Bachelorarbeit

Henrik Brauer

Design einer generischen

Visualisierung für Prozessorarchitekturen in Java

Henrik Brauer

Design einer generischen Visualisierung für Prozessorarchitekturen in Java

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Michael Schäfers
Zweitgutachter : Prof. Dr. Thomas Canzler
Abgegeben am 24.08.2007

Henrik Brauer

Thema der Bachelorarbeit

Design einer generischen Visualisierung für Prozessorarchitekturen in Java

Stichwort

Prozessor, CPU, ARM, 3D Grafik, Lernprogramm, Architekturen, Java

Kurzzusammenfassung

Diese Arbeit befasst sich mit dem Design und der Implementierung einer Entwicklungsumgebung für die Visualisierung von Prozessorarchitekturen. Die Entwicklungsumgebung wird so gestaltet, dass beliebige Prozessoren implementiert werden können. Mit Hilfe der Entwicklungsumgebung wird ein interaktives Lernprogramm für den ARM Prozessor implementiert.

Henrik Brauer

Title of the paper

Design of a generic visualization for CPU design in Java

Keywords

Processor, CPU, ARM, 3D computer graphics, E-Learning, Java

Abstract

This work relates to the design and implementation of a development environment for the visualization of CPU design. The development environment is configured in such a way that any desired processor can be implemented. An interactive learning program for the ARM processor is being implemented with the help of the development environment.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Prozessoren sind ein wichtiger Bestandteil unsers Lebens geworden. Ein Prozessor ist nicht nur das Gehirn eines jeden Computers, sondern findet sich auch in mehr und mehr Gegenständen des täglichen Lebens wieder. Angefangen beim Handy oder Auto bis hin zur Kaffemaschine und dem Geschirrspüler. Als Informatiker ist es elementar wichtig die Funktionsweise eines Prozessors zu kennen. Durch das Verständnis eines Prozessors sind viele Beschränkungen, denen wir als Informatiker unterliegen, erklärbar. Diese Arbeit verfolgt das Ziel, die Funktionsweise eines Prozessors zu visualisieren, so dass sie auch für Personen, die noch keine Erfahrungen in dem Bereich haben, verständlich wird.

1.1 Motivation

Die Funktionsweise eines Prozessors ist sehr komplex, gerade das Pipelining macht es sehr schwer die Funktionsweise zu verstehen. Diese Arbeit soll beim Verständnis helfen, indem es das große komplexe System in viele kleine Teilbereiche aufteilt, die einzeln verstanden werden können. Diese Teilbereiche sollen es ermöglichen, das gesamte System leichter zu verstehen.

1.2 Ziel

Das Ziel ist es eine Software zu entwickeln, die es Studienanfängern und Interessierten ermöglicht, die Funktionsweise eines Prozessors zu verstehen, hier im Speziellen eines ARM

[Fur00] Prozessors. Insbesondere muss darauf geachtet werden, dass das Pipelining erläutert wird. Hierbei sollen auch alle Randbereiche, wie z.B. der Arbeitsspeicher, mit einbezogen werden. Es ist wichtig, dass die Software eine möglichst genaue Abstraktion des Original Prozessors darstellt. Trotzdem soll die Software nicht überladen sein.

Außerdem soll eine Umgebung geschaffen werden, die erweiterbar ist, so dass weitere Prozessoren unterschiedlicher Typen leicht integriert werden können. Zusätzlich ist es wichtig, dass viele Teile der Software so allgemein gehalten werden, dass sie in verschiedenen Prozessoren wieder verwendet werden können, damit beim Einfügen eines neuen Prozessors vorhandene Teile nicht noch einmal implementiert werden müssen.

2 Analyse

In diesem Kapitel werden zunächst die genauen Anforderungen abgesteckt. Desweiteren werden die Grundlagen vermittelt, die für das Verständnis der Lösung notwendig sind.

2.1 Anforderungen

2.1.1 Die Allgemeinen Anforderungen

Das Ziel dieser Bachelorarbeit ist es ein interaktives Lernprogramm zu erstellen.
Dieses Lernprogramm soll im Rahmen der Lehrveranstaltungen an der Hochschule für Angewandte Wissenschaften Hamburg (HAW Hamburg) genutzt werden, um Studierenden die Funktionsweise eines Prozessors zu erläutern. In diesem Fall bedeutet das konkret, dass die Funktionsweise eines ARM Prozessors implementiert werden soll.

Als zusätzliche Anforderung wurde gefordert, dass das System erweiterbar ist, so dass die Möglichkeit besteht weitere Prozessortypen zu implementieren. Hierbei soll möglichst viel übernommen werden können. Um das System zu realisieren sind im speziellen folgende Anforderungen aufgestellt worden, die erfüllt werden sollen:

- Das System soll den Original Prozessor wiederspiegeln. Bei Abweichungen soll drauf hingewiesen werden.
- Die Darstellung soll inhaltlich klar strukturiert sein.
- Jede Pipeline Stufe soll einzeln klar erkennbar sein und es soll deutlich werden, was in der Pipeline Stufe gerade passiert.
- Alle relevanten Informationen, die dem besseren Verständnis dienen, sollen dargestellt werden.
- Es soll gezeigt werden, was sowohl im Arbeitsspeicher als auch in den Registern passiert und weshalb es passiert.
- Die Grafikelemente sowie die Basisbestandteile (Register, Arbeitsspeicher, Alu usw.) sollen bei der Implementierung von neuen Prozessoren ohne großen Aufwand wieder genutzt werden können.
- Die Anforderungen an die Hardware sollen nicht zu hoch sein.
- Es soll kostenlos sein und es soll ohne Lizenzprobleme weiter gegeben werden können.
- Es soll auf verschiedenen Plattformen laufen.

2.1.2 Anforderungen an den „virtuellen“ Prozessor

Der virtuelle Prozessor ist das Herzstück des ganzen Systems. Je näher er dem Original Prozessor kommt, umso einfacher ist hinterher die Darstellung des Prozessors. Aus diesem

Grund wurde auf diesen Aspekt besonders viel Wert gelegt. Folgende Anforderungen wurden gestellt:

- Der Prozessor soll originalgetreu arbeiten: das heißt, der Prozessor soll genau die gleichen Werte liefern, die auch ein Original Prozessor liefern würden. Dazu gehört auch, dass die Parallelität des Original Prozessors nachgebildet wird. Falls der Prozessor diese Anforderungen an gewissen Stellen nicht erfüllt, muss auf diese Fehler aufmerksam gemacht werden.
- Der Prozessor soll mit „Echten Daten“ arbeiten: Gemeint ist damit, dass der Prozessor mit Originaldaten arbeitet, so wie sie ein Compiler erstellt.
- Die Daten müssen abgreifbar sein: Das bedeutet, dass es eine Möglichkeit geben muss, die Daten mitlesen zu können, damit sie hinterher zur Darstellung genutzt werden können.

2.1.3 Anforderungen bezüglich der Erweiterbarkeit

Eine wichtige Anforderung an das System war, dass hinterher weitere Prozessoren hinzu gefügt werden können. Das sollte möglichst mit wenig Aufwand verbunden sein. Aus diesem Grund sind folgende drei Anforderungen bezüglich der Erweiterbarkeit gestellt worden:

- Alle Teile eines Prozessors, die in jedem Prozessor vorhanden sind (z.B. ALU, Register), sollen so generisch gestaltet werden, dass sie ohne Probleme in anderen Prozessoren wiederverwendet werden können.
- Das System zur Darstellung des Prozessors soll so gestaltet sein, dass es ohne großen Aufwand möglich ist, den Aufbau eines anderen Prozessors nachzubilden.
- Die GUI (Graphical User Interface) soll so gestaltet werden, dass sie ohne Veränderungen auch von anderen Prozessoren ähnlichen Typs genutzt werden kann.

2.1.4 Die Anforderungen an die grafische Benutzeroberfläche

Gerade bei einem Prozessor, der sehr komplex ist, kann es schnell passieren, dass man mit Informationen überflutet wird. Darum ist es wichtig, dass die GUI so strukturiert ist, dass man nicht mit Informationen überfrachtet wird, aber trotzdem auf alle Informationen Zugriff hat. Folgende Anforderungen wurden für die GUI festgelegt:

- Gut strukturiert: Informationen, die zusammen gehören sollen auch zusammen dargestellt werden. Damit ist gemeint, dass beispielsweise die einzelnen Pipeline Stufen zusammen dargestellt werden, so dass man direkt sieht, wie der Verlauf ist.
- Intuitiv: Es soll klar werden, wie das ganze System funktioniert, ohne dass vorher Benutzerhandbücher gelesen werden müssen.
- Flexibel gestaltet: Damit ist gemeint, dass möglichst viele Teile der GUI austauschbar bzw. ein- und ausblendbar sind, so dass der Nutzer wirklich nur die Fenster sieht, die ihn interessieren und er alle anderen ausblenden kann.

2.1.5 Anforderungen an die visuelle Darstellung des Prozessors

"Ein Bild sagt mehr als 1000 Worte"

Das Grafiksystem, welches die Darstellung des Prozessors übernimmt, ist ein sehr wichtiger Faktor. Es muss sehr flexibel ausgelegt sein, damit man Änderungen möglichst schnell vornehmen kann. Das Grafiksystem muss den Entwickler in verschiedenen Bereichen unterstützen, damit es ihm möglich ist, eine übersichtliche Darstellung des Prozessors zu erstellen. Um das zu gewährleisten, wurden folgende Kriterien festgelegt, die das Grafiksystem erfüllen soll:

- Das Erstellen der Grafikelemente soll möglichst ohne große Konfiguration möglich sein. Das heißt, dass möglichst nur die Position angegeben werden muss.
- Die Grafikelemente sollen in Größe und Farbe (Texturen) flexibel sein. Durch die Veränderung der Größe und Farbe soll es möglich sein, verschiedene Teile unterschiedlich hervorzuheben.
- Die Grafikelemente sollen sich leicht aneinander ausrichten lassen. Es soll also ein Mechanismus vorhanden sein, der den Programmiere beim Ausrichten unterstützt.
- Die Ausrichtung soll beliebig verändert werden können.

2.2 Grundlagen

In diesem Abschnitt werden Grundlagen zum ARM Prozessor und zu Prozessoren im Allgemeinen vermittelt, die notwendig sind, um die Funktionsweise des Gesamtsystems zu verstehen.

2.2.1 Die ARM-Architektur

Die ARM-Architektur ist ein Kern-Design für eine Familie von 32-Bit-Mikroprozessoren [Fli04], die dem RISC-Konzept folgen.

2.2.1.1 Entstehung

Das ARM-Design wurde 1983 vom englischen Computerhersteller Acorn Computers Ltd. als Entwicklungsprojekt gestartet. Das Team, unter der Leitung von Roger Wilson und Steve Furber, begann die Entwicklung eines leistungsfähigen Prozessors. Anstatt, wie die Konkurrenz, auf Prozessoren der Firmen Intel oder Motorola zurückzugreifen, entwickelte man einen eigenen Prozessor, den ARM (*Advanced RISC Machines*), mit 32 Bit und einer geplanten Taktfrequenz von 4 MHz. Die Tests mit den Prototypen verliefen derart erfolgreich, dass die späteren Serienprozessoren (ARM2), die in den neu entwickelten Rechnern (Acorn Archimedes) verbaut wurden, direkt mit 8 MHz getaktet wurden. Tests

ergaben, dass diese Rechner bei praktisch gleicher Taktfrequenz etwa achtmal schneller waren als die Konkurrenten Commodore Amiga und Atari ST.

2.2.1.2 Befehlssatz und Programmiermodell

Die ARM-CPU ist eine RISC-Architektur [Ric07] und kennt als solche drei Kategorien von Befehlen:

- Befehle zum Zugriff auf den Speicher (Load/Store)
- arithmetische oder logische Befehle auf Werte in Registern
- Befehle zum Ändern des Programmflusses (Sprünge, Subprogrammaufrufe)

Die ARM verfügt über einen 3-Adress-Architektur, alle arithmetisch/logischen Befehle akzeptieren also ein Zielregister und zwei Operandenregister. Beispiel:

```
ADD r0, r1, r2;  r0 := r1 + r2
```

Die ARM ist sowohl Little-Endian als auch Big-Endian kompatibel, kann also mit beiden „Byte Alignments“ umgehen, was angesichts des Einsatzzwecks als Standard-CPU in Kommunikationsgeräten ein deutlicher Vorteil ist. Der Standardmodus der ARM ist little endian.

2.2.1.3 Registersatz und Ausführungs-Modi

Dem Programmierer stehen 15 „general purpose“-Register zur Verfügung (r0–r14), wobei im Register r13 standardmäßig der Stackpointer gehalten wird und das Register r14 als „Link Register“ benutzt wird, in dem die Rücksprungadresse bei Prozeduraufrufen (mit BL „branch with link“) gespeichert wird, um später zurück in den Programmzähler geschrieben zu werden (Rückkehr zum aufrufenden Programmcode). Das Register r15 fungiert als Programmzähler (Program Counter, PC). Zusätzlich zu diesen direkt veränderbaren Registern gibt es das Status-Register (CPSR, Current Program Status Register), das die Statusbits und andere Informationen, wie z. B. den momentanen Ausführungsmodus, enthält.

2.2.1.4 Befehlsbreite und Adressierungsarten

Der Load/Store-Befehl des ARM unterstützt die üblichen Adressierungsmodi. Bei der unmittelbaren Adressierung und der absoluten Adressierung gibt es jedoch einige Einschränkungen, die im Folgenden näher erklärt werden sollen.

Sämtliche Befehle im ARM-Befehlssatz sind 32 Bit lang (außer Thumb Befehle, die hier nicht weiter behandelt werden). Dies bedeutet auf der einen Seite, dass jede Instruktion mit einem Speicherzugriff geladen werden kann, wodurch sich das Design der Pipeline und die Instruction Fetch-Unit vereinfachen. Auf der anderen Seite können 32-Bit-Adressen oder 32-Bit-Werte nicht in einem 32 Bit breiten Befehl angegeben werden. Stattdessen hilft man sich wie folgt: Es können keine 32-Bit-Werte direkt im Befehl codiert werden. Stattdessen werden für Direktwerte 8 Adressbits und 4 Shift-Bits angegeben. Andere Direktwerte

müssen im Speicher gehalten und vor dem eigentlichen Befehl in ein Register geladen werden, welches dann als Operand angegeben wird.

Der Sprungbefehl enthält einen 24-Bit-Offset, so dass im Bereich von ± 32 MB von der aktuellen Stelle im Programm aus gesprungen werden kann (wobei der Programmzähler der aktuellen Instruktion immer um 8 Byte vorauseilt). Der Programmzähler kann als Zielregister bei datenverarbeitenden Instruktionen angegeben werden (was gleichbedeutend mit einem Sprung Befehl ist). Falls das S Bit gesetzt ist wird automatisch das 'saved program status register' zurück in das CPSR (aktuelle Statusregister) kopiert. Das ist die Standard-Methode um aus einer Exception zurückzukehren. Der Programmzähler kann als Zielregister bei Load-Operationen angegeben werden. So können ebenfalls 32-Bit-Sprungadressen angegeben werden.

Bei den Load/Store-Befehlen kann ein 12-Bit-Offset auf eine Basisadresse addiert werden, welche aus einem Register gelesen wird. Auf diese Weise kann auf Adressen, die im Abstand von bis zu 2^{12} = 4096 Byte zu der Basisadresse liegen, zugegriffen werden. Als Basisregister kann auch der Programmzähler verwendet werden. So lässt sich z. B. ein 32-Bit-Sprung durch den Assembler umsetzen, indem die absolute Sprungadresse hinter dem Ende der Funktion gespeichert wird. Diese kann dann durch einen PC-relativen Load-Befehl mit Ziel Program-Counter durch einen einzigen Befehl angesprungen werden. Vorher wird durch Store-Multiple-Befehl der komplette Registersatz (inklusive PC = Rücksprungadresse) auf dem Stack gespeichert. Alle Variablen, die außerhalb der 4 kB um die aktuelle Stelle im Programm liegen, können nur geladen werden, indem zuerst ihre Adresse in ein Register geladen wird und dieses in nachfolgenden Zugriffen als Basisregister verwendet wird.

2.2.1.5 Besonderheiten des Befehlssatzes

Der ARM-Befehlssatz verfügt über einige Besonderheiten, die zur höheren Codedichte der Architektur beitragen.Sämtliche Befehle können bedingt ausgeführt werden. Damit entfällt die Notwendigkeit für Programmsprünge in vielen Standardsituationen, z. B. If-else-Abfragen, man vermeidet Programmsprünge, weil diese die Pipeline des Prozessors leeren, und dadurch Wartezyklen entstehen. Das hat aber nur einen Vorteil, wenn das Leeren der Pipeline Stufe länger dauert als die entsprechenden Befehle zu bearbeiten (bearbeiten bedeutet an dieser Stelle, dass der Befehl eingelesen wird, aber aufgrund der bedingten Ausführung nicht ausgeführt wird). Kodieren der Bedingung, werden die ersten 4-bit eines jedes Befehles verwendet. Beispiel:

```
SUBS   r0, r0, r1;    r0 := r0 -r1 (setzt Bedingungsbits)
ADDGE  r2, r2, r3;    if (r0 >= r1) then r2 := r2 + r3;
ADDLT  r2, r2, r4;    else r2 := r2 + r4;
```

Die ARM verfügt über einen Barrel-Shifter im B-Pfad der ALU, also dem Pfad über den der zweite Registerwert läuft (im ersten Beispiel r1): Sämtliche Befehle, die mit dem zweiten Operanden arbeiten, erlauben also auch die Angabe eines 4-bit-weiten Shift- oder Roll-Faktors.

2.3 Auswahl der Grafik Engine

Eine Grafik Engine (Grafikkern\Grafikmodul) ist ein mitunter eigenständiger Teil eines Computerprogramms, welcher für dessen Darstellung von Computergrafik zuständig ist, meist möglichst realitätsgetreuer 3D-Computergrafik, wie Gegenstände, Umwelt und Personen (Stichwort: Virtuelle Realität). Im Zusammenhang mit 3D-Computergrafik bezeichnet man die Grafik-Engine daher dann auch häufig als 3D-Engine. Sie bietet einem Programmierer eine große Palette von grafischen Funktionen und Effekten (geometrische Objektbeschreibung, Oberflächentexturen, Licht und Schatten, Transparenz, Spiegelungen usw.), so dass er für seine spezielle Anwendung diese nicht stets neu programmieren muss.

2.3.1 2D oder 3D Grafik

Als erstes stand die Entscheidung an, ob für die Darstellung des Prozessors 2D oder 3D Grafik genutzt wird. Beide Varianten haben gewisse Vor- und Nachteile.

