichert. Dadurch wird eine zwischenzeitliche Deaktivierung des Threads wirkungslos, da unmittelbar danach wiederum eine Thread Aktivierung ausgeführt wird.

4.4 Die Mikrocode-Unit

In der Mikrocode-Unit sind alle Mikroprogramme fest verdrahtet. Insgesamt sind dort 22 Mikroprogramme vorhanden, von denen einige mehrfach verwendet werden. Ein Programm wird für den Aufruf von Trap-Routinen und ISR-Routinen verwendet, zwei weitere sind für die Rückkehr aus Trap- und ISR-Routinen notwendig. Laufzeiten und Latenzen sind aus der im Anhang zu findenden Tabelle zu entnehmen. Realisiert ist der Mikrocode nicht in Form von Java Bytecode Befehlen, sondern in der direkten Generierung der für die restliche Pipeline nötigen Steuersignale. Diese dienen während der Ausführung eines Mikroprogramms als Eingangssignale für die in 4.2 beschriebene Operand-Fetch-Unit. Dadurch sind Funktionen möglich, die mittels Bytecode nicht oder nur sehr aufwändig realisiert werden können. Erfüllen einzelne Mikroprogramme nicht mehr die gewünschte Aufgabe (aufgrund einer Änderung der Anforderung), so können sie auf die in Abschnitt 4.1 beschriebene Art und Weise deaktiviert werden. Dies gilt allerdings nicht für die Programme zum Aufruf und Verlassen von Trap/ISR Routinen.

5 Arbeitsweise und Programmierung der Peripherie

In diesem Abschnitt wird die Programmierung der einzelnen peripheren Komponenten erläutert. Jeder Einheit sind im I/O-Bereich des Mikrocontrollers 16 Adressen zugeordent. Innerhalb dieses Adressbereichs ist an der letzten Adresse (0x0f) ein ID-Code abgelegt, der über einen Lesezugriff abgefragt werden kann. Mittels dieses Codes ist die Art der Schnittstelle in diesem Adressbereich eindeutig zuordenbar. Im Anhang ist zusätzlich eine Tabelle mit den im FPGA-Prototyp verwendeten Adressen aufgeführt.

5.1 Zähler/Zeitgeber

Die Zähler/Zeitgeber Einheit verfügt über 4 separate 16-Bit Zähler, die jeweils mit einem Reload-Register versehen sind. Als Takteingang kann einer von drei Eingängen dienen. Ein Eingang ist dabei auf Pin C23 am Extension-Header X2 gelegt. Jeder Kanal besitzt ein Zähl-Register, ein Reload-Register und ein Konfigurations-Register. Das Zähl- und Reload-Register sind unter einer gemeinsamen Adresse anzusprechen; gelesen wird das Zähl-Register, bei einem Schreibzugriff wird das Reload-Register modifiziert. Abbildung 4 zeigt die Belegung des Konfigurations-Registers. Das Signal für den hier angesprochene Pin P ist am Extension-Header X2 an Pin C22 anzulegen. Tabelle 8 gibt die Belegung des Zähler/Zeitgeber Adressbereichs wieder.

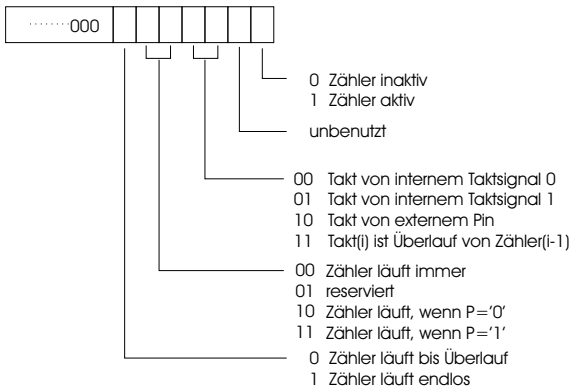


Abbildung 4: Belegung des Zähler/Zeitgeber Konfigurationsregisters

Adresse	Register
0	Kontrollregister A
1	Reloadregister/Zählerstand A
2	Kontrollregister B
3	Reloadregister/Zählerstand B
4	Kontrollregister C
5	Reloadregister/Zählerstand C
6	Kontrollregister D
7	Reloadregister/Zählerstand D
15	Identifikationskennung Zähler/Zeitgeber

Tabelle 8: Adressbelegung des Zähler/Zeitgebers

WICHTIG: Die beiden internen Taktsignale sind im FPGA-Prototyp an den internen Pipelinetakt gekoppelt, d.h. sie sind abhängig vom aktuellen Energiesparzustand des Mikrocontrollers. Sobald die Energiesparoption aktiviert ist, kann keine zuverlässige Zeitmessung stattfinden.

