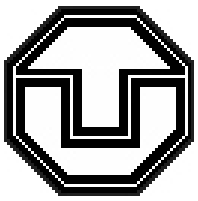

Entwicklung eines Simulators für den Spartan Mikrocontroller

Großer Beleg

Vorgelegt von:	Stefan Spickereit
Matrikel:	2943427
Betreuer:	Dr. Siegmar Schöne
Prüfer:	Prof. Dr. Christian Hochberger

Professur Mikrorechner

Institut für Technische Informatik

Fakultät Informatik

TU Dresden

Sommersemester 2006

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

ASCII	American Standard Code for Information Interchange
CISC	Complex Instruction Set Computer
CPI	Clocks per Instruction
CPU	Central Processing Unit
CSV	Comma Separated Values
DMA	Direct Memory Access
ExMem	Execute and Memory
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GPR	General Purpose Register
GUI	Graphical User Interface
HDL	Hardware Description Language
HTML	Hypertext Markup Language
IfId	Instruction Fetch and Decode
IRQ	Interrupt Request
LSB	Least Significant Bit
MC	Mikrocontroller
MSB	Most Significant Bit
NOP	No Operation
PC	Program Counter
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
SFR	Special Function Register
UART	Universal Asynchronous Receiver / Transmitter
Wb	Write Back
XML	Extensible Markup Language

1 Einführung

Der Spartan Mikrocontroller (MC) ist ein spezieller CPU-Core, entwickelt an der Professur für Mikrorechner der TU Dresden, für die Bausteine der Spartan-Reihe des FPGA-Herstellers Xilinx. Er ist speziell auf die Ressourcen zugeschnitten, die in diesen FPGAs vorhanden sind. Alternativ sind natürlich auch FPGAs mit ähnlichen Hardwareressourcen (18 Bit breite Speicher und Multiplizierer) anderer Hersteller, etwa Lattice und Altera, nutzbar.

Für den Mikrocontroller sollte ein Simulator entworfen werden, welcher die taktgenaue Abarbeitung der Befehle innerhalb der Pipeline umsetzen und darstellen kann. Als Ein- und Ausgabe sollten zunächst nur die UART-Schnittstelle sowie ein demonstratives DMA-Gerät nachgebildet werden.

Diese Arbeit unterteilt sich in fünf Kapitel. Zunächst werden die theoretischen Hintergründe erläutert und die wesentlichen Merkmale des Spartan Mikrocontrollers vorgestellt. Schließlich wird die Realisierung der Simulationsanwendung selbst beschrieben, zunächst das Backend, danach das grafische Frontend. Wichtige Entwurfsentscheidungen und technische Besonderheiten werden ausführlich dargelegt. Daran schließt sich eine kurze Anleitung an, um einen schnellen Einstieg in die Bedienung des Simulators zu finden. Im letzten Abschnitt werden die Ergebnisse bewertet und auch alternative Ansätze bzw. Erweiterungsmöglichkeiten angesprochen. Im Anhang werden die einzelnen Instruktionen des Mikrocontroller-Befehlssatzes aufgeführt und insbesondere deren Auswirkungen auf die Pipeline detailliert beschrieben.

2 Der Spartan Mikrocontroller

2.1 RISC-Prozessorarchitekturen

Anfang der Achtziger Jahre entwickelte sich eine Rechnerarchitektur, die mit wenigen kurzen und schnellen Befehlen arbeitete: Reduced Instruction Set Computers (RISC). Im Gegensatz zu anderen Architekturen (z.B. CISC – Complex Instruction Set Computers) waren die Befehlssätze äußerst kompakt und performant gestaltet.

2.1.1 Allgemeine Eigenschaften

RISC-Architekuren charakterieren sich durch drei wesentliche Merkmale:

- Ziel ist es, eine Instruktion innerhalb eines Taktes abarbeiten zu können, d.h. CPI (Clocks per Instruction) = 1. Aus dieser nicht-trivialen Forderung entwickelte sich die Idee des sogenannten Pipelinings, einer Hardwarestruktur, welche die wichtigsten Funktionalitäten des Prozessorkerns zu parallelisieren versucht.

- Jeder Befehl wird aufgrund dessen in mehrere (mindestens zwei) Bestandteile gegliedert und mittels der Pipeline abgearbeitet. Diese Pipeline kann parallel jeweils einen Befehlsteil in einer sogennanten Pipeline-Stufe (Pipeline Stage) abarbeiten, sodass mehrere versetzte Instruktionen simultan bearbeitet werden.

- Um langsame Speicherzugriffe so gut wie möglich zu vermeiden, steht bei RISC-Architekturen ein relativ großer Satz von Universalregistern (General Purpose Registers, GPR) zur Verfügung. Die meisten Instruktionen besitzen als Operanden einen

oder mehrere Universalregister. Der Speicherzugriff wird durch explizite Load/Store-Befehle realisiert.

Eine einzelne Instruktion ist, verglichen mit einer CISC-Architektur, sehr schlank. Um einfache Funktionalitäten umzusetzen, sind im Allgemeinen mehrere aufeinander folgende Befehle nötig. Mit anderen Worten, man erhöht den Software-Aufwand zugunsten der dafür benötigten Hardware.

CISC-Befehlssätze versuchen die Anzahl der notwendigen Instruktion für eine Aufgabe (ein Programm) zu minimieren, müssen dazu aber die CPI entsprechend erhöhen. Bei RISC-Architekturen ist es genau umgekehrt: Sehr kurze Befehle minimieren die CPI, jedoch müssen für die gleiche Aufgabe wesentlich mehr Instruktionen verwendet werden.

Aufgrund des massiven Preisverfalls bei Speicherhardware und der Mächtigkeit moderner Compilerwerkzeuge spielt eine minimale Programmgröße nur noch eine untergeordnete Rolle. Viel wesentlicher ist es geworden, bei gegebenen Taktfrequenzen möglichst viele Operationen pro Zeiteinheit umsetzen zu können.

2.1.2 Pipelining

Durch Pipelining können mehrere Befehle quasi-parallel verarbeitet werden. Eine ideale Pipeline-Architektur ermöglicht eine Leistungssteigerung gegenüber einer sequenziellen Abarbeitung von

$$\frac{\text{Sequenzielle Befehlsausführungszeit}}{\text{Anzahl der Pipeline-Stufen}}$$

Die meisten RISC-Befehlssätze unterteilen ihre Instruktionen ähnlich wie eine DLX-Maschine in die folgenden Phasen:

1. Instruction Fetch (IF): Die Instruktion wird via Program Counter (PC) aus dem (Befehls-)Speicher ausgelesen.

2. Instruction Decode (ID): Anschließend wird diese decodiert, d.h. abhängig von ihrer Semantik werden beispielsweise Steuersignale gesetzt. Insbesondere werden hier gegebenenfalls die Operanden decodiert bzw. aus den Registern ausgelesen.

3. Execute (EX): Die Operation wird im Rechenwerk (z.B. in der ALU oder FPU) aus-

geführt. Gegebenenfalls können hier auch Verzweigungs- bzw. Sprung-Berechnungen ausgeführt werden.

4. Memory (MEM): Ist ein Speicherzugriff erforderlich, so wird er in dieser Phase durchgeführt.

5. Write Back (WB): Soll ein berechnetes Ergebnis auf ein Register zurückgeschrieben werden, so erfolgt dies in dieser letzten Phase.

Anhand dieser Segmentierung kann eine Pipeline konstruiert werden, bei der jeder Instruktionsphase eine gekapselte Pipeline-Stufe zugeordnet wird. Jeweils an den Schnittstellen zweier Stufen werden Register eingefügt, welche unterbinden, dass Signale mehrere Stufen „durchwandern“. Sobald eine Instruktionsphase abgearbeitet wurde, rückt der gesamte Befehl innerhalb der Pipeline eine Stufe weiter. Das bedeutet also, dass bei voller Auslastung in jedem Schritt ein Befehl fertig bearbeitet wird – nämlich der, der sich zuvor in seiner letzten Ausführungsphase befand – und ein neuer Befehl begonnen wird. Abbildung 2.1 verdeutlicht dies schematisch.

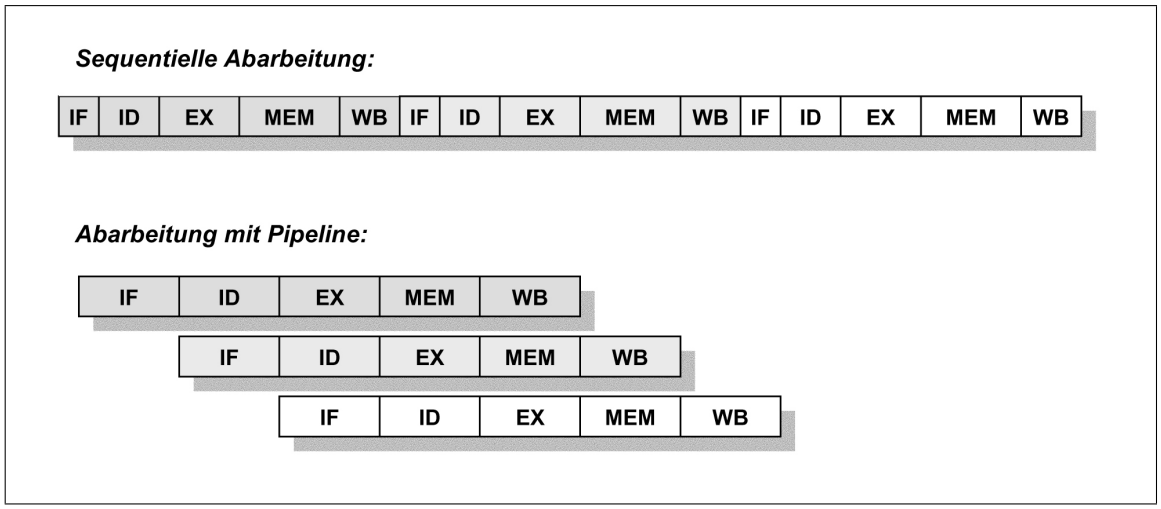

Abbildung 2.1: Prinzip einer Befehls-Pipeline

Sämtliche Pipeline-Stufen müssen gleich lang sein, d.h. sie müssen die gleiche Zahl von Taktzyklen benötigen. Orientiert wird sich dabei an der Ausführungszeit der längsten (langsamsten) Stufe. Meistens wird ein Pipeline-Schritt innerhalb von einem oder zwei Taktzyklen ausgeführt. Durch die Komplexität der Pipeline-Architektur ist ein Steuerungs-Overhead unvermeidbar, welcher zusammen mit den Halte- und Verzögerungszeiten der zusätzlichen Schnittstellenregister die Abarbeitungszeit eines einzelnen Befehls verlängert.

Trotzdem ist durch die erzielte Parallelität der Befehlsdurchsatz größer, d.h. das Programm wird letztendlich signifikant schneller ausgeführt.

Die reibungslose Abarbeitung des Befehlsstroms setzt allerdings voraus, dass alle Instruktionen voneinander völlig unabhängig sind. Dies ist natürlich nur selten der Fall, sodass die Pipeline-Architektur vor prinzipiellen Problemen steht, den sogenannten Hasards. Es können drei verschiedene Typen differenziert werden:

- Struktur-Hasards: Die Hardware kann meist nicht beliebig Ressourcen, etwa einen Hauptspeicherzugriff, für mehrere Befehle gleichzeitig verfügbar machen. Müssen Pipeline-Stufen auf ihre Ressourcen warten, so schwindet der Vorteil der Parallelität.

- Steuerungs-Hasards: Die Pipeline setzt einen kontinuierlichen linearen Strom vorraus. Sobald jedoch ein bedingter Verzweigungsbefehl auftritt, kann nicht entschieden werden, welcher Befehl als Nachfolger aufgenommen wird, bevor die Bedingung fertig berechnet und geprüft wurde.

- Daten-Hasards: Viele Befehle arbeiten mit Datenwerten (z.B. Registerinhalten), die von vorangegangenen Instruktionen unmittelbar verändert wurden. Es muss garantiert sein, dass keine veralteten Werte genutzt werden.

Die konkrete Prozessor-Architektur muss Lösungen für alle Hasards bereitstellen.

2.2 Eigenschaften des Spartan MC

Der Spartan Mikrocontroller ist eine Universalregister-Architektur und orieniert sich mit seinem Befehlssatz stark an einer DLX-Maschine. Die Wortbreite beträgt 18 Bit. Er verfügt über einen RAM-Hauptspeicher, in welchem sowohl Instruktions- als auch Datenworte abgelegt werden. Der Speicher kann nur über explizite Load/Store-Instruktionen genutzt werden.

Der Hardware-Core ist rekonfigurierbar. Das heißt, es ist möglich seine peripheren Komponenten beliebig zu variieren und zu vervielfältigen. Dadurch kann ein auf die jeweiligen Anforderungen des vorgesehenen Anwendungsgebietes angepasster Entwurf synthetisiert werden.

Der Prozessorkern nutzt eine dreistufige Pipeline bestehend aus Instruction Fetch and Decode (IF/ID), Execute and Memory (EX/MEM) sowie Write Back (WB).

Der Spartan MC arbeitet üblicherweise mit einer Taktfrequenz von bis zu 30 MHz. Abbildung 2.2 zeigt ein typisches FPGA-Entwicklungsboard.

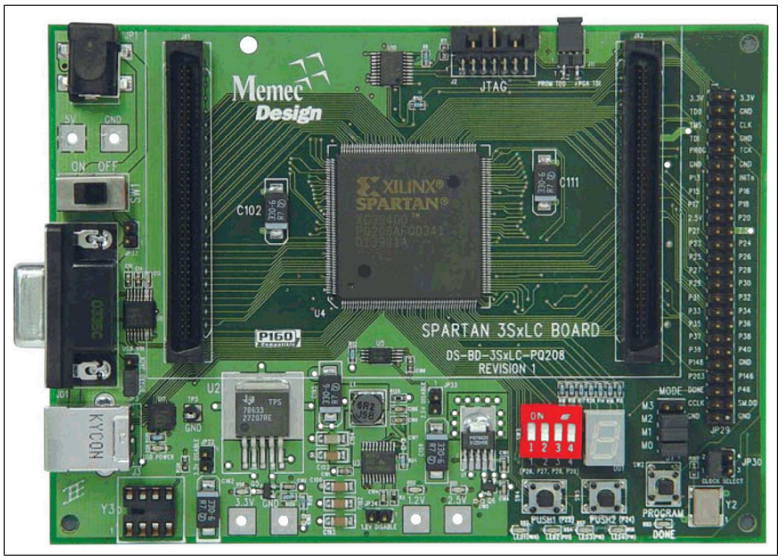

Abbildung 2.2: Xilinx Spartan 3 FPGA Entwicklungsboard

2.3 Register

2.3.1 General-Purpose-Register

Für die Abarbeitung des Programmes stehen 16 General-Purpose-Register (GPR) mit je 18 Bit Breite zur Verfügung: `R0` bis `R15`. Diese Register sind der Ausschnitt des zu dem Zeitpunkt aktiven Registerfensters (Register Window). Tatsächlich stehen weitaus mehr physikalische Register im RAM zur Verfügung. Eine Instruktion kann jedoch stets nur auf 16 von ihnen zugreifen, abhängig von der sogenannten Register-Window-Base, einem ganzzahligen sechsbittigen Offset-Faktor. Wird dieser inkrementiert (falls möglich), so ergibt sich das nachfolgende Fenster; wird er stattdessen dekrementiert, das Vorgängerfenster.

Die verfügbaren Register untergliedern sich in insgesamt vier Kategorien mit jeweils vier Registerpositionen. Siehe dazu auch Abbildung 2.3.

- Globale Register (`global`): `R0` bis `R3`. Sie sind unabhängig vom Registerfenster und

sind damit immer nutzbar.

- Lokale Eingangsregister (`in`): `R4` bis `R7`. Sie stellen den ersten Teil des Registerfensters dar. Angrenzende Fenster überlappen sich um je vier Register. Die Eingangsregister, d.h. die ersten vier Positionen, sind identisch mit den vier letzten Positionen des Vorgängerfensters.

- Lokale Ausgangsregister (`out`): `R12` bis `R15`. Analog sind dies die letzten vier Positionen. Das nachfolgende Registerfenster übernimmt diese als Eingangsregister.

- Rein lokale Register (`local`): `R8` bis `R11`. Diese sind nur im aktuellen Fenster zu erreichen.

Der Befehlssatz des Spartan MC umfasst Sprungbefehle, welche die Register-Base inkrementieren bzw. dekrementieren können und das Fenster damit jeweils um genau acht Positionen verschieben. Die Idee dahinter ist, Unterprogramm-Aufrufe zu optimieren. Die Fensterüberlappungen können also gegebenenfalls bis zu vier Registertransfer-Befehle einsparen.

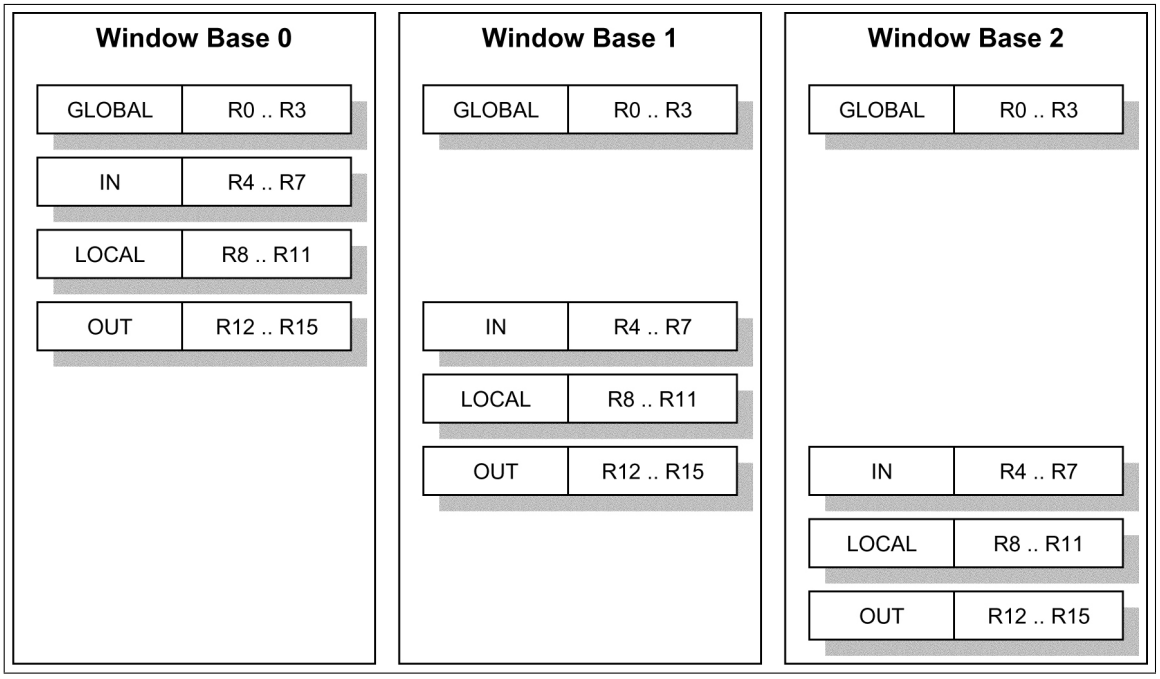

Abbildung 2.3: Register Window

2.3.2 Special-Function-Register

Der Prozessor kann neben den General-Purpose-Registern auch auf Sonderregister zugreifen, die Special-Function-Register (SFR). Dies sind:

`MUL`	18 Bit	Bei der Multiplikation zweier 18-bittiger Operanden wird das Ergebnis 36 Bit breit. Das niederwertige Wort wird als ALU-Ergebnis zurückgeliefert. Das höherwertige Wort wird in `MUL` abgespeichert, um nachfolgend darauf zugreifen zu können.
`LEDS`	7 Bit	Während der Entwicklung ist es hilfreich, Debug-Informationen einfach ausgeben zu können. Dafür dient das `LEDS`-Register, dessen Bits an die Sieben-Segment-Anzeige des FPGA-Boards weitergeleitet werden.
`WND`	6 Bit	Das Register Base Window wird hier abgelegt.
`MM`	1 Bit	Es existieren mehrere Formate zur Speicheradressierung. Das `MM`-Register wird dabei für die Berechnungen verwendet. Details dazu sind in Abschnitt 2.4.2 zu finden.
`CC`	1 Bit	Für Verzweigungs- und Vergleichsbefehle wird `CC` als Flag genutzt.
`INT`	1 Bit	Sämtliche Hardware-Interrupts können mit dem Interrupt-Enable-Bit aktiviert bzw. deaktiviert werden. Nach einem Reset ist `INT` stets auf eins, d.h. Interrupts werden zugelassen. Trifft eine Interruptforderung (Interrupt Request, IRQ) ein, so wird das Bit durch die Hardware automatisch auf null gesetzt. Es kann durch Software wieder anschließend auf eins zurückgesetzt werden.

2.4 Speicher

2.4.1 Hauptspeicher, DMA-Speicher und IO-Adressraum

Der Spartan MC arbeitet mit den RAM-Speicherblöcken des FPGAs. Der gesamte Adressierungsbereich von 18 Bit Breite kann, mathematisch, bis zu 256 K = 262.144 Adressen umfassen.

Die RAM-Blöcke des Spartan FPGAs sind dual-port. Dies bedeutet, dass in einem Takt bis zu *zwei* Zugriffe parallel durchgeführt werden können. Da es keinen separierten Instruktionsspeicher gibt, müssen gegebenenfalls innerhalb der Pipeline die IF/ID- und die EX/MEM-Stufe gleichzeitig auf den Hauptspeicher zugreifen, was durch den RAM unterstützt wird. Der Hauptspeicherbereich wird beginnend ab `0x00000` adressiert. Er setzt sich aus zwei neun Bit breiten Blöcken zusammen und kann damit auch halbwortweise gelesen bzw. beschrieben werden.

An den Hauptspeicherbereich kann sich ein DMA-Speicher anschließen. Die EX/MEM-Stufe hat auch auf diesen RAM Zugriff; IF/ID hingegen nicht. Dies bedeutet, es können keine Instruktionen aus dem DMA-Bereich geladen werden. Der Port wird für das DMA-Gerät bzw. die DMA-Geräte verwendet, welche, unabhängig von der CPU, Daten lesen und schreiben können. Für EX/MEM ist es jedoch nicht möglich, auf einzelne Halbworte des DMA-Speichers zu schreiben, da dieser im Gegensatz zum Hauptspeicher aus einem 18 Bit breiten Block besteht und beim Schreibzugriff alle Bits überschrieben werden. [1]

Desweiteren gibt es den IO-Adressbereich. Diese Adressen werden nicht auf einen RAM geführt, sondern stehen frei, um von IO-Geräten genutzt zu werden. Prinzipiell kann ein und dieselbe Adresse von mehreren Komponenten belegt werden. Das Datum eines Schreibzugriffes wird dabei an alle „verdrahteten“ Einheiten geleitet. Das Ergebnis eines Lesezugriffes wiederum ergibt sich aus der OR-Veknüpfung der einzelnen jeweils angelegten Bits. Lese- und Schreibzugriffe auf IO-Geräte müssen nicht zwingendermaßen als „lese von bzw. schreibe auf Speicherplatz“ interpretiert werden. Vielmehr kann ein Zugriff ein (nahezu) beliebiges Verhalten der entsprechenden Komponente induzieren.

[1] Es ist natürlich stets möglich, eine Art Strobe-Zugriff in Software auszuführen, d.h. ein Lesezugriff, gefolgt von einer Modifikation des entsprechenden Halbwortes und schließlich einem Schreibzugriff.

2.4.2 Adressierungsformate

Der Spartan MC besitzt zwei prinzipielle Adressierungsformate. Der Zugriff auf Instruktionen erfolgt mit einer Wortlänge von 18 Bit. Das bedeutet, dass sich jede Adressangabe auf ein vollständiges 18 Bit Wort bezieht. Entsprechend wird der Program Counter (PC) um genau eins erhöht, um den Nachfolgebefehl anzusprechen. Gleichermaßen können auch 18 Bit breite Datenworte aus dem Speicher ausgelesen werden. Halbworte werden innerhalb dieses Adressierungsformats mit High (Bits 17..9) bzw. Low (Bits 8..0) differenziert.

Mit der sogenannten Datenadressierung können neun Bit breite Datenworte aus dem Speicher ausgelesen werden. Dabei wird das `MM`-Registerbit der SFR herangezogen. Abbildung 2.4 verdeutlicht dies grafisch. Aus der ursprünglichen Adresse wird eine neue gewonnen, indem das niederwertigste Bit (LSB, an Position 0) entfernt und gleichzeitig das `MM`-Bit an die höchstwertigste Stelle (MSB, Position 17) gesetzt wird. Das entfernte Bit dient zur Auswahl eines der beiden Halbworte, aus denen das adressierte Datum zusammengesetzt ist. Ist das Auswahlbit gleich eins, so wird das niederwertige Halbwort ausgewählt (Low), andernfalls das höherwertige Halbwort (High). Es handelt sich demgemäß um ein Big-Endian-Format.

Die 18-Bit Addressierung nutzt ebenso das `MM`-Bit als MSB, wobei jedoch das (alte) LSB verworfen wird, da eine Auswahl zwischen High- oder Low-Halbwort nicht notwendig ist.