Vorteile von 2D:

- Einfacher zu handhaben
- Ressourcen sparend

Nachteile von 2D:

- Wenig Auswahl an Bibliotheken
- Es gibt keine wirkliche Software zur Erstellung von 2D Objekten nur die bekannten Grafikbearbeitungsprogramme

Vorteile von 3D:

- 3D ist mächtiger. Es kann mehr damit gemacht werden.
- Die Community ist größer. Somit ist die Chance, dass es jemanden gibt, der einem bei einem Problem helfen kann, größer.
- Viele verschiedene Grafik Engines stehen zu Verfügung.
- Bietet die Möglichkeit die Grafik so zu gestalten, dass sie mehr Interesse weckt, was dazu führt, dass die Motivation steigt, sich damit zu beschäftigen.
- Es gibt sehr gute Grafik Engines, die dem Nutzer viel Arbeit abnehmen.
- Es gibt sehr gute Software zur Erstellung von 3D Objekten und eine Große Community, die einen unterstützen kann.

Nachteile von 3D:

- Braucht aktuellere Hardware (fordert die Ressourcen mehr).
- Ist anfälliger für Fehler.
- Im Allgemeinen komplizierter.

2.3.1.1 Entscheidung

Nach verschiedenen Tests hat sich 3D als bessere Lösung herausgestellt. 3D bietet dem Programmierer viel mehr Möglichkeiten. Die Auswahl an verschiedenen Engine ist weitaus größer. Es ist eine viel größere Community vorhanden, die einem bei Problemen helfen kann. Die Grafik in 3D wirkt besser und durch eine gute 3D Engines ist die 3D Grafik auch relativ einfach zu handhaben.

2.3.2 Auswahl der Richtigen 3D Engine

3D-Engines gibt es wie Sand am Meer möchte man fast meinen. Allein eine Suche auf Google mit dem Stichwort “3D Engine” ergibt 106.000 Ergebnisse (Stand: Juni 2007) und die Zahl ist steigend.

Man kann sich vorstellen, dass unter diesen Vorrausetzungen das Finden der passenden Engine sehr schwer zu bewerkstelligen war. Um die passenden 3D Engine für den Prozessor zu finden, wurden einige Kriterien erstellt. Im Einzelnen sind es folgende Kriterien:

- Zugriff über Java möglich.
- Einfach zu handhaben: Die Grafik Engine soll einem so viel wie möglich an Arbeit abnehmen.
- Aktuell (sie wird weiter entwickelt): Die Software soll auch noch in Zukunft auf neuen Betriebssystemen laufen. Dafür ist Aktualität zwingend notwendig.
- Ausgereift: Gerade im Bereich Grafik Engine für Java gibt es viele, die noch nicht ausgreift genug sind.
- Sie sollte einfach in SWT [Nor04] (siehe dazu Abschnitt 8.1.3) einzubetten sein.
- Schnell: Geschwindigkeit ist ein entscheidender Faktor, denn das beste Programm bringt einem nichts, wenn es das ganze System ausbremst.
- Einfach sich darin einzuarbeiten. Es sollte Tutorials usw. geben, die einen bei der Einarbeitung unterstützen.
- Portierbar auf verschiedene Betriebssysteme: Windows ist weit verbreitet, aber längst nicht jeder arbeitet unter Windows. Darum sollte die Möglichkeit bestehen, das Programm auch auf anderen Betriebssystemen laufen zu lassen.
- Unterstützung vieler verschiedener 3D Grafik Formate: Gerade bei 3D Editoren findet jeder einen anderen gut. Aus diesem Grund ist es einfach von Vorteil, wenn man wählen kann.
- Open Source: Egal wie viele Tests man vorher macht, es kann immer sein, dass später eine Anforderung hinzukommt, die die Grafik Engine nicht erfüllt. In so einem Moment ist es sehr von Vorteil, wenn man die Funktionalität selbst einfügen kann.
- Eine Lizenz, die die Nutzung ohne Einschränkungen ermöglicht.
- Mit älterer Hardware kompatibel: Die Software soll hinterher auch auf Computern laufen, die nicht den aktuellsten Stand der Technik wiederspiegeln.
- Große Community: Viele Nutzer sind wichtig. Die Chance erhöht sich somit, dass einem bei Problemen geholfen werden kann.

Viele Grafik Engine sind nur Hobby Projekte. Deshalb sind sie in der Regel weder ausgereift noch werden sie regelmäßig aktualisiert. Aus dem Grund wurde ein großer Teil der Grafik Engine schon von vornerein aussortiert und nur ein kleiner Teil ist in die engere Auswahl gekommen.

2.3.2.1 Java 3D

Java 3D [Kla05] ist die Standard Bibliothek zur Erzeugung, Manipulation und Darstellung dreidimensionaler Grafiken innerhalb von Java-Applikationen. Allerdings ist Java 3D keine wirkliche Grafik Engine, sondern eher eine Grafik Bibliothek, die einem den Zugriff auf OpenGL ermöglicht. Die Unterstützung die Java 3D einem zur Verfügung stellt, wurde als zu gering empfunden.

2.3.2.2 Jirr

Jirr [Jirr07] ist eine Java Portierung der Irrlich [Irr07] Engine. Sie ist keine Neuimplementierung der Irrlich Engine in Java, sondern bietet nur den Zugriff auf die Irrlich Engine über Java. Irrlich ist eine plattformunabhängige 3D Grafik Engine, die sowohl DirectX 8 & 9 als auch Open GL unterstützt. Außerdem sind noch 2 Software Renderer vorhanden. Da die Jirr Engine direkt auf Funktionen zugreift, die in C++ geschrieben sind, ist sie sehr schnell und leistungsfähig.

Die Community ist groß genug, um den Programmierer unterstützen zu können. Sie ist sehr einfach zu handhaben und die Tutorials, die als Starthilfe vorhanden sind, erklären alles, was man für das Projekt wissen muss. Sie unterstützt die meisten bekannten 3D Formate.

2.3.2.3 Ogre4j

Ogre4j [Ogr07a] ist genau wie Jirr eine Portierung einer in C++ geschrieben 3D Engine. Ogre4j basiert auf der bekannte Engine Ogre [Ogr07b], die wohl eine der größten und bekanntesten Open Source Engine ist, die es gibt. Insgesamt hat Ogre4j einen sehr guten Eindruck hinterlassen. Allerdings war die aktuellste Version zum Zeitpunkt der Recherchen vom 1.8.2005.

2.3.2.4 jMonkeyEngine (jME)

jME [Jme07] ist eine in Java geschriebene 3D Engine. Sie nutzt LWJGL (Lightweight Java Game Library) [Lwj07], um auf Open GL [Orl04] zuzugreifen. jME ist auf 3D Spiele optimiert und liefert somit sehr gute Performancewerte. jME wird stetig weiter entwickelt. Die aktuellste Version stammt vom 23. April 2007. Insgesamt unterstützt jME 6 3D Formate. Die Community ist groß genug, um den Programmierer bei Fragen und Problemen zu unterstützen.

2.3.3 Entscheidung

Nach den Tests hat sich Jirr als beste Lösung herausgestellt. Sie hat alle Kriterien, die am Anfang aufgestellt wurden, erfüllt:

- Jirr ist sehr einfach zu handhaben. Zum Beispiel braucht man für das Laden eines 3D Modells nur 1 Zeile:
  ```
  smgr.addAnimatedMeshSceneNode(meshDatei,pos,rot,scale);
  ```

  Wie oben schon angesprochen zeigt Java 3D gerade in dem Bereich Schwächen. Auch Joe konnte in dem Bereich nicht mit Jirr mit halten.
- Jirr steht unter der LGPL-Lizenz [Lgp07]. Diese Open-Source Lizenz ist sehr liberal und erlaubt sogar den kommerziellen Einsatz. Die Engine ist also frei für jedermann und kann von allen herunter geladen, benutzt und sogar verändert werden.
- Die Engine stellt ein gutes Framework zur Verfügung und ist leicht zu benutzen.
- Jirr wird regelmäßig weiterentwickelt. Allein in den letzten 4 Monaten gab es zwei neue Releases. Was zum Beispiel bei Ogre4j nicht der Fall war (Bei Ogre4j stammt die aktuellste Version von 2005 und kann somit nicht als aktuell bezeichnet werden).
- Jirr unterstützt alle gängigen 3D Formate. Keine andere der getesteten Engine unterstützt so viele wie Jirr.
- Jirr läuft durch DirectX 8 Unterstützung und 2 Software Renderer auch auf älterer Hardware flüssig.
- Die Einbettung von Jirr in SWT ist durch wenig Zeilen Code ohne Probleme möglich.

2.4 Erbrachte Leistungen

In diesem Abschnitt werden kurz die erbrachten Leistungen aufgelistet.

- Analyse des Marktes der Grafik Engines.
- Beurteilung und Festlegung der in Frage kommenden Grafik Engines.
- Untersuchung der Funktionsweise der in Fragen kommenden Engine und Auswahl der Besten.
- In die Thematik des ARM Prozessor einarbeiten.
- Assembler für den ARM Prozessor suchen und mit der Funktionsweise vertraut machen.
- Verschiedene 3D Grafik Editor analysieren, den Besten auswählen.
- In den 3D Grafik Editor einarbeiten.
- Erstellen der 3D Modelle.
- Texturen erstellen.
- Java *Instruction Set Architecture* des ARM Prozessors implementieren.
- Grafik implementieren
- Implementierung eines Disassembler zum Testen und für die Visuelle Darstellung.
- In Java Nativ einarbeiten um die Jirr Engine zu erweitern.

3 Der Prozessor

3.1 Pipelining

Eine Frage, die auftritt, wenn man eine ISA in Java implementieren will ist: „Wie bildet man das Pipelining nach?“ Die einzelnen Pipeline Stufen laufen im Prozessor parallel ab. Das muss in Java nachgebildet werden.

Um das gleiche Ergebnis wie bei der Pipelining im Prozessor zu erhalten, werden die Pipeline Stufen hintereinander abgearbeitet. Also Fetch, Decode, Execute, Memory Access und dann Write Back (wobei sich in jeder Stufe eine jeweils eine andere Instruktion befindet). Die Abfolge der Stufen wird im weiteren Verlauf als ein Clock Zyklus bezeichnet. Die Werte, die in den einzelnen Stufen berechnet werden, werden nicht direkt an die nächste Pipeline Stufe übergeben, sondern erst beim folgenden Clock Zyklus. Also z.B. die Instruktion, die in der Fetch Stufe eingelesen wird, wird erstmal zwischengespeichert. Dann werden alle anderen Stufen durchgearbeitet. Beim nächsten Clock Signal wird der Wert, der in der Fetch Stufe eingelesen wurde an die Decode Stufe übergeben. Genau so funktioniert es auch bei den andern Pipeline Stufen.

	Clock Zyklus 0	Clock Zyklus 1	Clock Zyklus 2	Clock Zyklus 3	Clock Zyklus 4	Clock Zyklus 5
Fetch	Instruktion 1	Instruktion 2				
Decode		Instruktion 1	Instruktion 2			
Execute			Instruktion 1	Instruktion 2		
Memory Access				Instruktion 1	Instruktion 2	
Writeback					Instruktion 1	Instruktion 2

Abb. 1. Das Bild zeigt zwei Instruktionen in den einzelnen Stufen

Dieses Konzept hat den Vorteil, dass die berechneten Werte direkt an den grafischen Teil weiter gegeben werden können. Der Benutzer sieht also genau das, was gerade in dem „virtuellen“ Prozessor berechnet wurde. Das vermeidet, dass vorher große Datenmengen verarbeitet und hinterher gespeichert werden müssen. Ein anderer Vorteil ist, dass man dadurch sehr nah an die Funktionsweise eines „echten“ Prozessors heran kommt. Das führt dazu, dass man alle Informationen, die zur Darstellung benötigt werden, direkt entnehmen kann. Es muss hierbei also nichts zusätzlich ausgerechnet werden.

Der Nachteil dieses Konzeptes ist, dass man sich teilweise mit genau den gleichen Problemen rumschlagen muss, die man hätte, wenn der virtuelle Prozessor wirklich parallel arbeiten würde. Ein gutes Beispiel hierfür ist das Forwarding.

Die Werte müssen zwischen den Pipeline Stufen abgelesen werden, was die Komplexität der Implementierung erhöht. Diese Probleme hätte man nicht, wenn man einen anderen Ansatz wählen würde und jeweils eine Instruktion nach der anderen komplett durchrechnet (siehe Abb. 2).

	Instruktion Zyklus 0	Instruktion Zyklus 1
Fetch	Instruktion 1	Instruktion 2
Decode	Instruktion 1	Instruktion 2
Execute	Instruktion 1	Instruktion 2
Memory Access	Instruktion 1	Instruktion 2
Write Back	Instruktion 1	Instruktion 2

Abb. 2. Eine Instruktion wird nach der andern durchgeführt

Aber leider hat der Ansatz zwei große Nachteile. Erstens müssten sehr viel Statusinformationen gespeichert werden, wie z. B. die Flag Stellungen, weil die nachfolgende Instruktion diese Statusinformationen braucht. Das zweite Problem tritt bei der Synchronisation zwischen „virtuellen“ Prozessor und der visuellen Darstellung des Prozessors auf. Der „virtuelle“ Prozessor kann durch die veränderte Abarbeitung der Instruktionen nicht mehr direkt die Darstellung steuern, was dazu führt dass ein zusätzlicher Mechanismus eingebaut werden muss, der diese Funktion übernimmt.

Eine dritte Variante, die möglich ist, wäre eine Kombination der beiden Konzepte. Der einzige Punkt an dem Daten wirklich verändert werden ist die Execute Stufe. Man könnte also Fetch und Decode als eine Stufe implementieren und die Write Back und Memory Access Stufe zur Execute Stufe hinzufügen. Das würde dazu führen, dass der Prozessor übersichtlicher wird. Die Nachteile der zweiten Variante sind damit zum größten Teil nicht mehr da.

Allerdings sind die Vorteile nur noch teilweise vorhanden. Forwarding muss trotzdem noch betrieben werden, wenn auch nur begrenzt. Insgesamt ist der Vorteil nur noch sehr gering, vor allem weil in der Fetch, Memory Access und Write Back nur relativ wenig passiert, was zu Problemen führt.

Dafür wird ein Nachteil mit übernommen und zwar ist es nicht mehr möglich, die Werte der Pipeline stufen direkt auf die GUI zu übertragen. Schlussendlich ist die Entscheidung getroffen worden die Klassische Methode zu wählen und eine fünf stufige Pipeline zu implementieren.

3.2 Der Aufbau des „virtuellen“ Prozessors

In diesem Abschnitt geht es um den „virtuellen“ Prozessor, um Design Entscheidungen und um die Implementierung.

3.2.1 Erweiterbarkeit des Prozessors

Da eine der Anforderungen an die Software war, dass sie erweiterbar ist, steht natürlich die Frage im Raum, wie weit das auf die Implementierung des Prozessors zutreffen soll. Über diese Frage wurde ausführlich diskutiert. Das Ziel war es natürlich, soviel wie möglich allgemein zu gestalten, um hinterher die Möglichkeit zu haben sehr schnell und einfach neue Prozessoren einfügen zu können.

3.2.1.1 XML Interpreter

Der erste Ansatz war es, den Prozessor so zu gestalten, dass man nur ein XML [XML 05] Document anlegen muss, in der die Spezifikation für den Prozessor festgelegt ist. Dann sollte ein Art Interpreter erstellt werden, der das XML-Dokument einliest und dann das entsprechende Prozessorverhalten modelliert.

Dieser Ansatz wurde aber schnell als nicht machbar verworfen. Der Grund dafür war, dass man in das XML-Dokument einen Spezialteil für jeden Prozessor, den man bauen will, einbauen müsste. Beim ARM Prozessor war das z. B. die Condition (siehe Abschnitt 3.2.3.5). Das wiederum würde voraussetzen, dass die Spezifikation jedes Prozessors, der möglicherweise dargestellt werden soll, durchgegangen wird, um alle möglichen Besonderheiten festzustellen. Da aber nicht genau festgelegt ist, welche Prozessoren das sind, ist dies nicht durchführbar. Selbst wenn Prozessoren festgelegt wären, würde der Aufwand vermutlich höher sein, als wenn man jeden Prozessor einzeln implementieren würde.

3.2.1.2 Interface für die Pipeline Stufen

Der zweite Ansatz war, ein Interface für jede der 5 Pipeline Stufen zu erstellen. Die Pipeline Stufen müssten für jeden Prozessor einzeln programmiert werden. Alles andere sollte dann von einer vordefinierten Klasse bearbeitet werden. Problematisch an diesem Ansatz war die Kommunikation zwischen den einzelnen Pipeline Stufen. Da jede Implementierung der Pipeline Stufen vermutlich unterschiedliche Daten übergeben muss (abhängig vom Prozessor der implementiert wird) bzw. in dem Bereich keine Einschränkungen gemacht werden sollten, wären eigentlich nur zwei Datentypen übergeblieben. Der eine wäre Objekt gewesen, der andere wäre String. In dem String könnte z.B. ein XML-Dokument enthalten sein, in dem die Daten gespeichert sind. Beide Varianten sind keine guten Programmierstile.
Bei den Prozessoren, die auf dem System laufen sollen, steckt die Logik fast nur in den Pipeline Stufen. Deshalb ist auch der größte Programmieraufwand mit dem Implementieren

der Pipeline Stufen verbunden, somit ist der Vorteil durch diesen Ansatz nicht sehr groß. Aus dem Grund wurde auch dieser Ansatz verworfen.

Letztendlich wurde die Entscheidung getroffen, dass die verschiedenen Prozessoren komplett neu implementiert werden müssen. Die Prozessoren sind in der Regel so verschieden, dass man keinen bzw. nur einen sehr kleinen gemeinsamen Nenner für alle Prozessortypen findet, der nicht ausreichend ist um eine gemeinsame Basis zu finden.
Allerdings wurde festgelegt, dass Teile, die in der Regel in allen Prozessoren vorkommen als eigenständige Klassen implementiert werden, so dass sie hinterher in allen Prozessoren genutzt werden können. Diese Teile werden im nachfolgenden Abschnitt erläutert.

3.2.2 Allgemeine Klassen

Wie im Abschnitt zuvor besprochen, wurde die Entscheidung getroffen keine allgemeine Entwicklungsumgebung für alle möglichen Prozessoren zu erstellen, sondern nur die Teile, die auch andere Prozessoren nutzen können, auszulagern und unabhängig von der ARM Architektur zu erstellen. In den folgenden Abschnitten werden diese Teile vorgestellt.

3.2.2.1 Die ALU Klasse

Die ALU (*arithmetic logic unit*) ist eigentlich in jedem Prozessor vorhanden. In ihr werden die Arithmetisch Befehle abgearbeitet, wie z.B. das Addieren, Subtrahieren, Multiplizieren und in der Regel auch Shiften. Das ist bei der ARM Architektur anders. Bei der ARM Architektur gibt es einen Barrel Shifter, der das Shiften des zweiten Operanten schon vor der ALU übernimmt (so dass der Operant, wenn er in der ALU ankommt, schon geshiftet wurde).