5.2 Capture/Compare

Dieses Modul verfügt über vier gleichwertige Capture/Compare-Kanäle. Jedem Kanal ist ein eigenständiger 16-Bit Zähler/Zeitgeber zugeordnet, der die nötigen Referenzwerte liefert. Dabei stehen zwei interne und ein externes Taktsignal zur Verfügung. Zur Konfiguration besitzt jeder Kanal ein eigenes Kontroll- und ein Datenregister, das die zur Funktion benötigten Daten enthält, sowie ein Reload-Register für den jeweiligen Zähler.

Compare-Modus Im Compare-Modus wird der Stand des Referenzzählers (ZRef) mit dem Inhalt des Datenregisters (WComp) verglichen. Je nach Ergebnis dieses Vergleichs wird der Ausgang (A) des Capture/Compare-Kanals auf den im Pegel-Bit (BPegel) des Kontrollregisters angegebenen Wert gesetzt. So gilt:

$$\begin{aligned} A = \text{not}(\ BPegel\ ) \quad & \text{für ZRef} < \text{WComp} \\ A = BPegel \quad & \text{für ZRef} >= \text{WComp} \end{aligned}$$

Außerdem kann festgelegt werden, ob dieser Vorgang einmalig ausgeführt oder periodisch wiederholt werden soll. Zum Zeitpunkt, wenn ZRef = WComp gilt, wird eine Signalisierung an den Mikrocontrollerkern durchgeführt. In dieser Periode wird keine Signaländerung am Ausgang mehr vorgenommen, so dass durch die Software ein neuer Compare-Wert für die nächste Periode festgelegt werden kann.

Adresse	Funktion
0	Kontrollregister Kanal A
1	Datenregister Kanal A
2	Reloadregister Zähler A
3	Kontrollregister Kanal B
4	Datenregister Kanal B
5	Reloadregister Zähler B
6	Kontrollregister Kanal C
7	Datenregister Kanal C
8	Reloadregister Zähler C
9	Kontrollregister Kanal D
10	Datenregister Kanal D
11	Reloadregister Zähler D
15	Identifikationskennung Cap/Comp

Tabelle 9: Adressraumbelegung der Capture/Compare-Einheit

Capture-Modus Ist die Capture-Funktion eines Capture/Compare-Kanals gewählt, so wird der Stand des Referenz-Zählers während des Pegelübergangs von not(BPegel) nach BPegel in das Capture-Register übernommen. BPegel ist dabei das im Kontrollregister angegebene Pegel-Bit. Zum Zeitpunkt des Übergangs wird außerdem ein Signal an die Signaleinheit des Mikrocontrollerkerns gesendet. Auch hierbei ist es möglich, die Capture-Funktion für nur eine Periode oder fortlaufend zu aktivieren.

Für jeden Capture/Compare-Kanal ist ein eigenständiges Kontrollregister und ein Datenregister vorhanden. Im Compare-Modus kann das Datenregister gelesen und beschrieben werden, während im Capture-Modus Schreibzugriffe keine Wirkung haben. Das Kontrollregister erlaubt in jedem Fall beide Zugriffsarten. Abbildung 18 zeigt den Aufbau der Kontrollregister der Capture/Compare-Kanäle und Tabelle 9 gibt die Adressen der Register im Speicherbereich der Capture/Compare-Einheit an. Die Pinbelegung ist im Anhang nachzulesen.

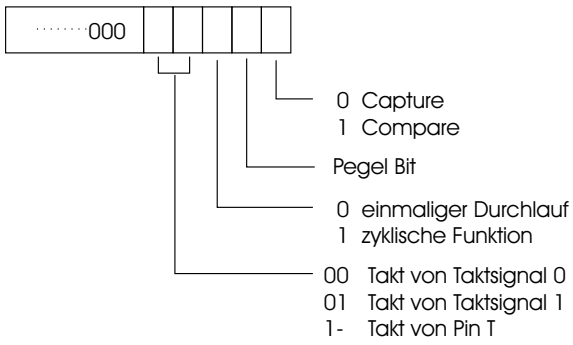


Abbildung 5: Kontrollregister eines Capture/Compare-Kanals

WICHTIG: Die beiden Taktsignale 0 und 1 sind im FPGA-Prototyp an den externen Takt des Mikrocontrollers gekoppelt; der "externe Takt" hingegen ist auf GND-Potenzial gelegt, d.h. er ist hier nicht zu gebrauchen.