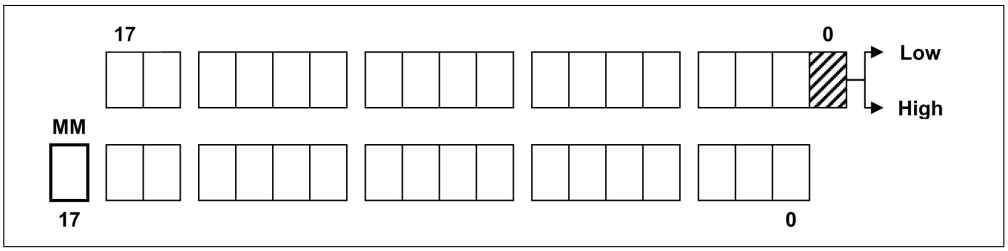

Abbildung 2.4: Umwandlung der Datenadressierung mittels `MM`-Bit

Weiteres zur Datenadressierung wird in Abschnitt 4.3.2 besprochen.

2.5 Befehlssatz

Die Befehle des Spartan MC orientieren sich zum großen Teil am DLX-Befehlssatz. Die Instruktionen nutzen bis zu drei Operanden. Die meisten arbeiten mit nur zwei Operanden

(dyadisch) und lassen sich als Register-Register, Register-Immediate und Register-Speicher klassifizieren. Jeder Befehl ist exakt 18 Bit lang. Eine Übersicht ist Abbildung 2.5 zu entnehmen.

Es stehen drei Adressierungsarten zur Verfügung:

- Bei der Register-Adressierung werden die zu nutzenden Werte in die Universalregister der Register-File abgelegt.
  Beispiel: `ADD R0, R1`

- Eine Immediate-Adressierung verwendet stattdessen eine Konstante, welche direkt in das Instruktionswort gesetzt wird.
  Beispiel: `ADDI R0, 0x47`

- Der Speicher wird mittels einer Displacement-Adressierung angesprochen. Dabei wird sowohl ein Register als auch eine Konstante (das Displacement) angegeben; die auszuwählende Speicheradresse berechnet sich anhand dieser beiden Werte.
  Beispiel: `L18 R0, 0x2(R1)`

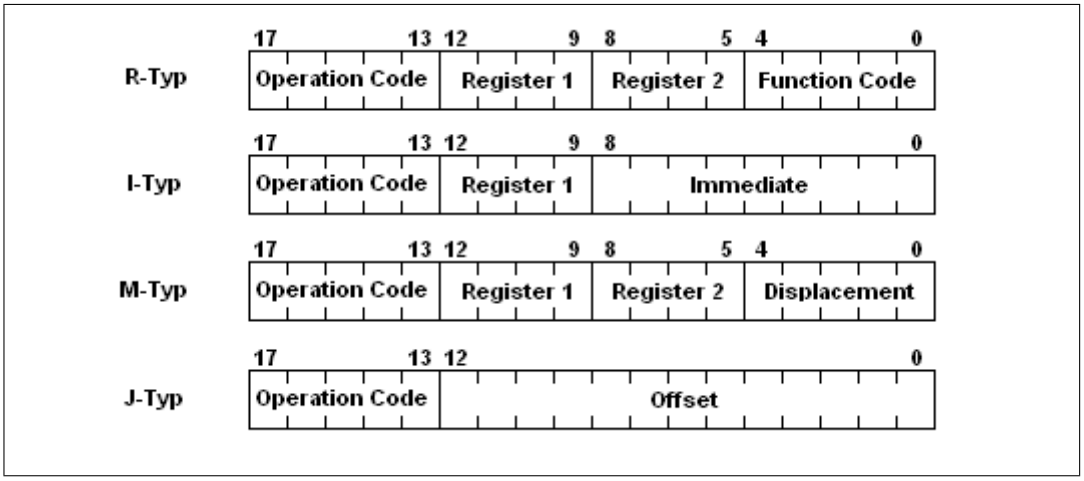

Abbildung 2.5: Einteilung der Befehlsformate

Die Instruktionen sollen kurz vorgestellt werden. Weitere Informationen sind dem Anhang sowie den Unterlagen zum Spartan MC zu entnehmen.

Arithemtisch-logische Befehle:

ADD, ADDU, ADDI, SUB, SUBU, MUL, MULI Es stehen die Grundrechenarten Addition, Subtraktion sowie Multiplikation zur Verfügung. Das Ergebnis wird jeweils in das erste Quellregister zurückgeschrieben. Bei der Multiplikation ist das Ergebnis $18 + 18 = 36$ Bit breit. Zurückgeschrieben wird nur das niederwertigste Wort, die oberen 18 Bit werden, wie bereits erwähnt, in das SF-Register `MUL` abgelegt und können von da aus wieder ausgelesen werden. `ADD` und `ADDU` sowie `SUB` und `SUBU` verhalten sich gegenwärtig noch identisch und rechnen vorzeichenlos. Es ist vorgesehen, dass `ADD` und `SUB` erweitert werden, sodass sie zwischen arithmetischen Vorzeichen und Wertebereichsüberlauf differenzieren können.

AND, ANDI, OR, ORI, XOR, XORI Als bitweise logische Operationen dienen AND, OR und XOR. Ihr Ergebnis wird analog zu arithmetischen Berechnungen in das erste Quellregister zurückgeschrieben.

IFADDUI, IFSUBUI Die beiden Befehle addieren bzw. subtrahieren einen Registerinhalt mit einem Immediate-Operanden. Das Ergebnis wird jedoch nur zurückgeschrieben, wenn das SF-Register `CC` gleich eins ist.

SIGEX Mittels `SIGEX` (Sign Extension) ist es möglich, die höherwertigsten Bits eines Registerinhaltes, definiert durch den ersten Operanden, abzuändern. Als zweiter Operand wird ein Immediate angegeben, welcher eine Bitbreite n definiert (16, 9 oder 8 Bit). Die nächst höheren Stellen werden durchgängig mit dem Bitwert, welcher an Position $n - 1$ steht, überschrieben.

LHI Als erster Operand wird ein Universalregister angegeben, als zweiter ein neun Bit breiter Immediate-Wert. Dieser zweite Operand wird in das höherwertige Halbwort des Registers geschrieben, das niederwertige Halbwort wird mit Nullen aufgefüllt (Load High Immediate).

SLL, SLLI, SRL, SRLI, SRA, SRAI Logische Shift-Operationen verschieben den Wert des Quellregisters bitweise um n Stellen nach links (`SLL`, Shift Left Logically) oder nach

rechts (SRL , Shift Right Logically). Die Schiebebreite n wird mit den niedersten fünf Bit des zweiten Operanden – ein Register- oder Immediate-Wert – definiert. „Frei werdende" Stellen werden mit Nullen aufgefüllt. Bei SRA (bzw. SRAI), Shift Right Arithmetically, werden die höherwertigsten Bits entsprechend dem bisherigen Vorzeichen besetzt. Damit wird garantiert, dass negative Zweierkomplement-Zahlen auch negativ bleiben. Es besteht die Möglichkeit, in den Hardware-Definitionen das Attribut CONFIG_PROC_SINGLE_SHIFT zu setzen. In diesem Fall wird stets nur um *eine* Bitposition geshiftet, unabhängig vom zweiten Operanden.

SEQ, SEQU, SEQI, SNE, SNEU, SNEI Diese Vergleichsbefehle prüfen die Operanden auf Gleichheit oder Ungleichheit (Set Conditional Equals bzw. Set Not Equals). Sollte dies der Fall sein, so wird das SF-Register CC auf eins gesetzt, andernfalls auf null.

SGT, SGTU, SGTI, SLT, SLTU, SLTI, SGE, SGEU, SGEI, SLE, SLEU, SLEI Analog zu den obigen Instruktionen prüfen diese Vergleichsbefehle auf kleiner (Set Less Than), kleiner gleich (Set Less Equal), größer (Set Greater Than) sowie größer gleich (Set Greater Equal) bei den zwei angegebenen Operanden. Ist der Vergleich positiv, wird CC auf eins gesetzt, andernfalls auf null.

Sprung- und Verzweigungsbefehle

BEQZ, BNEZ, BEQZC, BNEZC Die Verzweigungsbefehle BEQZ (Branch Equal Zero) und BNEZ (Branch Not Equal Zero) prüfen, inwiefern der Registerinhalt des ersten Operanden gleich bzw. ungleich null ist. Ist die Bedingung erfüllt, so wird dem nachfolgenden PC der zweite Operand, ein Immediate, aufaddiert. Andernfalls wird der PC einfach nur inkrementiert. Analog dazu arbeiten BEQZC (Branch Equal Zero CC) und BNEZC (Branch Not Equal Zero CC). Sie prüfen jedoch das CC-Register auf null. Das Instruktionswort beinhaltet nur einen einzigen Operanden, ein 13 Bit Offset, welcher bei erfüllter Bedingung dem nachfolgenden PC aufsummiert wird.

J, JR, JALR, JALS, JALRS, JRS Bei J (Jump) und JALS (Jump and Link with Shift) wird dem nachfolgenden PC ein Offset aufaddiert. Bei JR (Jump Register), JALR (Jump and Link Register), JALRS (Jump and Link Register with Shift) sowie JR (Jump Register)

wird der PC mit dem Registerinhalt des zweiten Operanden (der erste Operand bleibt ungenutzt) überschrieben. Bei einigen Befehlen wird der Nachfolge-PC-Wert im Register `R11` abgespeichert sowie die Basis des Registerfensters inkrementiert oder dekrementiert (Unterprogramm-Arbeit).

TRAP `TRAP` ist eine Art Software-Interrupt. Konkateniert man beide angegebenen Operanden, so ergibt sich eine Sprungadresse, welche auf den PC geschrieben wird. Der ursprüngliche Nachfolge-PC wird in `R11` abgelegt. Die Register-Window-Base wird inkrementiert.

RFE `RFE` ist ein Sprungbefehl, bei welchem das zweite Operandenregister (ersteres ist ungenutzt) auf den PC geschrieben wird. Gleichzeitig wird ein Signal an den Interrupt-Controller gesendet (Gegenwärtig ist diese Funktionalität in der Hardware noch nicht implementiert). Der Registerfenster-Offset wird dekrementiert. Der Befehl ist vergleichbar mit `JRS`.

Transportbefehle

L9, L18, S9, S18 Lade- und Speicherinstruktionen legen Daten im Hauptspeicher ab bzw. lesen sie aus. Der Zugriff erfolgt entweder wortweise, d.h. mit 18 Bit Breite, oder halbwortweise mit neun Bit.

MOV, MOVI Die beiden Move-Befehle überschreiben das im ersten Operanden definierte Register mit dem Wert des zweiten Operanden. Dieser kann sowohl ein Registerinhalt als auch ein Immediate-Wert sein.

MOVI2C, MOVC2I `MOVI2C` setzt das SF-Register `CC` auf den Wert des niederwertigsten Bits des angegebenen Operandenregisters. `MOVC2I` arbeitet genau umgekehrt; im Inhalt des angegebenen Registers wird die niederwertigste Stelle mit dem `CC`-Bit ersetzt.

MOVI2S, MOVS2I Um auf die Special-Function-Register direkt zugreifen zu können, werden `MOVI2S` sowie `MOVS2I` genutzt. Bei `MOVI2S` definiert der zweite Operand, wel-

ches Register überschrieben werden soll. Der entsprechende Wert wird dazu im ersten Universalregister-Operanden abgelegt. `MOVS2I` arbeitet invers und schreibt das bzw. die angesprochenen SF-Register in das General-Purpose-Register.

Steuerbefehle

`SBITS`, `CBITS` Vergleichbar zu den obigen Befehlen sind `SBITS` (Set Bits) und `CBITS` (Clear Bits). Im zweiten Operanden (der erste bleibt ungenutzt) wird definiert, welches SF-Register explizit zu setzen bzw. welches zu löschen ist.

2.6 Pipelining

Die Prozessor-Pipeline des Spartan MC besteht aus drei Stufen (siehe auch Abbildung 2.6):

1. Instruction Fetch and Decode (IF/ID): Die aktuelle Instruktion wird aus dem Speicher geladen und ihre Bestandteile (Opcode und Operanden) aufgeschlüsselt. Gegebenenfalls werden notwendige Universalregister geladen. Anhand des Opcodes werden sämtliche Steuerungsflags bestimmt. Bedingte Verzweigebefehle werden bereits hier ausgewertet und ausgeführt (d.h. der PC entsprechend modifiziert). Steuerungs-Hazards werden dadurch vermieden; es wird nie eine „unnütze“ Operation geladen. Falls ein Hardware-Interrupt anliegen sollte, und dieser auch freigeschaltet ist (`INT = 1`), wird statt dem eigentlichen Befehl die Interrupt-Abarbeitung gestartet.

2. Execute and Memory (EX/MEM): Abhängig vom Befehl wird entweder eine Berechnung in der ALU (bzw. der davon separaten Multiplikationseinheit) ausgeführt, oder es wird ein Lese- oder Speicherzugriff auf die SF-Registerbank oder den Haupt-, DMA- oder IO-Speicher ausgeführt.

3. Write Back (WB): Hier wird gegebenenfalls ein berechneter oder geladener Wert in ein GPR zurück geschrieben.

Die Pipelinestufen sind exakt gleich lang; sie benötigen jeweils genau einen CPU-Takt (dieser wird intern noch weiter geteilt). Stalls o.ä. aufgrund von Verzögerungen bei der Berechnung oder Speicherzugriffen sind *nie* notwendig.

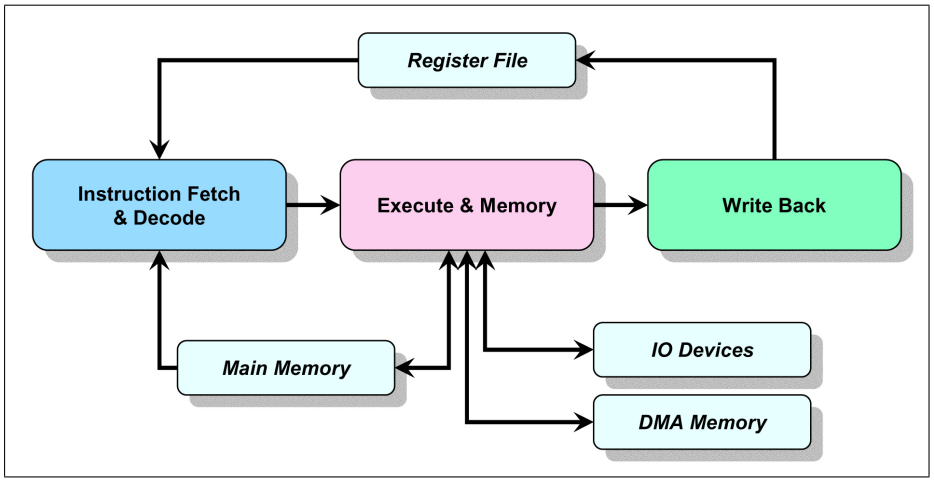

Abbildung 2.6: Schematischer Prozessoraufbau

Die bedingten Verzweigungen nutzen als Bedingung das `CC`-Bit im SFR, welches von EX/-MEM beschrieben wird. EX/MEM tut dies am Ende seiner Ausführung; die Überprüfung in IF/ID efolgt jedoch zu Beginn. Dies bedeutet, dass die Verzweigung gegebenenfalls nicht richtig ausgeführt wird (Daten-Hazard). Sollte eine solche Befehlskombination auftreten, muss aus diesem Grunde *vor* dem Verzweigebefehl ein NOP (z.B. `MOV R0, R0`) ausgeführt werden.

Wie bereits in Abschnitt 2.4.1 erwähnt, können auf den Hauptspeicher bis zu zwei Zugriffe gleichzeitig ausgeführt werden. Konflikte zwischen IF/ID und EX/MEM (Strukturelle Hazards) treten nicht auf. Zugriffe auf General-Purpose-Register und auf Special-Function-Register treten bei den Pipeline-Stufen stets zeitversetzt auf, sodass es auch da zu keiner Überschneidung kommen kann.

2.7 I/O und Peripherie

2.7.1 Interrupt Controller

Der Spartan MC besitzt einen einfachen Interrupt-Controller. Diesem werden, fest verdrahtet, sämtliche Interrupt-Leitungen zugeführt. Der Bitindex einer Leitung definiert gleichzeitig seine Priorität. Liegt eine Leitung an Position null, so ist seine Priorität gleich eins. Dies bedeutet also, bei n IRQ-Quellen liegen alle Prioritäten zwischen eins und n. Ein Interrupt mit größerem Prioritäts-Zahlenwert dominiert jeden mit niederem Wert, falls diese

im gleichen Takt anliegen.

Der Interrupt-Controller besitzt zwei IO-Register. Das erste zeigt die aktuell anliegenden IRQs. Jede Bitposition repräsentiert dabei eine Leitung, mit jeweils einer Eins, falls ein IRQ anliegt, und einer Null andernfalls. Das zweite IO-Register zeigt den binär codierten Prioritätswert des dominanten Interrupts.

Liegt mindestens ein Interrupt an, so meldet dies der Controller an den Prozessor, falls das Interrupt-Enable-Bit `INT` im SFR gesetzt ist. Dieses wird anschließend automatisch gelöscht, und die Interrupt-Abarbeitung beginnt. Dabei wird der PC auf die Interrupt-Sprungadresse gesetzt. Die Software kann nun gegebenenfalls die Register des IRQ-Controllers abfragen, um herauszufinden, welcher Interrupt genau anliegt.

2.7.2 UART

Die UART ermöglicht eine einfache Kommunikation über die serielle Schnittstelle. Sie besteht aus folgenden Komponenten:

- ein lesbares Status-Register für Statusinformationen
- ein schreibbares Control-Register für Steuerinformationen
- ein lesbares Rx-FIFO-Register, acht Bytes, als Empfangspuffer sowie ein Empfangs-Schieberegister, auf welches ankommenden Bits geschrieben werden
- ein schreibbares Tx-FIFO-Register, acht Bytes, als Sendepuffer sowie ein Sende-Schieberegister, auf welches zu übermittelnde Bits geschrieben werden
- eine Interruptleitung, gemeinsam genutzt vom Rx-IRQ und Tx-IRQ (mittels einer Status-Abfrage kann gegebenenfalls zwischen ihnen differenziert werden)

Für die beiden FIFO-Register wird jeweils ein Zeiger mitgeführt, der auf das aktuelle Byte verweist. Wird ein Datum empfangen, so wird es vom Rx-Schieberegister in den Rx-FIFO verschoben und der dazugehörige Zeiger entsprechend aktualisiert. Soll ein Datum gesendet werden, so wird das „älteste“ dem Tx-FIFO entnommen und dem Schiebregister zugeführt; der Tx-Zeiger wird aktualisiert.

Die UART unterstützt Baudraten bis zu 115.200 Bit/s. Es können wahlweise zwischen fünf und acht Datenbits versendet bzw. emfangen werden. Das Paritätsbit sowie ein oder zwei Stopbits können konfiguriert werden. Es gibt keine Flusskontrolle.

Werden Empfangs-Interrupts aktiviert, so wird ein IRQ ausgelöst, wenn der Rx-Puffer nicht leer ist. Durch einen Lesezugriff auf den FIFO wird das Signal wieder zurückgesetzt.

Der Sende-IRQ funktioniert etwas komplizierter. Wird auf den Tx-FIFO geschrieben, so wird ein Vorbereitungssignal gesetzt (ein gegebenenfalls schon anliegender IRQ wird gelöscht). Liegt im nächsten UART-Takt dieses Signal an *und* ist zudem der Tx-FIFO nicht voll, so wird ein IRQ veranlasst. Das Vorbereitungssignal wird gelöscht. Bei einem Lesezugriff auf das Status-Register werden beide Signale auf null gesetzt. Nach einem Reset liegt unmittelbar ein Vorbereitungssignal an, um kurz danach einen Interrupt auslösen zu können (nach einem Reset ist der Tx-Puffer natürlich geleert).

3 Entwurf und Implementierung des Backends

3.1 Anforderungen

Um die Portierbarkeit des Simulators zu ermöglichen, sollte die Anwendung in Java geschrieben werden. Dies offerierte die Vorteile, die vorgefertigten Werkzeuge und Funktionalitäten der SDK-Bibliotheken nutzen zu können. Die „Programmierfreundlichkeit“ von Java sowie insbesondere die dafür verfügbare leistungsstarke Workbench Eclipse unterstützten das Vorhaben, innerhalb von kurzer Zeit eine umfangreiche stabile Applikation zu entwickeln.

Der Quellcode war derart aufzubauen, zu gliedern und zu dokumentieren, dass es einfach möglich ist, den Simulator gegebenenfalls später einmal abändern bzw. erweitern zu können.

Bei der Implementierung stand der Quelltext des Mikrocontrollers in der Hardware Description Language (HDL) Verilog zur Verfügung. Prinzipiell boten sich zwei Strategien bei der Umsetzung des Simulators an: eine HDL-nahe oder eine hinreichend abstrahierte Realisierung. Bei der ersten Möglichkeit gilt es, den Referenzcode zu kopieren, natürlich entsprechend der Java-Syntax und Objektorientierung angepasst. Sämtliche Details müssen dabei übernommen werden, beispielsweise werden Steuerungssignale als Hilfsvariablen eins zu eins umgesetzt. Der große Vorteil dabei ist die Äquivalenz zum Original. Die Logik muss kein zweites Mal entworfen werden, sondern kann im Idealfall vom fehlerfreien Original ohne semantische Interpretation „blind“ übernommen werden. Genau daraus resultiert jedoch gleichermaßen der Nachteil, dass sämtlicher Implementierungs-Overhead, bedingt

durch die Anforderungen der Hardware-Umsetzung, mit übernommen wird sowie weitere Optimierungen durch die objektorientierte Hochsprache vermieden werden.

Für die hier vorgestellte Lösung ist der alternative Ansatz gewählt worden. Nur wesentliche Registerwerte sind aus der HDL-Umsetzung übernommen. Die Code-Struktur wurde an vielen Stellen aufgebrochen und vereinfacht, um sie den Möglichkeiten des objektorientierten Designs anzupassen. Der Schwerpunkt wurde auf Übersichtlichkeit und Erweiterbarkeit gelegt, weniger auf Performanz. Die ursprüngliche Hardware-Struktur ist fast nicht mehr nachvollziehbar. Insbesondere verwendet die simulierte Pipeline keine Steuersignale für die Abarbeitung, sondern nutzt direkt die entsprechenden Befehlsobjekte. Dem Java-Quellcode wurde an einigen Zeilen der dazugehörige HDL-Code (Verilog) als Kommentar zum besseren Verständnis beigefügt.

3.2 Allgemeine Entwurfsrichtlinien

Beim Gesamtentwurf ist darauf geachtet worden, die Funktionalitäten des Simulators in Backend und Frontend zu gliedern. Dabei kapselt das Frontend sämtliche grafischen Benutzerelemente und stellt Funktionalitäten zur Verfügung, um die Komponenten des Simulators bequem ansprechen zu können. Die eigentliche Modellierung der Hardware und die Ausführung der Simulation wird entsprechend durch das Backend übernommen.

Für die Bezeichnung der Klassen und Variablen sind Namen gewählt worden, die möglichst aussagekräftig sind. Klassen oder Variablen mit ähnlicher Funktionalität haben meist gleiche Wortstämme. Beispielsweise besitzen sämtliche GUI-Ansichtsklassen den Wortstamm `View`. Interfaces werden stets mit dem Postfix `_I` kenntlich gemacht.

Im Quellcode selbst sind zahlreiche Assertions verwendet. Dies hat mehrere Gründe. Zum einen wird die Fehlersuche bei der Entwicklung erheblich vereinfacht. Inkonsistente bzw. fehlerhafte Zustände können so kurz nach ihrem Auftreten abgefangen und damit lokalisiert werden. Es hat sich insbesondere bewährt, Registerwerte auf ein korrektes Format hin zu überprüfen (Bitbreite kleiner gleich 18). Zum anderen wird die Lesbarkeit des Codes erheblich verbessert. Assertions sind meist selbsterklärender als Kommentare. Wird das finale Programm mit einer virtuellen Maschine ohne Assertions-Schalter gestartet, so werden sämtliche dieser Anweisungen ignoriert. Es ergeben sich keine Laufzeitnachteile. Aus

diesen Gründen wurde sehr großzügig mit der Verwendung von Assertions umgegangen.

Bei der Implementierung ist eine große Zahl von Konstanten sowie Enumerationen verwendet worden. Werden diese von mehr als einer Klasse genutzt, so sind sie in separate Interfaces ausgelagert worden (z.B. `SpartanDefinitions_I`). Sollte eine Klasse die Konstanten bzw. Enumerationen benötigen, so implementiert sie einfach das dazugehörige Interface.

Für die Speicheradressierung existieren das Instruktionsformat und das Datenformat. Bei der Implementierung wird stets das Instruktionsformat als Standard benutzt. Im weiteren soll davon auch als sogenannte Default-Adressierung gesprochen werden.

Registerwerte sind durchweg als Integer implementiert, d.h. 32 Bit breit. Register, deren Inhalt ungültig ist (bzw. die als Fehler markiert werden sollen), werden mit `INVALID` belegt. Diese Konstante hat den Dezimalwert -1. Dies bedeutet also, dass sie nicht dem Format von 18 Bit genügt, da sie intern im Zweierkomplement abgelegt wird. Eine Formatprüfung schlägt dadurch fehl:

```
assert((someRegister & ~BITMASK_18) == 0);
```

Der gesamte Entwurf hat sich im Laufe seiner Entwicklung schrittweise herausgebildet. Viele Eigenschaften der jetzigen Umsetzung ergaben sich aus kontinuierlichem Refactoring („Agile Development“). Für die implementierten Lösungen ist stets ein Mittelweg zwischen Aufwand und Ertrag gewählt worden.