Das Ziel der ALU Klasse war, dass sie so allgemein ist, dass jeder Prozessor sie nutzen kann. Darum wurde der Barrel Shifter beim Design der ALU Klasse nicht weiter beachtet. Bei der Implementierung des ARM Prozessors wird die ALU zweimal aufgerufen. Einmal um zu Shiften und einmal für den normalen ALU Aufruf. Beim Aufruf der ALU Klasse können zwei 32 Bit Werte (Integer) übergeben werden. Zusätzlich kann noch ein Carry Bit mit übergeben werden. Die Operation, die ausgeführt werden soll, wird als Enum übergeben. Als Ergebnis bekommt man ein Result Objekt, dass das eigentliche Ergebnis und das Overflow, Zero, Carry und Negativ Flag beinhaltet.

3.2.2.2 Die Register Klasse

Eigentlich hat jeder Prozessor Register, nur die Anzahl ist unterschiedlich (teilweise auch die Größe, aber die Größe 32 Bit wurde festgelegt, weil die Prozessoren, die auf dem System dargestellt werden sollen in der Regel 32 Bit Register haben). Darum wurde die Register Klasse so aufgebaut, dass eine beliebige Anzahl von Registern erstellt werden kann, auf die alle einzeln zugegriffen werden kann.

3.2.2.3 Die RAM Klasse

Der Arbeitsspeicher (RAM) gehört nicht direkt zum Prozessor, ist aber ein wichtiger Teil, der notwendig ist, damit der Prozessor funktioniert. In ihm werden neben den Daten auch die Instruktionen für den Prozessor gespeichert.

Größe der Speichereinheiten

Im „echten“ Arbeitsspeicher ist eine Speichereinheit immer 8 Bit groß. Also auf jeden 8 Bit Wert kann einzeln zugegriffen werden. Für den Testbetrieb, für den das System entwickelt wird, wird in der Regel nur mit 32 Bit Werten gearbeitet. Da stellt sich natürlich die Frage, ob es sinnvoll ist, diese Funktionalität zu unterstützen, wenn sie eigentlich gar nicht genutzt wird. Wenn man sich nur auf 32 Bit Werten beschränkt, hätte das den Vorteil, dass man direkt mit Integern arbeiten könnte und nicht daraus erst Bytes machen muss und hinter wieder Bytes in Integer umwandeln muss.

Um unnötiges Umwandeln zu vermeiden wurde die Entscheidung getroffen, dass alle Werte in Integer gespeichert werden. Um trotzdem noch die Chance zu haben auf Byte Werte zuzugreifen, gibt es eine Methode, die einem, mit Hilfe von Bitmaskierung, die Möglichkeit gibt auf einzelne Bytes zuzugreifen.

3.2.3 Merkmale des Prozessors

In diesem Abschnitt werden die verschieden Merkmale des „virtuellen“ Prozessors besprochen.

3.2.3.1 Aufrufen der einzelnen Pipeline Stufen

Wie im Abschnitt 3.1 schon beschrieben laufen in einem „echten“ Prozessor die einzelnen Pipeline Stufen parallel ab. Es gibt also keine Reihenfolge. Für den „virtuellen“ Prozessor muss diese Funktionalität nachgebildet werden.

Umgesetzt wurde das ganze indem am Ende jeder Pipeline Stufe die darauf Folgende aufruft. Falls Pipeline Stufen aufgrund von Sprüngen nicht ausgeführt werden dürfen, wird direkt die nächste Stufe aufgerufen (nach einem Sprungbefehl werden alle Instruktionen hinter dem Sprungbefehl verworfen, z.B. findet ein Sprung in der Execute Stufe statt, werden die Instruktionen, die sich in der Fetch und Decode Stufe befinden, verworfen).

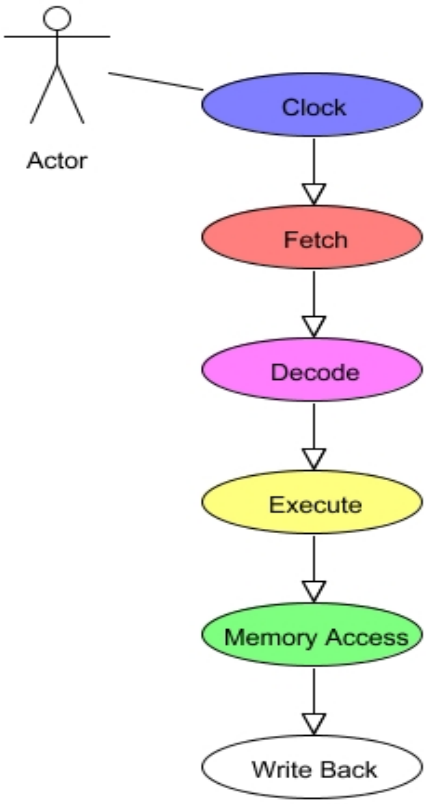


Abb. 3. Ablaufplan der Pipeline Stufen Aufrufe

3.2.3.2 Clock signal

Sobald ein Clock Signal eintrifft (Aufruf der Methode nextClock()), bekommen alle Variablen ein Update. Das bedeutet, dass ihnen der Wert der nächsten Instruktion (natürlich abhängig davon, um was für eine Variable es sich handelt) zugewiesen wird. Ein Beispiel:

```
exZielReg = exNextZielReg;
```

Hier wird Variable exZielReg, die das aktuelle Zielregister der Execute Stufe repräsentiert, das neue Zielregister zugewiesen. Nachdem alle Variablen ihren neuen Wert haben, wird die Fetsch Stufe aufgerufen.

3.2.3.3 Reset

Das Reset, also das Zurücksetzen aller Werte auf den Startzustand, ist theoretisch nur das Zurücksetzen aller Werte auf 0. In der Praxis ist es allerdings so, dass bei einfachen Datentypen (wie int, float usw.) der Wert 0 ein zulässiger Wert ist. Aus dem Grund mussten alle einfachen Datentypen durch Objekte ersetzt werden, die den Wert Null annehmen können und somit ist sichergestellt, dass der Prozessor nicht weiter arbeiten kann. Nach dem Reset fängt der Prozessor wieder an der Adresse 0 an und arbeitet das Programm durch.

3.2.3.4 Das Decodieren

Nachdem die Instruktion in der Fetch Stufe eingelesen wurde, wird sie in der zweiten Stufe decodiert. Es gibt verschiedene Varianten wie ein Aufbau einer Decode Stufe aussehen kann. Die erste Variante ist, dass das Decodieren und die Auswertung in einem Schritt gemacht werden, also z.B. Register Nummer 5 wird decodiert und dann sofort eingelesen.

Der Vorteil dieser Variante ist es, dass alles, was zusammen gehört an einer Stelle steht. Der Nachteil ist, dass das Decodieren sehr komplex ist und wenn man das Decodieren noch zusätzlich mit in die Decode Stufe einfügt, neben dem was in der Decode Stufe sowieso erledigt werden muss, wie beispielsweise das Setzen der Werte für die Execute Stufe, führt es dazu, dass die Komplexität steigt und somit auch die Fehleranfälligkeit.

Eine andere Möglichkeit ist es, das Ganze in zwei Schritte aufzuteilen. Also als erstes kommt das Decodieren der Instruktion. Dann werden die Werte zwischen Speicher und im zweiten Schritt die Daten weiterverarbeitet. Ein Beispiel: im ersten Schritt wird festgestellt, dass Register 5 gelesen werden soll und im zweiten Schritt wird dann Register 5 gelesen. Das hat den Vorteil, dass man ein großes Problem in zwei Kleinere aufteilen kann und die Teile auch unabhängig voneinander bearbeiten kann.

Der Nachteil ist, dass der Aufwand größer wird. Jeder Wert muss zweimal bearbeitet werden, außerdem muss eine Möglichkeit gefunden werden, die Daten zwischen den beiden Teilen auszutauschen. Das ist in sofern problematisch, weil es sehr viele verschiedene Werte bei den unterschiedlichen Instruktionstypen gibt.

Dies führt dazu, dass man entweder für jeden Instruktionstyp einen eigenen Datentyp definieren muss, in dem die verschiedenen Werte gespeichert werden oder man muss immer mit Objekten arbeiten und casten. Als dritte Variante kann man einen Datentyp erstellen, der alle verschiedenen Werte aufnehmen kann. Keine der drei Varianten ist ohne Vorbehalt zu empfehlen.

Implemtierung

Es wurde die Entscheidung getroffen, die zweite Variante zu nehmen und die Decodierung in zwei Schritte aufzuteilen. Der Grund für diese Entscheidung ist, dass alles dadurch übersichtlicher wird und die Komplexität sinkt. Zusätzlich wird der Datentyp genutzt, der alle verschiedenen Werte aufnehmen kann. Der Grund dafür ist, dass dieser Datentyp zu mehr Übersichtlichkeit führt.

3.2.3.5 Condition

Zu jeder Instruktion im ARM Prozessor gehört eine sogenannte Condition. Durch die Condition kann jede Instruktion bedingt ausgeführt werden. Die Ausführung hängt von den Werten der Flags ab. Die Werte der Flag sind abhängig von dem letzten in dem ALU berechneten Wert. Zum Beispiel wurde in der ALU 5 – 5 = 0 berechnet, dann ist die Zero Flag gesetzt. Das Überprüfen findet immer vor der Ausführung der Execute Stufe statt.

3.2.3.6 Forwarding

Als Forwarding wird das Abgreifen von Ergebnissen bezeichnet, die zwar schon berechnet wurden, aber noch nicht in die Register übertragen wurden. Das kann jeweils am Anfang der Memory Accesse und Write Back Stufe stattfinden (bei anderen Prozessoren kommt es oft vor, dass das Forwarding schon in der Decode Stufe stattfindet und somit die Werte am Ende der Execute, Memory Accesse und Write Back Stufe stattfinden).

Durch die sequenzielle Abarbeitung kommt es zu einem Problem, das bei einem „echten“ Prozessor nicht auftaucht. Wie im Abschnitt 3.2.2.1 besprochen, werden die Pipeline Stufen alle hintereinander aufgerufen. Das Forwarding findet in der Execute Stufe statt. Müssen jetzt Werte geforwardet werden, die in der Memory Access oder in der Write Back Stufe berechnet werden, ist das zu dem Zeitpunkt in dem die Execute Stufe abgearbeitet wird noch gar nicht passiert (beim „echten“ Prozessor läuft es so ab, dass die Werte sobald sie berechnet wurden übertragen werden, was vor dem Ende des Clock Zyklus ist) .

Das Problem wurde durch Aufteilung des Forwarding gelöst. Das Forwarding des Execute Wertes findet direkt in der Execute Stufe statt. Am Ende der Execute Stufe wird überprüft, ob der berechnete Wert geforwardet werden muss und gegebenenfalls wird der Wert geforwardet. Das Forwarding in den anderen Stufen funktioniert, indem nicht der Wert der eigentlichen Stufe geforwardet wird, sondern indem der Wert der Stufe davor geforwardet

wird. Also muss z. B. die Write Back Stufe geforwardet werden, wird die Memory Access geforwardet.
Diese Implementierung entspricht zwar nicht dem Original Prozessor, aber in der späteren Darstellung bemerkt man den Unterschied nicht. Eine andere Möglichkeit dieses Problem zu lösen ist, das Forwarding als letztes zu machen. Also nachdem alle Pipeline Stufen durchgelaufen sind. Das hat den Vorteil, dass man das Forwarding nicht aufteilen muss.

3.2.4 Die Pipeline Stufen

Insgesamt gibt es 5 Pipeline Stufen und zwar die Fetch, die Decode, die Execute, die Memory Access und die Write Back Stufe. In diesen Abschnitt werden die Stufen vorgestellt.

3.2.4.1 Fetch

Die Fetch Stufe ist die erste Stufe. In ihr wird die Instruktion aus dem Arbeitsspeicher eingelesen und an die Decode Stufe weiter gereicht. Außerdem wird der PC Count um 4 hochgezählt (4 für 4 Byte = 32 Bit Länge einer Instruktion)

3.2.4.2 Decode

Die zweite Stufe ist die Decode Stufe. Wie oben besprochen ist das Decodieren in zwei Teile aufgeteilt. Den ersten Teil übernimmt die Klasse ArmInstructionsDecoder, die die Instruktion einliest und alle wichtigen Werte in der Wrapper Klasse DecodeInstrucktionWrapper speichert. Danach werden dann die Register eingelesen und alle Werte für die Execute Stufe werden gesetzt.

3.2.4.3 Execute

Die Execute Stufe ist die dritte Stufe. Als erstes wird überprüft, ob die Execute Stufe aktiv ist. Wie im Abschnitt zu den Condition schon beschrieben wurde, ist dies nicht immer der Fall. Wenn sie nicht aktiv ist, wird sie nicht ausgeführt. Der zweite Schritt ist der Barrel Shifter. Falls er aktiv ist, wird das Shiften durchgeführt. Was nach dem Barrel Shifter passiert, ist abhängig von dem Instruktionstypen. Zurzeit unterstützt der Prozessor 3 Arten von Instruktionen.

Der erste Instruktionstyp Data Processing Typ. Die Data Processing Instruktionen, sind bis auf die Mov (Wert von einem Register in ein anders Register verschieben) alles ALU Operationen. Also wird abgefragt, ob es eine Mov Operation ist. Falls es eine ist, wird einfach nur der Wert des Operanten 1 dem Zieloperanten zugewiesen. Ist es keine Mov Operation wird die ALU aufgerufen. Der nächste Instruktionstyp ist der Single Data Transfer Type. Dieser Typ ist für das Laden und das Speichern von Werten aus bzw. in den Arbeitsspeicher verantwortlich. In der Execute Stufe wird hierfür nur die Adresse berechnet. Der letzte ist der Branch Type. Er repräsentiert einen Sprungbefehl. Hier wird der PC Count mit dem neuen Wert geladen.

3.2.4.4 Memory Access

Die Memory Access Stufe ist die vierte Stufe. Sie ist für das Laden und das Speichern in und aus dem Arbeitsspeicher verantwortlich. Falls kein Wert geladen oder gespeichert werden soll, werden hier nur die Werte der Execute Stufe zur Write Back stufe weiter gereicht. Falls ein Wert geladen wurde, wird der Wert an die Write Back Stufe weitergeleitet.

3.2.4.5 Write Back

Die Write Back Stufe ist die letzte Stufe. In ihr werden die berechneten bzw. geladenen Werte in die Register geschrieben.

4 Die Grafik

In diesem Abschnitt wird die Grafik besprochen dazu gehört die visuelle Darstellung des Prozessors und die grafische Benutzeroberfläche.

4.1 Das Grafik System

Beim Grafik System handelt es sich um den Teil der Grafik der für die visuelle Darstellung des Prozessors verantwortlich ist.

4.1.1 Das Koordinaten System

In den meisten Fällen ist es so, dass bei einer 3D Grafik, Grafik Modell beliebig im Raum verteilt werden können. Theoretisch ist das bei dem implementierten Grafiksystem auch möglich.

Praktisch wurden aber zwei Einschränkungen gemacht. Die erste ist, dass die y-Achse nicht genutzt wird. Der Grund dafür ist, dass alle Grafikmodelle auf einer Höhe sein sollen, damit ein einheitliches Bild entsteht. Die zweite Einschränkung ist, dass die Werte nur im positiven Bereich liegen. Das hat den Vorteil, dass die Werte für die Rotation der einzelnen Grafikmodelle einfacher zu berechnen sind (die Rotation wird genutzt, um festzulegen, in welche Richtung ein Grafikmodell zeigt).

4.1.2 Die Grafikklasse

Die Grafikklasse ist die Hauptklasse. Alle weiteren Grafikklassen erben von ihr. Die Grafikklasse stellt die Funktionalität zur Erzeugung des eigentlichen Grafikobjektes (mit eigentlichen Grafikobjektes ist das Grafikobjekte der Jirr Grafik Engine gemeint) dar. Der gesamte Zugriff auf das eigentliche Grafikobjekt findet über die Grafikklasse statt. Das hat den Vorteil, dass nur die Grafikklasse geändert werden muss, falls man die Grafik Engine ändern will.

Die Grafikklasse stellt eine Hand voll Methoden zur Verfügung, mit denen man die Texturen verändern kann und das Grafikmodell sichtbar und unsichtbar machen kann (ein Grafikmodell ist ein mittels 3D Editor erstelltes Modell. Es kann beliebige Formen haben, z. B. ein Rechteck oder ein Zylinder).

Außerdem können in der Grafikklasse globale Einstellungen verändert werden. Genutzt wurde das zum Beispiel um alle Grafikmodelle zu verkleinern. Dafür wurde in der Grafikklasse beim Erstellen des Grafikobjektes der Skalar Faktor durch 2 geteilt und alle Objekte waren nur noch halb so groß.

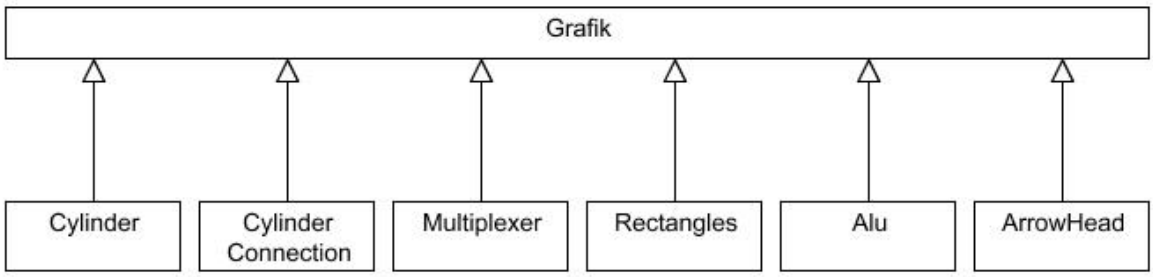


Abb. 4. UML Diagramm der Grafik Klasse und aller Klassen die von ihr erben

4.1.3 Unterklassen der Grafikklasse

Das Grafiksystem setzt sich aus verschiedenen Klassen zusammen, die alle von der Grafikklasse erben. In den Klassen sind die Grafikmodelle festgelegt. Außerdem definiert jede Klasse verschiedene Methoden und Konstruktoren, die speziell auf dieses Grafikmodell zugeschnitten worden sind. Dies gewährleistet, dass die Erzeugung dieser Grafikmodelle ohne großen Aufwand möglich ist. In der Regel müssen nur ein paar Werte im Konstruktor übergeben werden. Werte wie Rotation und Skalar werden dann selbstständig berechnet. Insgesamt gibt es sechs Unterklassen der Grafikklasse, die jetzt kurz vorgestellt werden (Eine Demonstration wie die einzelnen Klassen aufgerufen werden befindet ich auf der beigelegten CD unter Sourcecode\BA\Beispiele).

4.1.3.1 Cylinder

Die Cylinder Klasse repräsentiert ein Grafikobjekt in Form eines Zylinders. Es muss der Startpunkt und der Zielpunkt angegeben werden und die Cylinder Klasse erzeugt dann einen Zylinder, der diese beiden Punkte verbindet. Zusätzlich kann noch der Durchmesser des Zylinders angegeben werden. Die Cylinder Objekte werden für die Arrow Objekte gebaucht (siehe Abschnitt 4.1.4.1 Arrow).