Adresse	Funktion
0	Lesen von Pin I0, schreiben auf Pin O0
1	Lesen von Pin I1, schreiben auf Pin O1
2	Lesen von Pin I2, schreiben auf Pin O2
3	Lesen von Pin I3, schreiben auf Pin O3
4	Lesen von Pin I4, schreiben auf Pin O4
5	Lesen von Pin I5, schreiben auf Pin O5
6	Lesen von Pin I6, schreiben auf Pin O6
7	Lesen von Pin I7, schreiben auf Pin O7
15	Identifikationskennung IO-Port

Tabelle 10: Adressraumbelegung des IO-Ports

Adresse	Funktion
0	Reload für Datenübertragungsrate
1	Konfigurationsregister
2	Sende/Empfangsdaten
3	Startadresse für Datenblockübertragung
4	Länge des Datenblocks [Zeichen]
15	Identifikationskennung serielle Schnittstelle

Tabelle 11: Adressraumbelegung der seriellen Schnittstelle

5.3 IO-Port

Diese Einheit ist für einfache Ein- und Ausgaben zuständig. Dazu sind hier jeweils 8 separate Eingangs-Pins und Ausgangs-Pins vorhanden. Jeder Pin wird über eine eigene Adresse angesprochen, wobei Ein- und Ausgabe jeweils paarweise angeordnet sind. Dies bedeutet, dass ein Lesen eines Port den Wert des Eingangs-Pins als Resultat liefert, ein Schreiben auf die gleiche Adresse ändert den Wert des Ausgangs-Pins.

5.4 Serielle Schnittstelle

Die serielle Schnittstelle unterstütz nur asynchrone Datenübertragung. Dabei wird zu Beginn einer Übertragung ein Startbit gesendet. Darauf folgen 2 bis 14 Datenbits, abgeschlossen von einem optionalen Paritätsbit und ein oder zwei Stoppbits. Ein Übertragung kann jeweils nur in eine Richtung (Senden oder Empfangen) stattfinden. Jede Übertragung kann einzeln oder als Blockübertragung durchgeführt werden; dabei stammen die Daten aus dem internen Pufferspeicher oder werden dort hin abgelegt. Um eine Datenübertragung z.B. mittels RS485 durchführen zu können, wird während eines Sende-Vorgangs ein externes Direction-Bit gesetzt. Dieses ist direkt an Bit 0 des Konfigurationsregisters geknüpft, dessen Aufbau in Abbildung 6 gezeigt wird. Für die Festlegung der Übertragungsrate muss ein Wert in das Reloadregister eingetragen werden, mittels dessen die Frequenzteilung erfolgt. Dieser Wert berechnet sich wie folgt:

$$\text{Reload} = \text{0xFFF} - (\text{externe Frequenz} / \text{Baud-Rate})$$

Dabei ist zu beachten, dass jede Schnittstelle einen separaten externen Frequenzeingang besitzt.

5.5 Pufferspeicher

Der Pufferspeicher dient als Zwischenspeicher für die Übermittlung von Datenblöcken über eine serielle Schnittstelle. In der Version des FPGA ist der Speicher 32 Bit breit und hat eine Größe von

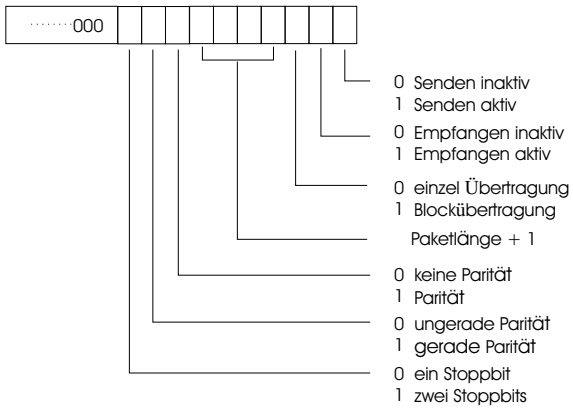


Abbildung 6: Kontrollregister der seriellen Schnittstelle

Adresse	Funktion
0	Bits 15-10 der Basisadresse
15	Identifikationskennung Pufferspeicher