3.3 Strukturierung und Funktionalitätenverteilung

3.3.1 SimulationManager und Abarbeitung

Hauptklasse des Backends ist der `SimulationManager`. Er übernimmt die Verwaltung sämtlicher Simulationskomponenten sowie die Ausführung des Simulationsschrittes selbst. Es stehen die folgenden grundlegenden Funktionalitäten zur Verfügung:

- Durch Aufruf des Konstruktors werden die Kernkomponenten des Backends initialisiert. Ein Zeiger auf jede dieser Klassen kann angefordert werden.

- Die Simulation ist schritt-orientiert, d.h. es wird genau nur *ein* Schritt ausgeführt (`executeStep()`). Während dieses Schrittes, im folgenden auch als Takt bezeichnet, wird jede Pipeline-Stufe genau einmal berechnet sowie gegebenenfalls Peripherie und Interrupt-Controller angesprochen. Eine fortlaufende Simulation wird durch das Backend nicht unterstützt. In Kapitel 4 wird dargelegt, wie darauf das Frontend anknüpft, um eine kontinuierliche Simulation zu ermöglichen.

- Durch Aufruf der Methode `restartSimulation()` kann die vollständige Simulation neu gestartet werden. Dies bedeutet, dass die Pipeline geleert wird und die Ausführungsposition wieder an den Ausgangspunkt (initialen PC) gesetzt wird. Desweiteren werden alle Komponenten angewiesen, sich ebenfalls zurückzusetzen. Registerwerte werden dabei gelöscht. Einzig der Inhalt des Hauptspeichers und des DMA-Speichers bleiben erhalten. Der Zustand nach einem Neustart ist identisch mit dem Zustand unmittelbar nach Laden einer Assembler-Datei.

- Mittels `setupSimulation()` kann ein Setup erfolgen. Dies bedeutet, dass elementare Simulationsparameter geändert worden sein könnten. Alle Komponenten werden aufgefordert, gleichermaßen ein Setup auszuführen. Gegebenenfalls müssen sie verifizieren, wie die gegenwärtige Konfiguration aussieht; sie haben dafür Sorge zu tragen, in einen konsistenten Zustand überzugehen. Unter Umständen müssen sie ihren Zustand eigenständig verändern, um einen Konflikt aufzulösen (bspw. bei einer Veränderung der Adressraumaufteilung). Per Definition zieht jedes Setup einen Neustart nach sich.

Die Abarbeitung eines Simulationsschrittes läuft folgendermaßen ab:

1. Die Access-Manager des Haupt- und DMA-Speichers, der Register-File und die SFR werden zurückgesetzt (mehr dazu später).

2. Die Interrupts werden berechnet. Der Interrupt-Controller wird aufgefordert, die gegebenenfalls anliegenden Hardware-Interrupts zu überprüfen. Bestätigt der Controller einen anliegenden Interrupt und ist gleichzeitig das Interrupt-Enable-Bit in den SFR gesetzt, so wird diese Interrupt-Anforderung vorgemerkt. Das Enable-Bit wird in diesem Falle zudem auf null gesetzt.

3. Anschließend werden die drei Pipeline-Stufen nacheinander ausgeführt. Der IF/ID-

Stufe wird dabei mitgeteilt, inwiefern ein Hardware-Interrupt anliegt. Sollte bei der Abarbeitung der Pipeline-Stufen eine Fehlersituation auftreten (z.B. ein ungültiger Befehl), so wird eine `SpartanException` geworfen.

4. Danach wird die `onClock()`-Methode des IO-Managements aufgerufen, welcher wiederum seine IO-Geräte informiert. Die IO-Geräte ihrerseits können ebenso eine Exception werfen.

5. Zuletzt wird überprüft, ob Break- oder Watchpoints berührt worden sind. Ist dies der Fall, wird eine `SimulationBreakException` geworfen. Diese beihnaltet Informationen zu sämtlichen betroffenen Haltepunkten.

Die `SpartanException` findet an vielen Stellen im Backend Verwendung. Sie wird dann geworfen, wenn eine fehlerhafte oder inkonsistente Situation auftritt. Das heißt insbesondere auch, dass sie geworfen wird, wenn die Ursache nicht beabsichtigt sein kann (z.B. Schreibzugriff auf ein Read-Only-Register). Die Originalhardware muss nicht zwingend auch einen Fehler erkennen. Nach einer `SpartanException` ist die Simulation abzubrechen und darf erst wieder nach einem Neustart fortgeführt werden.

3.3.2 Zentrale Komponenten

Der Simulator besteht aus den Kernkomponenten (siehe Grafik 3.1):

- der Hauptspeicher (`MainMemory`),
- der DMA-Speicher(`DmaMemory`),
- die General-Purpose-Register (`RegisterFile`),
- die Special-Function-Register (`SfRegisters`),
- die Pipeline-Stufen samt Schnittstellen-Klasse (`IfIdStage`, `ExMemStage`, `WbStage`, `PipelineUI`),
- der Interrupt-Controller (`InterruptController`),
- der IO-Manager (`IoManager`)
- sowie die Statistiken (`Statistics`).

Sämtliche hierbei simulierten Hardware-Komponenten werden von der abstrakten Superklasse `AbstractHardwareComponent` abgeleitet. Aufgrund der Trennung zwischen Backend und der Ansichtsdarstellung ist es notwendig, einen Mechanismus zur Aktualisierung

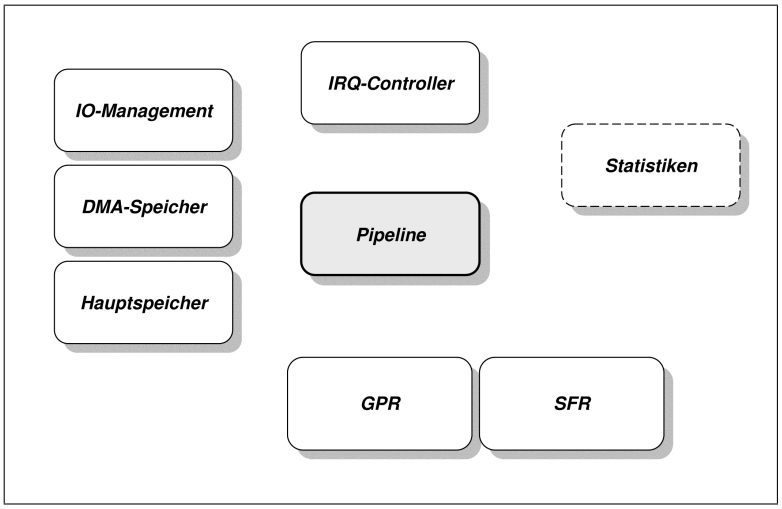

Abbildung 3.1: Kernkomponenten des Backends

einzuführen. Anzeigeelemente können sich als sogenannter `RefreshListener_I` bei einer `AbstractHardwareComponent` registrieren. Sollte sich der interne Zustand der Backend-Klasse verändern, so löst diese ein Event aus. Die dazugehörigen Listener sind in einer Liste abgespeichert und werden nacheinander über die Änderung(en) informiert. Fakultativ kann als Zusatzinformation eine Adressangabe mitgegeben werden, um die notwendige Aktualisierung einzuschränken. Im Kapitel zum Frontend wird näher darauf eingegangen.

3.3.3 Settings

Globale Simulationsparameter werden in der Klasse `Settings` statisch abgespeichert. Wie bereits erwähnt, ist nach Änderung mindestens einer dieser Parameter ein Simulations-Setup notwendig. Es stehen die folgenden Konfigurationswerte zur Verfügung:

- `SINGLE_SHIFT`: Gibt an, ob Shift-Befehle, wie z.B. SLL, ihren Operanden um eine Stelle oder um mehrere Stellen verschieben.
- `SIMULATION_SPEED`: Definiert den Geschwindigkeitsmodus der Abarbeitung.
- `PC_INITIAL`: Definiert den initialen PC nach einem Simulationsneustart.
- `INTERRUPT_JUMP_ADDRESS`: Definiert die Sprungadresse beim Auftreten eines Hardware-Interrupts.

- `MEMORY_ADDRESS_MIN` und `MEMORY_ADRESS_MAX`: Definieren die Anfangs- und die Endadresse des Hauptspeichers. Die Anfangsadresse kann durch den Benutzer nicht verändert werden, sie bleibt fest auf `0x00000`. Die GUI stellt eine Einstellungsmaske zur Verfügung, die es nur gestattet, ganzzahlige Kilo-Blöcke auszuwählen, d.h. die Adressbereiche des Hauptpeichers, des DMA-Speichers sowie des IO-Bereiches sind stets Vielfache von 1024. Der Hauptspeicher muss mindestens 1 K groß sein.

- `DMA_ADDRESS_MIN` und `DMA_ADDRESS_MAX`: Definieren die Anfangs- und die Endadresse des DMA-Speichers.

- `USING_DMA`: Gibt an, ob die Adressraumaufteilung einen DMA-Bereich enthält. Ist der Wert gleich `false`, so ist die spezifizierte DMA-Anfangsadresse und die DMA-Endadresse als ungültig anzusehen.

- `IO_ADDRESS_MIN` und `IO_ADDRESS_MAX`: Definieren die Anfangs- und die Endadresse des IO-Adressraums. Es sei betont, dass dies nicht bedeutet, dass die Adressen innerhalb dieses Intervalles automatisch nutzbar sind. Sie müssen erst von einem IO-Gerät okkupiert und registriert werden. Der Adressbereich muss mindestens 1 K groß sein.

- `INTERRUPT_LIST`: Definiert eine Liste mit allen registrierten interruptfähigen Geräten. Näheres dazu im Abschnitt zum Interrupt-Controller.

- `INTERRUPT_IO_BASE_ADDRESS`: Definiert die Basisadresse der zwei IO-Register des Interrupt-Controllers.

- `REFRESH_ENABLE`: Gibt an, ob die Listener der `AbstractHardwareComponents` über eine Änderung informiert werden sollen.

- `REGISTER_FONT_PLAIN` und `REGISTER_FONT_BOLD`: Definieren die Standardschrift.

Aus Gründen der Zweckmäßigkeit speichert `Settings` auch Konfigurationsparameter ab, die semantisch dem Frontend zugeordnet werden. Desweiteren können Parameter geladen und gespeichert werden. Näheres dazu im Kapitel 4.

3.3.4 AccessManager

Der Haupt- und DMA-Speicher, die Register-File und die SFR bieten die Möglichkeit, Watchpoints zu setzen. Wird auf eine Variable, d.h. einen Registerwert, auf welche ein Watchpoint gesetzt worden ist, lesend oder schreibend zugegriffen, so soll dies „bemerkt“ werden. Analog dazu sollen zusätzlich im Hauptspeicher Breakpoints gesetzt werden können. Wird eine Instruktion (kein Datenwert) von einer Hauptspeicheradresse, die mit einem Breakpoint verbunden ist, durch IF/ID geladen, so soll auch dies kenntlich gemacht werden.

Diese Funktionalitäten werden in die Klasse `AccessManager` ausgelagert. Die obigen Komponenten besitzen jeweils ihre eigene private `AccessManager`-Instanz. Der Manager verwaltet eine Liste von Breakpoints sowie von Watchpoints. Vor jedem Simulationsschritt muss der Manager initialisiert (vorbereitet) werden. Registerzugriffe auf die dazugehörige Komponente erfolgen stets mit einem Access-Type-Parameter: `TRANSPARENT`, `BREAKPOINT` oder `WATCHPOINT`. Abhängig davon informiert diese Komponente dabei ihren `AccessManager`, dass der Lese- oder Schreibzugriff stattgefunden hat. Sollte auf der Adresse ein Haltepunkt definiert worden sein, so bemerkt dies der Manager und speichert den Treffer ab. Am Ende des Simulationsschrittes wird jeder Manager abgefragt und gibt seine Trefferlisten zurück. Durch den Initialisierungsvorgang beim nächsten Durchlauf werden eben diese Listen wieder gelöscht. Es kann in einem Takt mehrere berührte Watchpoints geben, hingegen nur maximal einen Breakpoint, da bei einem Ausführungsschritt (höchstens) eine Befehlsadresse geladen wird. Breakpoints werden ausschließlich für den Hauptspeicher verwendet.

Der `AccessManager` bleibt für außenstehende Klassen unsichtbar. Die Kommunikation (z.B. die Aufforderung zum Setzen eines Watchpoints) hat stets mit der eigentlichen Komponente zu erfolgen.

3.3.5 Statistics

Um Aussagen über die Instruktionsnutzung bzw. -verteilung machen zu können, werden sämtliche Instruktionen, die von der IF/ID-Pipeline-Stufe geladen werden, protokolliert. Dies übernimmt die Klasse `Statistics`.

Bei jedem Schritt wird ihr der neue Befehlstyp mitgeteilt. Die Statistik kann zu jedem Zeitpunkt gelöscht werden. Intern wird ein Array mit der Anzahl der Instruktionstypen sowie ein zweidimensionales Array mit der Anzahl von Folgeinstruktionen genutzt. Das bedeutet, dass es auch möglich ist, nach Vorgänger-Nachfolger-Paaren abzufragen. Dies kann für Analysen des Befehlssatzes genutzt werden.

3.4 Speicher

3.4.1 Hauptspeicher und DMA-Speicher

Der Hauptspeicher wird durch die Klasse `MainMemory`, der DMA-Speicher durch die Klasse `DmaMemory` modelliert. Beide besitzen jeweils ein Array, in dem sie ihre Datenwerte abspeichern. Zugriffe erfolgen über `getValueAt()` bzw. `setValueAt()`. Wie bereits angesprochen kommt dabei auch ein `AccessManager` zum Einsatz.

Beide Klassen sind im Stande, ihre Größe anzupassen, sobald die Adressraumaufteilung (in `Settings`) geändert wurde. Dabei werden, soweit möglich, alte Speicherdaten erhalten. Entscheidet sich der Benutzer, auf den DMA-Speicher zu verzichten, so bleibt die `DmaMemory`-Instanz natürlich erhalten, liefert aber bei einer Anfrage `containsAddress()` für jeden Adresswert stets **`false`** zurück.

Der Hauptspeicher protokolliert, an welchen Speicheradressen sich die drei Pipeline-Stufen jeweils befinden. Auf den Hauptspeicher kann sowohl von der IF/ID-Stufe als auch von der EX/MEM-Stufe zugegriffen werden. Der DMA-Speicher kann von beliebigen DMA-Geräten genutzt werden. Die EX/MEM-Stufe kann gleichermaßen Worte als auch Halbworte aus den DMA-Bereich lesen, darf hingegen nur ganze Worte schreiben. Instruktionen können aus dem DMA-Speicher nicht gelesen werden. Siehe dazu auch Abschnitt 2.4.1.

3.4.2 Instruktionen

Der Hauptspeicher bietet die Möglichkeit, statt eines Datenwertes eine Instruktion zu laden. Dabei wird versucht, das Datum an der betreffenden Adresse als Instruktion zu interpretieren. Diese wird, falls erfolgreich, zurückgeliefert.

CPU-Befehle werden durch die Klasse `Instruction` dargestellt. Eine Instruktion ist definiert durch das dazugehörige Instruktionswort, einen Instruktionstyp (z.B. `ADD`), bis zu drei Operanden sowie durch eine mnemonische String-Beschreibung.

Instruktionen werden mittels einer statischen Methode erzeugt, welcher das Instruktionswort übergeben wird. Sollte anhand dieser Bitfolge eine Interpretation erfolgreich sein, so wird eine `Instruction`-Instanz zurückgegeben, andernfalls **`null`**. Zu Debug-Zwecken ist es auch möglich, eine Instruktion anhand ihres Typs und ihrer Operanden zu instanziieren.

Es besteht die Möglichkeit, Daten bzw. Instruktionen aus einer Datei in den Hauptspeicher zu laden bzw. umgekehrt vom Speicher in eine Datei zu schreiben. Die Daten in der Datei müssen im `sph`-Format vorliegen. Das heißt, jedes Datum liegt als Textzeichen hexadezimal formatiert jeweils in einer Zeile vor. Es kommt die Klasse `AssemblyParser` zum Einsatz. Diese liest die Quelldatei aus und schreibt sie fortlaufend in den Hauptspeicher. Sollte dabei ein Fehler auftreten, wird eine `ParsingException` geworfen. Gleichermaßen kann der Parser den Hauptspeicherinhalt in eine Zieldatei im gleichen Format zurückschreiben. Sollte hierbei ein Fehler auftreten oder der Hauptspeicher gänzlich leer sein, wird ebenso eine entsprechende Exception geworfen.

3.5 Register

3.5.1 Registerfile

Die General-Purpose-Register werden in der `RegisterFile` gespeichert. Dabei werden wieder ein Datenarray und ein `AccessManger` genutzt. Die Hardware des Originalprozessors arbeitet mit einem Register-Window. Dies wird auch hier übernommen. Es wird zwischen einer lokalen und einer globalen Registeradressierung bzw. -nummerierung differenziert. Erstere ist im Kontext des Registerfensters zu interpretieren, d.h. die Nummerierung bleibt im Intervall 0..15. Die lokale Registernummer wird abhängig vom Registerfenster auf eine globale Adresse abgebildet. Die Register-Window-Basis wird innerhalb der `SfRegisters`-Klasse gespeichert.

3.5.2 SFR

Die Special-Function-Register werden in der Klasse `SfRegisters` implementiert. Ein `AccessManager` steht zur Verfügung, um Watchpoints setzen zu können. Sollte die Register-Window-Basis während der Simulationsabarbeitung durch Inkrementieren oder Dekrementieren außerhalb ihres zulässigen Wertebereiches gelangen, so wird unmittelbar eine `SpartanException` geworfen.

3.6 Prozessor-Pipeline

3.6.1 Pipeline-Stufen

Die Berechnungen der Pipeline werden in drei unterschiedlichen Klassen ausgeführt, welche jeweils eine Pipeline-Stufe simulieren: `IfIdStage`, `ExMemStage` und `WbStage`. Es wurde versucht, die Komplexität der Original-Hardware-Pipeline zu reduzieren, indem die wichtigsten Register einer jeden Stufe ausgewählt wurden, und sich ausschließlich diese in der Simulation wiederfinden. Auf nahezu alle Kontroll- bzw. Steuersignale wurde gänzlich verzichtet.

Vorrang gegenüber dem technischen Detailgrad hat die Übersichtlichkeit für den Anwender. Registerwerte, die für die konkret bearbeitete Instruktion nicht benötigt werden, erhalten stets den Wert `INVALID` (Das daran ansetzende Frontend wird diese ungültigen Register erst gar nicht anzeigen).

IF/ID-Stufe	
`pc`	die Adresse des aktuell geladenen Befehls
`pc_new`	die Adresse des Nachfolgebefehls
`ir`	das aktuell geladene Instruktionswort
`offset`	Offset-Operand
`imm`	Immediate-Operand
`imm_high`	Immediate-Operand, links-vershiftet um neun
`disp`	Displacement-Operand
`reg_no_a`	erste GP-Register-Nummer
`reg_no_b`	zweite GP-Register-Nummer
`reg_a`	erster GP-Register-Inhalt
`reg_b`	zweiter GP-Register-Inhalt
`alu_opa`	erster Operand (meist für nachfolgende ALU)
`alu_opb`	zweiter Operand (meist für nachfolgende ALU)

EX/MEM-Stufe	
`alu_op_a`	erster ALU-Operand
`alu_op_b`	zweiter ALU-Operand
`alu_result`	Ergebnis der ALU
`reg_no_a`	erste GP-Register-Nummer
`reg_no_b`	zweite GP-Register-Nummer
`reg_a`	erster GP-Register-Inhalt
`sfr_address`	Adresse zur Auswahl eines SFR
`data_addr`	Adresse zur Auswahl von Speicher / IO
`write_data_data`	Register-Wert bei Schreibzugriff auf Speicher / IO
`use_bypass_opA`	1, wenn Bypass für `alu_op_a` aktiv
`use_bypass_opB`	1, wenn Bypass für `alu_op_b` aktiv
`use_bypass_regA`	1, wenn Bypass für `reg_a` aktiv (für `S9` und `S18`)

WB-Stufe	
`write_reg_number`	zurückzuschreibende GP-Register-Nummer
`write_reg_data`	zurückzuschreibender GP-Register-Inhalt
`do_wb`	1, wenn GP-Register zurückgeschrieben werden soll

Es sei angemerkt, dass die Befehle `MOV` sowie `MOVI` keinen `alu_op_a` für die ALU benötigen, trotzdem ist bei ihrer Abarbeitung in der Original-Hardware der dazugehörige Bypass freigeschaltet. Dies hat jedoch keinerlei Auswirkungen auf die Abarbeitung. Aus diesem Grunde wird der Bypass in der Simulation ignoriert; d.h. die EX/MEM-Stufe zeigt bei beiden Befehlen niemals `use_bypass_opA` an, obgleich die Bedingungen vorliegen könnten. Das gleiche gilt analog für `alu_op_b` bei den Befehlen `MOVI2S`, `MOVS2I`, `SBITS`, `CBITS` sowie `INTERRUPT`.

Die Bezeichnungen sind den HDL-Quellen entnommen. Für weitere Details sei darauf verwiesen.

Die Registerwerte der Pipeline-Stufen sind jeweils package-scoped, d.h. die Klassen können untereinander darauf zugreifen. Nach außen hin sind sie jedoch nicht sichtbar. Um dem Frontend eine sinnvolle Schnittstelle zur Verfügung zu stellen, wurde die Klasse `PipelineUI` implementiert. Sie ist von `AbstractHardwareComponent` abgeleitet. Bei ihr können sich `RefreshListener_I` für die Pipeline-Stufen registrieren; bei jedem Simulationsschritt wird ein Aktualisierungs-Event gefeuert. Die Klasse offeriert Funktionen, um *lesend* auf die Pipeline-Register zuzugreifen und einen neuen PC-Wert zu setzen.

3.6.2 Sequenzialisierung

Hauptproblem bei der Umsetzung war die Tatsache, dass die Pipeline-Stufen parallel arbeiten und sowohl untereinander als auch mit externen Komponenten in Beziehung stehen. Es musste eine Sequenzialisierung erarbeitet werden, die exakt das gleiche Verhalten aufzeigte.

Es wurde die folgende Lösung implementiert:

```
wb.executeStep();                                    // may throw SpartanException

final int old_cc = sfr.getCC(AccessType.TRANSPARENT);
exMem.executeStep(wb.getOverwrittenRegNo(),
                  wb.getOverwrittenRegValue());      // may throw SpartanException

ifId.executeStep(old_cc, doInterrupt, statistics);   // may throw SpartanException
```

Der Code ist im `SimulationManager` wiederzufinden.

Es können folgende Bedingungen, gemäß Hardware-Vorlage, gewährleistet werden:

- Jede Pipeline-Stufe kann auf die notwendigen Registerwerte bzw. Signale der Vorgängerstufe zugreifen, d.h. EX/MEM auf IF/ID und WB auf EX/MEM. Dabei wird garantiert, dass es sich bei diesen Werten um die *alten* Belegungen handelt, die einen Ausführungsschritt zuvor berechnet worden sind. In der Hardware wird diese Eigenschaft durch die separierenden Zwischenregister angrenzender Stufen umgesetzt.

- Die WB-Stufe schreibt, falls erforderlich, dass Ergebnis zu Beginn der Ausführung in die Registerfile. Genauso wie in der Hardware-Vorlage kann die (quasi-parallele) IF/ID-Stufe, die gegen Ende Ihrer Ausführungsperiode die Registerfile wieder ausliest, bereits auf die *aktuellen* zurückgeschriebenen Werte zugreifen.

- Die IF/ID-Stufe benötigt für die Entscheidung über bedingte Sprünge den Wert des `CC`-Registers, der prinzipiell von EX/MEM überschrieben werden könnte. Dabei greift sie jedoch auf den *alten* Wert zurück, da in Hardware der Zugriff zu Beginn der IF/ID-Periode erfolgt, eine etwaige Aktualisierung durch EX/MEM jedoch erst zum Ende der Periode.

- Sollte EX/MEM Operanden aus Registern benötigen, die durch WB überschrieben werden sollen, so können diese über einen Bypass geholt werden.

3.7 Interrupt-Controller

Sämtliche Komponenten, die einen Interrupt anfordern können, müssen die Schnittstelle `InterruptSource_I` implementieren. Dieses Interface verlangt 1) eine Methode zur Berechnung einer Interrupt-Forderung, 2) zur Anfrage nach einer Forderung sowie 3) zur Bereitstellung eines simulationsweit eindeutigen Interrupt-Namens. Die Komponente kann sich beim Interrupt-Controller registrieren. Dieser verwaltet eine Liste mit sämtlichen potentiellen Interrupt-Quellen.

Der Interrupt-Controller besitzt zwei IO-Register: Das erste IO-Register zeigt alle anliegenden Interrupts (eine Bitposition pro Interrupt), der zweite zeigt den Prioritätenwert des dominierenden Interrupts an (d.h. also 101, falls als höchstpriorer IRQ die Nummer fünf anliegt). Auf beide Register darf nur lesend zugegriffen werden.