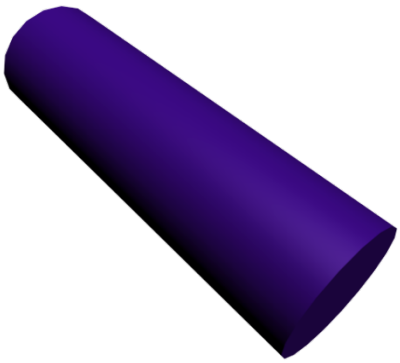


Abb. 5. Cylinder Objekt

4.1.3.2 CylinderConnection

Die CylinderConnection Klasse repräsentiert ein Verbindungsstück zwischen zwei Zylindern, die im 90° Winkel zueinander stehen. Zur Erstellung müssen nur zwei Zylinder übergeben werden, die im 90 ° Winkel zueinander stehen. Genau wie die Cylinder Objekte werden die CylinderConnection Objekte für die Arrow Objekte gebraucht (siehe Abschnitt 4.1.4.1 Arrow).

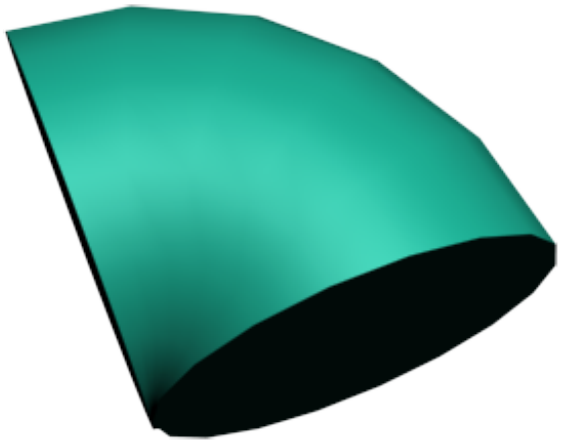


Abb. 6. CylinderConnection Objekt

Abb. 7. CylinderConnection Objekt verbunden mit 2 Cylindern Objekten

4.1.3.3 Multiplexer

Die Multiplexer Klasse repräsentiert einen Multiplexer, der mehrere Eingänge und einen Ausgang hat. Die Größe ist variabel und er kann in jede Richtung gedreht werden (die „spitze“ kann in X positiv und negativ sowie in Z positiv und negativ Richtung zeigen).

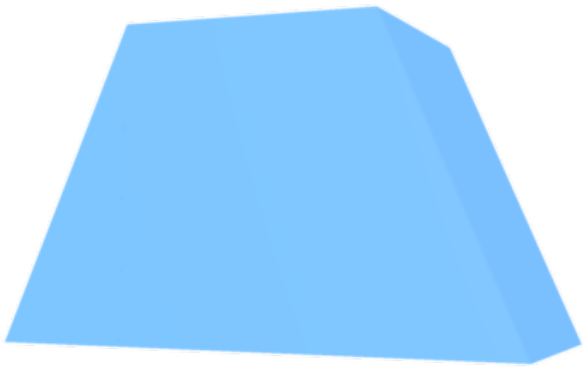


Abb. 8. Multiplexer

4.1.3.4 Rectangles

Rectangles repräsentiert ein Rechteck. Ein Rechteck steht für verschiedene Bauteile des Prozessors. Beispielsweise steht ein Rechteck für die Register. Die Seitenlängen (Höhe, Breite, Länge) können beliebige Werte haben.

Abb. 9. Rectangles Objekt

4.1.3.5 Alu

Die Alu Klasse stellt die ALU dar. Sie hat zwei Eingänge und einen Ausgang. Genau wie der Multiplexer kann die Größe verändert werden und die Alu kann in jede Richtung zeigen.

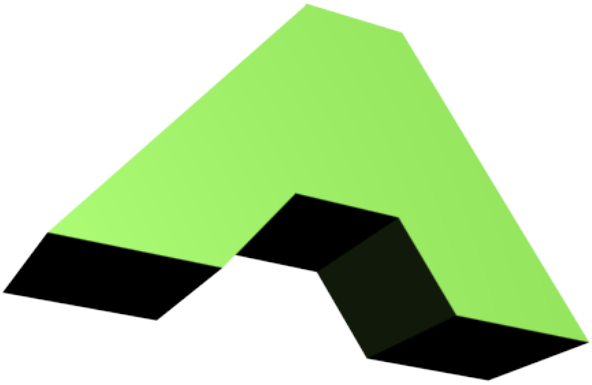


Abb. 10. Alu Objekt

4.1.3.6 ArrowHead

Die Klasse ArrowHead repräsentiert eine Pfeilspitze. Sie wird nur im Zusammenhang mit dem Arrow genutzt (siehe Abschnitt 4.1.4.1 Arrow).

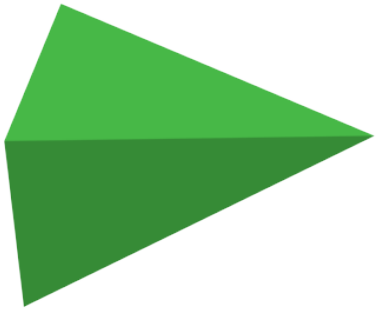


Abb. 11. ArrowHead Objekt

4.1.4 GrafikList

Alle bis jetzt vorgestellten Grafikelemente sind sehr statisch. Es ist zwar möglich die Größe, die Ausrichtung und die Position zu verändern, aber im Endeffekt bleiben es genau die gleichen Grafikelemente. Dieses Konzept ist für die Erstellung der Pfeile, die die Busse darstellen, viel zu unflexibel. Um die Möglichkeit zu haben flexible Grafikelemente zu erzeugen, wurde ein Konzept entwickelt, dass es ermöglicht, ein Grafikelement aus vielen verschiedenen Grafikelementen zusammen zusetzten. So können die einzelnen Bestandteile vorher mittels 3D Editor erstellt werden und werden dann entsprechend den Anforderungen zusammen gesetzt. Hierfür gibt es eine Hauptklasse die GrafikList heißt. Sie bietet die gleichen Methoden wie die Grafikklasse an und ermöglicht dadurch, dass die beiden Klassen austauschbar sind. Die GrafikList Klasse ermöglicht es beliebig viele Grafikelemente hinzuzufügen. Die GrafikList Klasse erstellt selbst keine Grafikelemente, sondern ist nur eine Container Klasse für die Grafikelemente.

4.1.4.1 Arrow

Die Arrow Klasse erbt von der Klasse GrafikList. Ein Arrow, also ein Pfeil, besteht aus verschiedenen Teilen, und zwar aus einer Pfeilspitze (ArrowHead), aus Zylindern (Cylinder) und aus Zylinderverbindungsstücken (CylinderConnection). Die Arrow Klasse stellt in der grafischen Darstellung des Prozessors die Busse dar. Ein Arrow kann eine oder zwei Spitzen haben (zwei Spitzen um eine bidirektionale Verbindung darzustellen).

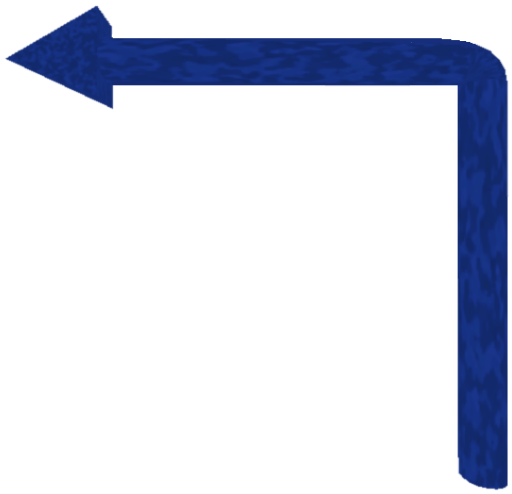


Abb. 12 Arrow

Es gibt zwei Möglichkeiten um den Pfeil darzustellen. Entweder ist er gerade oder er geht über Ecken. Ist der Pfeil gerade, gibt man nur Start- und Zielkoordinaten an. Dann wird ein Pfeil erstellt, der zwischen diesen beiden Koordinaten liegt. Geht der Pfeil über Ecken, gibt es zwei verschiedene Möglichkeiten diese Eckpunkte festzulegen. Die erste Möglichkeit ist, die Koordinaten direkt anzugeben, an den die Ecken sein sollen. Die zweite Möglichkeit ist, einen Float Wert anzugeben, der die Eckpunkte in Abhängigkeit zum letzten Eckpunkt (oder beim ersten in Abhängigkeit zum Startpunkt) festlegt. Nachfolgend wird folgendes Beispiel aufgeführt:

Folgende Voraussetzungen sind gegeben:

- Koordinaten des letzten Eckpunktes: x: 100 y:2,5 z: 100.
- Richtung des nächsten Eckpunktes Positive x Richtung.
- Der Float Wert, der die Verschiebung angibt ist 50.

Koordinaten des zweiten Eckpunktes:
x = x des letzten Eckpunktes + Float Wert = 100 + 50 = 150
y und z behalten ihren Wert.
Die Koordinaten für den zweiten Eckpunkt sind also x: 150 y: 2,5 z: 100

4.1.5 Positionierung der Grafikelemente

Eines der größten Probleme bei der Erstellung von Grafiken ist Positionierung der einzelnen Grafikelemente. Die Koordinaten der einzelnen Grafikelemente müssen aufeinander abgestimmt werden und das möglichst mit geringem Aufwand. Es gibt verschiedene Varianten, um das zu realisieren.

Variante 1: Werte selbst berechnen

Die „einfachste“ Lösung für dieses Problem ist, die Koordinaten vorher selbst zu berechnen und direkt einzutragen. Diese Variante hat den Vorteil, dass kein zusätzlicher Programmieraufwand vorhanden ist. Der Nachteil dieser Variante ist allerdings, dass sie vollkommen unflexibel ist. Will man beispielsweise alle Elemente um 100 Pixel nach oben verschieben, muss man an jedem Element die Werte einzeln verändern. Ein zusätzlicher Nachteil dieser Variante ist, dass es relativ aufwendig ist alle Koordinaten selbst auszurechen. Besonders komplex wird es, wenn verschiedene Grafikelemente in Abhängigkeit voneinander sind. Wie beispielsweise die Register und die Alu, die durch einen Pfeil verbunden sind.

Variante 2: Entwicklung eines Editors

Die beste und eleganteste Lösung wäre vermutlich ein Editor gewesen, in dem man die einzelnen Grafikelemente positionieren kann. Diese Variante hätte viele Vorteile. Es wäre sehr einfach die Grafikelemente zu positionieren. Das System wäre sehr flexibel und man würde sofort sehen, wie alle Elemente zusammen passenAllerdings hat dieser Ansatz auch einen sehr großen Nachteil. Der Programmieraufwand ist sehr hoch, wenn man einen vernünftig funktionierenden Editor haben möchte. Da die Zeit dafür nicht vorhanden war, musste dieser Ansatz auch verworfen werden.

Variante 3: System zur Berechnung der Koordinaten

Die 3. Variante ist ein System, das dem Programmierenden die Berechnung der Koordinaten weitgehend abnimmt. Hierbei ist es wichtig, dass es möglich ist, Abhängigkeiten von einzelnen Grafikelementen festzulegen und das durch diese Abhängigkeiten die Koordinaten berechnet werden. Auch bei Veränderungen von Positionen einzelner Grafikelemente sollte dies sichergestellt sein.

Dieses Konzept hat den Vorteil, dass es flexibel ist und sich gleichzeitig der Programmieraufwand in Grenzen hält. Der Nachteil diese Konzeptes ist, dass man wie bei

der ersten Variante nur im Code arbeitet und erst beim Starten des Programmes sieht, wie die Grafikelemente wirklich angeordnet sind bzw. ob sie so angeordnet sind, wie man es gerne möchte. Dieses Konzept ist der ersten Variante weit überlegen und weil die zweite Variante aufgrund von Zeitmangel nicht implementiert werden konnte, wurde die dritte Variante implementiert.

Der ConnectionPoint

Für die Implementierung der dritten Variante wurde der UML Editor UMLet als Vorbild genommen. Durch UMLet können Pfeile an beliebigen Stellen der Außenkante eines Grafikelements (UML Element) befestigt werden. Die Pfeile befestigen sich automatisch mit der Außenkante, sobald man in die Nähe einer Außenkante klickt und lassen sich dann an der Außenkante beliebig verschieben.

Abb. 13. UMLet. Pfeil ist noch nicht befestigt

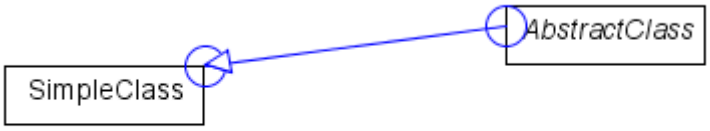


Abb. 14. UMLet. Pfeil ist befestigt

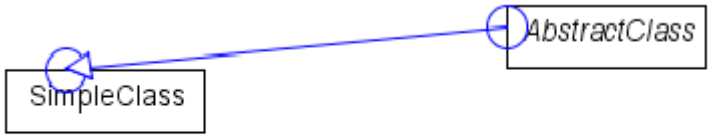


Abb. 15. UMLet. Pfeil wurde verschoben

Der Ansatz wurde übernommen und zwar kann jedes Grafikelement sogenannte ConnectionPoint erstellen. Ein ConnectionPoint ist ein Punkt an der Außenkante eines Grafikelementes. An welchen Stellen der Außenkante diese Punkte genau erzeugt werden können, ist abhängig von dem Grafikelement.

Das Konzept wird anhand eines Rechtecks (die Rectangles Klasse) erläutert. Das Rechteck stellt zwei verschiedene Möglichkeiten zur Verfügung, um Connection Points zu erstellen. Die erste Möglichkeit ist die Erstellung eines ConnectionPoints an einer beliebigen Stelle der

Außenkante. Hierfür wird die Kante angegeben (die Achse in die Kante Zeigt (AXIS)) und der Abstand zum Nullpunkt des Rechtecks.

Die zweite Möglichkeit ist eine Möglichkeit, die eigentlich nur zur Arbeitserleichterung dient. Hierbei wird, wie bei der ersten Möglichkeit, die Kante angegeben und als zweiter Wert wird die Position angegeben. Es gibt jeweils 3 Positionen und zwar links, Mitte und rechts (die Position Angaben beziehen sich auf die Ansicht eines Betrachters (Draufsicht) der direkt vor der jeweiligen Seite steht). Die Werte werden dann abhängig von der Größe des Rechtecks berechnet.

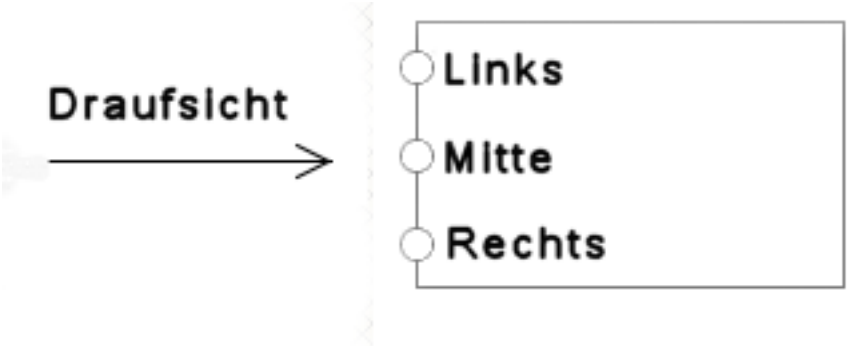


Abb. 16. Rechteck mit 3 ConnectionPoints

Für jedes Grafikelement gibt es speziell auf das Element bezogende Methoden, abhängig davon, was nötig ist. Zum Beispiel gibt es bei der Alu nur 3 Möglichkeiten, ConnectionPoints zu erstellen. Einen jeweils für die beiden Eingänge und einen für den Ausgang. Das Konzept ist nicht nur darauf beschränkt Grafikelemente direkt zu verbinden. Es ist auch sehr hilfreich, wenn man verschiedene Elemente aneinander ausrichten möchte. Für diesen Zweck gibt es auch spezielle Methoden, die einen dabei unterstützen.

4.2 Die Grafische Darstellung des Prozessors

Der entscheidende Punkt zum Verständnis des Prozessors ist die Darstellung. Je klarer zu erkennen ist, was im Prozessor passiert, umso einfacher ist die Funktionsweise des Prozessors zu verstehen.

4.2.1 Der Ansatz

Die grundsätzliche Idee bei der Grafischen Darstellung war es ein System zu entwickeln, welches einem echten Prozessor nachempfunden ist. Es sollte modellhaft zu erkennen sein, wie die verschiedenen Komponenten eines Prozessors verbunden sind. Außerdem soll deutlich zu erkennen sein, wie die Instruktion durch den Prozessor wandert und wie die einzelnen Komponenten zusammen wirken.

4.2.2 Detailtiefe

Eine der wichtigsten Entscheidungen war die Detailtiefe. Ist sie zu gering, kann man die Funktionsweise des Prozessors nicht verstehen, weil wichtige Informationen fehlen. Ist sie zu groß, wird der Prozessor zu unübersichtlich. Um einen guten Kompromiss zu finden wurde entschieden, dass alle wichtigen Komponenten dargestellt werden, aber in der grafischen Darstellung nicht darauf eingegangen wird, was innerhalb der Komponenten passiert. Bei der Darstellung des Prozessors wurde als Grundlage eine Zeichnung aus dem Buch ARM system-on-chip architecture [Fur00] genommen.