Tabelle 12: Adressraumbelegung des internen Pufferspeichers

1024 Worten. Da der Speicher an sich aus Gründen der Flexibilität nicht fest in den Adressraum eingebunden ist, muss vor der ersten Nutzung eine Basisadresse festgelegt werden. Über diese kann anschließend das Beschreiben und Auslesen des Speichers erfolgen. Die Basisadresse muss in ein Konfigurationsregister in dem fest zugeordneten Adressraum des Pufferspeichers geschrieben werden. Dabei ist zu beachten, dass nur die Bits 15-10 der gewünschten Adresse dort eingetragen werden (ein Rechtsverschiebung um 10 Bit ist erforderlich).

A Liste der Hardware-Befehle

Tabelle 13: Standard-Java Bytecodes

Befehl	Bytecode (dez)	Bytecode (hex)	Länge (Byte)	Instruktion	Latenz	Beschreibung
nop	0	00	1	nop		nop
aconst_null	1	01	1	push_op		push null object
iconst_m1	2	02	1	push_op		push integer constant -1
iconst_0	3	03	1	push_op		push integer constant 0
iconst_1	4	04	1	push_op		push integer constant 1
iconst_2	5	05	1	push_op		push integer constant 2
iconst_3	6	06	1	push_op		push integer constant 3
iconst_4	7	07	1	push_op		push integer constant 4
iconst_5	8	08	1	push_op		push integer constant 5
bipush	16	10	2	push_op		push byte-sized value
sipush	17	11	3	push_op		push two-byte value
iload	21	15	2	push_d1		load local integer variable
aload	25	19	2	push_d1		load local object variable
iload_0	26	1A	1	push_d1		load local integer variable #0
iload_1	27	1B	1	push_d1		load local integer variable #1
iload_2	28	1C	1	push_d1		load local integer variable #2
iload_3	29	1D	1	push_d1		load local integer variable #3
aload_0	42	2A	1	push_d1		load local object variable #0
aload_1	43	2B	1	push_d1		load local object variable #1
aload_2	44	2C	1	push_d1		load local object variable #2
aload_3	45	2D	1	push_d1		load local object variable #3
iaload	46	2E	1	Mikro		load from array of integer
aaload	50	32	1	Mikro/t18		load from array of object
baload	51	33	1	Mikro		load from array of (signed) bytes
caload	52	34	1	Mikro		load from array of chars
saload	53	35	1	Mikro		load from array of (signed) shorts
istore	54	36	2	store_v		store local integer variable
astore	58	3A	2	store_v/t17		store local object variable
istore_0	59	3B	1	store_v		store local integer variable #0
istore_1	60	3C	1	store_v		store local integer variable #1
istore_2	61	3D	1	store_v		store local integer variable #2
istore_3	62	3E	1	store_v		store local integer variable #3
astore_0	75	4B	1	store_v/t17		store local object variable #0
astore_1	76	4C	1	store_v/t17		store local object variable #1
astore_2	77	4D	1	store_v/t17		store local object variable #2
astore_3	78	4E	1	store_v/t17		store local object variable #3
iastore	79	4F	1	Mikro		store into array of int
aastore	83	53	1	Mikro/t19		store into array of object
bastore	84	54	1	Mikro		store into array of (signed) bytes
castore	85	55	1	Mikro		store into array of chars
sastore	86	56	1	Mikro		store into array of (signed) shorts
pop	87	57	1	nop		pop top element
pop2	88	58	1	nop		pop top two elements
dup	89	59	1	push_d1		dup top element
dup_x1	90	5A	1	Mikro/t22		dup top element. Skip one
dup_x2	91	5B	1	Mikro/t23		dup top element. Skip two
dup2	92	5C	1	Mikro		dup top two elements.