Bei jedem Simulationsschritt wird *vor* Ausführung der Pipeline eine Interrupt-Anfrage an den Controller gestellt. Dieser fordert alle registrierten Quellen auf, ihren Interrupt neu zu berechnen. Anschließend werden alle wiederum befragt, ob ein Interrupt anliegt. Falls ja, wird dies notiert. Entsprechend werden die beiden IO-Register angepasst.

Beim Anmelden einer neuen Interrupt-Quelle wird ein neues `Interrupt`-Objekt erzeugt und der internen Liste hinzugefügt. Ein `Interrupt` ist definiert durch seine Priorität, seinen Namen (d.h. den String-Bezeichner, der durch die Interrupt-Quelle bei Anfrage zurückgeliefert wird) sowie einen Zeiger auf die Interrupt-Quelle selbst. Jeder neu hinzugefügte Interrupt enthält die Anfangspriorität eins, d.h. die niederste Priorität.

Es besteht die Möglichkeit, die Priorität jederzeit zu inkrementieren oder zu dekrementieren. Dabei müssen sämtliche anderen Prioritäten neu berechnet werden, um Inkonsistenz zu vermeiden. Bei Entfernen einer Interrupt-Quelle werden die Prioritäten abermals neu berechnet (alle niederen Interrupts rücken auf).

Ein Problem bei der Interrupt-Verwaltung war die Wiederherstellung nach einem Programm-Neustart. Dieses Thema setzt an das Präferenzen-Management des Frontends an. Die Interrupt-Liste sowie die IO-Basis-Adresse des Controllers werden in der `Settings`-Klasse gespeichert. Bei einem Programm-Neustart wird die IRQ-Liste korrekt wiederhergestellt. Jedoch werden sich nachfolgend `Interrupt`-Instanzen erneut anmelden, obgleich diese bereits in der Liste vorhanden sind (inklusive der vorher eingestellten Prioritätenordnung). Die `Settings`-Klasse hat beim Wiederherstellungsvorgang aus diesem Grunde bei den `Interrupts` die Zeiger auf die Quellobjekte mit **`null`** ersetzt. Sobald sich nun Quellen registrieren, wird überprüft, inwiefern der zu registrierende Interrupt-Name bereits existiert. Wenn bereits ein Eintrag vorhanden ist, muss dies bedeuten, dass der dazugehörige Verweis auf die Quelle **`null`** ist, d.h. es handelt sich um den Platzhalter für die sich jetzt registrierende Komponente.

Wenn sich in der IF/ID-Stufe ein `RFE`-Befehl befindet, so wird der Controller mit einem Methodenaufruf informiert. Der Methodenrumpf ist nicht implementiert, er steht für spätere Erweiterungen zur Verfügung.

3.8 IO-Geräte

Der Vorteil des Spartan MC, Konfigurationen angepasst an das jeweilige Einsatzgebiet zu erstellen, schließt insbesondere auch das Hinzufügen bzw. Entfernen von IO-Komponenten mit ein. Alle bisher besprochenen Simulationselemente sind statisch, d.h. fest im Simulationsmanager verankert. Es kann keine Systemkonfiguration geben, die nicht alle Kernkomponenten umfasst.

Zusätzlich dazu gibt es Geräte (im weitesten Sinne), die dynamisch einer Simulationseinstellung hinzugefügt oder entfernt werden können.

3.8.1 IO-Manager

Sämtliche peripheren Komponenten werden durch den `IoManager` verwaltet. IO-Geräte können sich bei diesem an- bzw. abmelden. Dabei können sie fakultativ eine IO-Adresse angeben. Die Adresse muss innerhalb des durch die Simulationsparameter festgelegten IO-Adressraums liegen. Ein Gerät kann sich mit beliebig vielen Adressen anmelden.

Anmeldungen für eine Adresse dürfen je IO-Gerät nur einmal erfolgen. Zur Vereinfachung der Implementierung dürfen Abmeldungen jedoch beliebig oft geschehen, nachfolgende Aufrufe bleiben schlichtweg ohne Wirkung. Ebenso ist es nicht „schädlich“, sich abzumelden, ohne vorher registriert worden zu sein.

Um auf eine IO-Adresse lesend oder schreibend zuzugreifen, wendet sich der Prozessor an den IO-Manager. Sollte sich für die angegebene Adresse kein Gerät registriert haben, so wird eine `SpartanException` geworfen. Bei einem Schreibzugriff (`setValueAt()`) wird der zu schreibende Wert an *alle* IO-Komponenten weitergeleitet, die sich für die Adresse registriert haben. Bei Lesezugriffen wird das durch den IO-Manager zurückgegebene Ergebnis aus einer OR-Verknüpfung aller Teilergebnisse der an diese Adresse gebundenen IO-Komponenten bestimmt.

Bei jedem Simulationsschritt wird *nach* Abarbeitung der Pipeline der `IoManager` mit der Funktion `onClock()` aufgerufen. Dieser wiederum leitet den Aufruf an alle registrierten IO-Geräte weiter. Dadurch sind diese im Stande, taktabhängige Operationen bzw. Berechnungen auszuführen. Es sei angemerkt, dass jedes Gerät genau einmal aufgeru-

fen wird, unabhängig davon, mit wievielen IO-Adressen es sich registriert hat. Bei einem Simulations-Neustart fordert der Manager alle IO-Komponenten analog zum Neustart auf. Bei einem Simulations-Setup werden wiederum alle Geräte informiert; gleichzeitig werden jedoch die internen Listen gelöscht und damit alle Registrierungen unwiderruflich aufgelöst. Bei einem Setup ist das IO-Gerät angewiesen sich neu zu registrieren.

Eine Sonderrolle bei den IO-Geräten nimmt der `InterruptController` ein. Er ist nicht dynamisch einsetzbar, jedoch wird er auch über IO-Adressen angesprochen. Aus diesem Grund besitzt der Manager einen Zeiger auf die gültige Instanz und überprüft bei jedem IO-Lese- oder -Schreibzugriff ebenso, ob gegebenenfalls die Adressen des Controllers angesprochen werden.

3.8.2 AbstractPeriphery

IO-Komponenten werden stets von der abstrakten Superklasse `AbstractPeriphery` abgeleitet. Diese Superklasse ist ihrerseits eine `AbstractHardwareComponent`, damit sich gegebenenfalls Frontend-Ansichten registrieren lassen können.

Die `AbstractPeriphery` gibt die Schnittstellenfunktionen vor, die vom `IoManager` genutzt werden (Abbildung 3.2). Dies ist zum einen eine Methode zum Neustart sowie zum anderen eine Setup-Methode. Innerhalb der Setup-Methode muss sich das Gerät beim Manager registrieren, sowie auf Inkonsistenzen bei den IO-Adressen prüfen. Jedes Setup führt anschließend einen Neustart aus.

IO-Geräte werden durch den Benutzer mit Hilfe des Frontends hinzugefügt und auch entfernt. Beim Entfernen wird die Funktion `tearDown()` aufgerufen. Diese muss sämtliche „Aufräumarbeiten“ erledigen und sich beim Manager abmelden. Nach diesem Aufruf darf das IO-Objekt nicht mehr benutzt werden.

Es ist bereits angesprochen worden, dass die IO-Komponente in jedem CPU-Takt aufgerufen wird. Dafür steht die Methode `onClock()` zur Verfügung. Der Aufruf erfolgt gegen Ende des Simulationsschrittes, d.h. die Pipeline (sowie der Interrupt-Controller) ist zuvor vollständig abgearbeitet worden. Sollte ein IO-Gerät simuliert werden, das langsamer arbeitet als der Prozessor, so kann innerhalb von `onClock()` mit Zählern gearbeitet werden.

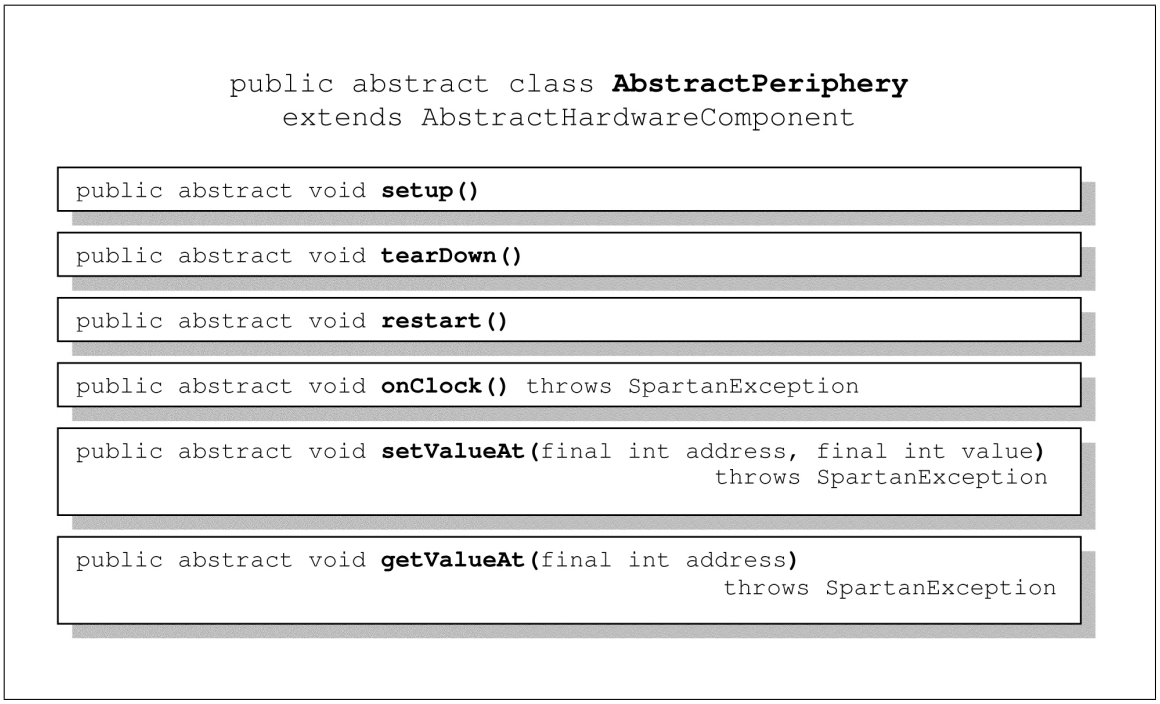

Abbildung 3.2: Klassenübersicht `AbstractPeriphery`

Es stehen ferner die beiden Methoden `getValueAt()` und `setValueAt()` für Lese- bzw. Schreibzugriffe zur Verfügung. Sie werden vom Manager aufgerufen, wenn eine der registrierten IO-Adressen zutreffend ist. Es wird nicht vorgeschrieben, was in diesen Methoden zu tun ist, d.h. es muss sich nicht zwingend nur um Datenoperationen handeln, sondern es können beliebige Seiteneffekte auftreten. Beide Methoden können darüber hinaus eine `SpartanException` werfen, beispielsweise, wenn auf ein Read-Only-Register schreibend zugegriffen werden soll.

3.8.3 UART

Die Klasse `Uart` simuliert eine serielle Schnittstelle. Sie besitzt vier IO-Register, für die sie sich entsprechend beim `IoManager` registriert: das Status-Register, das Control-Register, das Empfangs-FIFO und das Sende-FIFO. Zusätzlich wird auch das Empfangs- und das Sende-Shift-Register simuliert. Um IRQs zu ermöglichen, registriert sich die UART zudem beim Interrupt-Controller.

Um die Ein- und Ausgabe von Bytes zu simulieren, wird eine weitere Klasse benötigt. Diese muss das Interface `UartStream_I` implementieren. Soll ein Byte verschickt wer-

den, so wird `onTransmitNextByte()` aufgerufen. Um herauszufinden, ob und gegebenenfalls welches Datum empfangen wird, werden die Funktionen `hasNextByte()` sowie `onReceiveNextByte()` genutzt. Das Frontend wird daran mit schlichten Textkomponenten anknüpfen.

Um eine Baudrate nachzubilden, wird ein Zähler verwendet, *unabhängig* von dem im Control-Register stehenden Baudwert. Es soll keine Aussage über den Prozessortakt gemacht werden. Da sich IO-Takte nur aus dem CPU-Takt – gegebenenfalls mit einem Vorteiler – gewinnen lassen, ist es nicht möglich und auch nicht sinnvoll, eine „authentische“ Baudrate zu erzeugen.

Bei jedem Taktaufruf wird der folgende Ablauf ausgeführt:

1. Ist die UART aktiviert, d.h. dass im Control-Register mindestens das Read-Enable- oder das Write-Enable-Bit gesetzt ist, so wird der Baudraten-Zähler inkrementiert.

2. Ist das Read-Enable-Bit gesetzt, so wird, falls im Rx-Shift-Register ein Wert steht, dieser in das dazugehörige FIFO geschoben. Dieser Vorgang ist unabhängig von der simulierten Baudrate.

3. Wenn das Read-Enable-Bit gesetzt ist und der Baudraten-Zähler abgelaufen ist, wird überprüft, inwiefern das Stream-Objekt ein neues Datenbyte gesendet hat. Wenn dem so ist, wird es in das Rx-Schieberegister übernommen. Ist die UART mit einer Wortlänge kleiner acht Bit eingestellt, wird das empfangene Byte entsprechend gekürzt.

4. Wenn das Write-Enable-Bit gesetzt ist und der Baudraten-Zähler abgelaufen ist, wird überprüft, ob im Tx-Shift-Register ein Datum zum Senden ansteht. Gegebenenfalls wird dieses an das Stream-Objekt übermittelt und das Schieberegister invalidiert. Falls im Tx-FIFO weitere Daten zum Senden anstehen, wird ein Byte entfernt und dem Schieberegister zugeführt. Erst beim nächsten Ablauf des Baudraten-Zählers kann dieses wiederum übermittelt werden.

5. Falls notwendig wird zum Schluss ein Aktualisierungs-Event getriggert.

Schreibzugriffe auf das Status-Register sowie auf das FIFO-Empfangs-Register sind unzulässig (es wird eine Exception geworfen). Gleichermaßen sind Lesezugriffe auf das Control-Register und das Tx-FIFO untersagt.

Die beiden FIFO-Register werden intern durch Arrays implementiert, es wird jeweils ein Indexzeiger mitgeführt. Bei jeder Veränderung werden gegebenenfalls Statusflags aktualisiert.

Die UART kann einen Hardware-Interrupt anfordern, wenn entweder der Empfangs-FIFO nicht leer ist oder der Sende-FIFO nicht voll ist. Intern werden zwei Interrupt-Signal-Variablen verwendet:

- Das Rx-Interrupt-Signal ist gesetzt, sobald das Rx-FIFO nicht mehr leer ist. Es wird bei jedem Rx-FIFO-Lesezugriff zurückgesetzt.

- Das Tx-Interrupt-Signal wird gesetzt, sobald das Tx-FIFO nicht voll ist und ein Prepare-Flag bereits gesetzt ist; in diesem Moment wird das Prepare-Flag wieder gelöscht. Das Interrupt-Signal wird bei jedem Lesezugriff auf das Status-Register und bei jedem Schreibzugriff auf den Tx-FIFO *gelöscht*. Das Prepare-Flag seinerseits wird bei jedem Schreibzugriff auf das Tx-FIFO *gesetzt*.

Die Anfrage des Interrupt-Controllers auf einen IRQ wird nur dann bejaht, wenn mindestens ein Signal plus dem dazugehörigen Interrupt-Enable-Bit im Control-Register gesetzt ist.

Dem Konstruktor der `Uart` wird unter anderem ein Geräte-Index übergeben, mit dem die Instanz von anderen UARTs unterschieden werden kann. Dieser Index ist simulationsweit unter den UART-Instanzen eindeutig.

Die Klasse stellt mehrere Lese- und Schreibmethoden zur Verfügung, mit welchen das Frontend transparent (d.h. ohne Seiteneffekte) auf die Registerbelegungen zugreifen kann.

Die Basis-IO-Adresse kann jederzeit modifiziert werden. Anhand der Offsets berechnen sich alle vier IO-Adressen automatisch. Bei einer Adressänderung werden die alten Adressen beim `IoManager` abgemeldet, die neuen entsprechend angemeldet.

Für die Einstellungen im Control-Register wurden als Default (bei fehlenden Angaben) für die Baudrate 115.200 Baud sowie für die Wortlänge acht Bit definiert. Die Originalhardware besitzt keinerlei Defaultwerte.

3.8.4 DMA-Demonstrator

Der `DmaDemonstrator` dient als einfaches Anschauungsbeispiel für die Implementierung eines IO-Gerätes. Er schreibt, unabhängig von der Prozessor-Pipeline, aller drei Takte ein zufälliges Datum in den DMA-Speicherbereich. Die DMA-Adresse wird dabei fortlaufend inkrementiert. Wird das Ende des Speichers erreicht, so wird ein Hardware-Interrupt angefordert. Der Addresszeiger springt anschließend automatisch auf die Anfangsadresse zurück, und das Prozedere beginnt von neuem.

Der Demonstrator hat die folgenden Eigenschaften:

- Es handelt sich um ein IO-Gerät, welches sich (ohne IO-Adresse) beim `IoManager` registriert.
- Die Klasse implementiert das Interface `InterruptSource_I`, kann somit Hardware-Interrupts anfordern, und muss sich demgemäß beim IRQ-Controller registrieren.
- Das Gerät hat (unbeschränkten) Zugriff auf den DMA-Speicher.
- Es besitzt einen Geräte-Index, eindeutig unter allen DMA-Demonstratoren.

Die einzelnen Methoden sollen nun schrittweise erläutert werden, um als Anleitung für etwaige Erweiterungen zu dienen.

Dem Konstruktor wird der Geräte-Index, der DMA-Speicher, der Interrupt-Controller sowie der IO-Manager übergeben. Es wird sich hier einmalig beim Interrupt-Controller registriert. Da diese Registrierung konfigurationslos ist, wird es nie nötig sein, sie irgendwie abzuändern. In der `tearDown()`-Methode wird diese Registrierung wieder gelöscht. Zuletzt wird im Konstruktor die `setup()`-Methode aufgerufen.

```
public DmaDemonstrator( final int id, final IoManager ioMgr,
                        final DmaMemory dmaMemory, final InterruptController irqCtrl) {

        this.ioMgr = ioMgr;
        this.dmaMemory = dmaMemory;
        this.irqCtrl = irqCtrl;
        this.id = id;

        irqCtrl.registerInterruptSource(this);

        rand = new Random();
        setup();
}
```

Die Setup-Prozeduren werden, wie gerade erwähnt, bei der Instanziierung und bei jedem Simulations-Setup aufgerufen. Zunächst wird sich beim IO-Manager adresslos (Dummy-Adresswert `NO_IO_ADDRESS`) registriert. Zuvor muss sich, per Definition, abgemeldet werden. Anschließend wird der DMA-Anfangszeiger auf die erste Adresse des DMA-Bereiches gesetzt, insofern ein DMA-Bereich verwendet wird (sonst auf `INVALID`). Sollten die Adressbereiche geändert werden, so wird diese Methode (durch den IO-Manager) definitiv aufgerufen, d.h. es kann nie zu ungültigen Adresswerten kommen. Anschließend wird das Gerät neu gestartet.

```
@Override
public void setup() {

    ioMgr.unregister(IoManager.NO_IO_ADDRESS, this);
    ioMgr.register(IoManager.NO_IO_ADDRESS, this);
    dmaStartAddress = (Settings.USING_DMA)? Settings.DMA_ADDRESS_MIN : INVALID;
    restart();
}
```

Bei jedem Neustart wird der DMA-Zeiger auf die gespeicherte Startadresse gesetzt. Der Taktzähler wird zurückgesetzt. Ein Aktualisierungs-Event wird getriggert.

```
@Override
public void restart() {

    dmaPointer = dmaStartAddress;
    clockCounter = 0;
    fireRefreshEvent(COMPLETE_REFRESH);
}
```

In der Methode `onClock()` wird, insofern ein DMA-Bereich genutzt wird, der Taktzähler

inkrementiert (modulo drei). Sollte der Taktzähler abgelaufen sein, so wird auf die Adresse, die durch den DMA-Zeiger definiert wird, eine Zufallszahl geschrieben. Der Zeiger wird inkrementiert. Sollte dieser danach den DMA-Adressbereich verlassen, wird er auf den Anfangswert zurückgesetzt. Ein Aktualisierungs-Event wird gefeuert.

```
@Override
public void onClock() throws SpartanException {

    if (dmaPointer == INVALID) return;      // no DMA range used
    if (clockCounter++ < 2) return;

    clockCounter = 0;

    assert(DmaMemory.containsAddress(dmaPointer));
    dmaMemory.setValueAt(dmaPointer++, rand.nextInt(BITMASK_18 + 1),
                         AccessType.WATCHPOINT);

    if (dmaPointer > Settings.DMA_ADDRESS_MAX) dmaPointer = dmaStartAddress;

    fireRefreshEvent(COMPLETE_REFRESH);
}
```

Ein Interrupt wird genau dann angefordert, wenn der Zeiger auf die Endadresse des DMA-Speichers zeigt. Dafür ist nur ein Vergleich, keinerlei zustandsverändernde Berechnung, nötig; die Methode `calculateInterrupt()` bleibt leer. Fällt dieser Vergleich positiv aus, so liefert `hasInterruptRequest()` **`true`** zurück.

```
public void calculateInterrupt() {
    // nothing to do here
}
public boolean hasInterruptRequest() {
    return (dmaPointer == Settings.DMA_ADDRESS_MAX);
}
```

Die Funktionen `setValueAt()` und `getValueAt()` werden nicht benötigt; der IO-Manager wird sie nie aufrufen. Die Registrierungen beim IO-Manager und beim Interrupt-Controller werden schließlich in der `tearDown()`-Methode wieder gelöscht.

```
@Override
public void tearDown() {
    irqCtrl.unregisterInterruptSource(this);
    ioMgr.unregister(IoManager.NO_IO_ADDRESS, this);
}
```

4 Entwurf und Implementierung des Frontends

4.1 Aufbau und Steuerung

Als Frontend des Simulators ist das grafische Java-Swing-Framework genutzt worden. Die Applikation wird mit einem Hauptfenster gestartet. Mittels eines Menüs und einer Steuerleiste sind die globalen Simulationsfunktionen ansprechbar. Um die einzelnen Komponenten des simulierten Mikrocontrollers darzustellen, können weitere Unterfenster geöffnet werden. Es stehen ingesamt acht verschiedene Ansichten (Views) zur Verfügung. Die Fenster können in beliebiger Anzahl platziert (und natürlich auch wieder entfernt) sowie auch zu einem Icon minimiert (und wieder vergrößert) werden.

4.1.1 Hauptfenster, Menuleiste und Steuerleiste

Das Hauptfenster `MainWindow` ist der Kern der gesamten Anwendung. Es besitzt eine Instanz des `SimulationManagers`, um damit auf die Funktionalitäten des Backends zuzugreifen. Seine Aufgaben bestehen u.a. aus:

- Starten und Beenden der Gesamtanwendung
- Laden und Speichern von Präferenzen
- Grafische Anzeige aller GUI-Komponenten
- Dynamische (Unter-)Fensterverwaltung

- Globale GUI-Aktualisierung (Refreshing)

Auf die Details soll im Laufe dieses Kapitels eingegangen werden. Sämtliche zentralen Konstanten und Enumerationen, die für die GUI von Interesse sind, sind in `GuiDefinitions_I` abgelegt.

Zur administrativen Unterstützung des Hauptfenster gibt es einen zentralen Ereignishandler: den `GlobalEventHandler`. Dieser bearbeitet simulationsweite Ereignisse, beispielsweise das Neustart- und Setup-Prozedere.

Es stehen eine Steuerleiste `ToolBar` sowie ein Menü `MenuBar` zur Verfügung. Beide Klassen kommunizieren sowohl mit dem Hauptfenster als auch mit dem Ereignishandler.

Die Toolbar bietet die folgenden Funktionalitäten:

- Die Abarbeitung der Simulation kann gesteuert werden:
  RUN – startet die Simulation mit der gewählten Geschwindigkeit
  PAUSE – pausiert eine laufende Simulation
  STEP – führt exakt einen Simulationsschritt aus
  RESTART – erzwingt einen Neustart

  Bei Betätigen der Taste wird der Ereignishandler aufgerufen. Dieser übernimmt die notwendigen Arbeiten.

- Die Simulationsgeschwindigkeit kann gesteuert werden. Die gewählte Geschwindigkeit wird in `Settings` abgespeichert. Die höchste Geschwindigkeitsstufe unterbindet eine GUI-Aktualisierung und lässt zur Bestätigung ein Mitteilungsfenster erscheinen. Die Geschwindigkeit kann auch während einer laufenden Abarbeitung verändert werden; in diesem Falle wird (intern) die Abarbeitung unterbrochen, die Einstellung abgespeichert und die Abarbeitung wieder fortgesetzt. Weiteres dazu im Abschnitt 4.1.2.

- In einer Auswahlbox werden alle gegenwärtig geöffneten Unterfenster dargestellt. Durch Selektierung eines Eintrages wird das dazugehörige Fenster aktiviert, d.h. es wird ausgewählt und rückt in den Vordergrund.