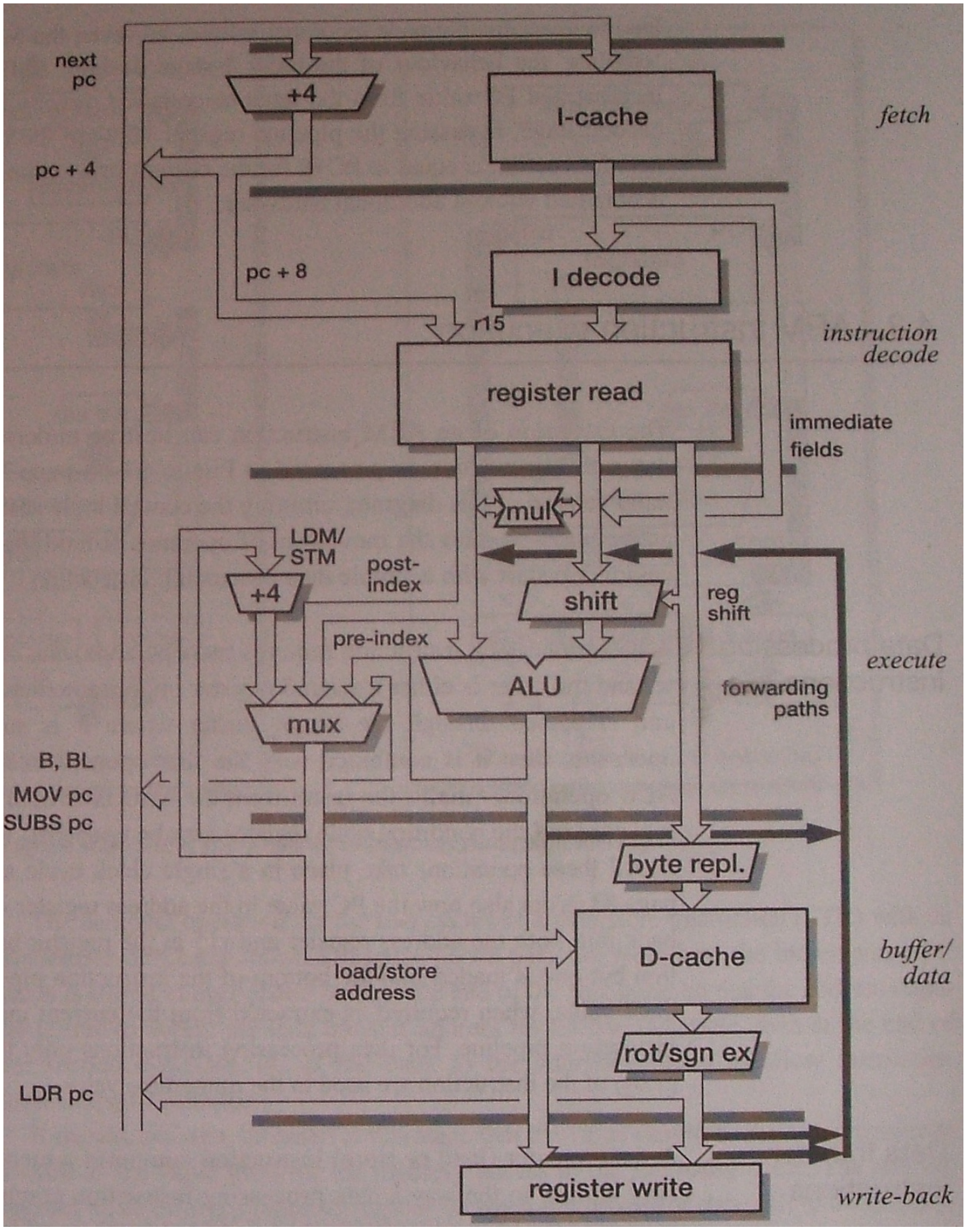


Abb. 17. Abbildung aus dem Buch ARM system-on-chip architecture

4.2.3 Die einzelnen Komponenten

Die einzelnen Komponenten (Alu, Register usw.) des Prozessors sind statisch. Sie verändern sich nicht. Auf jeder Komponente steht, um was es sich dabei handelt.

4.2.4 Die Busse

Die Busse sind einer der wichtigsten Teile des Prozessors. Sie stellen die Verbindung zwischen den einzelnen Komponenten dar. Die Busse werden durch Pfeile dargestellt. Die Pfeilspitze zeigt an, in welche Richtung der Bus läuft. Ein Bus kann zwei verschiedene Zustände haben, aktiv und inaktiv.

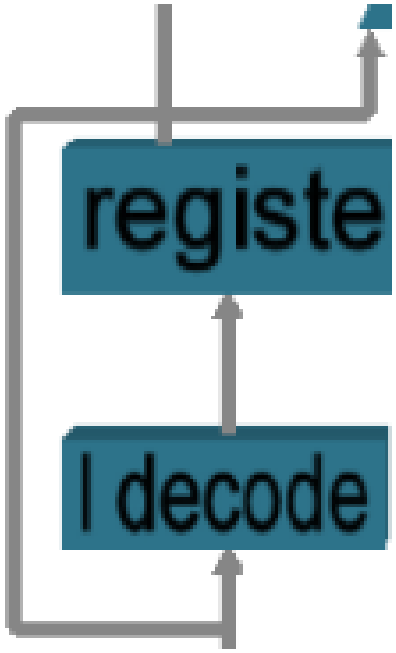


Abb. 18. Inaktiver Bus

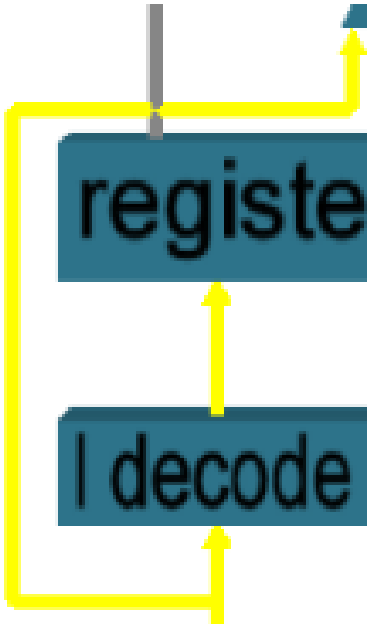


Abb. 19. Aktiver Bus

Wenn ein Bus aktiv ist bedeutet das, dass in der aktuellen Instruktion Daten durch diesen Bus geflossen sind (die aktuelle Instruktion ist die Instruktion, die sich gerade in der Pipeline Stufe befindet, zu der der Bus gehört). Was für Daten durch den Bus geflossen sind, ist abhängig von den verschiedenen Bussen. Beispielsweise ist der linke Bus in Abb. 19 der Immediate Operant.

4.2.4.1 Die Farben der Busse

Jeder Instruktion wird beim Einlesen in die Fetch Stufe eine Farbe zugeordnet. Diese Farbe begleitet die Instruktion durch alle Stufen. Die Farbe entspricht auch der Farbe, die die Busse im aktiven Zustand haben. Die farbliche Markierung der einzelnen Instruktionen zieht sich durch die ganze GUI. Jedes mal, wenn klar gemacht werden soll, dass bestimmte Daten zu einer Instruktion gehören, werden die Daten mit der Farbe, die der Instruktion zugordnet wird, markiert.

4.2.5 Eine Instruktion

Jede Instruktion durchläuft die 5 verschieden Pipeline Stufen. In jeder Stufe werden verschiedene Werte berechnet oder ausgelesen, die dann über den Bus weitergeleitetet werden. Abhängig davon, was die einzelne Instruktion genau macht, werden die Leiterbahnen aktiviert. Liest zum Beispiel eine Instruktion den zweiten Operanten für die Alu aus dem Register, wird der entsprechende Bus aktiviert. Um die einzelnen Instruktionen voneinander unterscheiden zu können, ist jeder Instruktion eine Farbe zugeordnet (in der Farbe werden die Busse gefärbt). Anhand der Farben der Leitbahnen erkennt man, in welcher Pipeline Stufe sich die Instruktion gerade befindet.

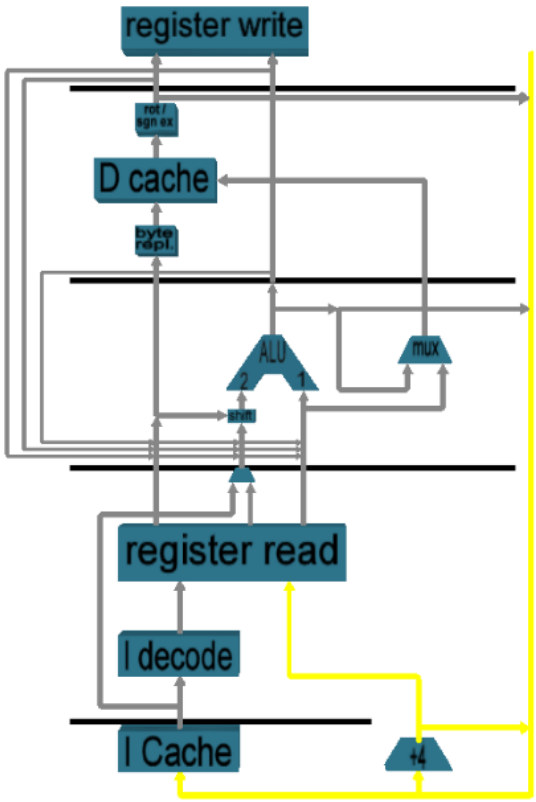


Abb. 20. Durchlauf einer Instruktion - Fetch

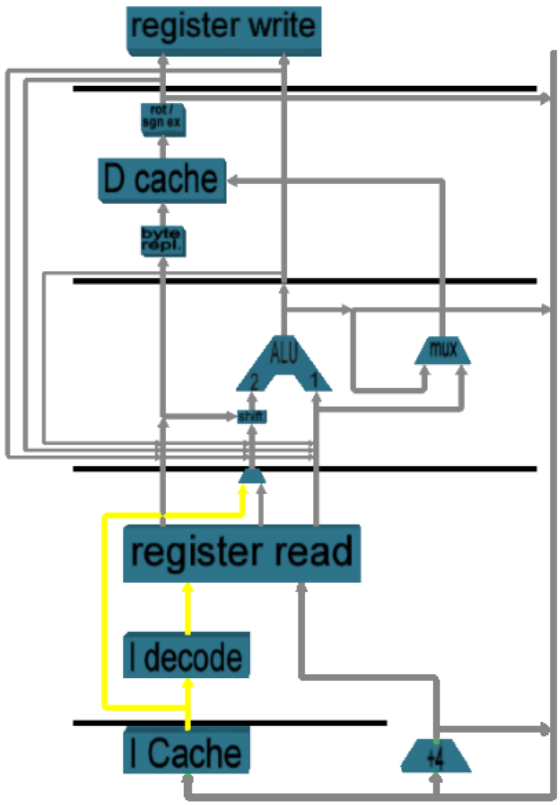


Abb. 21. Durchlauf einer Instruktion - Decode

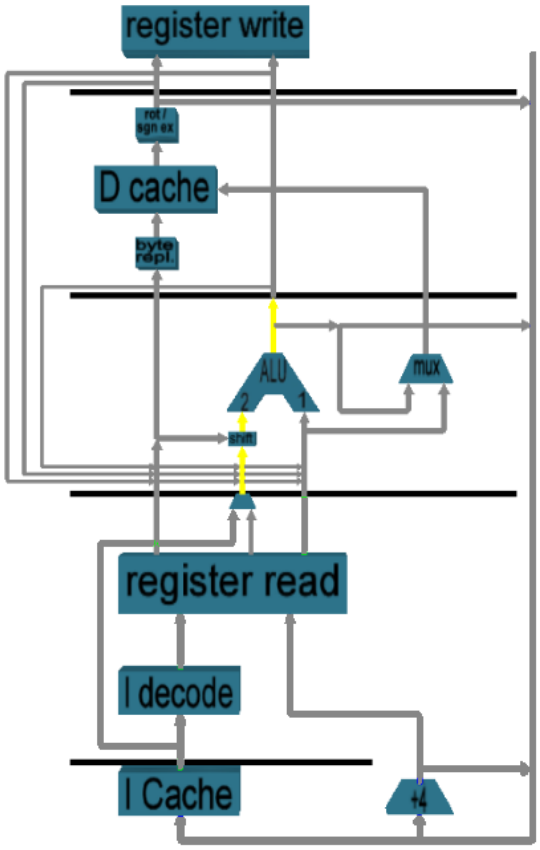


Abb. 22. Durchlauf einer Instruktion - Execute

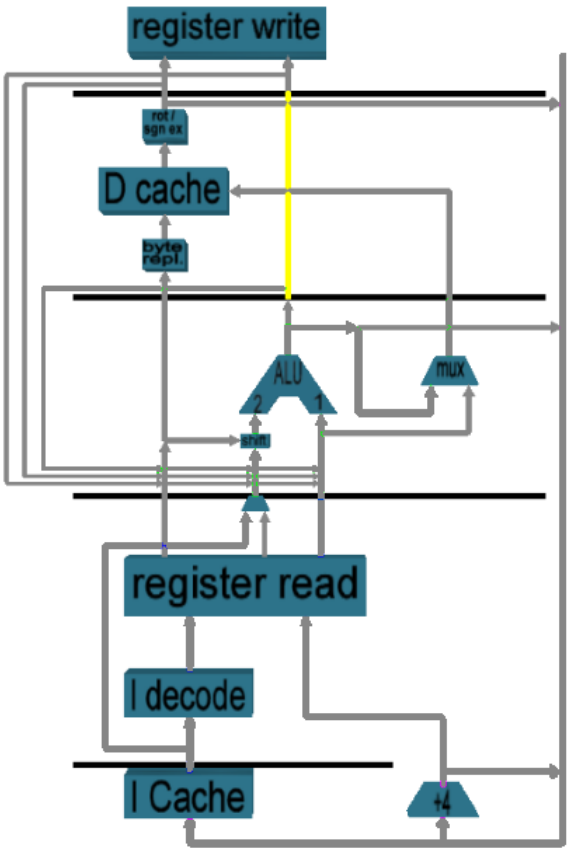


Abb. 23. Durchlauf einer Instruktion - Memory Access

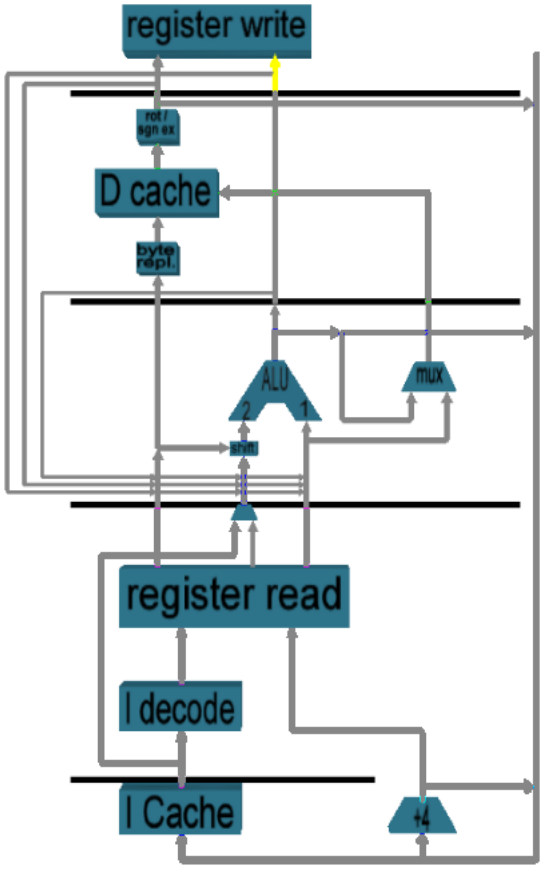


Abb. 24. Durchlauf einer Instruktion - Write Back

Im den Abb. 20-24 sieht man den durchlauf einer Mov Instruktion (bei der Mov Instruktion wird ein Register mit dem Wert eines Anderen Registers geladen oder mit einem Immediate Wert). In der ersten Abbildung sieht man das einlesen der Instruktion und das der Programmzähler um 4 erhöhte wird. In der zweiten Abbildung sieht man das die Instruktion decodiert wird und das der zweite Operant ein Immediate Wert und kein Register ist (linker Pfeil). In der nächsten Abbildung ist zu sehen wie der zweite Operant durch die ALU geschleust wird, der Wert des Operant wird dabei nicht verändert. In der vierten Abbildung kann man beobachten das der Wert nur zwischen gespeichert wird um dann in der letzten Abbildung in ein Register übertragen zu werden.

4.3 Das Tab System

Ein wichtiger Faktor für das Verständnis ist die gute Aufbereitung der Informationen. Einerseits muss die Möglichkeit bestehen an alle relevanten Informationen heran zu kommen, anderseits ist es natürlich auch wichtig, dass die Nutzer nicht mit Informationen überladen werden. Um eine guten Aufbau zu finden wurden als erstes einmal verschiedene GUI's von anderen Programmen analysiert. Besonders wichtig war dabei eine einfache Handhabung und die Möglichkeit Informationen ein- und auszublenden.

Besonders positiv ist das Tab System von dem 3D Editor 3D Studio Max [Wen06] aufgefallen. Die Darstellung mit Hilfe von Tabs ist den *Registerkarten* aus Aktenschränken nachempfunden. Tabs dienen dazu, Informationen und Eingabefelder eines Programmfensters auf mehreren, hintereinander liegenden Dialogfeldern anzuordnen (die Dialogfelder befinden sich hierbei auf den Tabs, zum Tab gehört also ein Reiter zur Auswahl des Tabs und eine Fläche auf der ein Dialogfeld angeordnet werden kann). Dabei befindet sich immer ein Tab im Vordergrund. Durch Anklicken des entsprechenden Reiters kann man einen anderen Tab in den Vordergrund holen, wobei die Informationen und ggf. getätigten Einstellungen auf den vorher genutzten Tab bestehen bleiben.

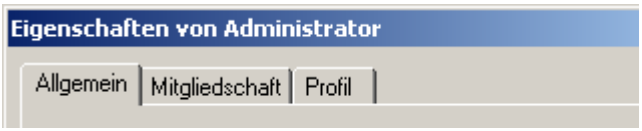


Abb. 25. Tabs (die Reiter der Tabs) unter Windows 2000

Innerhalb der Tabs bei 3D Studio Max gibt es verschiedene Menüpunkte, die alle ein- und ausgeblendet werden können. Dieses Konzept ist besonders positiv aufgefallen, weil so die Möglichkeit alles, was im Moment nicht wichtig ist, auszublenden besteht. Dies erlaubt dem Nutzer sich auf das Wesentliche zu konzentrieren. Da diese sehr vorteilhaft ist wurde auch versucht die Optik von dem gesamten 3D Studio Max Tab System zu übernehmen (soweit es für die Anwendung sinnvoll war).

Das Tab System von 3D Studio Max wurde für das Implementierte Tab System als Vorbild genommen.

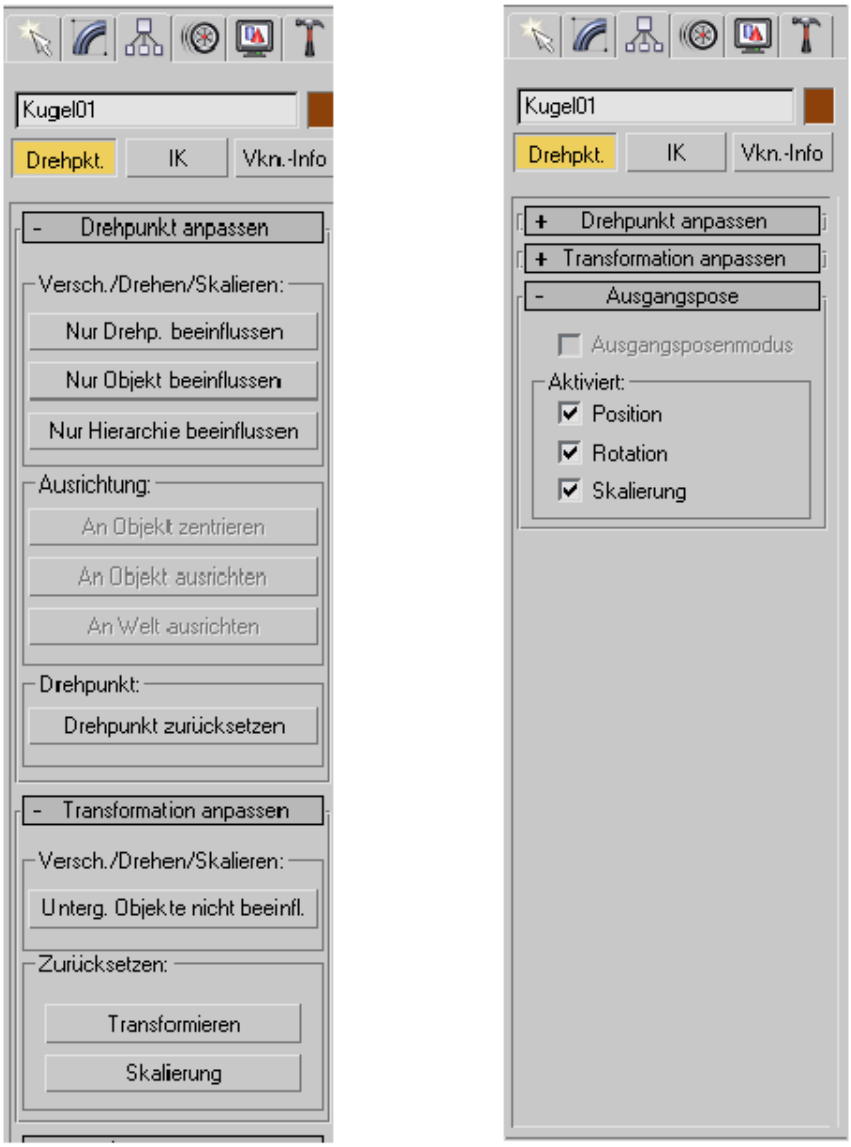


Abb. 26. 3D Studio Max Tab System

4.3.1 Die verschiedenen Tabs

Insgesamt hat gibt es 4 verschiedene Tabs. Die in diesem Abschnitt vorgestellt werden.