Tabelle 13: Standard-Java Bytecodes

Befehl	Bytecode (dez)	Bytecode (hex)	Länge (Byte)	Instruktion	Latenz	Beschreibung
dup2_x1	93	5D	1	Mikro/t24		dup top two elements. Skip one
dup2_x2	94	5E	1	Mikro/t25		dup top two elements. Skip two
swap	95	5F	1	Mikro/t21		swap top two elements of stack.
iadd	96	60	1	add		integer add
isub	100	64	1	sub		integer subtract
imul	104	68	1	mul		integer multiply
idiv	108	6C	1	Mikro		integer divide
irem	112	70	1	Mikro		integer mod
ineg	116	74	1	neg		integer negate
ishl	120	78	1	shl		shift left
ishr	122	7A	1	shr		shift right
iushr	124	7C	1	ushr		unsigned shift right
iand	126	7E	1	and		boolean and
ior	128	80	1	or		boolean or
ixor	130	82	1	xor		boolean xor
iinc	132	84	3	iinc		increment lcl variable by constant
i2b	145	91	1	i2b		integer to byte
i2c	146	92	1	i2c		integer to character
i2s	147	93	1	i2s		integer to signed short
ifeq	153	99	3	braeq	4	goto if equal
ifne	154	9A	3	brane	4	goto if not equal
iflt	155	9B	3	bralt	4	goto if less than
ifge	156	9C	3	brage	4	goto if greater than or equal
ifgt	157	9D	3	bragt	4	goto if greater than
ifle	158	9E	3	brale	4	goto if less than or equal
if_icmpeq	159	9F	3	braeq	4	compare top two elements of stack
if_icmpne	160	A0	3	brane	4	compare top two elements of stack
if_icmplt	161	A1	3	bralt	4	compare top two elements of stack
if_icmpge	162	A2	3	brage	4	compare top two elements of stack
if_icmpgt	163	A3	3	bragt	4	compare top two elements of stack
if_icmple	164	A4	3	brale	4	compare top two elements of stack
if_acmpeq	165	A5	3	braeq	4	compare top two objects of stack
if_acmpne	166	A6	3	brane	4	compare top two objects of stack
goto	167	A7	3	goto	4	unconditional goto
ireturn	172	AC	1	Mikro		return integer from procedure
areturn	176	B0	1	Mikro/t20		return object from procedure
return	177	B1	1	Mikro		return (void) from procedure
arraylength	190	BE	1	Mikro		get length of array
ifnull	198	C6	3	braeq	4	goto if null
ifnonnull	199	C7	3	brane	4	goto if not null
ldc_quick	203	CB	2	Mikro		load a const from constant table

Tabelle 14: Erweiterte Bytecodes

Befehl	Bytecode (dez)	Bytecode (hex)	Instruktion	Latenz	Beschreibung
load_ubyte	0	00	load		load unsigned byte from mem
load_byte	1	01	load		load signed byte from mem
load_char	2	02	load		load unsigned char from mem

Tabelle 14: Erweiterte Bytecodes

Befehl	Bytecode (dez)	Bytecode (hex)	Instruktion	Latenz	Beschreibung
load_short	3	03	load		load signed short from mem
load_word	4	04	load		load int from mem
priv_ret_from_trap	5	05	Mikro		return from trap
io_read	6	06	load		read 32bit from io bus
priv_ret_from_ISR	7	07	Mikro		return from ISR
store_byte	32	20	store		store byte to mem ..,d,a
store_short	34	22	store		store short to mem ..,d,a
store_word	36	24	store		store int to mem ..,d,a
io_write	38	26	store		write 32bit to io bus ..,d,a
shl_carry	62	3E	shlcarry		(D1<<1)+carry-bit aus EXE
xchg	63	3F	Mikro		[D2]<=[D2]orD1,=>[D2]alt;locked
read_pc	64	40	push_d1		read PC register
read_vars	65	41	push_d1		read VARS register
read_frame	66	42	push_d1		read FRAME register
read_optop	67	43	push_d1		read OPTOP register
read_tsr	68	44	push_d1		read TS register
read_cp	69	45	push_d1		read CONSTPOOL register
read_dp	70	46	push_d1		read Dribbler register
sw_resume	82	52	sw_resu		set sw_active bit of thread
hw_resume	83	53	hw_resu		set hw_active bit of thread
atom_unlock	84	54	Aunlock		end of atomic region
read_stack	88	58	rd_cross		read from cross stack (cross++)
read_cross	89	59	push_d1		get cross pointer from other stack
read_global0	90	5A	push_d1		read GLOBAL0 register
read_global1	91	5B	push_d1		read GLOBAL1 register
read_tt	92	5C	push_op		read thread tag
read_sct	93	5D	push_d1		read stack cross tag
read_d	94	5E	push_d1		read from dribbler (dribbler++)
read_refbit	95	5F	rd_ref		read reference bit of optop
write_pc	96	60	wr_pc	4	write PC register
write_vars	97	61	wr_base	2	write FRAME register
write_optop	99	63	wr_base	2	write OPTOP register
write_tsr	100	64	wr_base	2	write GLOBAL0 to TS register
write_cp	101	65	wr_base	2	write CONSTPOOL register
write_dp	102	66	wr_base	2	write Dribbler register
sw_suspend	114	72	sw_susp	4	clear sw_active bit of thread
hw_suspend	115	73	hw_susp	4	clear hw_active bit of thread
atom_lock	116	74	Alock		begin of atomic region
sw_suspend_unlock	117	75	sw_susp	4	clear sw_active bit & atom_unlock
write_stack	120	78	wr_cross		write to cross stack (cross++)
write_cross	121	79	wr_base	2	set new cross pointer in other stack
write_global0	122	7A	wr_base	2	write GLOBAL0 register
write_global1	123	7B	wr_base	2	write GLOBAL1 register
write_tt	124	7C	nop		write thread tag
write_sct	125	7D	wr_base	2	write stack cross tag
write_d	126	7E	wr_dp		write to dribbler (dribbler–)
write_refbit	127	7F	wr_base		.., obj, refbit =¿ (refbit, obj)