Das Menü bietet seinerseits die folgenden Möglichkeiten:

- Laden und Speichern von Assembler-Dateien (`.sph`)
- Beenden der Applikation
- Befehle zum Steuern der Simulation; identisch mit denen der Toolbar
- Konfiguration der Simulationseinstellungen (`Settings`)
- Öffnen neuer Unterfenster
- Anzeige von Informationen über die Anwendung selbst

Viele Benutzeraktionen werden auch hier nur direkt an den Eventhandler weitergegeben. Dadurch ist es insbesondere für die weiteren Maßnahmen transparent, ob beispielsweise ein STEP-Befehl vom Menü oder der Steuerleiste induziert worden ist.

4.1.2 Zustände und Ereignisverarbeitung

Der `GlobalEventHandler` verwaltet den Applikationsszustand und führt globale Steuerungsaktionen aus bzw. leitet sie entsprechend weiter.

Es gibt drei Zustände (siehe auch Diagramm 4.1):

- `DEFAULT` – Der Standardzustand; es läuft keine kontinuierliche Simulation, und es ist kein Fehler aufgetreten.
- `RUNNING` – Die Simulation wird gerade ausgeführt.
- `EXCEPTION` – Es ist eine `SpartanException` aufgetreten. Die Simulation muss neu gestartet werden.

Der Eventhandler trägt dafür Sorge, dass dem aktuellen Zustand entsprechend Menüeinträge bzw. Buttons gegebenenfalls disabled (nicht anklickbar) werden. Z.B. ist es im Fehlerfall nicht mehr möglich, ein START oder STEP zu betätigen. Die Benutzerführung wird dadurch wesentlich vereinfacht und inkonsistente Zustände von vornherein vermieden.

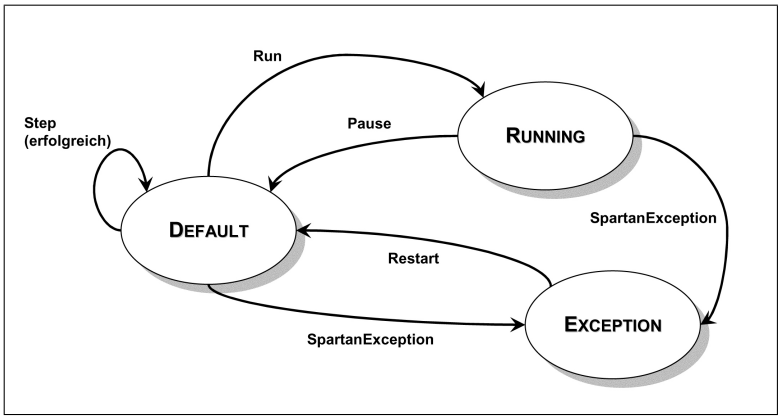

Abbildung 4.1: Zustandsdiagramm der Gesamtanwendung

Betätigt der Benutzer STEP, so wird die Methode `onStep()` zum Berechnen eines einzelnen Simulationsschrittes aufgerufen. Bei einer kontiniuierlichen Simulationsausführung wird ein separater Thread beauftragt, eben diese Methode immer wieder aufzurufen. Dies bedeutet also, das Verhalten bei einem manuellen Schritt ist mit dem einer automatisierten Abarbeitung identisch, da beide Vorgehensweisen letztendlich auf dieselbe Funktion abgebildet werden.

Innerhalb von `onStep()` wird der `SimulationManager` mit `executeStep()` aufgerufen. Sollte eine `SpartanException` geworfen werden, so wird diese unmittelbar abgefangen. Es erscheint eine `MessageBox` mit einem Hinweis für den Benutzer. Der Zustand wechselt in `EXCEPTION`. Sollte der Simulationstakt erfolgreich verlaufen sein und ein Watchpoint oder ein Breakpoint berührt worden sein, so wird die dabei geworfene `SimulationBreakException` abgefangen und als Hinweismeldung ausgegeben; der Zustand wechselt zu (oder bleibt gegebenenfalls in) `DEFAULT`.

Bei fast allen Ereignissen, etwa bei einem Neustart oder Setup, wird eine globale GUI-Aktualisierung angefordert. Eine Sonderrolle spielt `onStep()`: die Aktualisierungen werden in diesem Falle dezentral und „bedarfsgerecht“ gesteuert. Diesbezüglich wird ein, im Backend bereits angesprochener, Aktualisierungs-Mechanismus eingesetzt, welcher in Abschnitt 4.1.3 näher betrachtet wird. Alle anderen Refreshings erfolgen nicht regelmäßig; es wird großzügig damit umgegangen.

Beim Laden und Speichern von `sph`-Dateien wird vom EventHandler ein Auswahlfenster offeriert, mit welchem der Benutzer einen entsprechenden Dateinamen definieren kann. Anschließend wird einen `AssemblyParser`-Instanz des Backends beauftragt, den Haupt-

speicher zu füllen bzw. auszulesen. Im Falle einer `ParsingException` wird der Vorgang abgebrochen und eine `MessageBox` angezeigt. Nach einem erfolgreichen Ladevorgang wird die Simulation neu gestartet.

Anmerkung: Während der laufenden Simulation arbeiten der Simulationsthread sowie die GUI-Threads nebenläufig. Aus Gründen der Performance sind keine Mutexe zum Schutz von Registerwerten u.ä., welche sowohl automatisiert als auch durch den Benutzer (via GUI) verändert werden können, eingesetzt worden. [1] Obgleich bisher keinerlei Probleme festgestellt werden konnten, wird geraten, die Simulation bei Benutzer-Manipulationen zu unterbrechen.

4.1.3 Subwindows und AbstractViews

Grundidee des Frontends ist es, auf ein und denselben simulierten Mikrocontroller verschiedene Ansichtsmöglichkeiten (Views) zu geben. Diesbezüglich lassen sich zwei Arten von Views unterscheiden:

1. *Transparente Ansichten* können in beliebiger Zahl genutzt werden. Sie beeinflussen das darunter liegende Modell einzig durch explizite Manipulationen. Veränderungen des Modells werden unmittelbar bei allen Ansichten sichtbar.

2. *Instanziierte Ansichten* sind direkt mit einer Komponente, genauer: einem IO-Gerät, verknüpft. Jede neue Ansicht ein und desselben Typs erzeugt eine neue Komponente mit ihren eigenen Eigenschaften. Unterschiedliche Komponenten-Instanzen sind prinzipell unabhängig voneinander (ungeachtet etwaiger Wechselwirkungen miteinander).

Die nachfolgenden Abschnitte befassen sich ausgiebiger mit den verschiedenen Ansichtsklassen. Jede dieser Klassen ist abgeleitet von der abstrakten Superklasse `AbstractView`. Diese ist wiederum ein `javax.swing.JPanel`, d.h. die konkreten GUI-Elemente (Buttons, Tabellen, Labels etc.) können direkt eingefügt werden. Abbildung 4.2 zeigt eine Übersicht.

[1] Es wurde in der Tat versucht, an nur *einer* Stelle einen Mutex einzuführen. Jedoch war der Performance-Verlust bereits da bei schnelleren Ausführunggeschwindigkeiten deutlich spürbar und nicht akzeptabel.

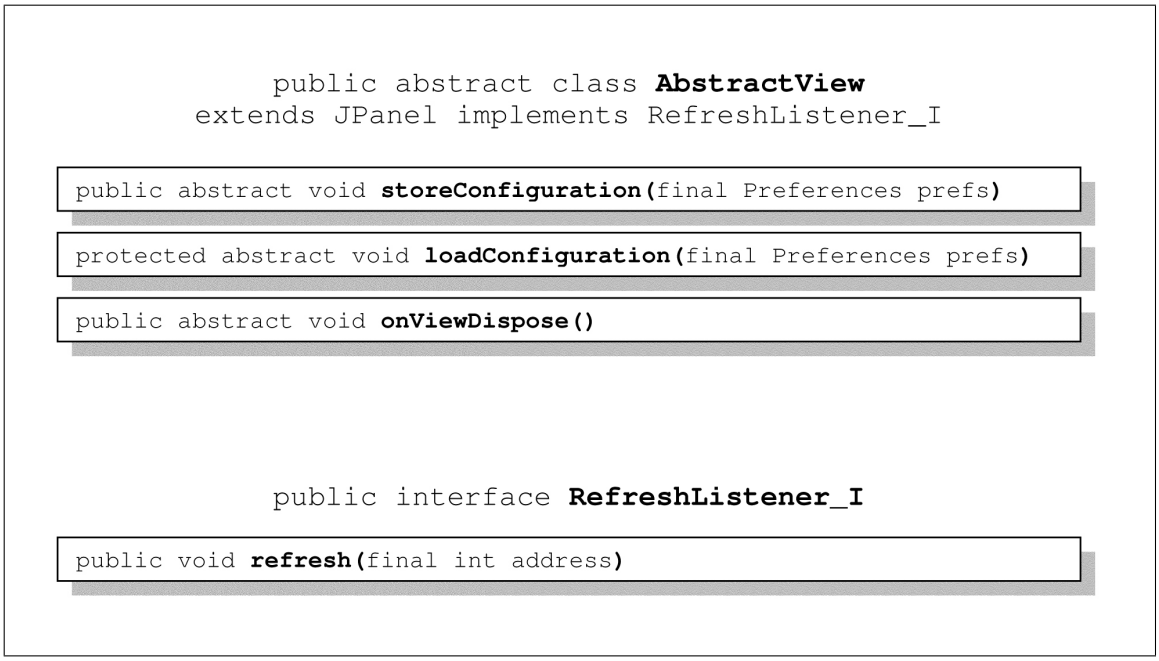

Abbildung 4.2: Klassenübersicht `AbstractView` und `RefreshListener_I`

Zwei Funktionen, `storeConfiguration()` und `loadConfiguration()`, sind für die Präferenzen-Verwaltung notwendig. Mehr dazu in Abschnitt 4.1.5.

Die abstrakte Klasse implementiert das Interface `RefreshListener_I`. Die Ansicht muss sich bei sämtlichen `AbstractHardwareComponents` als Listener registrieren, deren Inhalte sie anzeigen soll, um über ihre Zustandsänderungen regelmäßig informiert zu sein. Sobald ein Update erfolgen soll, wird `refresh()` aufgerufen. Wird dem Aufruf eine Adresse übergeben, so muss ausschließlich diese aktualisiert werden. Dies wurde insbesondere für den Speicher und die Register-File entwickelt, da es die Performance *merklich* beeinträchtigt, wenn bei einer einzelnen Registeränderung Tausende von Werten unnütz aktualisiert werden. Alternativ dazu kann statt einer Adresse `COMPLETE_REFRESH` (:= -1) angegeben werden. Um Fehler nicht unnötig zu provozieren, wird an den meisten Stellen mit nur wenigen zu aktualisierenden Registern ein `COMPLETE_REFRESH` angefordert.

Die Funktion `onViewDispose()` wird aufgerufen, sobald das Fenster geschlossen wird. Jede Ansicht muss sich hier zwingend von allen Hardware-Komponenten abmelden, für die sie sich als `RefreshListener_I` registriert hatte. Bei instanziierten Ansichten muss desweiteren das zugrunde liegende IO-Gerät korrekt beendet und abgemeldet werden.

Dem gegenüber steht die Klasse `SubWindow`. Sie ist eine unmittelbare Ableitung von `javax.swing.JInternalFrame` und implementiert damit ein grafisches Unterfenster. Der dazugehörige `javax.swing.JDesktopPane` wird vom Hauptfenster zur Verfügung gestellt. SubWindows sind konkrete Instanzen, die Verwaltungsfunktionalitäten kapseln und als internes Anzeigeelement ein `AbstractView` aufnehmen. Dies bedeutet also: `AbstractViews` sind prinzipielle *Darstellungsflächen*, `SubWindows` hingegen ermöglichen eine spezielle *Darstellungsart*, die sie dem zugehörigen View zur Verfügung stellen (in diesem Fall kleine interne Fenster).

Es stehen zwei statische Factory-Methoden zur Verfügung, mit welchen ein Fenster instanzziert werden kann: `create()` sowie `createByPreferences()`. Erste wird aufgerufen, sobald der Benutzer ein *neues* Fenster hinzufügen möchte. Letztere hingegen wird im Rahmen des Präferenzen-Managements genutzt, um gespeicherte Fenster bei einem Programmstart wiederherzustellen. Beiden Methoden wird u.a. ein `ViewType` als Parameter übergeben, anhand dessen die gewünschte Ansichtsklasse erzeugt und dem Fenster hinzugefügt werden kann.

Das `MainWindow` hat kein Wissen über Views, es kennt ausschließlich `SubWindows` und kann nur diese verwalten. Informationen über den Anzeigeinhalt und dazugehörige Eigenschaften gelangen also nicht über das Unterfenster hinaus. Sollte das Hauptfenster (meist durch Anweisung des EventHandlers) ein globales Refreshing fordern, so wird dies an die `SubWindows` via `refreshCompletely()` geleitet, welche wiederum ihre konkrete View-Instanz informieren müssen. Abbildung 4.3 stellt das Aktualisierungsverhalten schematisch dar.

Sowohl bei der Erstellung eines neuen Fensters als auch bei dessen Entfernung ist es notwendig, eine gegebenenfalls laufende Simulation zu stoppen und anschließend wieder fortzuführen. Es kann und soll nicht garantiert werden, dass andernfalls die Konsistenz nicht gefährdet wird. Gleichermaßen wird bei beiden Ereignissen stets ein globales Refreshing veranlasst.

Anmerkung: Bei Programmstart werden auch „Minimiert“-Zustände der Fenster gegebenenfalls restauriert. Das Hauptfenster befindet sich jedoch während der Wiederherstellungsphase noch im Konstruktor. Swing ermöglicht es nicht, den minimierten Zustand eines `JInternalFrames` zu diesem Zeitpunkt anzuwenden. Aus diesem Grunde wird die Methode `onMainWndSetVisible()` genutzt, welche aufgerufen wird, sobald das Haupt-

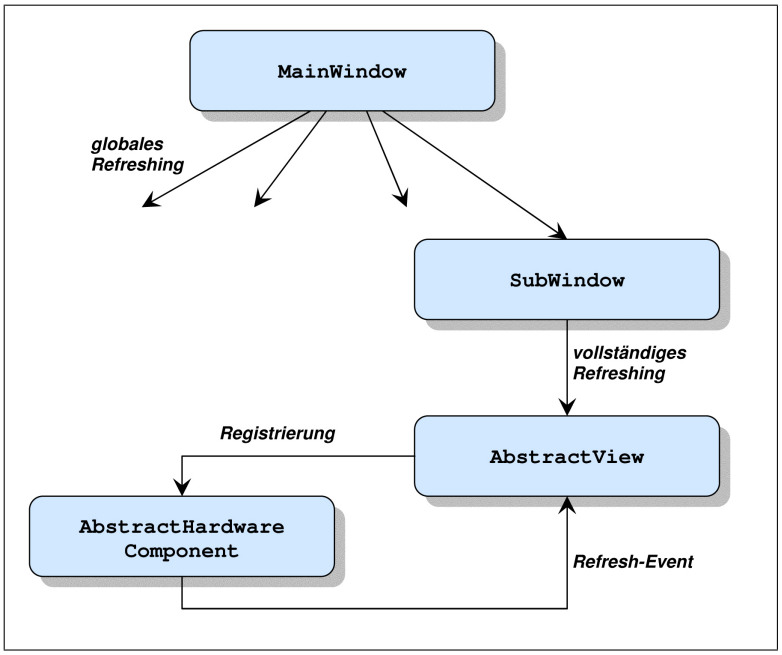

Abbildung 4.3: Prinzip des GUI-Refreshings

fenster sichtbar wird.

4.1.4 Konfigurationsmöglichkeiten

Die Simulationsparameter, gespeichert in der statischen Klasse `Settings`, können über das Menü mit einem separaten modalen Fenster bearbeitet werden (Abbildung 4.4). Die dazu genutzten GUI-Klassen befinden sich im Packet `gui.settings`.

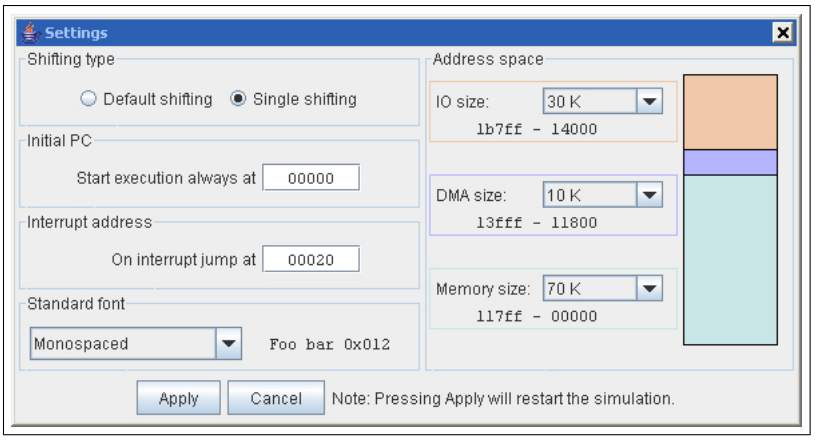

Abbildung 4.4: Screenshot Settings-Dialog

Es sind die folgenden Einstellungen möglich:

- Shifting-Modus (für `SLL` u.ä.)
- Anfangs-PC sowie IRQ-Sprungadresse: Sollten ungültige Eingaben gemacht werden (insbesondere unter Berücksichtigung der Adressraumaufteilung), so werden diese nicht akzeptiert.
- Standard-Schriftart: Sie wird meist für Tabelleninhalte u.ä. verwendet. Es wird empfohlen, eine nicht-proportionale Schriftart zu verwenden, um Registerwerte deutlicher darstellen zu können.
- Adressraumaufteilung: Der Gesamtadressraum von 18 Bit kann entsprechend in Hauptspeicher, fakultativ DMA-Speicher sowie in IO-Bereich aufgeteilt werden. Es ist nicht möglich, inkonsistente Adressaufteilungen zu generieren. Falls der Anfangs-PC oder die Interrupt-Adresse ungültig werden sollten, so wird dies bemängelt und korrigiert.

Wird die Änderung mit einem Apply bestätigt, dann wird daraufhin ein Simulations-Setup erzwungen, um sämtlichen Komponenten die Möglichkeit zu geben, sich gegebenenfalls neu zu konfigurieren. Unmittelbar darauf wird, per Definition, die Simulation neu gestartet. Parallel dazu wird auch die GUI aufgefordert, sämtliche Anzeigen *vollständig* zu aktualisieren. Da die Standard-Schriftart sich potentiell geändert haben könnte, müssen Komponenten gegebenenfalls im Rahmen ihrer `refresh()`-Methode die Schriftattribute neu setzen.

4.1.5 Präferenzen-Verwaltung

Da die Anwendung mit zahlreichen Einstellungsmöglichkeiten ausgestattet ist, ist es erforderlich, Benutzerkonfigurationen, d.h. Präferenzen, auch speichern und beim nächsten Programmstart wieder laden zu können.

Es gibt die folgenden Präferenzen:

- Simulationseinstellungen, definiert in `Settings`
- die Größe und der Zustand des Hauptfensters

- das zuletzt genutzte Arbeitsverzeichnis
- die Größe und der Zustand von Unterfenstern sowie der Typus ihrer jeweiligen Ansichtklasse
- Einstellungen der Ansichtsklassen

Es wurde ein Präferenzen-Management entwickelt, dessen verzweigte Struktur sich durch das gesamte Frontend zieht. Die Lade- und Speicher-Funktionalitäten werden innerhalb einer verwaltenden Klasse, dem `PreferencesManager`, durch die zwei wesentlichen Methoden `loadAllPreferences()` und `storeAllPreferences()` gekapselt. Der Ladevorgang wird innerhalb des Konstruktors von `MainWindow` gestartet. Die Speicherung ihrerseits erfolgt im `GlobalEventHandler`, sobald der Benutzer das Programm beenden will.

Es wird die `java.util.prefs.Preferences`-Klasse genutzt, einer Schlüssel-Wert-Menge, die zusätzlich mit Knoten baumartig strukturiert werden kann. Dieses Präferenzen-Objekt kann als (XML-)Datei persistent abgelegt und daraus auch wieder restauriert werden.

Wird der `PreferencesManager` angewiesen, die Konfigurationen zu laden, so sucht er die Datei `.spartansim` im aktuellen Ausführungsverzeichnis. Wird diese nicht gefunden, so wird der Wiederherstellungsvorgang abgebrochen, und es werden die implementierten Vorgabe-Werte verwendet. Andernfalls jedoch wird daraus wieder ein `Preferences`-Objekt importiert. Dieses wird nun an alle notwendigen Komponenten übergeben. Die Präferenzen sind dabei als Hierarchie gegliedert, und es ist möglich, entsprechend hierarchisch niederen Klassen nur ihren jeweiligen „Teilbaum“ zu übergeben, anhand dessen sie sich ihre (und nur ihre) Informationen laden.

Abbildung 4.5 zeigt schematisch den Aufbau der Präferenzen-Hierarchie. Das dargestellte Beispiel besitzt drei Unterfenster.

Diesbezüglich besitzen `AbstractViews` stets eine Lade- und Speicher-Methode für Präferenzen (siehe auch Abschnitt 4.1.3). Dem Konstruktor wird *immer* ein `Preferences`-Objekt übergeben, mit welchem die Lademethode abgearbeitet wird. In dieser Methode sollen alle Konfigurationen vorgenommen werden: entweder durch Extraktion der Information aus den Präferenzen oder durch Setzen von Default-Werten, falls die Ansicht neu

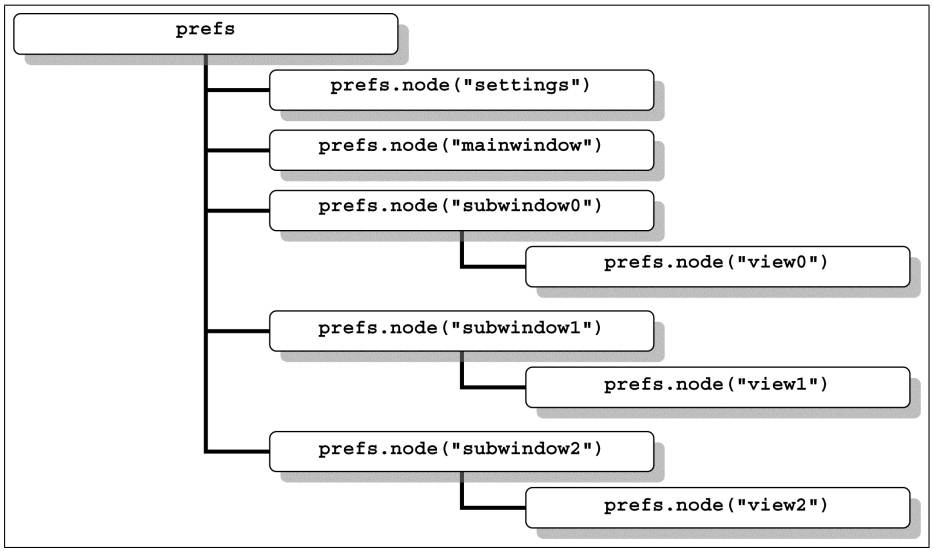

Abbildung 4.5: Schematische Struktur der Präferenzen-Map

erzeugt wurde. In letzterem Fall ist der Präferenzen-Parameter auf `null` festgelegt.

Ein abschließendes Beispiel soll den Sachverhalt noch einmal verdeutlichen. Es ist aus der Klasse `AssemblyView` entnommen; diese lädt bzw. speichert als Benutzereinstellung die Start- und Endadresse des anzuzeigenden Speicherausschnittes. Als Standardwerte sind die Anfangs- und Endadresse des vollständigen Hauptspeicherbereiches definiert. Ist das dazugehörige Fenster z.B. mit dem Index drei abgespeichert worden, so entdeckt der `PreferencesManager` bei den importierten Präferenzen einen Knoten `"subwindow3"` und erzeugt ein neues Subwindow. Dieses erkennt anhand seines Knotens, wie seine grafischen Ausmaße sein sollen, und dass es eine Ansicht vom Typ `AssemblyView` beinhalten soll. Es startet den dazugehörigen Konstruktor und übergibt ihm die Präferenzeneinträge unterhalb des Knotens `"view3"`. Die weiteren Details sehen wie folgt aus:

```
@Override
public void storeConfiguration(final Preferences prefs) {

    assert(prefs != null);
    prefs.putInt("startingAddress", startingAddress);
    prefs.putInt("endingAddress", endingAddress);
}
```

```
@Override
protected void loadConfiguration(final Preferences prefs) {

    if (prefs != null) {
        final int start = prefs.getInt("startingAddress", Settings.MEMORY_ADDRESS_MIN);
        final int end = prefs.getInt("endingAddress", Settings.MEMORY_ADDRESS_MAX);

        startingAddress = (MainMemory.containsAddress(start))?
                                start: Settings.MEMORY_ADDRESS_MIN;
        endingAddress = (MainMemory.containsAddress(end))?
                                end: Settings.MEMORY_ADDRESS_MAX;

    } else {
        // brand new view:
        startingAddress = Settings.MEMORY_ADDRESS_MIN;
        endingAddress = Settings.MEMORY_ADDRESS_MAX;
    }

    startingAddressField.setText(String.format("%05x", startingAddress));
    endingAddressField.setText(String.format("%05x", endingAddress));
}
```

4.2 Hinweise und Entwurfsaspekte zu Ansichtselementen

Die folgenden Ansichten bestehen aufgrund ihrer Komplexität fast immer aus mehreren Klassen, die jeweils in einem Packet zusammengefasst werden. Die Hauptklasse trägt immer den Postfix `View` und implementiert `AbstractView`. Ihr wird per Definition der `SimulationManager`, das `MainWindow` sowie die `Preferences` übergeben. Alle für die spezifische Funktionalität notwendigen Komponenten lassen sich entweder über den Manager oder über das Hauptfenster anfordern.