4.3.1.1 Register und Pipeline Stufen Tab

Im ersten Tab findet man die Register, sowie die 5 Pipeline Stufen. Die einzelnen Stufen und die Register kann man ein- und ausblenden. Das hat den Vorteil, dass man alle Informationen, die im Moment nicht interessieren, ausblenden kann.

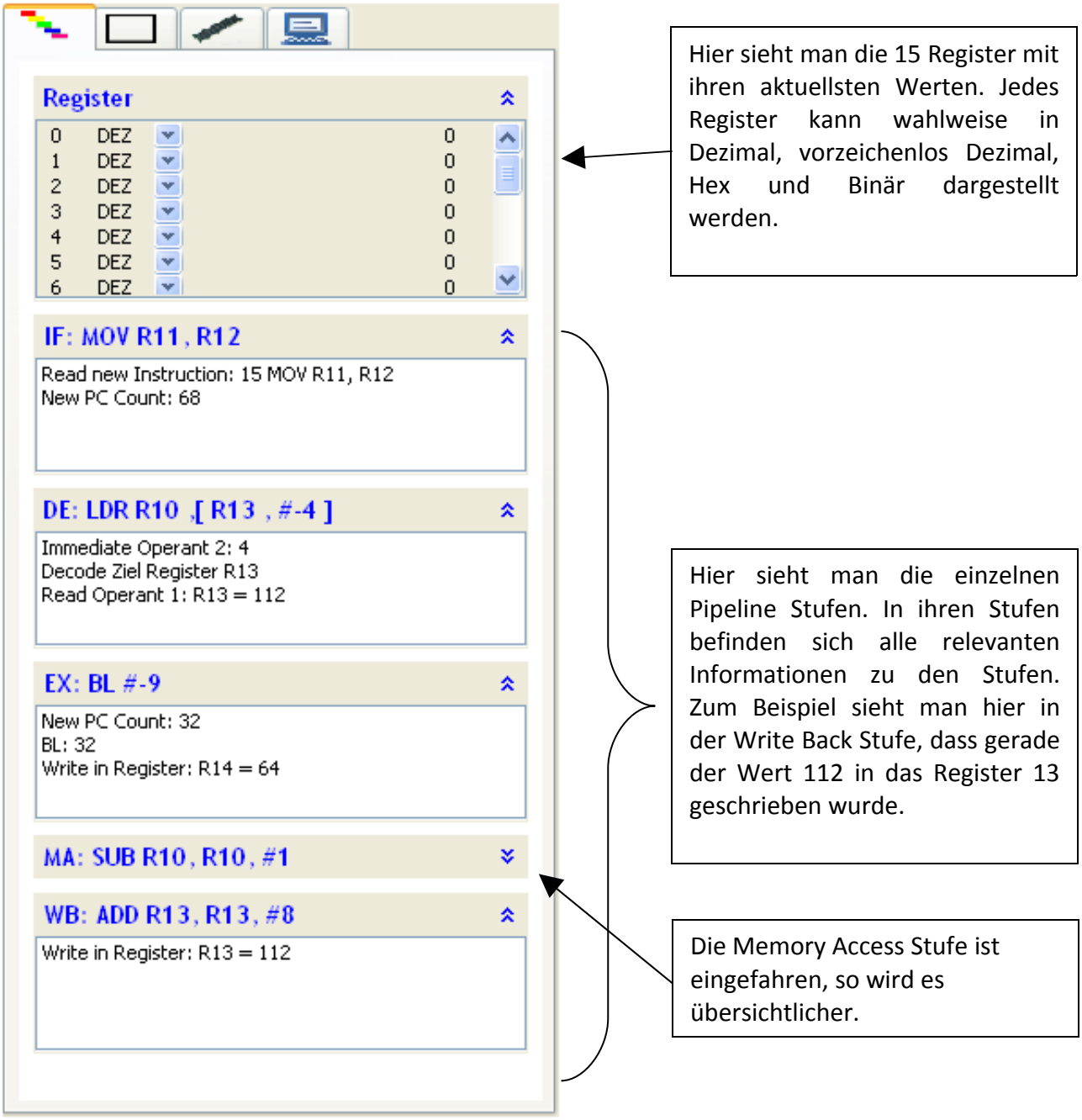


Abb. 27. Register und Pipeline Stufen Tab

4.3.1.2 Instruktion Tab

Das zweite Tab ist das Instruktion Tab. Hier kann man sich den Verlauf einer Instruktion anschauen. Die Auswahl der einzelnen Instruktionen erfolgt durch die Radio Button am oberen Ende des Tabs. Der erste Button steht immer auch für die erste Pipeline Stufe, der zweite für die zweite usw. Neben dem Radio Button wird noch die Farbe, die der Instruktion zugeordnet ist, angezeigt.

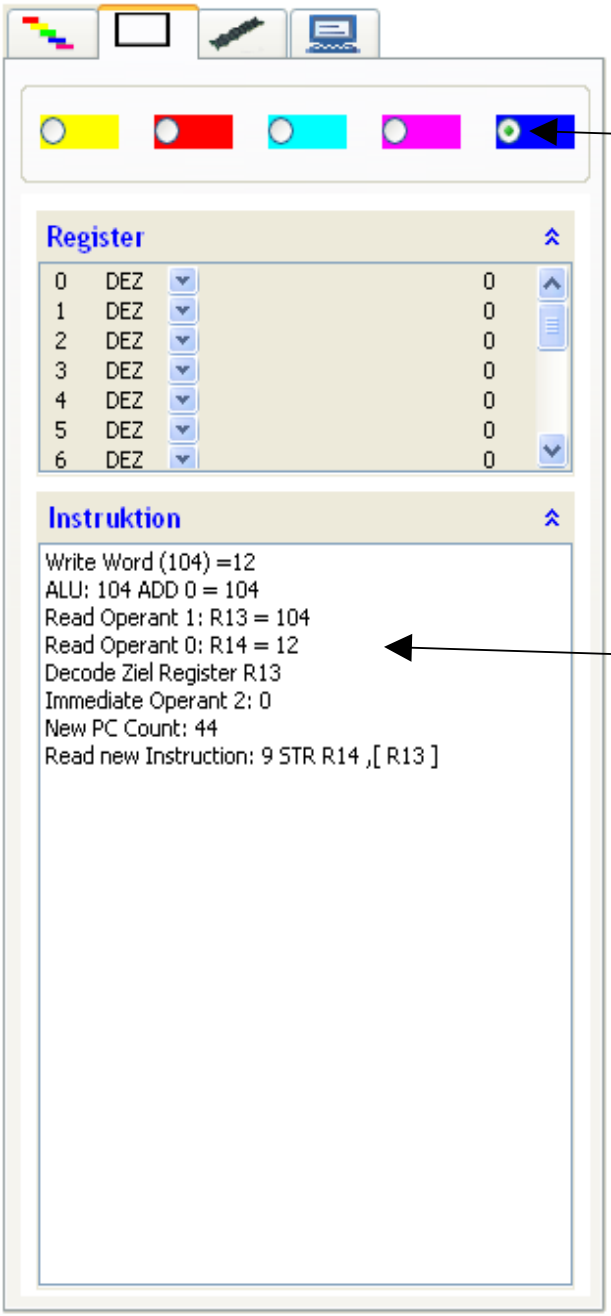
Abb. 28. Die Radio Button, die Instruktion die sich in der Write Back Stufe befindet ist ausgewählt

Wenn man eine Instruktion auswählt (Abb. 29), wird bei jedem Clock Signal entsprechend der nächste Radio Button eigenständig ausgewählt (Abb. 30). Diese führt zwar zu einem etwas unruhigen Bild, aber dadurch wird deutlich, welche Instruktion gerade angezeigt wird. Die Gefahr der Verwesung der Pipeline Stufen ist mit der Variante weitaus geringer.

Abb. 29. Gelbe Instruktion in der Decode Stufe

Abb. 30. Nach einem Clock Zyklus, Gelbe Instruktion in der Execute Stufe

Als nächstes befindet sich im Instruktion Tab, genau wie im Pipeline Stufen Tab, die Anzeige der Register. Als letztes sieht man das Instruktionsfenster. Hier werden alle Informationen zu dieser Instruktion dargestellt. Also alle Informationen der einzelnen Stufen, die die Instruktion schon durchlaufen hat, z.B. die Instruktion befindet sich grade in der Execute Stufe. Dann wird angezeigt, was in der Fetch, Decode und Execute Stufe passiert ist. Angefangen von der Fetch Stufe bis zur Write Back Stufe. Sobald eine Instruktion die Write Back Stufe erreicht hat, wird beim nächsten Clock Signal automatisch auf die Instruktion umgeschaltet, die neu in die Fetch Stufe kommt.



Die Instruktion, die sich in der Write Back Stufe befindet, ist ausgewählt.

In der Write Back Stufe befindet sich eine STR (Read new Instruction: 11 STR R12,[R13]) Instruktion, die den Wert aus Register 12 an die Adresse, die sich im Register 13 befindet, schreibt. Das heißt in diesem Fall, dass der Wert 12 an Adresse 104 geschrieben wird (Write Word (104) = 12). Außerdem sieht man anhand der Radio Button, dass sich die Instruktion gerade in der Write Back Stufe befindet.

Abb. 31. Instruktion Tab

4.3.1.3 Arbeitsspeicher Tab

Im Arbeitsspeicher Tab wird der gesamte Arbeitsspeicher dargestellt. Der Arbeitsspeicher wird mit 32 Bit Werten dargestellt, jede Zeile entspricht also 4 Byte. Vorne wird die Adresse angegeben (die Adresse kann als Dezimal und als Hexadezimal Wert dargestellt werden) und hinten befindet sich der Wert. Wie auch schon bei Register können die Werte in 4 verschiedenen Formen dargestellt werden: Dezimal, vorzeichenlos Dezimal, Hexadezimal und Binär. Allerdings kann hier nur der gesamte Arbeitsspeicher umgestellt werden. Das Umstellen von einzelnen Werten ist nicht möglich.

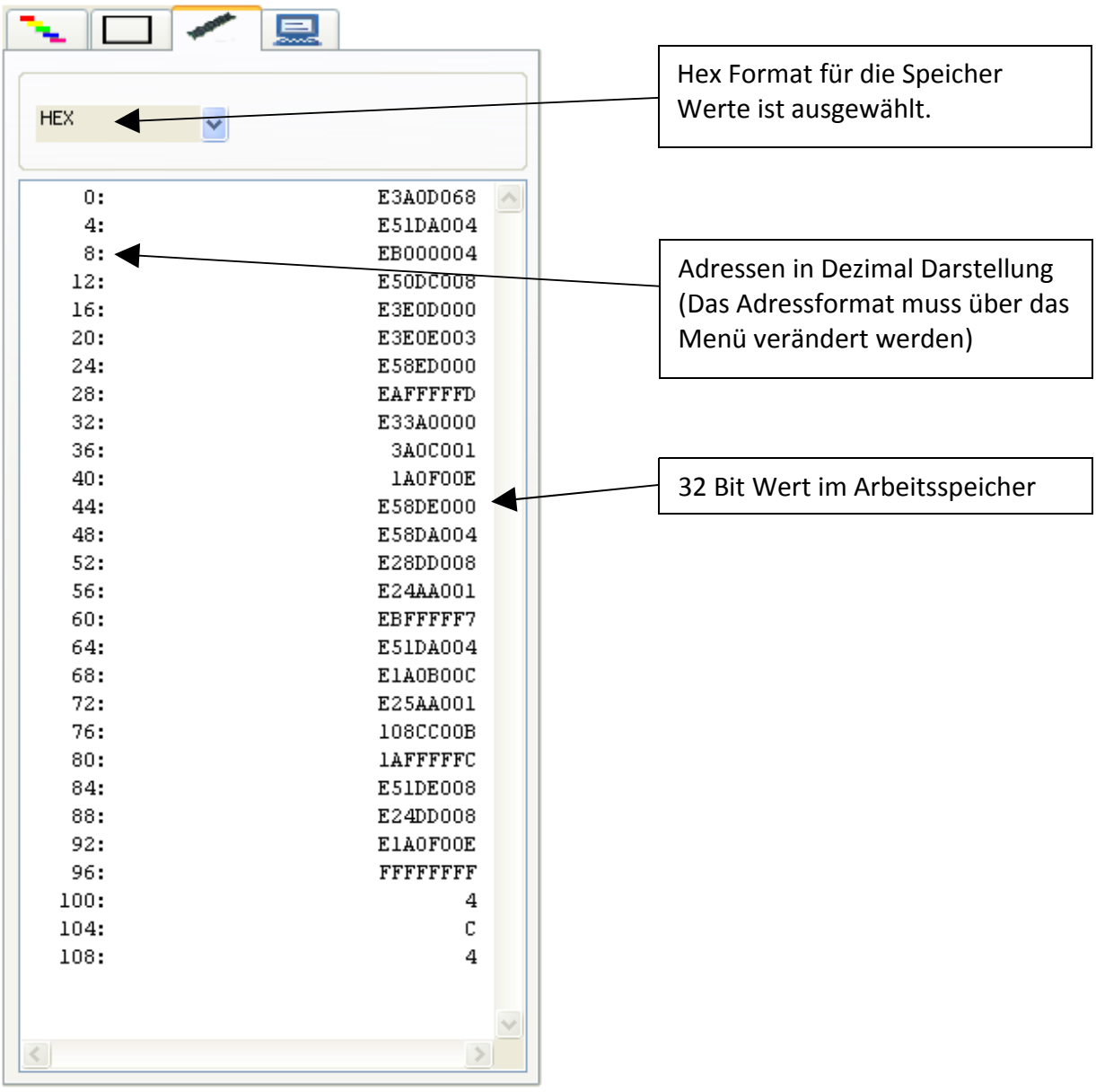


Abb. 32. Arbeitsspeicher Tab

4.3.1.4 Disassembler Tab

Dieses Tab ist ähnlich dem Arbeitsspeicher Tab. Allerdings wird hier nicht der Byte Code, sondern die Assembler Befehle angezeigt. Jede Zeile entspricht einem 32 Bit Word. Genau wie im Arbeitsspeicher Tab befindet sich vor der Instruktion die Adresse. Zeilen können durch verschiedene Farben markiert sein. Diese Farben repräsentieren die entsprechenden farbigen Instruktionen (die Instruktionen, die in der Prozessordarstellung die entsprechenden Farben haben).

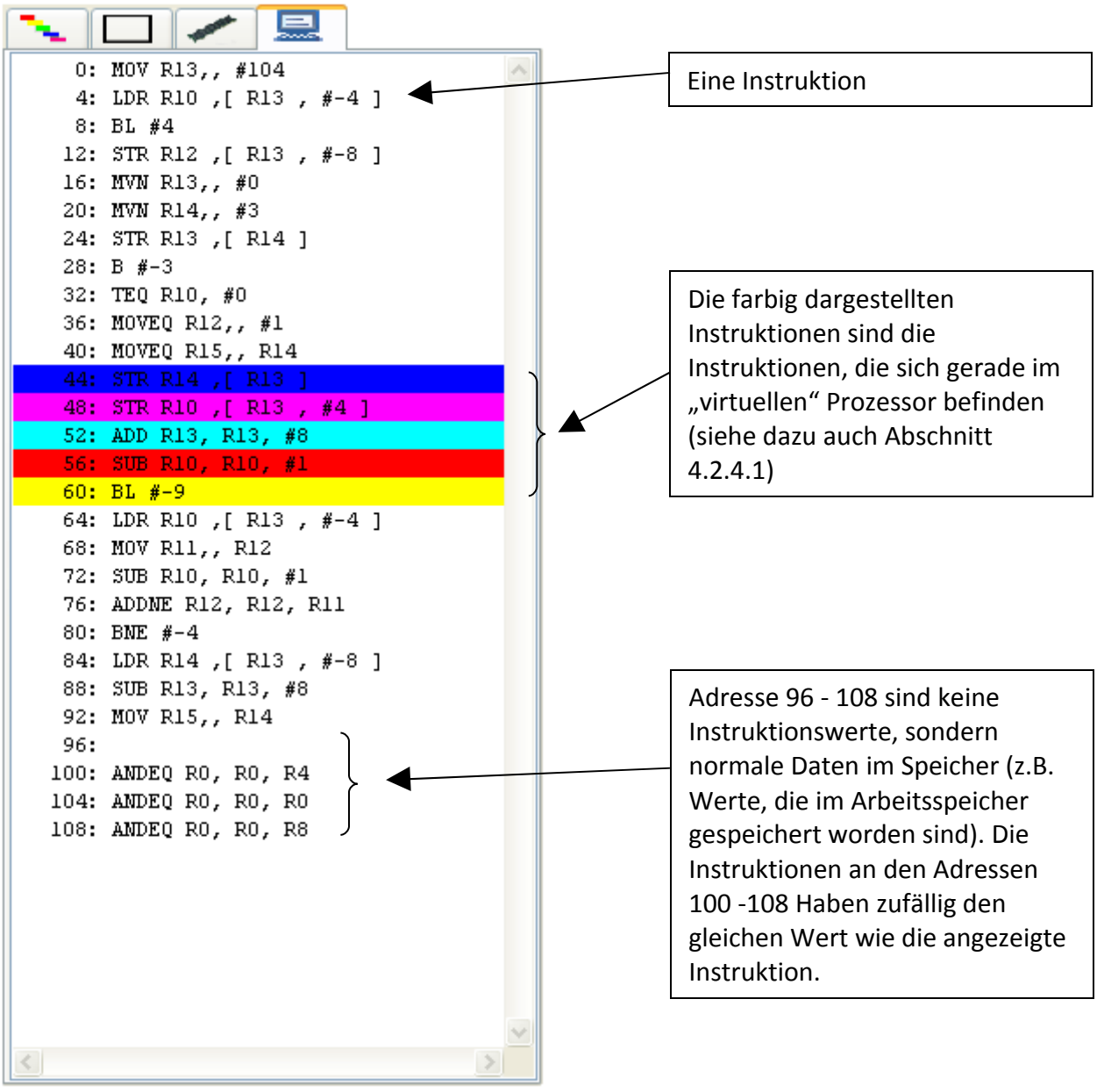


Abb. 33. Disassembler Tab

5 Die Kommunikation zwischen Grafik und Prozessor

In diesem Kapitel geht es um die Kommunikation zwischen Grafik und Prozessor. Wichtig war es eine Möglichkeit zu finden, die flexibel und erweiterbar ist.