B Liste der Mikro-Programme

Tabelle 15: Mikro-Programme

Programm	Laufzeit [Takte]	davon Latenzen	Beschreibung
iaload	6		Wird für iaload und aaload (nur ohne GC) verwendet
baload	4		
caload	6		
saload	6		
iastore	11	1	Wird für iastore und iaload (nur ohne GC) verwendet
bastore	9	1	
sastore	11	1	Wird für castore und sastore verwendet
arlen	3		
dupx1	4		
dupx2	6		
dup2	2		
dup2x1	5		
dup2x2	6		
swap	3		
ireturn	11	2	Wird für ireturn und areturn (nur ohne GC) verwendet
return	8	2	
div	561	136	Dividend < 0,Divisor < 0
		138	Dividend < 0,Divisor $>= 0$
			Dividend $>= 0$,Divisor < 0
	557	136	Dividend $>= 0$,Divisor $>= 0$
rem	473	136	Dividend < 0,Divisor < 0
			Dividend < 0,Divisor $>= 0$
			Dividend $>= 0$,Divisor < 0
	469		Dividend $>= 0$,Divisor $>= 0$
ldc_quick	9		
trap_invoke	15	4	Wird für alle Trap-/ISR-Aufrufe verwendet
ret_trap	13	6	
ret_isr	13	6	

C Liste der Signalquellen für die Signal-Unit

Tabelle 16: Signalquellen für die Signal-Unit

Bit-Nr.	Bit Maske	Signal
0	0x00000001	UART0 - Receive
1	0x00000002	UART0 - Transmit
2	0x00000004	UART1 - Receive
3	0x00000008	UART1 - Transmit
4	0x00000010	UART2 - Receive
5	0x00000020	UART2 - Transmit
6	0x00000040	Zähler/Zeitgeber 0
7	0x00000080	Zähler/Zeitgeber 1
8	0x00000100	Zähler/Zeitgeber 2
9	0x00000200	Zähler/Zeitgeber 3
10	0x00000400	Capture/Compare A
11	0x00000800	Capture/Compare B
12	0x00001000	Capture/Compare C
13	0x00002000	Capture/Compare D
14	0x00004000	Externer Wake-Up 0
15	0x00008000	Externer Wake-Up 1
16	0x00010000	Thread 0: Deadline missed
17	0x00020000	Thread 1: Deadline missed
...	...	...