Die Hauptklasse speichert in den meisten Fällen Zeiger auf die benötigten Backend-Objekte als package-scoped Membervariablen ab. Dadurch wird den weiteren Hilfsklassen eine einfache Zugriffsmöglichkeit offeriert.

Die Verantwortung des Refreshings liegt bei der Hauptklasse. Sie hat dafür Sorge zu tragen, dass sämtliche Anzeigeelemente aktualisiert und konsistent bleiben.

Jedem Packet ist ein Interface (Postfix `Definitions_I`) beigefügt, in dem gegebenenfalls Konstanten und Enumerationen abgelegt werden können. Aus Gründen der Übersicht-

lichkeit erweitert jedes dieser Interfaces stets sowohl `SpartanDefinitions_I` als auch `GuiDefinitions_I` und stellt damit gleichzeitig deren Definitionen zur Verfügung.

Insofern nicht explizit der Kontext etwas anderes vorgibt, sind sämtliche Register-Inhalte sowie Adresswerte in hexadezimaler Form dargestellt. Zugunsten der Lesbarkeit wurde meist auf einen `0x`-Präfix verzichtet. Bei der Eingabe von Hexadezimalzahlen steht es dem Benutzer frei, `0x` oder `x` (sowohl in Klein- als auch in Großschreibung) dem Wert voranzusetzen. Bei Binärzahlen gilt dies gleichermaßen mit `b`. Die Bezeichnernummern von General-Purpose-Registern hingegen werden stets dezimal angegeben – ein `R11` ist verständlicher als ein `RB`.

Die meisten Register-Inhalte sind mit einem Tooltip versehen. Dieser zeigt sowohl die Binärdarstellung als auch die, gegebenenfalls vorzeichenbehaftete, Dezimaldarstellung an.

Sollen Werte, die mittels Tabelleneinträgen dargestellt werden (das sind nahezu alle), verändert werden, sind sie doppelt anzuklicken, um damit in den Editiermodus zu wechseln. Nach Eingabe des neuen Wertes ist dieser mit einem ENTER zu bestätigen. Zweifelt der Benutzer während des Editierens, so kann er beliebig ungültige Zeichen (z.B. Buchstaben jenseits von F) eingeben oder schlichtweg den Wertebereich des dazugehörigen Registers überschreiten. Nach Bestätigung wird der Modus verlassen; der alte Wert bleibt jedoch erhalten, da die vorgenommene Eingabe unzulässig ist.

4.3 Speicheransichten

Es gibt mehrere Ansichten, die den Inhalt des Hauptspeichers reflektieren. Da der Spartan MC keinen separaten Instruktionsspeicher besitzt, kann der Wert einer Hauptspeicherzelle prinzipiell sowohl als einfaches Datum als auch als CPU-Befehl interpretiert werden.

4.3.1 AssemblyView

Die `AssemblyView` sieht den Hauptspeicher als Aufzählung von Instruktionen (Abbildung 4.6). Es wird versucht, jedes 18-Bit-Wort als Befehl zu interpetieren. Der zu betrachtende Speicherbereich kann eingeschränkt werden, indem eine Anfangs- und Endadresse definiert werden.

Zentrales Widget ist eine Tabelle mit den Speicheradressen und dazugehörigen mnemonischen Instruktions-Strings, falls eine Interpretation möglich ist. Befehle, die sich gegenwärtig in der Pipeline befinden, werden farbig markiert: blau für IF/ID, rot für EX/MEM sowie grün für WB.

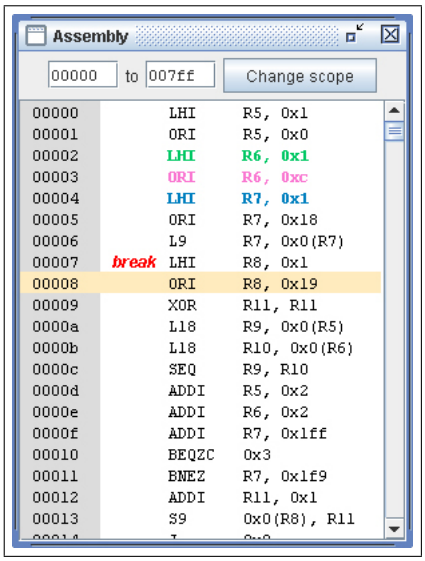

Abbildung 4.6: Screenshot Instruktionen

Durch Klicken (links oder rechts) auf eine Zeile im Bereich der vorderen beiden Spalten lässt sich ein Breakpoint positionieren. Durch abermaliges Klicken lässt sich dieser Breakpoint wieder entfernen. Trifft eine Abarbeitung auf einen Breakpoint, d.h. wird der Befehl der entsprechenden Zeile geladen, so erscheint ein Hinweisfenster und die Ausführung wird gegebenenfalls angehalten. Es können beliebig viele Breakpoints gesetzt werden. Sie bleiben auch über einen Simulationsneustart hinweg erhalten.

Das dazugehörige Packet `gui.assembly` besteht aus den folgenden Klassen:

`AssemblyView`	`AssemblyViewDefinitions_I`
`AssemblyCellRenderer`	`AssemblyTableModel`

4.3.2 MemoryView

`MemoryView` ermöglicht es, die Zahlenwerte des Haupt- sowie des DMA-Speichers (falls gegeben) einzusehen (Abbildung 4.8).

Es wurde bereits dargestellt, dass die Adressangaben sowohl als Default-Adressierung als auch als Daten-Adressierung interpretiert werden können. Die Speicheransicht bietet dies-

bezüglich vier Varianten an, jeweils dargestellt als separate Tabellen:

- Default-Adressierung mit 18 Bit Breite (ein Beispiel: ein Datum auf Adresse `0x000b4`, siehe Abbildung 4.7)
- Default-Addressierung mit neun Bit Breite, Differenzierung von oberem Halbwort (High) und unterem Halbwort (Low) (obiges Datum liegt hier auf `0x000b4` High und Low);
- Daten-Adressierung mit 18 Bit Breite (das Datum liegt auf `0x00168`)
- Daten-Adressierung mit neun Bit Breite (hier liegt der Wert auf `0x00168` und `0x00169`)

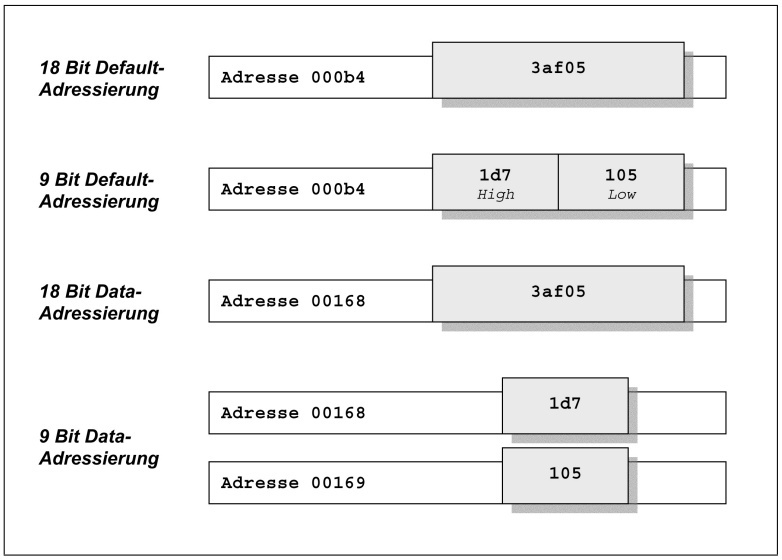

Abbildung 4.7: Varianten der Speicheradressierung

Der Hauptspeicher wird mit schwarzer Schriftart kenntlich gemacht, der DMA-Speicher mit blauer. Es steht dem Nutzer frei, Felder beliebig abzuändern. Alle Zahlenwerte ungleich null werden fett hervorgehoben, um die Übersichtlichkeit bei größeren Speicherbereichen zu verbessern.

Es lässt sich an jedem Speicherplatz ein Watchpoint durch einen Rechtsklick setzen. Ein weiterer Klick entfernt diesen wieder. Sobald lesend oder schreibend auf diese Adresse zugegriffen wird, erscheint ein Hinweisfenster und die Abarbeitung wird gegebenenfalls abgebrochen. Watchpoints bleiben auch nach einem Simulations-Neustart erhalten. Es ist egal, in welcher Registerkarte (d.h. Adressierungsformat) der Watchpoint gesetzt wird; er

wird überall entsprechend aktualisiert.

Es stehen zwei Buttons zur Verfügung, mit denen die gegenwärtig selektierten Zellen bzw. der gesamte Speicher gelöscht werden kann.

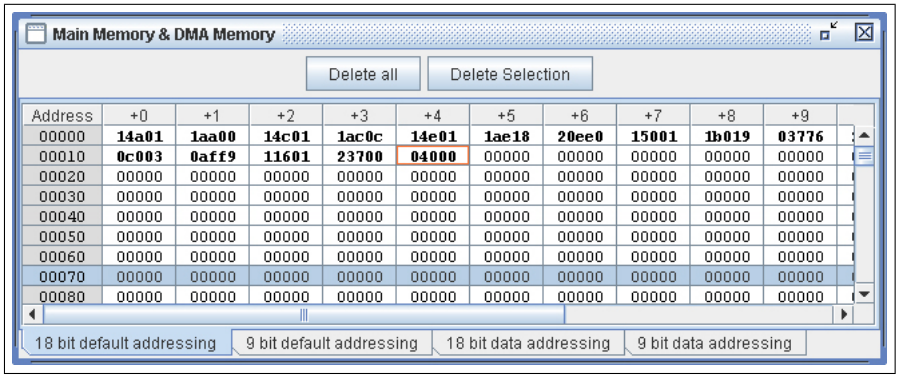

Abbildung 4.8: Screenshot Speicher

Das Packet `gui.memory` setzt sich wie folgt zusammen:

`MemoryView`	`MemoryViewDefinitions_I`
`Memory18DefaultPanel`	`Memory18DataPanel`
`Memory9DefaultPanel`	`Memory9DataPanel`

4.4 Registeransichten

Die General-Purpose-Register als auch die SFR-Register werden in einer gemeinsamen Ansicht dargestellt: `RegistersView` (Abbildung 4.9). Das Packet `gui.registers` unterteilt sich diesbezüglich in `gui.registers.registerfile` und `gui.registers.sfr`.

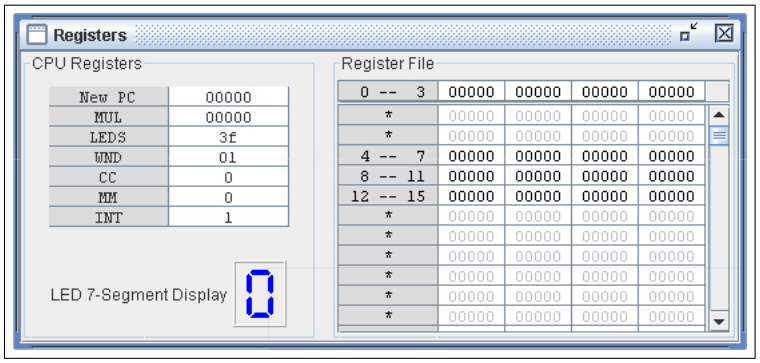

Abbildung 4.9: Screenshot CPU-Register und Register-File

4.4.1 RegisterFileView

Die Register-File beschreibt den RAM-Bereich, welcher für General-Purpose-Register genutzt wird. Die Bitbreite des Offsets für das Registerfenster beträgt sechs Bit, d.h. der maximale Offset-Wert beträgt 63.

Eine Tabelle zeigt den vollständigen Registerbereich an. Werte außerhalb des aktuellen Fensters werden grau gekennzeichnet. Die erste Spalte bildet die lokale Adresse, d.h. die Registernummer (0..15), ab. Die gesamte Tabelle besteht intern aus zwei Teiltabellen. Die globalen Register R0 bis R3 sind stets sichtbar und werden in einem separaten Widget gezeigt. Der lokale Bereich hingegen ist scrollbar, um das aktuelle Fenster stets im sichtbaren Ausschnitt zu behalten, auch wenn der Offset entsprechend groß wird.

Durch einen Rechtsklick können, analog zur Speicher-Ansicht, Watchpoints gesetzt und gelöscht werden.

Die Registerfile-GUI nutzt die folgenden Klassen:

`RegisterFileView`	`RegisterFileViewDefinitions_I`
`RegisterFileTableModel`	`RegisterFileCellRenderer`

4.4.2 SfRegistersView

Die Special-Function-Register werden in der linken Hälfte des `RegistersViews` präsentiert. Als Widget wird auch hier eine Tabelle genutzt. Neben den Registern `MUL`, `LEDS`, `WND`, `MM`, `CC` und `INT` wird auch der Wert des nächst zu holenden PCs angezeigt. Streng genommen ist dies kein SFR, es ist jedoch nicht sinnvoll, den Wert aufwendig auszulagern.

Alle Werte können durch den Benutzer im Rahmen ihres jeweiligen Wertebereiches abgeändert werden. Der PC muss zudem auf eine gültige Speicheradresse verweisen. Damit lässt sich der Programmfluss manuell lenken. Wird der `WND`-Inhalt modifiziert, so aktualisiert sich die Register-File inklusive ihrer grafischen Darstellung unmittelbar.

Zusätzlich wird eine Sieben-Segment-Anzeige dargestellt. Sie wird direkt von den Bitbelegungen des `LEDS`-Registers angesteuert. Es ist die allgemein übliche „Verdrahtung“ gewählt worden; für Details sei auf entsprechende Literatur verwiesen.

Die verwendeten Klassen:

`SfRegistersView`	`SfRegistersViewDefinitions_I`
`SfRegistersTableModel`	`SfRegistersCellRenderer`
`SfRegistersSegmentDisplay`	

4.5 PipelineView

Der Inhalt der Pipeline-Stufen lässt sich jederzeit anzeigen. Es werden für jeden Befehl, falls vorhanden, nur die jeweils sinnvollen internen Register angezeigt. Details dazu sind in der Beschreibung des Backends unter 3.6 zu finden.

Das Fenster (siehe Abbildung 4.10) zeigt im oberen Teil ein Schema des Prozessoraufbaus. Im unteren Teil befinden sich drei farbig gerahmte Kästen, die die drei Pipeline-Stufen anzeigen. Die Instruktionsverarbeitung ist von links nach rechts zu lesen. Es sei betont, dass stets der Zustand *nach* einem einzelnen Pipeline-Takt zu sehen ist. Das heißt also, wenn WB einen Zahlenwert zum Rückschreiben anzeigt, so ist dieser bereits erfolgreich zurückgeschrieben worden. Demzufolge hat es auch keinerlei Auswirkungen, wenn z.B. ein durch EX/MEM geladener Speicherwert in der Hauptspeicher-Ansicht manuell verändert wird.

Es besteht die Möglichkeit, die Pipeline-Details auszublenden. In diesem Fall wird nur noch die aktuelle Instruktionsbezeichnung für die jeweiligen Stufe angezeigt.

Die einzelnen Stufen arbeiten intern mit Tabellen, deren Inhalt und Größe bei jedem Takt dynamisch angepasst wird. Die Pipeline-Ansicht wird ausnahmslos bei jedem Takt aktualisiert.

Das Packet `gui.pipeline` setzt sich zusammen aus:

`PipelineView`	`PipelineViewDefinitions_I`
`IfIdPanel`	`ExMemPanel`
`WbPanel`	`PanelCellRenderer`

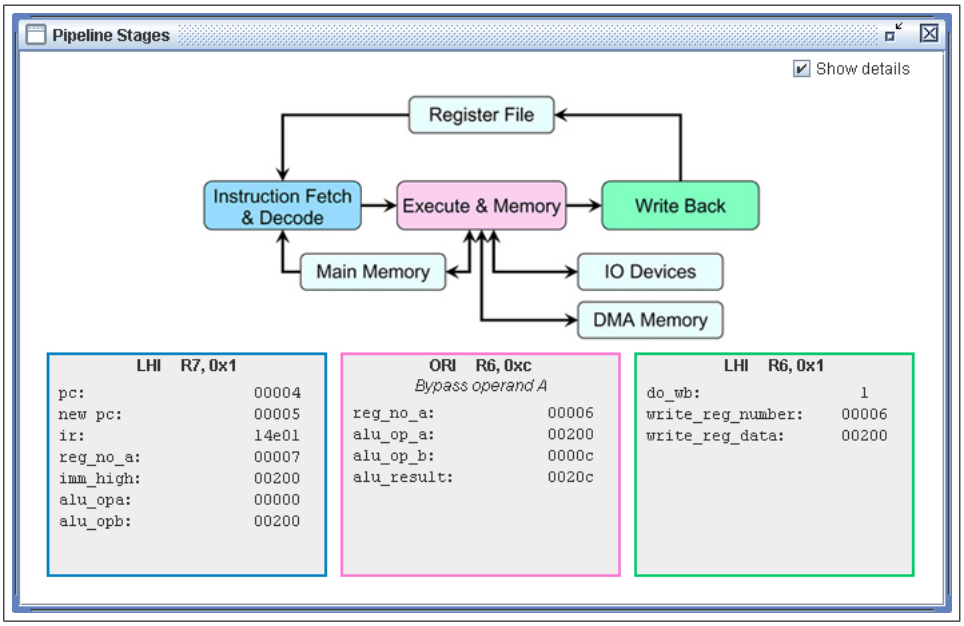

Abbildung 4.10: Screenshot Pipeline

4.6 InterruptView

Der Interrupt-Controller benötigt für seine Visualisierung die Abbildung seiner zwei Register sowie eine Auflistung mit allen in der aktuellen Konfiguration anliegenden Interrupt-Quellen (Abbildung 4.11).

Im oberen Teil wird die IO-Basis-Adresse, welche für die beiden Register als Offset verwendet wird, angezeigt und kann gegebenenfalls abgeändert werden. Es werden die daraus resultierenden zwei Registeradressen (d.h. Basis + 0, sowie Basis + 1) aufgezeigt.

Im mittleren Teil werden die Registerinhalte hexadezimal und binär dargestellt. Die erste Zeile zeigt alle anliegenden IRQs, die zweite die Priorität des dominanten. Es kann nur lesend darauf zugegriffen werden.

Im unteren Teil befindet sich eine Liste mit allen Interrupt-Quellen, die sich registriert haben. Sie sind absteigend nach Prioritäten geordnet. Als Name wird der String verwendet, welcher mit `getInterruptName()` von der dazugehörigen `InterruptSource_I`-Implementierung angefordert wurde. Ein einzelner IRQ kann selektiert werden und mittels der darunter befindlichen Buttons in seiner Priorität schrittweise erhöht oder vermindert werden. Die anderen Interrupts ordnen sich entsprechend neu ein.

Die Reihenfolge der IRQ-Anordnung und die IO-Adresse werden von den `Settings` gespeichert und können damit bei einem Programmstart ohne Probleme restauriert werden. Details dazu sind in den Beschreibungen des Backends zu finden.

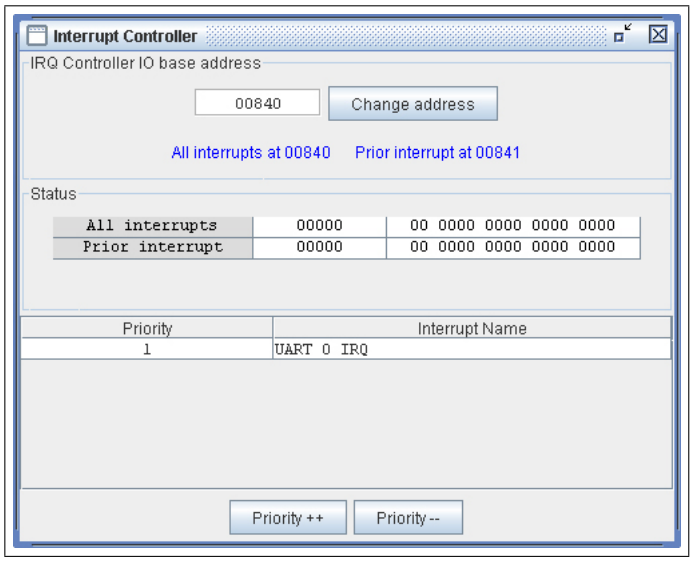

Abbildung 4.11: Screenshot Interrupt-Controller

Im Packet `gui.interrupt` befinden sich die folgenden Klassen:

`InterruptView`
`InterruptViewDefinitions_I`
`InterruptNorthPanel`
`InterruptTableModel`
`InterruptCellRenderer`

4.7 IO-Ansichten

Alle Peripherie-Ansichten sind, wie zu Anfang dieses Kapitels bereits erläutert, von den Instanzen abhängige Visualisierungen. Sie erzeugen, vorzugsweise in ihrem Konstruktor, die korrespondierende Backend-Komponente und übergeben ihr u.a. den klassenweit eindeutigen Geräteindex. Dieser Index ist der kleinste ungenutzte Zahlenwert; dementsprechend müssen alle gegenwärtig genutzten Indizes in einer statischen Liste verwaltet werden. Der Index eines genutzten Gerätes bleibt bei einem Programmende als Nutzer-Präferenz selbstverständlich erhalten.

Beim Schließen des `SubWindows` wird die Ansichtsklasse enstprechend aufgerufen. Sie hat die dazugehörige Komponente via `tearDown()` aufzulösen.

Wie bereits erwähnt, kann jeder Schreib- oder Lesezugriff auf eine IO-Adresse Seiteneffekte

hervorrufen. Der GUI der entsprechenden Peripherie-Komponente müssen Funktionen zur Verfügung gestellt werden, mit welchen Registerwerte ausgelesen oder modifiziert werden können, *ohne* dass dabei anderweitig Auswirkungen auftreten. Mit anderen Worten: Der Objektzustand darf bei einem Lesezugriff keinesfalls verändert werden, bei einem Schreibzugriff ausschließlich der entsprechende Wert.

4.7.1 UartView

Die UART-Schnittstelle wird durch die `UartView` dargestellt. Sie besteht aus zwei zentralen Teilen: der Register-Ansicht (Abbildung 4.12) und der Konsolen-Ansicht (Abbildung 4.13).

Die Register-Ansicht offeriert die folgenden Visualisierungen:

- Das Status- und das Control-Register werden jeweils mit einem Tabellen-Widget hexadezimal und binär angezeigt. Die einzelnen gesetzten Bits werden interpretiert, und deren Bedeutungen werden zusätzlich als Texthinweise angezeigt, z.B. „No parity“, falls kein Paritätsbit gesetzt worden ist. Die Werte des Control-Registers können manuell abgeändert werden, das Status-Register ist read-only.

- Es werden das FIFO-Empfangs-Register als auch das FIFO-Sende-Register samt dem jeweils dazugehörigen Schieberegister dargestellt. Eingehende Bytes werden von rechts nach links durch die Tabelle geschoben, ausgehende Byte wiederum von links nach rechts. Unbelegte Schieberegister werden mit „- -“ gekennzeichet. Um den jeweiligen FIFO-Index-Zeiger darzustellen, wird der Wert, auf welchen gegenwärtig verwiesen wird, fett hervorgehoben.

- Die IO-Basis-Adresse wird angezeigt und kann gegebenenfalls verändert werden. Die sich anhand der Basis ausrichtenden UART-Adressen werden simultan dazu berechnet und ausgegeben.

- Ein weiterer Bereich zeigt den gegenwärtigen Interrupt-Zustand an. Liegen intern die Bedingungen für ein Rx- oder Tx-Interrupt-Signal vor, so erscheint eine entsprechende Hinweismeldung. Ist zudem das dazugehörige Interrupt-Enable-Bit gesetzt, wird der Text „Interrupt Requested“ dargestellt (andernfalls „No Interrupt“). Genau dann liegt

eine Interruptforderung nach außen an, und der Interrupt-Controller wird darüber mit dem nächsten Takt informiert.