5.1 Art der Kommunikation

Als erstes wird die Art der Kommunikation festgelegt. Im Folgenden werden zwei Varianten erläutert.

5.1.1 Einfache Variante

Die einfachste Variante für die Kommunikation zwischen Prozessor und Grafik ist, dass man vom Prozessor aus direkt auf die Grafikelemente zugreift. Also jedes Mal, wenn der Prozessor etwas berechnet bzw. irgendwas passiert, was dargestellt werden soll, werden die Vorgänge direkt durch einen Methodenaufruf auf die GUI übertragen. Der ganz klare Vorteil dieser Variante ist, dass sie sehr einfach zu implementiert ist.

Allerdings hat dieser einfache Ansatz auch sehr viele Nachteile. Eines der Probleme ist, das die Funktionsweise des „virtuellen“ Prozessors nicht genau mit der des echten Prozessors übereinstimmt. Das führt dazu, dass die Reihenfolge, in der die verschiedenen Werte im „virtuellen“ Prozessor berechnet werden, nicht immer der Anzeige Reihenfolge in der GUI entspricht. Ein Beispiel hierfür ist, dass die Pipeline Stufen im „virtuellen“ Prozessor nacheinander berechnet werden, aber bei echten Prozessoren parallel ablaufen und auch so dargestellt werden sollen.

Das nächste Problem ist, dass der „virtuelle“ Prozessor sequenziell arbeitet, die Darstellung in der GUI aber parallel sein muss. Wenn also im Prozessor direkt die GUI angesprochen werden soll, kann es vorkommen, dass es so wirkt, als ob Sachen hintereinander passieren, die in Wirklichkeit parallel ablaufen. Das dritte Problem ist, dass es keine lose Kopplung gibt. Wird etwas an der GUI geändert, muss auch am Prozessor etwas geändert werden, was zu mehr Arbeitsaufwand führen würde. Das letzte Problem ist, dass der „virtuelle“ Prozessor unübersichtlicher wird, weil sehr viel GUI Code mit eingefügt werden muss.

5.1.2 Verbesserte Variante

Bei der verbesserten Variante ist es so, dass alle relevanten Informationen eines Clock Zyklus, die zur Darstellung gebraucht werden, gespeichert werden und am Ende des Clock Zyklus zur Verarbeitung weitergegeben werden. Ein Beispiel: in der Execute Stufe werden 5 und 10 addiert. Nun werden die Werte 5 und 10, das Ergebnis 15 und das addiert wurde

gespeichert. Sobald ein Clock Zyklus vorbei ist, werden die gespeicherten Daten weitergereicht. Dies hat den Vorteil, dass die GUI Klasse am Ende eines Clock Zyklus alle relevanten Informationen zur Darstellung gleichzeitig hat und selbst entscheiden kann, wann und wie sie die Informationen darstellt. Die GUI Klasse hat jetzt die vollkommene Freiheit. Sie kann sich entscheiden, wie und zu welchem Zeitpunkt sie, was darstellt.

Ein anderer Vorteil ist, dass bei Veränderungen der GUI Klasse in der Regel keine Veränderungen an der Prozessor Klasse vorgenommen werden müssen. Außerdem ist der zusätzliche Code für die Prozessor Klasse relativ gering. Das Konzept sorgt zwar für sehr viel mehr Freiheiten, ist dafür aber auch weitaus komplexer als die erste Variante. Es muss zusätzliche Logik in die GUI Klasse, die die Verarbeitung der Informationen übernimmt, was bei der ersten Variante nicht der Fall ist. Eine nützliche Erweiterung des Konzeptes ist es, wenn man die Informationen der vorherigen Pipeline Stufen speichert und den Informationen der aktuellen Stufe hinzufügt. Dies führt dazu, dass kein zusätzlicher Aufwand betrieben werden muss, um relevante Informationen zur Darstellung der aktuellen Stufe aus Informationen der vorangegangenen Stufe zu filtern. Ein Beispiel dafür ist die Darstellung des Forwarding. Das Forwarding findet beim ARM in der Execute Stufe statt, aber die Informationen darüber, was geforwardet wird, werden in der Decode Stufe decodiert.

5.2 Kommunikationsformat zwischen Prozessor und GUI

Wie oben schon beschrieben werden alle Daten eines Clock Zyklus am Ende des Clock Zyklus auf einmal übergeben. Um die Daten übergeben zu können, wird das richtige Format gebraucht. Es wird besonders auf dieses Format eingegangen, weil es ein sehr wichtiger Bestandteil der Kommunikation zwischen Prozessor und GUI darstellt. Alles, was im Prozessor passiert, wird der GUI über dieses Format mitgeteilt. Was dazu führt, dass eine sehr lose Kopplung möglich ist.

Das Format muss folgende Daten enthalten:

- Die Instruktion zu der die Daten gehören
- Operationen: Es muss die Möglichkeit bestehen anzugeben, was für Operationen der Prozessor ausgeführt hat. Also z.B. dass der Prozessor ein Register gelesen hat.
- Zusätzlich Daten: Beim Lesen eines Registers ist das z.B. die Nummer des Registers und der Wert, der gelesen wurde.

Es gibt zwei Möglichkeiten um Daten einer Instruktion zuzuordnen. Die Erste ist es, den Daten einen Identifier zuzuordnen, zum Beispiel die ID der Instruktion. Der Nachteil hierbei ist, dass wenn man Daten einer bestimmten Instruktion haben will, muss man die Daten filtern. Der Vorteil ist aber, dass man die Daten zusammen abspeichern kann.

Die zweite Möglichkeit ist es, die Daten der verschiedenen Instruktionen gesondert abzuspeichern. Also alle Daten einer Instruktion jeweils in einem Objekt gekapselt. Das hat den Vorteil, dass ganz klar abgegrenzt ist, welche Daten zu welcher Instruktion gehören und man auf Daten einer Instruktion separat zugreifen kann.

Die zweite Variante wurde ausgewählt, weil sie den Umgang mit den Daten leichter macht. Zur Speicherung der Daten gibt es eine Wrapper Klasse namens ArmInstrucktion. Um die Operationen darzustellen, wurde für jede Operationen ein Identifier festgelegt. Der Identifier für das Lesen eines Registers ist zum Beispiel ein Enum mit dem Namen READ_REGISTER. Dieses Konzept ist sehr einfach, aber es erfüllt seinen Zweck. Die zusätzlichen Daten müssen jeweils einer Operation zugeordnet werden. Also braucht man einen Datentyp, in dem man zwei Objekte einander zuordnen kann.

Der erste Ansatz war ein zweidimensionales Array. Dieser Ansatz wurde aber sehr schnell verworfen, weil der Identifier für die Operation grundsätzlich von einem anderen Typen ist als die zugehörigen Daten. Somit hätte man nur ein Objekt Array nehmen können, was bedeutet hätte, dass man sehr oft casten muss.

Der zweite Ansatz war eine Hashmap. Bei der Hashmap werden Objekte immer als Paar eingefügt. Das erste Objekt repräsentiert einen Schlüssel, mit dem man auf das zweite Objekt zugreifen kann. Das zweite Objekt ist ein beliebiges Objekt. Also können bei der Hashmap zwei Objekte einander zugeordnet werden. Mit Hilfe von Generics können die beiden Objekte auch von verschiedenen Typen sein. Im Prinzip erfüllt die Hashmap also die Anforderungen. Aber die Hashmap hat einen sehr großen Nachteil, es kann nicht direkt auf einzelne Objekte (bzw. Objektpaare) zugreifen werden. Der Zugriff geht immer nur über den Schlüssel. Was dazu führt das man nicht wie z.B. in einem Array, die Hashmap in einer Schleife durchlaufen kann. Da die Hashmap somit auch nicht in Betracht gekommen ist, wurde ein eigener Datentyp erstellt.

Der Datentyp hat den Namen TowSet. Er kann zwei Objekte beliebigen Typs aufnehmen. Durch Generics ist die Typsicherheit gewährleistet. Im Programm wird das TowSet so genutzt, dass der erste Wert die Operation ist und der zweite ein Objekt Array, das die verschiedenen Daten enthält. Das ist zwar nicht die schönste Variante, aber für die Daten jeder Operation noch mal einen Datentyp zu definieren ist einfach nicht praktikabel. Ein Objekt der Klasse ArmInstrucktion enthält dann für jede Pipeline Stufe eine Liste von TowSets. Zusätzlich ist noch die Originalinstruktion und eine ID die die Instruktion eindeutig identifiziert enthalten.

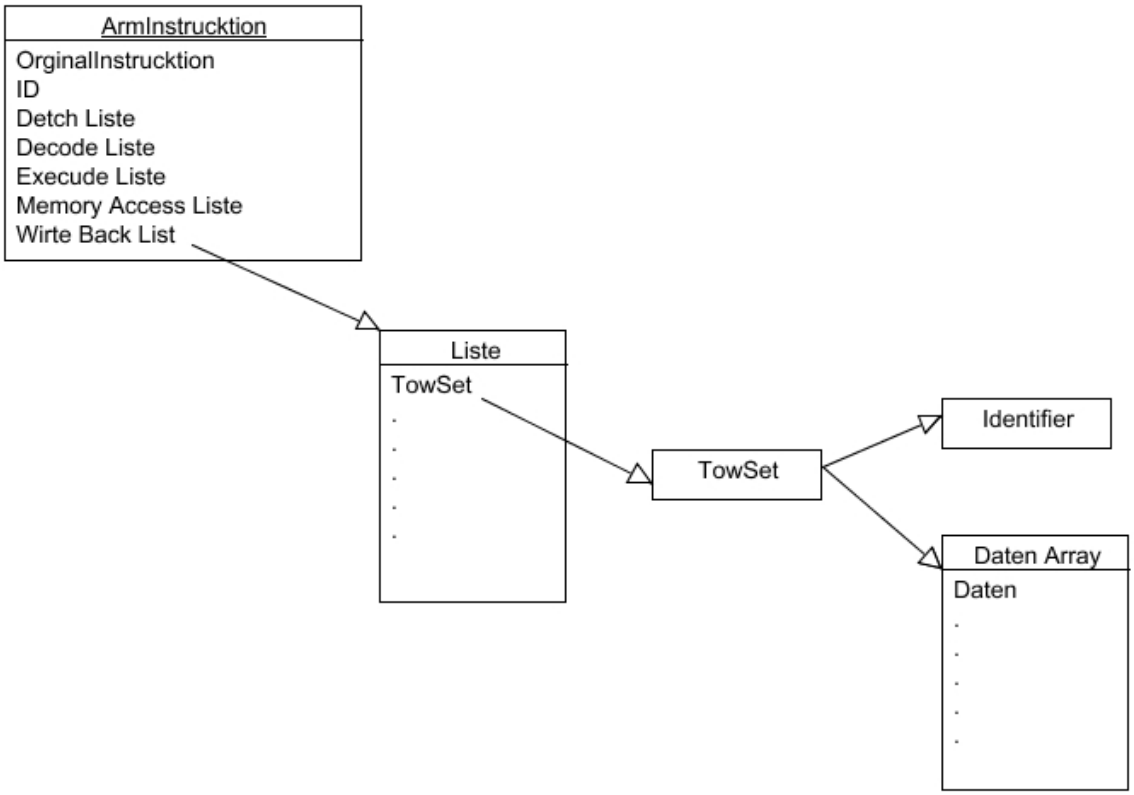


Abb. 34. Aufbau des Daten Formats

5.3 Die Verbindungsklasse

In der Anforderung bezüglich der Erweiterbarkeit wurde festgelegt, dass die GUI so gestaltet werden soll, dass sie ohne Veränderungen auch von anderen Prozessoren ähnlichen Typs genutzt werden kann. Um das zu gewährleisten, müssen alle Teile der GUI, die Prozessor spezifisch sind, ausgelagert werden und alle anderen Daten so aufgearbeitet werden, dass sie einem allgemeinen Format entsprechen.

Prozessorspezifisch ist vor allem die Darstellung des Aufbaus des Prozessors, alle anderen Sachen, wie der Arbeitsspeicher und die Register, sind in allen Prozessoren, die für die Darstellung in Frage kommen, vorhanden und können somit der GUI Klasse zugeordnet werden.

Dann gibt es noch die Fenster, die nur teilweise prozessorspezifisch sind wie z.B. die Pipeline Stufen (die Anzeige der Vorgänge innerhalb der Pipeline Stufen) und die Darstellung der einzelnen Instruktionen. Der Inhalt an sich ist prozessorspezifisch, aber die Fenster nutzt jeder Prozessor. Aus dem Grund wurde festgelegt, dass für diese Fenster die Daten in einem allgemeinen Format übergeben werden, das ohne Veränderung dargestellt werden kann. Die Klasse, die für die ausgelagerten Teile verantwortlich ist, heißt GrafikUnit.

5.3.1 GrafikUnit

Die Klasse GrafikUnit ist für alle Teile der GUI verantwortlich, die ARM-spezifisch sind. Außerdem ist sie dafür verantwortlich, dass die Daten so aufbereitet werden, dass sie in der GUI angezeigt werden können und letztendlich ist sie das Verbindungsstück zwischen GUI und „virtuellem“ Prozessor.

Abb. 35. Diagramm das die Verbindung zwischen GUI und Prozessor zeigt

Im Einzelnen heißt das, dass innerhalb der GrafikUnit die grafische Darstellung des Prozessors erstellt und gesteuert wird. Damit ist gemeint, dass die einzelnen Busse von der GrafikUnit aktiv und inaktiv geschaltet werden. Als nächstes verarbeitet die GrafikUnit die Instruktionsdaten des Prozessors und übergibt sie der GUI, die sie dann darstellt. Als letztes ist die GrafikUnit noch dafür verantwortlich, das Clock Signal von der GUI an den „virtuellen“ Prozessor weiterzuleiten.

Im Folgenden Bild sieht man den Ablauf zwischen dem Drücken des Clock Botton und der visuellen Darstellung.

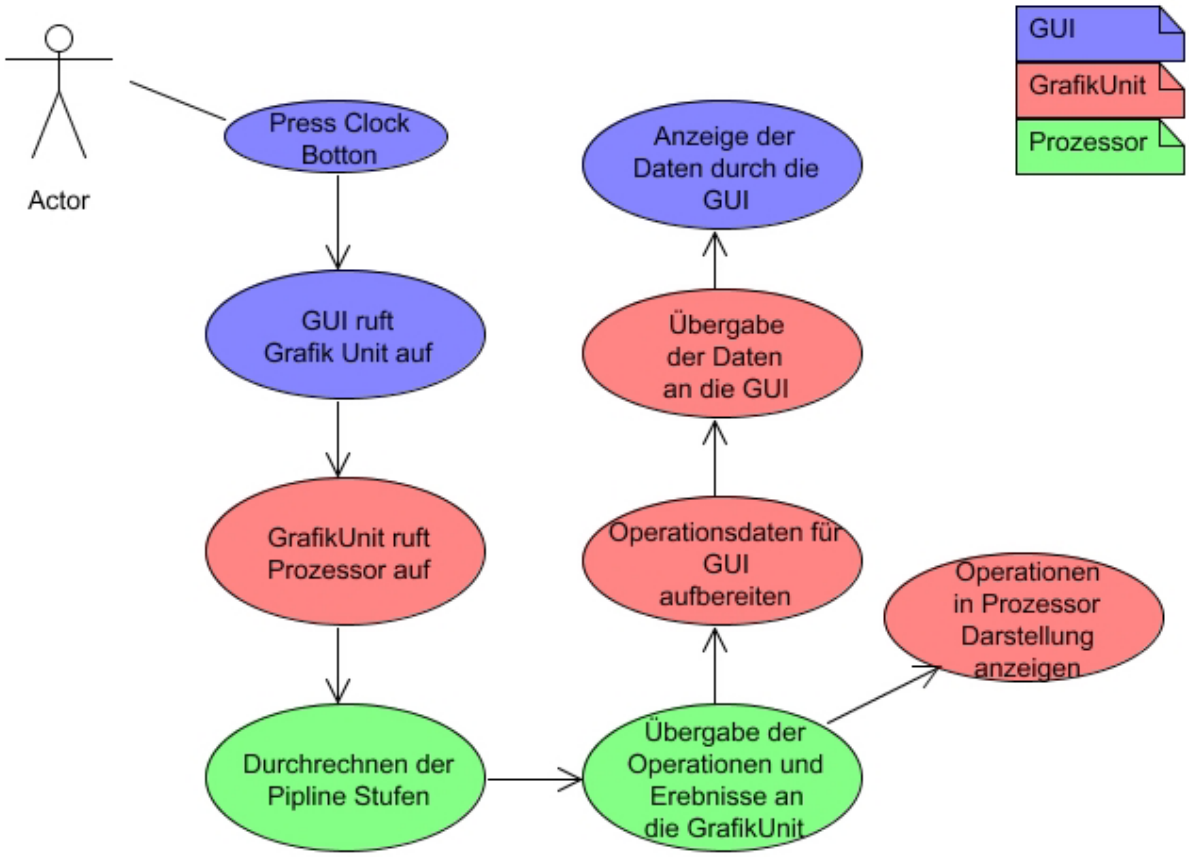


Abb. 36. Ablaufplan der den Verlauf vom drücken des Clock Botton bis zur Anzeige in der GUI zeigt

5.4 Gesamtübersicht

In diesem Abschnitt wird kurz der gesamte Aufbau des ganzen Systems vorgestellt. Dadurch sieht man, wie die einzelnen Klassen zusammen arbeiten bzw. welche Klassen untereinander kommunizieren.

In dem UML Klassen Diagramm sieht man die ArmProcessor Klasse, die direkt mit der RAM, ALU Decoder und Register Klasse kommuniziert außerdem kommuniziert sie noch mit Hilfe der ArmInstrucktion mit der GrafikUnit (das Diagramm ist aufgrund der Übersichtlichkeit leicht vereinfacht worden).

Die GrafikUnit greift direkt auf die RAM Klasse zu um sie mit den Daten, die sie mit Programm Daten, die sie über die FileImporter Klasse eingelesen hat, zu initialisieren. Werden durch den Prozessor die RAM oder Register Werte verändert, informiert die RAM bzw. die Register Klasse die GrafikUnit über die Veränderungen.

Die GrafikUnit erstellt bei ihrer Initialisierung mit Hilfe der verschiedenen Grafik bzw. GrafikList Klassen die visuelle Darstellung des Prozessors und stellt die Busse aktiv oder inaktiv abhängig von den Daten, die die GrafikUnit vom ArmProcessor bekommt. Außerdem updatet die GrafikUnit die GUI mit den Daten vom ArmProcessor, RAM und Register.