D Instruktionen der Execute-Unit

Tabelle 17: Instruktionen der Execute-Unit

Instruktion	Funktion	Ziel
NOP	keine	-
PUSH_OP	Schreiben des übergebenen Operationswerts	Top of Stack
PUSH_D1	Schreiben des ersten Quelloperanden	Top of Stack
STORE_V	Schreiben des ersten Quelloperanden	Relativ zum VARS-Zeiger
ADD	Addieren der beiden Operanden	Top of Stack
SUB	Subtrahiert ersten und zweiten Operanden	Top of Stack
MUL	Multipliziert ersten und zweiten Operanden	Top of Stack
NEG	Negiert ersten Operanden	Top of Stack
SHR	Verschiebung des zweiten Operanden um den Ersten nach rechts	Top of Stack
SHL	Verschiebung des zweiten Operanden um den Ersten nach links	Top of Stack
USHR	Verschiebung des zweiten Operanden um den Ersten nach rechts, Vorzeichen bleibt	Top of Stack
AND	Logische Und-Verknüpfung der beiden Operanden	Top of Stack
OR	Logische Oder-Verknüpfung der beiden Operanden	Top of Stack
XOR	Logische Exlusiv-Oder-Verknüpfung der beiden Operanden	Top of Stack
IINC	Addiert den Operationswert zum ersten Quelloperand	Ziel = Quelloperand
I2B	Konvertiert 8 auf 32 Bit	Top of Stack
I2C	Konvertiert 16 auf 32 Bit	Top of Stack
I2S	Konvertiert 16 auf 32 Bit	Top of Stack
BRAEQ	Springt zu PC+Operationswert, wenn Operand1 = Operand2	-
BRANE	Springt zu PC+ Operationswert, wenn Operand1 ! = Operand2	-
BRALT	Springt zu PC+ Operationswert, wenn Operand1 < Operand2	-
BRAGE	Springt zu PC+ Operationswert, wenn Operand1 => Operand2	-
BRALE	Springt zu PC+ Operationswert, wenn Operand1 <= Operand2	-
BRAGT	Springt zu PC+ Operationswert, wenn Operand1 > Operand2	-
WR_BASE	Schreibt den im ersten Operanden angegebenen Wert in das im Operationswert angegebene Kontroll-/Steuerregister	Das im Operationswert angegebende Kontroll-/Steuerregister
WR_PC	Unbedingter Sprung zur Adresse aus dem ersten Operanden	PC
LOAD	Ladebefehl aus dem Speicher. Der erste Operand gibt die Adresse an, die Arte des Zugriffs ist im Operationswert kodiert	Top of Stack

Tabelle 17: Instruktionen der Execute-Unit

Instruktion	Funktion	Ziel
STORE	Schreibbefehl in den Speicher. Die Adresse wird im ersten, die Daten im zweiten Operanden übergeben. Die Art (8,16,32 Bit) ist im Operationswert kodiert	-
SWSUSPEND	Das SW-Aktiv-Bit des im ersten Operanden angegebenen Thread wird gelöscht	-
SWRESUME	Das SW-Aktiv-Bit des im ersten Operanden angegebenen Thread wird gesetzt	-
HWSUSPEND	Das HW-Aktiv-Bit des im ersten Operanden angegebenen Thread wird gelöscht	-
HWRESUME	Das HW-Aktiv-Bit des im ersten Operanden angegebenen Thread wird gelöscht; zusätzlich wird ggf. die Deadline neu initialisiert	-
RD_CROSS	Wie PUSH_D1, lediglich der Quelloperand stammt aus einem fremden Stack; ist CROSSPOINTER = OPTOP des fremden Threads, so wird der Wert 0 mit gesetztem Ref-Bit geschrieben	Top of Stack
WR_CROSS	Schreibt den ersten Operanden auf einen fremden Stack	SCT:[CROSSPOINTER]
RD_REF	Liest das Ref-Bit des ersten Operanden und schreibt 1 oder 0	Top of Stack
WR_REF	Ist der erste Operand=1, so wird der Zweite mit gesetztem Ref-Bit geschrieben, andernfalls mit gelöschtem	Top of Stack
GOTO	Unbedingter Sprung zu PC+Operationswert	-
SHLCARRY	Linksverschiebung um 1 Bit mit Carry (wird für die Division benötigt)	Top of Stack
SUBIFLESS	Subtrahiert die beiden Operanden, wenn Operand1<Operand2	Top of Stack
WR_DP	Schreibt an die Adresse des Dribbler-Pointers; dieser wird anschließend decrementiert	[DP]