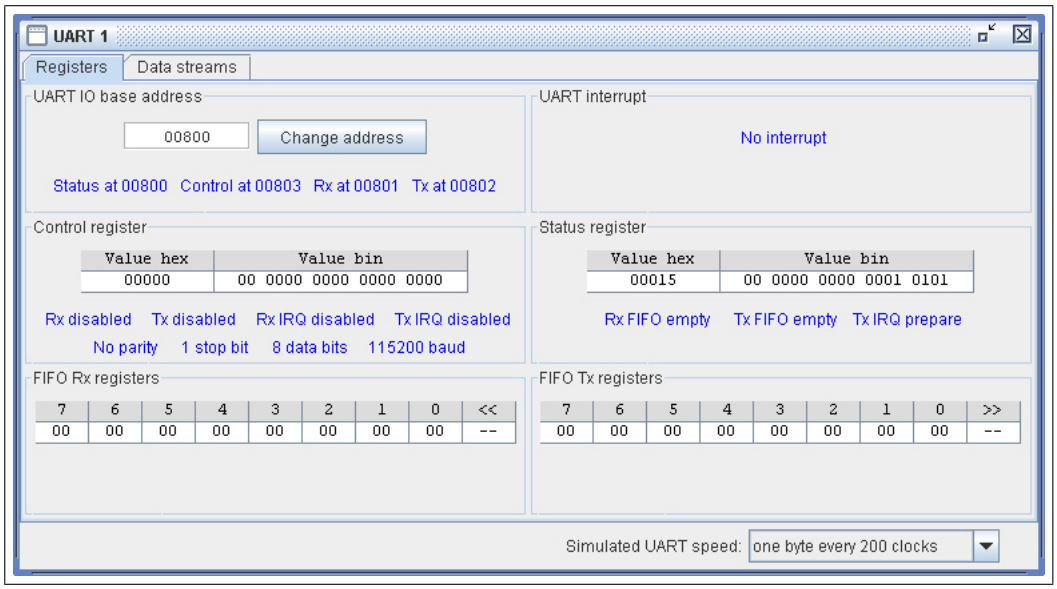

Abbildung 4.12: Screenshot UART Register

Theoretisch hätte man es zulassen können, das Statusregister, beide FIFO-Index-Zeiger sowie die beiden Schieberegister durch den Benutzer manipulieren zu können. Dadurch wäre es jedoch vom Prinzip her möglich, inkonsistente Zustände zu erzeugen, was durch zusätzliche nicht-triviale Bedingungen unterbunden werden müsste.

Die Konsolen-Ansicht knüpft an das Interface `UartStream_I` an und stellt die gesendeten Bytes als ASCII-Text im Empfangs-Textfeld dar. Ferner bietet es die Möglichkeit, Textzeichen in das Sende-Textfeld einzufügen. Die Bytes werden dann nacheinander an das UART übermittelt. Die Interface-Funktion `onTransmitNextByte()` ist mit dem Empfangsfeld gekoppelt, die Funktionen `hasNextByte()` und `onReceiveNextByte()` mit dem Sendefeld.

Es ist nicht möglich, eingegebene Zeichen aus dem Empfangs-Textfeld wieder zu entfernen. Da intern ein Zeichenzeiger mitgeführt wird, welcher protokolliert, was bereits gesendet worden ist und was noch ansteht, könnte ein Löschvorgang einen fehlerhaften Zustand provozieren.

Zugunsten der Bedienungsfreundlichkeit können zu sendende Textzeichen auch in das (eigentlich schreibgeschützte) Empfangsfeld eingegeben werden; die Tastatur-Events werden direkt an das Eingabefeld weitergeleitet.

Die Textbereiche sind abgeleitet von `javax.swing.JTextArea`. Diese Klasse benutzt als Newline-Zeichen prinzipiell `0A`. Mittels einer Auswahlbox kann jedoch zwischen den Varianten `0A`, `0D` sowie `0D0A` gewählt werden. Zu sendende Zeilenumbrüche werden abgefangen und gegebenenfalls entsprechend der Auswahl nachträglich modifiziert. Dem gegenüber werden alle empfangenen Varianten von Zeilenumbrüchen wieder starr zu `0A` konvertiert.

Es kann eine Datei angegeben werden, welche gesendet werden soll. Der Dateiinhalt wird dabei als Text interpretiert und zeilenweise in das Eingabefeld geschrieben. Nach dem Erreichen des Dateiendes wird eine zusätzliche Leerzeile hinzugefügt. Diese Funktionalität wird insbesondere benötigt, um über den Spartan-Monitor Programme laden zu können.

Mittels des Buttons Clear Streams können beide Textbereiche gelöscht werden. Da, wie oben erwähnt, die UART direkt mit den Textbuffern kommuniziert, werden die Textfelder bei einem Simulations-Neustart *nicht* gelöscht. Es liegt keine Backend-Komponente zugrunde, die auf ein `reset()` reagieren könnte. [2]

Am unteren Fensterrand befindet sich ein Auswahlmenü, mit dessen Hilfe die Übertragungsrate der Simulation eingestellt werden kann. Default-Wert ist ein Byte(!) aller 200 Takte; damit lassen sich auch längere Dateien im Allgemeinen hinreichend langsam versenden.

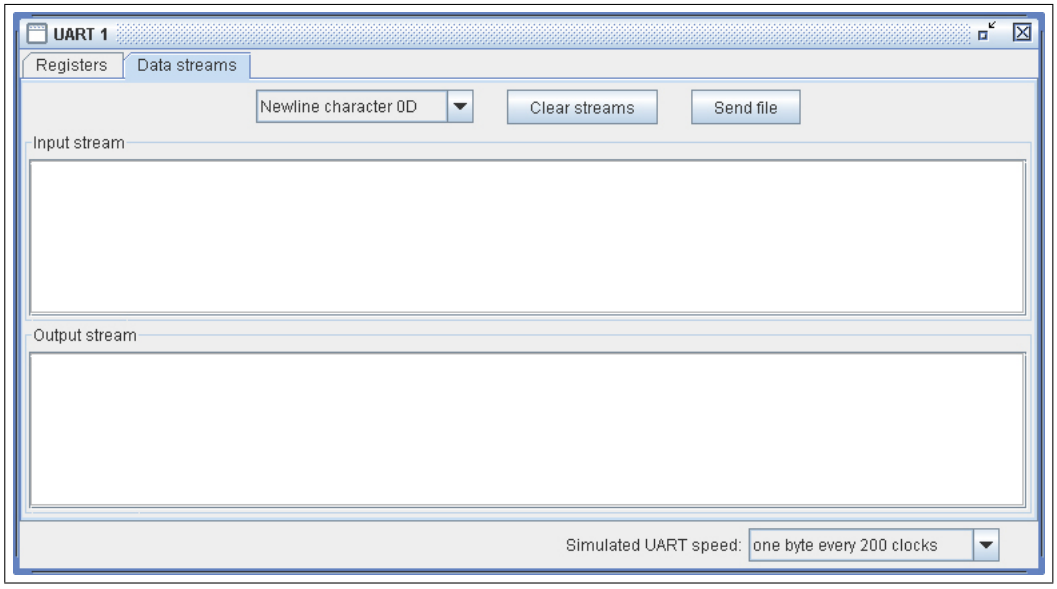

Abbildung 4.13: Screenshot UART Streams (Konsole)

[2] Man beachte: Die GUI-Fenster kennen weder Setup noch Restart, sie vermögen nur, ihre Ansicht zu aktualisieren.

Das Packet `gui.uart` setzt sich wie folgt zusammen:

`UartView`	`UartViewDefinitions_I`
`UartRegisterPanel`	`UartStreamPanel`
`UartButtonPanel`	`UartAddressSubPanel`
`UartInterruptSubPanel`	`UartControlSubPanel`
`UartStatusSubPanel`	`UartFifoRxSubPanel`
`UartFifoTxSubPanel`	

4.7.2 DmaDemonstratorView

Der DMA-Demonstrator im Packet `gui.dma` dient nur als einfaches Anschauungsobjekt und besitzt ausschließlich die GUI-Klasse `DmaDemonstratorView` (Abbildung 4.14). Im oberen Teil der Anzeige wird eine Textbeschreibung angezeigt. Im unteren Teil befindet sich der aktuelle Wert des DMA-Zeigers sowie ein Hinweis-String, ob gegenwärtig ein Interrupt angefordert wird oder nicht.

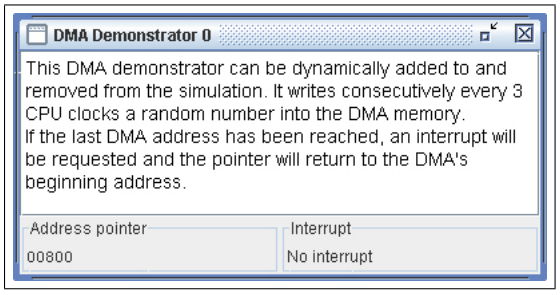

Abbildung 4.14: Screenshot DMA-Demonstrator

Um das Verständnis für die Entwicklung eigener GUI-Komponenten zu verbessern, soll auf die Details näher eingegangen werden.

Dem Konstruktor wird, wie jedem `AbstractView`, der Simulationsmanager, das Hauptfenster sowie die Nutzer-Präferenzen (bei einem neuen Fenster **`null`**) übergeben. Zunächst müssen die Präferenzen bzw. die Standardeinstellungen geladen werden. In diesem Zusammenhang wird auch die Backend-Instanz `dmaDemonstrator` erzeugt, inklusive einer eindeutigen ID. Danach müssen die entsprechenden Widgets hinzugefügt werden. Abschließend wird die Ansicht bei ihrem zugrundeliegendem Modell, dem Demonstrator, für Refreshings registriert.

Die initialen Belegungen der Widgets müssen bei ihrer Erzeugung nicht mit angegeben wer-

den. Sobald die Ansicht vollständig ist, wird sie mit ihrem `SubWindow` dem Hauptfenster eingegliedert. Dieses veranlasst unmittelbar darauf eine globale GUI-Aktualisierung für alle Fenster einschließlich dieser Klasse selbst. Dadurch werden alle Widgets von Beginn an korrekt dargestellt. Da eine weitere vollständige Aktualisierung mit dem Beenden der Ansicht induziert wird, ist damit insbesondere die Konsistenz der GUI des Interrupt-Controllers gewährleistet. Diese zeigt jeden DMA-Demonstrator in ihrer Liste an und entfernt ihn wiederum, sobald er geschlossen wird.

```
public DmaDemonstratorView( final SimulationManager simulationMgr,
                            final MainWindow mainWnd,
                            final Preferences prefs) {

    super();
    this.mainWnd = mainWnd;
    this.simulationMgr = simulationMgr;

    loadConfiguration(prefs);
    setName("DMA Demonstrator " + dmaDemonstrator.getIndex());

    /* Hier werden die GUI-Widgets erzeugt und dem Panel hinzugefuegt. */
    /* [...]                                                           */

    dmaDemonstrator.addRefreshListener(this);
}
```

Aus den Präferenzen wird einzig der Geräte-Index gewonnen. Sind die Präferenzen gleich **`null`** oder kann der entsprechende Eintrag nicht gefunden werden, so wird ein neuer Index gesucht. Anschließend wird die Demonstrator-Instanz erzeugt. Dieser wird der gefundene Index übergeben; die anderen notwendigen Parameter lassen sich einfach anhand des Managers anfordern. Schlussendlich wird der zu verwendende Geräte-Index in der statischen Klassenvariable (`java.util.HashSet`) abgelegt. Beim Speichern der Präferenzen muss nur der Index gesichert werden.

```
@Override
public void storeConfiguration(final Preferences prefs) {
    prefs.putInt("index", dmaDemonstrator.getIndex());
}

@Override
protected void loadConfiguration(final Preferences prefs) {

    final int unitIndex;
    if (prefs != null)  unitIndex = prefs.getInt("index", findMeAnUnusedIndex());
    else                unitIndex = findMeAnUnusedIndex();

    dmaDemonstrator = new DmaDemonstrator(unitIndex, simulationMgr.getIoManager(),
                simulationMgr.getDmaMemory(), simulationMgr.getInterruptController());

    DMA_DEMONSTRATOR_VIEW_INDEX_SET.add(new Integer(unitIndex));
}
```

Wird das Fenster wieder geschlossen, so springt die Ausführung in `onViewDispose()`. Die Funktion muss sich als `RefreshListener_I` wieder abmelden und das DMA-Gerät „herunterfahren“. Der genutzte Geräte-Index wird aus der statischen Liste wieder entfernt.

```
@Override
public void onViewDispose() {

    dmaDemonstrator.removeRefreshListener(this);
    dmaDemonstrator.tearDown();
    DMA_DEMONSTRATOR_VIEW_INDEX_SET.remove(new Integer(dmaDemonstrator.getIndex()));
}
```

Bei einer Aktualisierung wird der Parameter `address` ignoriert und alle (beiden) Werte aktualisiert. Zunächst wird der DMA-Zeiger abgefragt und angezeigt, danach der IRQ-Zustand. Wird kein DMA-Bereich genutzt, so wird der Zeiger als „Invalid“ dargestellt. Der Verweis auf die Standard-Schriftart ist prinzipiell zu aktualisieren, da nicht bekannt ist, ob das Refreshing gegebenenfalls eine Konsequenz von Einstellungsmodifikationen ist.

```
public void refresh(final int address) {

    pointerLabel.setFont(Settings.REGISTER_FONT_PLAIN);

    final int addr = dmaDemonstrator.getDmaPointer();
    if (addr != INVALID)    pointerLabel.setText(String.format("%05x", addr));
    else                    pointerLabel.setText("invalid");

    if (dmaDemonstrator.hasInterruptRequest())  irqLabel.setText("Interrupt requested");
    else                                        irqLabel.setText("No interrupt");
}
```

4.8 StatisticsView

Während der Simulationsabarbeitung werden Statistiken über die genutzten CPU-Befehle mitgeführt. Es ist jederzeit möglich, die dazugehörigen Übersichten einzusehen (siehe Abbildung 4.15). Es sei betont, dass das Schließen eines Statistik-Fensters natürlich keinerlei Einfluss auf die Statistik selbst hat; sie wird dadurch nicht gelöscht, sondern kann jederzeit wieder geöffnet werden. Um die Statistiken manuell zu löschen, steht der Button CLEAR zur Verfügung.

Es gibt zwei Tabellen. Erste zeigt die Anzahl der (durch IF/ID geholten) Instruktionen pro Instuktionstyp sowie die Gesamtzahl aller Befehle an. Zusätzlich wird der prozentuale Anteil berechnet. Die zweite Tabelle zeigt die Zahl aller möglichen Kombinationen von Folgebefehlen an. Die Tabellenzeilen repräsentieren dabei den ersten Befehl, die Spalten den zweiten. Mit anderen Worten: Trifft man in Zeile drei (`SUB`) und Spalte fünf (`MULI`) auf den Wert acht, so heißt dies, dass während der bisherigen Simulation die Situation, dass der Befehl `SUB` geholt wurde und unmittelbar darauf als nächstes der Befehl `MULI`, genau acht Mal aufgetreten ist.

Da die Ansicht selbst bei maximaler Vergrößerung nicht sehr ansprechend ist, besteht die Möglichkeit, die aktuellen Statistiken zu exportieren. Dabei wird, für jede Tabelle separat, jeweils eine HTML-Datei sowie eine CSV-Datei erstellt. Die HTML-Darstellung zeigt eine schlichte, übersichtliche Tabelle mit sämtlichen Werten an. Die CSV-Datei stellt den Tabelleninhalt mit Semikolon getrennt zeilenweise als Textzeichen dar. Es ist recht einfach, diese Datei anschließend in ein Tabellenkalkulations-Programm einzulesen.

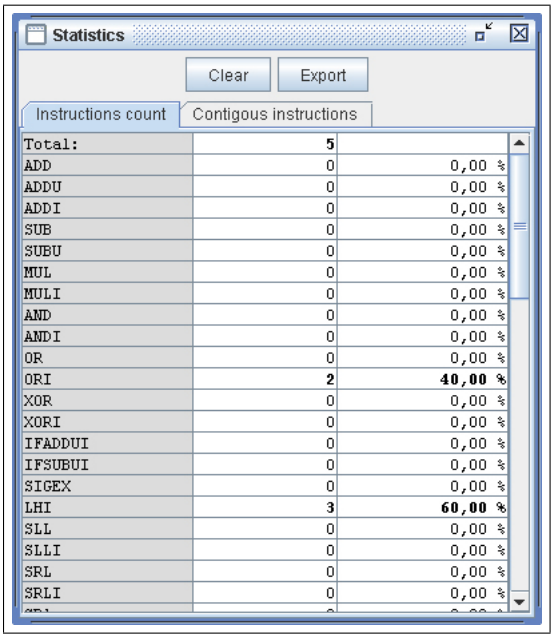

Abbildung 4.15: Screenshot Statistiken

Das Packet `gui.statistics` besteht aus den folgenden Klassen:

`StatisticsView`
`StatisticsViewDefinitions_I`
`FirstStatisticsTableModel`
`SecondStatisticsTableModel`
`FirstStatisticsCellRenderer`
`SecondStatisticsCellRenderer`
`StatisticsExporter`

5 Benutzungshinweise

Um einen schnellen Einstieg in die Benutzung des Simulators zu finden, sollen hier kurz die wichtigsten Funktionen vorgestellt werden.

Beim ersten Start der Anwendung erscheint ein leerer Arbeitsbereich.

1. Öffnen Sie über das Menü Window ▷ Assembly ein Assembly-Fenster. Beschränken Sie den Scope auf den Bereich `0x00000` bis `0x00020`. Sie können die Hexadezimalwerte mit oder ohne entsprechendem Präfix eingeben.

2. Fügen Sie eine Speicheransicht Window ▷ Memory hinzu. Sie sehen nun den Inhalt des Hauptspeichers. Jede Speicherzelle ist mit null initialisiert.

3. Öffnen Sie via File ▷ Load File die auf der CD befindliche Datei `example.sph`. Sie wird in den Hauptspeicher ab Adresse `0x00000` geladen. Beachten Sie die veränderten Werte in beiden Fenstern. Sie können in der Speicheransicht die verschiedenen Adressierungsformate wechseln, indem Sie die entsprechende Registerkarte auswählen. Bei der Code-Ansicht werden alle Speicherzellen als Befehl interpretiert. Es ist das folgende Programm zur Addition und Subtraktion jeweils zweier Ganzzahlen geladen worden:

```
.text (0x0)
start:  lhi     r1, A >> 9
        ori     r1, A & 0x1ff   ; Laden Adresse A = 0x210 nach r1
        l18     r2, 2(r1)       ; Laden b nach r2
        l18     r3, 4(r1)       ; Laden c nach r3
        add     r3, r2
        s18     0(r1), r3       ; a = b + c
        l18     r2, 8(r1)
        l18     r3, 10(r1)
        sub     r2, r3
        s18     6(r1), r2       ; d = e - f
stop:   j       stop            ; Endlosschleife

.data (0x210)
A:      .w18    0x0
B:      .w18    0x3
C:      .w18    0x5
D:      .w18    0x0
E:      .w18    0x9
F:      .w18    0x6
```

Die eigentlichen Instruktionen gehen bis Adresse `0x0000a`. Die Datenworte auf `0x00011`, `0x00012`, `0x00014` und `0x00015` werden jedoch gleichermaßen als (in diesem Fall nicht sinnvolle) Befehle interpretiert. Setzen Sie auf den Befehl bei `0x0000a` einen Breakpoint, indem Sie auf den linken Bereich der entsprechenden Zeile klicken.

4. Die CPU-Register lassen sich über Window ▷ Registers anzeigen. Beachten Sie, dass der Wert für den nächstgültigen PC `0x00000` ist. Alle Special-Function-Register sind gleichermaßen null. Die Register-File im rechten Bereich zeigt in der oberen Zeile die vier globalen Register. Darunter befinden sich je vier Register auf insgesamt drei Zeilen: IN, LOCAL und OUT. Verändern Sie manuell einen Wert im `WND`-Feld bei den SFR, um zu sehen, wie sich das Register-Fenster verschiebt.

5. Unter Window ▷ Pipeline können Sie eine Ansicht der Befehls-Pipeline öffnen. In den drei Kästen unterhalb der Grafik werden, sobald die Ausführung beginnt, die entsprechenden Befehle sowie die wichtigsten Pipeline-Register abgebildet.

6. Führen Sie mittels der Button Step auf der Steuerleiste oder unter Control ▷ Step insgesamt drei Simulationstakte aus. Im Assembly-Fenster werden die Instruktionen, die sich in der Pipeline befinden, farblich hervorgehoben. Nach den drei Schritten

befinden sich die ersten drei Befehle in den Pipeline-Stufen. Klicken Sie auf das Kontrollkästchen DETAILS, um die Einzelheiten aus- bzw. wieder einzublenden.

7. In der Steuerleiste können Sie die Geschwindigkeit der automatisierten Abarbeitung wählen. Stellen Sie den kleinsten (rechten) Wert ein. Starten Sie die Simulation mittels RUN. Die Ausführung ist langsam genug, um ihr mit bloßem Auge folgen zu können. Das Programm wird linear (d.h. ohne Sprünge oder Verzweigungen) bis `0x0000a` abgearbeitet. Danach stopt die Simulation, und es erscheint ein Hinweisfenster, dass der von Ihnen definierte Breakpoint erreicht wurde. Dies bedeutet, dass IF/ID den Befehl `J` geladen hat.

8. Deaktivieren Sie den Breakpoint. Führen Sie anschließend zwei weitere Schritte manuell aus, bis sich alle Befehle in der Endlosschleife bei `0x0000a` befinden.

9. Das Ergebnis der Addition von Speicherplatz `0x00011` (drei) und `0x00012` (fünf) ist mit dem Wert acht auf `0x00010` abgelegt worden. Analog dazu liegt das Ergebnis der Subtraktion, drei, auf Adresse `0x00013`. Die Adressangaben beziehen sich auf das Format 18 BIT DEFAULT ADDRESSING. Beachten Sie, im Assembler-Code stehen die Adressen im Format 18 BIT DATA ADDRESSING. Wechseln Sie in die dazugehörige Registerkarte in der Speicheransicht, um zu sehen, wie sich die Adressierung ein und derselben Speicherplätze verschiebt.

6 Bewertung und Erweiterungsmöglichkeiten

Die kleinste darstellbare Zeiteinheit der Simulation ist der Prozessortakt. Dieser wird durch den entsprechenden Methodenaufruf realisiert. Es ergeben sich dadurch zwei Nachteile:

- Es ist nicht möglich, eine kleinere Zeiteinheit umzusetzen, z.B. für schnelle IO-Geräte. Größere Zeiteinheiten können zudem nur ganzzahlige Vielfache des Prozessortaktes sein.

- Asynchrone Ereignisse (relativ zum Prozessortakt) sind prinzipiell nicht beliebig darstellbar.

Bei der Umsetzung wurde das Augenmerk auf Stabilität und die Korrektheit der Abarbeitung gelegt. Es hat sich gezeigt, dass die Swing-Bibliotheken einen erheblichen Performance-Verlust darstellen. Die Abarbeitung wird durch die leitungsfordernde GUI merklich träge. Im Rahmen von Erweiterungen bietet es sich an, an der Performanz der Gesamtanwendung weiter zu arbeiten. Insbesondere das Refreshing-Management bietet noch einige Ansätze, welche näher betrachtet werden können.

Funktionalitäten sind stets in Backend und Frontend gekapselt worden. Einzige Ausnahme ist die Klasse `Settings`. Hier wurde zugunsten der Einfachheit und Übersichtlichkeit die strikte Trennung verletzt. Es wäre darüber nachzudenken, inwiefern es sinnvoll ist, zwischen Attributen des Backends und Attributen des Frontends zu differenzieren, und die dazugehörige Lade-/Speicherfunktionalität im Rahmen des Präferenzen-Managements auszulagern.

Bei der Implementierung der Watchpoints und Breakpoints sind Registerzugriffe anhand der Enumeration `AccessType` als `TRANSPARENT`, `BREAKPOINT` und `WATCHPOINT` differenziert worden. An vielen Stellen, insbesondere bei IO-Geräten, ist es nötig, manuelle Registeränderungen (via GUI) ohne die sonst zwangsläufigen Seiteneffekte zu ermöglichen. Die gegenwärtige Implementierung offeriert in solchen Fällen stets separate Zugriffsmethoden. Eine bessere Lösung wäre es, das Konzept der Zugriffsklassifikation konsistent (und nicht notwendigerweise an einen AccessManager gebunden) bei *sämtlichen* Komponenten umzusetzen. Jeder lesende oder schreibende Registerzugriff müsste als transparent oder nicht-transparent klassifiziert werden. Die GUI könnte damit die gleichen Methoden nutzen, ohne dass sie Seiteneffekte auslöst.

A Vollständiger Befehlssatz

In der nachfolgenden Tabelle sind sämtliche Befehle aufgelistet, die vom Simulator ausgeführt werden können. Aufgeteilt auf die drei Pipeline-Stufen sind die relevanten Auswirkungen, d.h. die für die Funktionsweise essentiellen Registerbelegungen, dokumentiert. Sie finden sich in der Umsetzung des Simulators wieder. Zusätzlich zu den hier aufgeführten Registern beinhaltet die IF/ID-Stufe den aktuellen Program Counter PC sowie das Instruction Register IR; beide werden nachfolgend zugunsten der Übersichtlichkeit weggelassen.

Als letzte Abarbeitungsanweisung wird das Verhalten bei einem Hardware-Interrupt aufgeführt. Es sei angemerkt, dass `Interrupt` kein Teil des regulären Befehlssatzes ist.