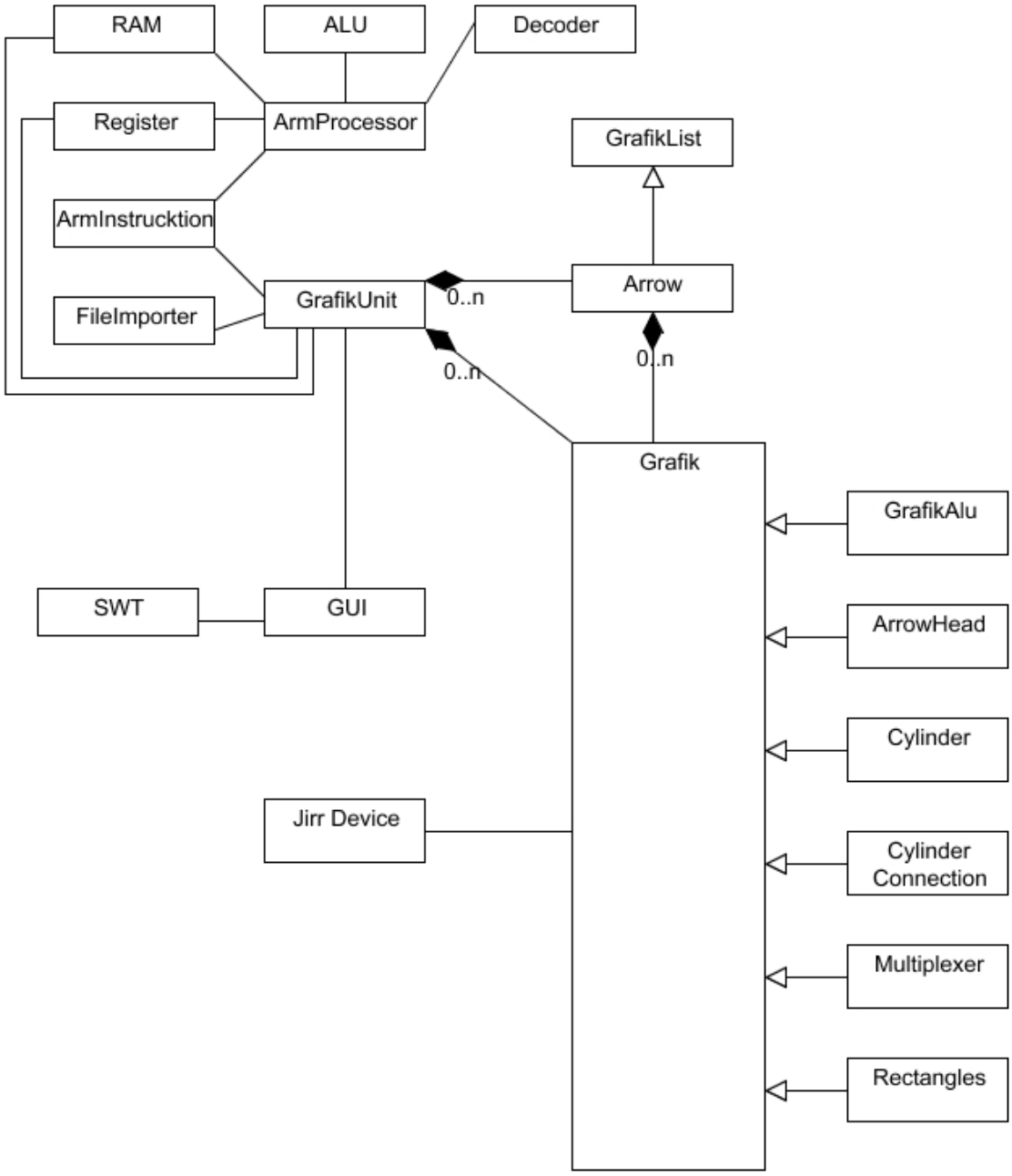


Abb. 37 Klassen Diagramm

.

6 Abschließende Bewertung

In diesem Abschnitt geht es um die nachträgliche Bewertung der wichtigsten Entscheidungen. Es wird bewertet, ob die Entscheidungen, die getroffen wurden sinnvoll waren und was man hätte besser machen können.

6.1 Jirr

Die Entscheidung Jirr zu nutzen hat sich bewährt. Jirr hat alle Anforderungen sehr gut erfüllt. Insgesamt gab es nur zwei etwas größere Probleme. Das erste Problem gab es mit den 3D Modellen. Wenn das Format 3ds genutzt wurde, wurden diese teilweise nicht komplett richtig angezeigt. Ob Jirr hier die Modelle falsch dargestellt hat oder ob der Exporter, der die Modelle in die verschiedenen Formate exportiert hat, nicht richtig funktioniert hat, ist nicht ganz klar. Das Problem wurde gelöst indem die Modelle in das Format ms3d exportiert worden. Das zweite Problem gab es bei den Texturen. Die Texturen sollten als erstes automatisch erstellt werden. Jirr bietet in dem Bereich allerdings keinerlei Unterstützung. Aus dem Grund wurden die Texturen von Hand erstellt.

6.2 Prozessor

Die Implementierung des Prozessors hat sich schwieriger herausgestellt als erwartet. Es musste ein sehr hoher Verwaltungsaufwand getrieben werden um die Daten zwischen den Pipeline Stufen auszutauschen. Im Nachhinein wäre eine andere Lösung vermutlich die besser gewesen. Das Konzept des Originalprozessors nachzubilden ist zwar eine solide Lösung, aber wahrscheinlich nicht die beste. Man muss sehr viele Probleme lösen, die bei einer sequenziellen Abarbeitung eigentlich gar nicht vorkommen dürften.

Man kann davon ausgehen, dass das Konzept, bei dem die Instruktionen nacheinander komplett abgearbeitet werden (siehe Abschnitt 3), die bessere Lösung gewesen wäre. Das Problem, das die Vorgänge des „virtuellen“ Prozessors bei diesem Konzept nicht direkt auf die Darstellung übertragen werden können, ist bei der verwendeten Kommunikationsmethode (siehe Abschnitt 5), zwischen Prozessor und GUI, so gut wie nicht mehr vorhanden. Die Vorteile sind hingegen ganz klar, man braucht so gut wie keinerlei Verwaltungsaufwand mehr betreiben um die Daten zwischen den Pipeline Stufen auszutauschen, was die Implementierung stark vereinfacht. Deshalb sollte bei einer Neu-Implementierung dieses Konzept dem verwendeten Konzept vorgezogen werden.

6.3 Grafik

Die Entscheidungen eine Hauptgrafikklasse zu erstellen, von der alle anderen Klassen erben, hat sich bewährt. Bei einer Neu-Implementierung wäre es sogar sinnvoll dieses Konzept noch zu erweitern, indem man noch mehr Logik in die Grafik Klasse verschiebt. Ein Beispiel

hierfür wären die ConnectionPoints. Man könnte die Erstellung der ConnectionPoints komplett von der Grafik Klasse übernehmen lassen und nur Konstanten für jede Unterklasse festlegen, die angeben, an welchen Stellen ConnectionPoints erstellt werden können. Dies würde dazu führen, dass die Codemenge erheblich sinkt und der gesamte Code wird übersichtlicher. Die ConnectionPoints haben sich auch als sehr gute Lösung herausgestellt und sie waren auf jeden Fall die beste Lösung, die im Hinblick auf die Zeit möglich war.

6.4 Kommunikations-Konzept

Das Komunikations-Konzept ist eine optimale Lösung gewesen. Es konnte einerseits sehr gut für die Darstellung des Prozessors genutzt werden, andererseits konnten auf diese Weise auch alle anderen Informationen sehr gut weitergeleitet werden (z.B. der aktuelle PC Count).

7 Fazit

Im Rahmen dieser Arbeit wurde ein Interaktives Lern Programm für Prozessoren entwickelt, mit dem verschiedene Prozessortypen dargestellt werden können. Als Beispiel Prozessor wurde ein ARM Prozessor implementiert. Das Programm ist auf Studienanfänger zugeschnitten und soll ihnen helfen, die Vorgänge in einem Prozessor zu verstehen. Es wurde eine Entwicklungsumgebung erstellt, die es dem Anwender erlaubt mit relativ wenig Aufwand verschiedene Prozessoren in das System zu integrieren. Hierbei wurde besonders darauf geachtet, dass das System aus verschiedenen Komponenten besteht, die unabhängig voneinander genutzt werden können.

Im Rückblick auf die Anforderungen ist es gelungen diese zu erfüllen. Wie gefordert wurde eine Umgebung geschaffen, die erweiterbar ist und die inhaltlich klar strukturiert ist. In Zukunft ist davon auszugehen, dass neben dem ARM Prozessor noch andere Prozessoren implementiert werden. Das Ziel soll es sein alle wichtigen Prozessortypen in das System zu integrieren.

8 Anhang

8.1 Die Entwicklungswerkzeuge

In diesem Abschnitt werden kurz die verwendeten Entwicklungswerkzeuge vorgestellt.

8.1.1 Programmiersprache

Die Programmiersprache Java wurde ausgewählt weil Java eine moderne objektorientierte Programmiersprache ist. Sie ist plattformunabhängig und ermöglicht einem damit, die Software ohne großen Aufwand für verschiedene Betriebssysteme anzubieten. Die Java Standard library ist sehr groß und ausgereift, so dass man kaum zusätzliche Bibliotheken benötigt.

Es gibt viele verschiedene und gute IDE für Java und die Community ist sehr groß, somit findet man Unterstützung in den meisten Bereichen. Zusätzlich hat der Autor in Java die meiste Programmiererfahrung.

8.1.2 Eclipse

Eclipse ist eine Open Source IDE, die seit 2001 von IBM entwickelt wird. Sie ist die meist genutzte IDE für Java Programmierung und bietet Plugins zu fast jedem Bereich. Eclipse bieten Methoden zur Code Generierung, die einem viele Aufgaben abnehmen.

8.1.3 SWT

Das Standard Widget Toolkit (SWT) ist eine Bibliothek für die Erstellung grafischer Oberflächen mit Java. SWT wurde im Jahr 2001 von IBM für die Entwicklungsumgebung Eclipse entwickelt und wird kontinuierlich gepflegt. SWT nutzt dabei im Gegensatz zu Swing die nativen grafischen Elemente des Betriebssystems – wie das AWT von Sun – und ermöglicht somit die Erstellung von Programmen, die eine Optik vergleichbar mit „nativen“ Programmen aufweisen. SWT wurde ausgewählt, weil der Autor mit SWT die meiste Erfahrung hat.

8.1.4 SWT Builder

SWT Builder ist ein Plugin für Eclipse, welches zur Gestaltung von SWT Fenstern genutzt wird. Alle wichtigen Elemente von SWT können durch einfaches Drag and Drop eingefügt werden. Es gibt eine Unterstützung, die einem hilft Elemente aneinander auszurichten. Die Elemente werden dann in einem Baum dargestellt und können von dort aus weiter verarbeitetet werden (Variablen verändern usw.). Die Code Generierung funktioniert sehr

gut, sie ist übersichtlich und es gibt auch keine Probleme, wenn man von Hand den Code verändert.

8.1.5 UMLet

UMLet ist ein Programm zu Erstellung von UML Diagrammen.

8.1.6 3D Studio MAX

3D Studio MAX (auch bekannt als 3ds Max) ist ein 3D-Computergrafik- und Animationsprogramm, das im Bereich Computerspiele, Comic, Animationen, Film (TV / Kino) als auch in gestalterischen Berufen wie Design oder Architektur seinen Einsatz findet.
Mit 3ds Max wurden 3D Modelle erstellt und in verschiedene Formate exportiert, so dass sie mit Hilfe von Jirr einlesen konnten.

8.2 Glossar

3ds - 3ds ist ein 3D Grafik Modell Format, das früher von 3D Studio Max genutzt wurde.

ALU - arithmetisch-logische Einheit (englisch arithmetic logic unit, daher oft abgekürzt ALU) ist ein elektronisches Rechenwerk, welches in Prozessoren zum Einsatz kommt.

ARM - ARM Steht für Advanced RISC Machines

ARM-Architektur - ARM-Architektur ist ein Kern-Design für eine Familie von 32-Bit-Mikroprozessoren, die dem RISC-Konzept folgen.

Barrel Shifter - Ein Barrel-shifter ist ein Bauteil aus der Digitaltechnik. Es handelt sich dabei um eine Schiebelogikschaltung. Er besteht aus 2-auf-1-Multiplexern und Dekodern.

Bus - Ein Bus ist ein Leitungssystem mit zugehörigen Steuerungskomponenten, das zum Austausch von Daten und zwischen Hardware-Komponenten dient.

Clock Zyklus - Ein Clock Zyklus ist der Zeitraum zwischen zwei Clock Signalen.

CPSR - Current Program Status Register. Das Register enthält den momentanen Ausführungsmodus und andere Status Informationen.

Dialogfeld - Als Dialogfeld bezeichnet man einen Teil einer GUI, auf welchem Informationen dargestellt werden oder in dem es möglich ist Informationen einzugeben.

Disassembler - Ein Disassembler ist ein Computerprogramm, das die binär kodierte Maschinensprache eines ausführbaren Programmes in eine für Menschen lesbarere Assemblersprache umwandelt.

Enum - Die Abkürzung enum steht für Enumeration und bezeichnet Aufzählungstypen in der Programmierung.

Grafik-Engine - Eine Grafik-Engine ist ein mitunter eigenständiger Teil eines Computerprogramms, welcher für dessen Darstellung von Computergrafik zuständig ist.

Grafik Modell - Ein Grafikmodell ist ein Mittels 3D Editor erstelltes Objekt. Es kann beliebige Formen haben, z.B. ein Rechteck.

GUI - Eine grafische Benutzeroberfläche (engl. Graphical User Interface, Abk. GUI) ist eine Softwarekomponente, die einem Computerbenutzer die Interaktion mit der Maschine über grafische Elemente erlaubt.

HAW - Hochschule für Angewandte Wissenschaften.

Instruktion - Eine Instruktion ist ein Maschinenbefehl, der von einem Mikroprozessor ausgeführt werden kann.

ISA - Eine Befehlssatzarchitektur (engl. Instruction Set Architecture, kurz: ISA) ist vereinfacht gesagt die formale Spezifikation bestimmter Verhaltensweisen eines Prozessors aus Sicht seines Programmierers.

Mikroprozessor - Ein Mikroprozessor (griech. mikros für „klein“) ist ein Prozessor in sehr kleinem Maßstab, bei dem alle Bausteine des Prozessors auf einem Mikrochip vereinigt sind.

ms3d - ms3d ist ein 3D Grafik Modell Format, das von dem 3D Editor Milkshape genutzt wird.

Multiplexer - Ein Multiplexer (kurz: MUX) ist ein Selektionsschaltnetz in der analogen Elektronik- und Digitaltechnik, mit dem aus einer Anzahl von Eingangssignalen eines ausgewählt werden kann.

SPSR - State Program Status Register. Dieses Register wird genutzt um den Status vom CPSR zu speichern wenn der Prozessor in den privilegierten Modus wechselt (siehe dazu [Fur00] Seite 107-108).

RISC - Reduced Instruction Set Computing , zu deutsch Rechnen mit reduziertem Befehlssatz, ist eine bestimmte Designphilosophie für Prozessoren.

Textur - Bei der 3D-Modell bezeichnet Textur das Bild, welches auf der Oberfläche eines virtuellen Körpers dargestellt wird.

Thumb - Thumb ist eine Erweiterung des Standard-ARM-Cores, welcher es erlauben soll, den 32-Bit-Code von häufig benutzten ARM-Befehlen in 16-Bit-Code auszudrücken.

"virtueller" Prozessor - Die Java Implementierung einer ISA

XML- XML steht für Extensible Markup Language (engl. für „erweiterbare Auszeichnungssprache“), XML ist eine Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten in Form von Textdateien.

8.3 Literaturverzeichnis

[Fli04] Mikroprozessortechnik und Rechnerstrukturen. Auflage: 7
Thomas Flik , Hans Liebig, M. Menge.
Springer, Berlin (September 2004)

[Fur00] ARM system-on-chip architecture seconde edition.
Steve Furber.
Addison-Wesley 2000.

[Irr07] Irrlicht Engine. Stand 29.07.2007
http://irrlicht.sourceforge.net/.
SourceForge, Inc. (SourceForge.com)

[Jag00] ARM Architecture Reference Manual. 2nd Edition
David Jagger.
Addison-Wesley, 2000.

[Jirr07] Jirr - fast 3D for Java. Stand 22.07.2007
http://jirr.sourceforge.net/
SourceForge, Inc. (SourceForge.com)

[Jme07] jMonkeyEngine. Stand 25.07.2007
http://www.jmonkeyengine.com/

[Kau04] ARM System Developer's Guide. Designing and Optimizing System Software.
Andrew N. Sloss, Dominic Symes, Chris Wright.
Morgan Kaufmann, 2004

[Kla05] Grundkurs Computergrafik mit Java. Die Grundlagen verstehen und einfach umsetzen mit Java 3D. 1 Auflage
Frank Klawonn.
Vieweg; Auflage (Oktober 2005)

[Lgp07] GNU Lesser General Public License. Stand 27.07.2007
http://de.wikipedia.org/wiki/LGPL
Wikimedia Foundation Inc.

[Lwj07] Lightweight Java Game Library (LWJGL). Stand 30.04.2007
http://lwjgl.org/.

[Nor04] SWT: The Standard Widget Toolkit Volume 1.
Steve Northover, Mike Wilson.
Addison-Wesley 2004.

[Ogr07a] Java Native Interfaces for OGRE. Stand 1.08.2007
http://ogre4j.sourceforge.net/.
SourceForge, Inc. (SourceForge.com)

[Ogr07b] Object-Oriented Graphics Rendering Engine (OGRE). Stand 8.07.2007
http://www.ogre3d.org/.

[Orl04] Computergrafik und OpenGL. Eine systematische Einführung. Auflage 1
Dieter Orlamünder, Wilfried Mascolus.
Hanser Fachbuchverlag (September 2004)

[Ric07] Reduced Instruction Set Computing (RISC). Stand 4.04.2007
http://de.wikipedia.org/wiki/RISC.
Wikimedia Foundation Inc.

[Wen06] discreet. 3ds max 8. Auflage: 1
Volker Wendt.
Vmi Buch (Mai 2006)

Anhang CD

Auf der Beigelegten CD befindet sich der Programmcode. Das ausführbare Programm sowie eine Benutzer- und Installationsanleitung.

- Verzeichnisstruktur
  - Anleitung: Hier befindet sich die Benutzer- und Installationsanleitung.
  - Arbeit: In dem Ordner befindet sich die Arbeit im PDF Format
  - Sourcecode: Hier befindet sich der Programmcode
  - Software: An dieser Stelle befindet sich die komplette Software

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 24. August 2007

_____ _____

Ort, Datum Unterschrift