E Pinbelegung des FPGA Extension-Headers X2

Tabelle 18: Pinbelegung des FPGA Extension-Headers X2

X2-Pin	FPGA-Pin	Richtung	Modul	Beschreibung
A9	AK20	In	UART0	Clock
C31	AD1	Out		Transmit
C30	AD4	In		Receive
A10	AJ19	In	UART1	Clock
C29	AE4	Out		Transmit
C28	AF4	In		Receive
A12	AH17	In	UART2	Clock
C27	AG3	Out		Transmit
C26	AK3	In		Receive
C25	AJ4	Out		Direction
C23	AK6	In	Zähler/Zeitgeber	Externe Clock
C22	AK7	In		Aktivierungspin (P)
C21	AH7	In	Capture/Compare	Capture A
C20	AJ8	In		Capture B
C19	AH9	In		Capture C
C18	AH10	In		Capture D
C17	AL12	Out		Compare A
C16	AH12	Out		Compare B
C15	AH13	Out		Compare C
C14	AH14	Out		Compare D
C13	AJ15	In	IO-Port	Input 0
C5	AH23	Out		Output 0
C12	AK17	In		Input 1
C4	AJ24	Out		Output 1
C11	AK18	In		Input 2
C3	AH25	Out		Output 2
C10	AH18	In		Input 3
C2	AJ26	Out		Output 3
C9	AH19	In		Input 4
a6	AJ23	Out		Output 4
C8	AJ20	In		Input 5
A5	AK24	Out		Output 5
C7	AJ21	In		Input 6
A4	AJ25	Out		Output 6
C6	AH22	In		Input 7
A3	AK26	Out		Output 7

F Belegung des IO-Adressraum

Tabelle 19: IO-Adressraums

Adresse	Modul	Beschreibung
0x0000-0x000n	Priority Manager	Konfiguration für Thread-Scheduling
0x0010-0x001n		GP Prozentanteile
0x0020-0x002n		EDF Deadline/ LLF Laxity
0x0030-0x003n		FPP Priorität
0x0080-0x008n	Signal	Unit Wake-Up Bitmaske
0x00A0-0x00An		ISR Bitmaske
0x00C0-0x00Cn		Wake-Up Statusregister
0x00E0-0x00En		ISR Statusregister
0x0100	UART 0	Reload Register
0x0101		Konfigurationsregister
0x0102		Send/Receive Daten
0x0103		Startadresse f. Blockdatenübertragung
0x0104		Blocklänge
0x010F		Identifikationswert UART
0x0110	UART 1	Reload Register
0x0111		Konfigurationsregister
0x0112		Send/Receive Daten
0x0113		Startadresse f. Blockdatenübertragung
0x0114		Blocklänge
0x011F		Identifikationswert UART
0x0120	UART 0	Reload Register
0x0121		Konfigurationsregister
0x0122		Send/Receive Daten
0x0123		Startadresse f. Blockdatenübertragung
0x0124		Blocklänge
0x012F		Identifikationswert UART
0x0130	Zähler/Zeitgeber	Konfiguration 0
0x0131		Reload/Zählerstand 0
0x0132		Konfiguration 1
0x0133		Reload/Zählerstand 1
0x0134		Konfiguration 2
0x0135		Reload/Zählerstand 2
0x0136		Konfiguration 3
0x0137		Reload/Zählerstand 3
0x013F		Identifikationswert Zähler/Zeitgeber
0x0140	Capture/Compare	Konfigurationsregister A
0x0141		Daten Register A
0x0142		Reload Register A
0x0143		Konfigurationsregister B
0x0144		Daten Register B
0x0145		Reload Register B
0x0146		Konfigurationsregister C
0x0147		Daten Register C
0x0148		Reload Register C
0x0149		Konfigurationsregister D
0x014A		Daten Register D
0x014B		Reload Register D
0x014F		Identifikationswert Capture/Compare

Tabelle 19: IO-Adressraums

Adresse	Modul	Beschreibung
0x0150	IO-Port	Ein-/Ausgabe Pin 0
0x0151		Ein-/Ausgabe Pin 1
0x0152		Ein-/Ausgabe Pin 2
0x0153		Ein-/Ausgabe Pin 3
0x0154		Ein-/Ausgabe Pin 4
0x0155		Ein-/Ausgabe Pin 5
0x0156		Ein-/Ausgabe Pin 6
0x0157		Ein-/Ausgabe Pin 7
0x015F		Identifikationswert IO-Port
0x0160	Pufferspeicher	Basisadressenregister
0x016F		Identifikationswert Pufferspeicher