Befehl	Instruction Fetch & Decode	Execute & Memory	Write Back
ADD **ADDU**	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) reg_no_b ← B alu_opb ← R(B)	reg_no_a ← A alu_op_a ← R(A) alu_op_b ← R(B) alu_result ← R(A) + R(B)	do_wb ← 1 write_reg_number ← A write_reg_data ← alu_result
ADDI	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) imm ← B_{signed} alu_opb ← imm	reg_no_a ← A alu_op_a ← R(A) alu_op_b ← imm alu_result ← R(A) + imm	do_wb ← 1 write_reg_number ← A write_reg_data ← alu_result
SUB **SUBU**	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) reg_no_b ← B alu_opb ← R(B)	reg_no_a ← A alu_op_a ← R(A) alu_op_b ← R(B) alu_result ← R(A) - R(B)	do_wb ← 1 write_reg_number ← A write_reg_data ← alu_result
MUL	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) reg_no_b ← B alu_opb ← R(B)	reg_no_a ← A alu_op_a ← R(A) alu_op_b ← R(B) alu_result ← $(R(A) * R(B))_{17..0}$ mul ← $(R(A) * R(B))_{35..18}$	do_wb ← 1 write_reg_number ← A write_reg_data ← alu_result
MULI	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) imm ← B_{signed} alu_opb ← imm	reg_no_a ← A alu_op_a ← R(A) alu_op_b ← imm alu_result ← $(R(A) * imm)_{17..0}$ mul ← $(R(A) * B)_{35..18}$	do_wb ← 1 write_reg_number ← A write_reg_data ← alu_result
AND	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) reg_no_b ← B alu_opb ← R(B)	reg_no_a ← A alu_op_a ← R(A) alu_op_b ← R(B) alu_result ← R(A) & R(B)	do_wb ← 1 write_reg_number ← A write_reg_data ← alu_result

Befehl	Instruction Fetch & Decode	Execute & Memory	Write Back
ANDI	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) imm ← $B_{unsigned}$ alu_opb ← imm	reg_no_a ← A alu_op_a ← R(A) alu_op_b ← imm alu_result ← R(A) & imm	do_wb ← 1 write_reg_number ← A write_reg_data ← alu_result
OR	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) reg_no_b ← B alu_opb ← R(B)	reg_no_a ← A alu_op_a ← R(A) alu_op_b ← R(B) alu_result ← R(A) | R(B)	do_wb ← 1 write_reg_number ← A write_reg_data ← alu_result
ORI	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) imm ← $B_{unsigned}$ alu_opb ← imm	reg_no_a ← A alu_op_a ← R(A) alu_op_b ← imm alu_result ← R(A) | imm	do_wb ← 1 write_reg_number ← A write_reg_data ← alu_result
XOR	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) reg_no_b ← B alu_opb ← R(B)	reg_no_a ← A alu_op_a ← R(A) alu_op_b ← R(B) alu_result ← R(A)^R(B)	do_wb ← 1 write_reg_number ← A write_reg_data ← alu_result
XORI	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) imm ← $B_{unsigned}$ alu_opb ← imm	reg_no_a ← A alu_op_a ← R(A) alu_op_b ← imm alu_result ← R(A)^imm	do_wb ← 1 write_reg_number ← A write_reg_data ← alu_result
IFADDUI	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) imm ← $B_{unsigned}$ alu_opb ← imm	reg_no_a ← A alu_op_a ← R(A) alu_op_b ← imm alu_result ← R(A) + imm	do_wb ← $\begin{cases} 1 & \text{wenn cc == 1} \\ 0 & \text{sonst} \end{cases}$ write_reg_number ← A write_reg_data ← alu_result

Befehl	Instruction Fetch & Decode	Execute & Memory	Write Back
IFSUBUI	pc_new $\leftarrow$ pc + 1 reg_no_a $\leftarrow$ A alu_opa $\leftarrow$ R(A) imm $\leftarrow$ $B_{unsigned}$ alu_opb $\leftarrow$ imm	reg_no_a $\leftarrow$ A alu_op_a $\leftarrow$ R(A) alu_op_b $\leftarrow$ imm alu_result $\leftarrow$ R(A) - imm	do_wb $\leftarrow \begin{cases} 1 & \text{wenn cc} == 1 \\ 0 & \text{sonst} \end{cases}$ write_reg_number $\leftarrow$ A write_reg_data $\leftarrow$ alu_result
SIGEX	pc_new $\leftarrow$ pc + 1 reg_no_a $\leftarrow$ A alu_opa $\leftarrow$ R(A) imm $\leftarrow$ $B_{unsigned}$ alu_opb $\leftarrow$ imm	reg_no_a $\leftarrow$ A alu_op_a $\leftarrow$ R(A) alu_op_b $\leftarrow$ imm alu_result $\leftarrow \begin{cases} (R(A)_{15})^2 \#\# R(A)_{15..0} & \text{wenn } B_{4..0} == 16 \\ (R(A)_8)^9 \#\# R(A)_{8..0} & \text{wenn } B_{4..0} == 9 \\ (R(A)_7)^{10} \#\# R(A)_{7..0} & \text{sonst} \end{cases}$	do_wb $\leftarrow$ 1 write_reg_number $\leftarrow$ A write_reg_data $\leftarrow$ alu_result
LHI	pc_new $\leftarrow$ pc + 1 reg_no_a $\leftarrow$ A alu_opa $\leftarrow$ R(A) imm_high $\leftarrow$ B## 0^9 alu_opb $\leftarrow$ imm_high	reg_no_a $\leftarrow$ A alu_op_a $\leftarrow$ R(A) alu_op_b $\leftarrow$ imm_high alu_result $\leftarrow$ imm_high	do_wb $\leftarrow$ 1 write_reg_number $\leftarrow$ A write_reg_data $\leftarrow$ alu_result
SLL	pc_new $\leftarrow$ pc + 1 reg_no_a $\leftarrow$ A alu_opa $\leftarrow$ R(A) reg_no_b $\leftarrow$ B alu_opb $\leftarrow$ R(B)	reg_no_a $\leftarrow$ A alu_op_a $\leftarrow$ R(A) alu_op_b $\leftarrow$ R(B) alu_result $\leftarrow$ R(A) << $R(B)_{4..0}$	do_wb $\leftarrow$ 1 write_reg_number $\leftarrow$ A write_reg_data $\leftarrow$ alu_result
SLLI	pc_new $\leftarrow$ pc + 1 reg_no_a $\leftarrow$ A alu_opa $\leftarrow$ R(A) imm $\leftarrow$ $B_{unsigned}$ alu_opb $\leftarrow$ imm	reg_no_a $\leftarrow$ A alu_op_a $\leftarrow$ R(A) alu_op_b $\leftarrow$ imm alu_result $\leftarrow$ R(A) << $imm_{4..0}$	do_wb $\leftarrow$ 1 write_reg_number $\leftarrow$ A write_reg_data $\leftarrow$ alu_result

Befehl	Instruction Fetch & Decode	Execute & Memory	Write Back
SRL	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) reg_no_b ← B alu_opb ← R(B)	reg_no_a ← A alu_op_a ← R(A) alu_op_b ← R(B) alu_result ← R(A) >> $R(B)_{4..0}$	do_wb ← 1 write_reg_number ← A write_reg_data ← alu_result
SRLI	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) imm ← $B_{unsigned}$ alu_opb ← imm	reg_no_a ← A alu_op_a ← R(A) alu_op_b ← imm alu_result ← R(A) >> $imm_{4..0}$	do_wb ← 1 write_reg_number ← A write_reg_data ← alu_result
SRA	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) reg_no_b ← B alu_opb ← R(B)	reg_no_a ← A alu_op_a ← R(A) alu_op_b ← R(B) alu_result ← $(R(A)_{17})^{hinreichend}$##R(A) >> $R(B)_{4..0}$	do_wb ← 1 write_reg_number ← A write_reg_data ← alu_result
SRAI	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) imm ← $B_{unsigned}$ alu_opb ← imm	reg_no_a ← A alu_op_a ← R(A) alu_op_b ← imm alu_result ← $(R(A)_{17})^{hinreichend}$##R(A) >> $imm_{4..0}$	do_wb ← 1 write_reg_number ← A write_reg_data ← alu_result
SEQ **SEQU**	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) reg_no_b ← B alu_opb ← R(B)	alu_op_a ← R(A) alu_op_b ← R(B) cc ← $\begin{cases} 1 & \text{wenn R(A) == R(B)} \\ 0 & \text{sonst} \end{cases}$	do_wb ← 0
SEQI	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) imm ← B_{signed} alu_opb ← imm	alu_op_a ← R(A) alu_op_b ← imm cc ← $\begin{cases} 1 & \text{wenn R(A) == imm} \\ 0 & \text{sonst} \end{cases}$	do_wb ← 0

Befehl	Instruction Fetch & Decode	Execute & Memory	Write Back
SNE **SNEU**	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) reg_no_b ← B alu_opb ← R(B)	alu_op_a ← R(A) alu_op_b ← R(B) cc ← $\begin{cases} 1 & \text{wenn R(A) != R(B)} \\ 0 & \text{sonst} \end{cases}$	do_wb ← 0
SNEI	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) imm ← B_{signed} alu_opb ← imm	alu_op_a ← R(A) alu_op_b ← imm cc ← $\begin{cases} 1 & \text{wenn R(A) != imm} \\ 0 & \text{sonst} \end{cases}$	do_wb ← 0
SLT	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) reg_no_b ← B alu_opb ← R(B)	alu_op_a ← R(A) alu_op_b ← R(B) cc ← $\begin{cases} 1 & \text{wenn R(A) < R(B)} \\ 0 & \text{sonst} \end{cases}$	do_wb ← 0
SLTI	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) imm ← B_{signed} alu_opb ← imm	alu_op_a ← R(A) alu_op_b ← imm cc ← $\begin{cases} 1 & \text{wenn R(A) < imm} \\ 0 & \text{sonst} \end{cases}$	do_wb ← 0
SLTU	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) reg_no_b ← B alu_opb ← R(B)	alu_op_a ← R(A) alu_op_b ← R(B) cc ← $\begin{cases} 1 & \text{wenn R(A)}_{unsigned} < \text{R(B)}_{unsigned} \\ 0 & \text{sonst} \end{cases}$	do_wb ← 0
SGT	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) reg_no_b ← B alu_opb ← R(B)	alu_op_a ← R(A) alu_op_b ← R(B) cc ← $\begin{cases} 1 & \text{wenn R(A) > R(B)} \\ 0 & \text{sonst} \end{cases}$	do_wb ← 0

Befehl	Instruction Fetch & Decode	Execute & Memory	Write Back
SGTI	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) imm ← B_{signed} alu_opb ← imm	alu_op_a ← R(A) alu_op_b ← imm cc ← $\begin{cases} 1 & \text{wenn R(A) > imm} \\ 0 & \text{sonst} \end{cases}$	do_wb ← 0
SGTU	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) reg_no_b ← B alu_opb ← R(B)	alu_op_a ← R(A) alu_op_b ← R(B) cc ← $\begin{cases} 1 & \text{wenn R(A)}_{unsigned} > \text{R(B)}_{unsigned} \\ 0 & \text{sonst} \end{cases}$	do_wb ← 0
SLE	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) reg_no_b ← B alu_opb ← R(B)	alu_op_a ← R(A) alu_op_b ← R(B) cc ← $\begin{cases} 1 & \text{wenn R(A) <= R(B)} \\ 0 & \text{sonst} \end{cases}$	do_wb ← 0
SLEI	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) imm ← B_{signed} alu_opb ← imm	alu_op_a ← R(A) alu_op_b ← imm cc ← $\begin{cases} 1 & \text{wenn R(A) <= imm} \\ 0 & \text{sonst} \end{cases}$	do_wb ← 0
SLEU	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) reg_no_b ← B alu_opb ← R(B)	alu_op_a ← R(A) alu_op_b ← R(B) cc ← $\begin{cases} 1 & \text{wenn R(A)}_{unsigned} <= \text{R(B)}_{unsigned} \\ 0 & \text{sonst} \end{cases}$	do_wb ← 0
SGE	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) reg_no_b ← B alu_opb ← R(B)	alu_op_a ← R(A) alu_op_b ← R(B) cc ← $\begin{cases} 1 & \text{wenn R(A) >= R(B)} \\ 0 & \text{sonst} \end{cases}$	do_wb ← 0

Befehl	Instruction Fetch & Decode	Execute & Memory	Write Back
SGEI	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) imm ← B_{signed} alu_opb ← imm	alu_op_a ← R(A) alu_op_b ← imm cc ← $\begin{cases} 1 & \text{wenn R(A) >= imm} \\ 0 & \text{sonst} \end{cases}$	do_wb ← 0
SGEU	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A) reg_no_b ← B alu_opb ← R(B)	alu_op_a ← R(A) alu_op_b ← R(B) cc ← $\begin{cases} 1 & \text{wenn R(A)}_{unsigned} >= \text{R(B)}_{unsigned} \\ 0 & \text{sonst} \end{cases}$	do_wb ← 0
BEQZ	pc ← $\begin{cases} \text{pc + imm} & \text{wenn R(A) == 0} \\ \text{pc + 1} & \text{sonst} \end{cases}$ reg_no_a ← A imm ← B_{signed} reg_a ← R(A)	–	do_wb ← 0
BNEZ	pc ← $\begin{cases} \text{pc + imm} & \text{wenn R(A) != 0} \\ \text{pc + 1} & \text{sonst} \end{cases}$ reg_no_a ← A imm ← B_{signed} reg_a ← R(A)	–	do_wb ← 0
BEQZC	pc ← $\begin{cases} \text{pc + offset} & \text{wenn cc == 0} \\ \text{pc + 1} & \text{sonst} \end{cases}$ offset ← A_{signed}	–	do_wb ← 0
BNEZC	pc ← $\begin{cases} \text{pc + offset} & \text{wenn cc != 0} \\ \text{pc + 1} & \text{sonst} \end{cases}$ offset ← A_{signed}	–	do_wb ← 0
J	pc ← pc + offset offset ← A_{signed}	–	do_wb ← 0

Befehl	Instruction Fetch & Decode	Execute & Memory	Write Back
JR	pc ← R(B) reg_no_b ← B reg_b ← R(B)	–	do_wb ← 0
JALR	pc ← R(B) reg_no_b ← B reg_b ← R(B) reg_no_a ← 11 alu_opb ← pc + 1	reg_no_a ← 11 alu_op_b ← pc + 1 alu_result ← pc + 1	do_wb ← 1 write_reg_number ← 11 write_reg_data ← alu_result
JALRS	pc ← R(B) reg_no_b ← B reg_b ← R(B) reg_no_a ← 11 alu_opb ← pc + 1	reg_no_a ← 11 alu_op_b ← pc + 1 alu_result ← pc + 1 window ← window + 1	do_wb ← 1 write_reg_number ← 11 write_reg_data ← alu_result
JALS	pc ← pc + offset offset ← A_{signed} reg_no_a ← 11 alu_opb ← pc + 1	reg_no_a ← 11 alu_op_b ← pc + 1 alu_result ← pc + 1 window ← window + 1	do_wb ← 1 write_reg_number ← 11 write_reg_data ← alu_result
JRS	pc ← R(B) reg_no_b ← B reg_b ← R(B)	window ← window - 1	do_wb ← 0
TRAP	pc ← A ## B reg_no_a ← 11 alu_opb ← pc + 1	reg_no_a ← 11 alu_op_b ← pc + 1 alu_result ← pc + 1 window ← window + 1	do_wb ← 1 write_reg_number ← 11 write_reg_data ← alu_result
RFE	pc ← R(B) reg_no_b ← B reg_b ← R(B) Signal an IRQ-Controller	window ← window - 1	do_wb ← 0

Befehl	Instruction Fetch & Decode	Execute & Memory	Write Back
L9	pc_new $\leftarrow$ pc + 1 reg_no_a $\leftarrow$ A reg_no_b $\leftarrow$ B disp $\leftarrow C_{unsigned}$ alu_opa $\leftarrow$ disp alu_opb $\leftarrow$ R(B)	reg_no_a $\leftarrow$ A alu_op_a $\leftarrow$ disp alu_op_b $\leftarrow$ R(B) data_addr $\leftarrow$ mm ## $(R(B)+C)_{17..1}$ alu_result $\leftarrow \begin{cases} M(data\_addr)_{17..9} & \text{wenn } (R(B)+C)_0 == 0 \\ M(data\_addr)_{8..0} & \text{sonst} \end{cases}$	do_wb $\leftarrow$ 1 write_reg_number $\leftarrow$ A write_reg_data $\leftarrow$ alu_result
L18	pc_new $\leftarrow$ pc + 1 reg_no_a $\leftarrow$ A reg_no_b $\leftarrow$ B disp $\leftarrow C_{unsigned}$ alu_opa $\leftarrow$ disp alu_opb $\leftarrow$ R(B)	reg_no_a $\leftarrow$ A alu_op_a $\leftarrow$ disp alu_op_b $\leftarrow$ R(B) data_addr $\leftarrow$ mm ## $(R(B)+C)_{17..1}$ alu_result $\leftarrow$ M(data_addr)	do_wb $\leftarrow$ 1 write_reg_number $\leftarrow$ A write_reg_data $\leftarrow$ alu_result
S9	pc_new $\leftarrow$ pc + 1 reg_no_a $\leftarrow$ A reg_no_b $\leftarrow$ B disp $\leftarrow C_{unsigned}$ alu_opa $\leftarrow$ disp alu_opb $\leftarrow$ R(B) reg_a $\leftarrow$ R(A)	alu_op_a $\leftarrow$ disp alu_op_b $\leftarrow$ R(B) data_addr $\leftarrow$ mm ## $(R(B)+C)_{17..1}$ write_data_data $\leftarrow \begin{cases} reg\_a_{8..0}\#\#M(data\_addr)_{8..0} & \text{wenn } (R(B)+C)_0 == 0 \\ M(data\_addr)_{17..9}\#\#reg\_a_{8..0} & \text{sonst} \end{cases}$	do_wb $\leftarrow$ 0
S18	pc_new $\leftarrow$ pc + 1 reg_no_a $\leftarrow$ A reg_no_b $\leftarrow$ B disp $\leftarrow C_{unsigned}$ alu_opa $\leftarrow$ disp alu_opb $\leftarrow$ R(B) reg_a $\leftarrow$ R(A)	alu_op_a $\leftarrow$ disp alu_op_b $\leftarrow$ R(B) data_addr $\leftarrow$ mm ## $(R(B)+C)_{17..1}$ write_data_data $\leftarrow$ reg_a	do_wb $\leftarrow$ 0
MOV	pc_new $\leftarrow$ pc + 1 reg_no_a $\leftarrow$ A reg_no_b $\leftarrow$ B alu_opb $\leftarrow$ R(B)	reg_no_a $\leftarrow$ A alu_op_b $\leftarrow$ R(B) alu_result $\leftarrow$ R(B)	do_wb $\leftarrow$ 1 write_reg_number $\leftarrow$ A write_reg_data $\leftarrow$ alu_result

Befehl	Instruction Fetch & Decode	Execute & Memory	Write Back
MOVI	pc_new ← pc + 1 reg_no_a ← A imm ← $B_{unsigned}$ alu_opb ← imm	reg_no_a ← A alu_op_b ← imm alu_result ← imm	do_wb ← 1 write_reg_number ← A write_reg_data ← alu_result
MOVI2C	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A)	alu_op_a ← R(A) cc ← $R(A)_0$	do_wb ← 0
MOVC2I	pc_new ← pc + 1 reg_no_a ← A alu_opa ← R(A)	reg_no_a ← A alu_op_a ← R(A) alu_result ← $R(A)_{17..1}$ ## cc	do_wb ← 1 write_reg_number ← A write_reg_data ← alu_result

MOVI2S	pc_new ← pc + 1
	reg_no_a ← A
	reg_no_b ← B
	alu_opa ← R(A)
		alu_op_a ← R(A)
		reg_no_b ← B
		sfr_addr ← $B_{2..0}$
		wenn sfr_addr == SFR_MUL:
		  mul ← R(A)
		wenn sfr_addr == SFR_LEDS:
		  leds ← $R(A)_{6..0}$
		sonst:
		  interrupt ← $R(A)_8$
		  mm ← $R(A)_7$
		  cc ← $R(A)_6$
		  window ← $R(A)_{5..0}$
			do_wb ← 0

Befehl	Instruction Fetch & Decode	Execute & Memory	Write Back

MOVS2I	pc_new ← pc + 1
	reg_no_a ← A
	reg_no_b ← B
	alu_opa ← R(A)
		reg_no_a ← A
		reg_no_b ← B
		alu_op_a ← R(A)
		sfr_addr ← $B_{2..0}$
		wenn sfr_addr == SFR_MUL:
		  alu_result ← mul
		wenn sfr_addr == SFR_LEDS:
		  alu_result ← 0^{11} ## leds
		sonst:
		  alu_result ← 0^{9} ## interrupt ## mm ## cc ## window
			do_wb ← 1
			write_reg_number ← A
			write_reg_data ← alu_result

SBITS	pc_new ← pc + 1
	reg_no_b ← B
		reg_no_b ← B
		wenn $B_{1..0}$ == 1:
		  mm ← 1
		wenn $B_{1..0}$ == 2:
		  interrupt ← 1
		sonst:
		  cc ← 1
			do_wb ← 0

CBITS	pc_new ← pc + 1
	reg_no_b ← B
		reg_no_b ← B
		wenn $B_{1..0}$ == 1:
		  mm ← 0
		wenn $B_{1..0}$ == 2:
		  interrupt ← 0
		sonst:
		  cc ← 0
			do_wb ← 0

Interrupt	pc ← Interrupt-Sprungadresse reg_no_a ← 11 alu_opb ← pc	reg_no_a ← 11 alu_op_b ← pc alu_result ← pc window ← window + 1	do_wb ← 1 write_reg_number ← 11 write_reg_data ← alu_result

B Befehlscodierungen

Die folgende Tabelle listet sämtliche durch den Simulator unterstützten Befehle in ihren Binärcodierungen auf. Es gelten die Abkürzungen a (Operand A), b (Operand B), i (Immediate), o (Offset), d (Disposition) sowie * (Don't Care).

Befehl	Instruktionswort	Befehl	Instruktionswort	Befehl	Instruktionswort
MOVI2C	00000 aaaa **** 00010	AND	00001 aaaa bbbb 10100	ANDI	01100 aaaa iiii iiiii
MOVC2I	00000 aaaa **** 00011	OR	00001 aaaa bbbb 10101	ORI	01101 aaaa iiii iiiii
SLL	00000 aaaa bbbb 00100	XOR	00001 aaaa bbbb 10110	XORI	01110 aaaa iiii iiiii
MOV	00000 aaaa bbbb 00101	MUL	00001 aaaa bbbb 10111	MULI	01111 aaaa iiii iiiii
SRL	00000 aaaa bbbb 00110	SEQ	00001 aaaa bbbb 11000	L9	10000 aaaa bbbb ddddd
SRA	00000 aaaa bbbb 00111	SNE	00001 aaaa bbbb 11001	S9	10001 aaaa bbbb ddddd
SEQU	00000 aaaa bbbb 01000	SLT	00001 aaaa bbbb 11010	L18	10010 aaaa bbbb ddddd
SNEU	00000 aaaa bbbb 01001	SGT	00001 aaaa bbbb 11011	S18	10011 aaaa bbbb ddddd
SLTU	00000 aaaa bbbb 01010	SLE	00001 aaaa bbbb 11100	SLLI	10100 aaaa iiii iiiii
SGTU	00000 aaaa bbbb 01011	SGE	00001 aaaa bbbb 11101	SRLI	10110 aaaa iiii iiiii
SLEU	00000 aaaa bbbb 01100	MOVI2S	00001 aaaa bbbb 11110	SRAI	10111 aaaa iiii iiiii
SGEU	00000 aaaa bbbb 01101	MOVS2I	00001 aaaa bbbb 11111	SEQI	11000 aaaa iiii iiiii
CBITS	00000 **** bbbb 10110	J	00010 oooo oooo ooooo	SNEI	11001 aaaa iiii iiiii
SBITS	00000 **** bbbb 10111	JALS	00011 oooo oooo ooooo	SLTI	11010 aaaa iiii iiiii
RFE	00001 **** bbbb 00000	BEQZ	00100 aaaa iiii iiiii	SGTI	11011 aaaa iiii iiiii
TRAP	00001 aaaa aaaa 00001	BNEZ	00101 aaaa iiii iiiii	SLEI	11100 aaaa iiii iiiii
JR	00001 **** bbbb 00010	BEQZC	00110 oooo oooo ooooo	SGEI	11101 aaaa iiii iiiii
JALR	00001 **** bbbb 00011	BNEZC	00111 oooo oooo ooooo	IFADDUI	11110 aaaa iiii iiiii
JRS	00001 **** bbbb 00100	ADDI	01000 aaaa iiii iiiii	IFSUBUI	11111 aaaa iiii iiiii
JALRS	00001 **** bbbb 00101	MOVI	01001 aaaa iiii iiiii	ADDU	00001 aaaa bbbb 10001
ADD	00001 aaaa bbbb 10000	LHI	01010 aaaa iiii iiiii	SUBU	00001 aaaa bbbb 10011
SUB	00001 aaaa bbbb 10010	SIGEX	01011 aaaa iiii iiiii		

Literaturverzeichnis

[1] *Rechnerarchitektur – Analyse, Entwurf, Implementierung, Bewertung* J. L. Hennessy, D. A. Patterson. Vieweg Verlag, Braunschweig/Wiesbaden, 1994.

[2] *Mikroprozessortechnik* H. Müller, L. Walz. 4. Auflage. Vogel Verlag, Würzburg, 1992.

[3] *RISC Architecture* cse.stanford.edu/class/sophomore-college/projects-00/risc

[4] *Handbuch der Java-Programmierung* G. Krüger. HTML-Ausgabe 4.0.2. Addison-Wesley, 2004.

[5] *Java ist auch eine Insel* C. Ullenboom. HTML-Ausgabe 5. Galileo Computing, 2005.

[6] Unterlagen zum Spartan Mikrocontroller. Lehrstuhl für Mikrorechner, TU Dresden, 2006. www.mr.inf.tu-dresden.de/wiki/page.php?page=SpartanMC