Elektro-/Informationstechnisches Seminar/Projekt

Automated Test System Adaptation for Power Emulation

Daniel Reitz

Institut für Technische Informatik
Technische Universität Graz
Vorstand: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhold Weiß

Begutachter: Ass.-Prof. Dipl.-Ing. Dr.techn. Christian Steger
Betreuer: Dipl.-Ing. Christian Bachmann

Graz, im November 2009

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz,am ....

(Unterschrift)

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Die Bestimmung der Leistungsaufnahme von integrierten Schaltkreisen spielt während der Entwicklungsphase eine bedeutende Rolle. Einerseits will man möglichst früh erkennen, wo Potential für Leistungseinsparungen vorhanden ist und zum anderen sollen leistungskritische Vorgänge, die sich negativ auf die Systemstabilität auswirken können, ebenso rechtzeitig erkannt werden um entsprechend gegenzusteuern.

Im Rahmen von Schätzverfahren zur Bestimmung der Leistungsaufnahme (engl: Power Estimation), werden auf unterschiedlichen Ebenen Simulationen durchgeführt, um entsprechende Leistungsprofile bzw. Modelle der Komponenten zu erstellen, die eine Abschätzung der Leistung ermöglichen. Je nach Simulationsebene (z.B. Transistor, Gatter, RTL, usw.) erhält man mehr oder weniger genaue Ergebnisse. Das Problem an diesem Konzept ist die erhöhte Simulationszeit auf unteren Ebenen was sich natürlich unmittelbar negativ auf Kosten der Entwicklungszeit des Produkts auswirkt.

Um einen schnelleren Ablauf dieses Designschrittes der Leistungsabschätzung zu ermöglichen, setzt man auf das neuere Konzept der Power Emulation. Dabei werden für die Leistung relevante Hardwarekomponenten um Zusatzhardware erweitert, deren Aufgabe darin besteht, Signalwechsel zu erfassen und anhand integrierter Power-Profile die Leistungsaufnahme zur Laufzeit zu berechnen. Mit entsprechender Software lassen sich zeitliche Verläufe der Leistungsaufnahme aufzeichnen und können in weiterer Folge in einem Leistungsdiagramm graphisch dargestellt werden. [1]

Auf System-Ebene lassen sich mithilfe von Power-Sensoren Informationen von Signalen erfassen und per Software kann aus diesen Daten die Leistungsaufnahme der Schaltung bestimmt werden. Durch Maskierung lassen sich auch Signale (bzw. zugehörende Module oder Sub-Module) bei der Bestimmung der Leistung ausblenden. [1]

Da digitale Schaltkreise zumeist mit VHDL beschrieben werden, und diese Hardwarebeschreibungssprache hierarchisch aufgebaut ist, ergibt sich das Problem, Signale aus tieferen Ebenen des Designs am Top-Level verfügbar zu machen. Um die Signalinformationen

Power-Sensoren tatsächlich zugänglich zu machen, muss ein Weg gefunden werden, die Signale auf den Top-Level zu routen. Dies sollte automatisiert erfolgen, um den Designer zusätzliche Arbeit zu ersparen, was insbesondere bei größeren Designs eine erhebliche Zeit- und Kostenersparnis bedeutet.

1.2 Gliederung der Arbeit

Die vorliegende Arbeit gliedert sich folgendermaßen:

In Kapitel 2 wird die Thematik der Leistungsabschätzung (Power Estimation und Power Emulation) über eine Literaturrecherche behandelt werden.

In weiterer Folge werden die Möglichkeiten der automatisierten VHDL-Analyse erläutert. Konkret wird in Kapitel 2.4 das Prinzip des Parsings und die in Frage kommenden Parsergeneratoren erwähnt.

Die Problemstellung ist in Kapitel 3 beschrieben. Aus den Ergebnissen der Recherche werden die unterschiedlichen Ansätze miteinander verglichen und etwaige Lösungen oder Optimierungen dargestellt.

In Kapitel 4 wird die Umsetzung der Aufgabenstellung dargelegt, eine Demonstration der Ergebnisse anhand eines Designs des Mikrocontrollers 8051 mit dem Hinblick auf etwaige Verbesserungsmöglichkeiten schließt die Arbeit ab.

Kapitel 2

Literaturrecherche

2.1 Leistungsaufnahme von integrierten Schaltkreisen

Nachfolgend sollen kurz die wesentlichen Ursachen für die Leistungsaufnahme in integrierten Schaltkreisen erwähnt werden. [2]

Schaltvorgänge an Kapazitäten

Unter vereinfachten Bedingungen lässt sich für einen Kondensator folgende Gleichung aufschreiben:

$$P_{sweap} = \frac{1}{2} \cdot C_{load} \cdot \alpha \cdot V_{dd}^2 \cdot f$$

C_{load} gibt die Gesamtkapazität der Anordnung an, V_{dd} ist die Versorgungsspannung und f die Schaltfrequenz. Die Variable α gibt die Schaltwahrscheinlichkeit während eines Taktzyklus an.

Kurzschlussströme

Diese resultieren aus kurzzeitigen, leitenden Verbindungen zwischen pull-up und pull-down Zweig während des Umschaltvorganges in CMOS-Schaltkreisen. Es kann folgend beschrieben werden:

$$P_{shortcircuit} = \frac{\beta}{12}(V_{dd} - 2V_{th})^3 \cdot \frac{\tau}{T}$$

Diese Gleichung stellt nur eine grobe Abschätzung dar. Für präzise Bestimmungen müssen Transistormodelle und Transientenanalysen herangezogen werden.

Leckströme

Leckströme stellen eines der größten Probleme im Chipentwurf der heutigen Zeit dar. Auf Grund der nicht-idealen Schaltereigenschaften eines MOS Transistor kommt es zu verschiedensten Stromflüssen (Hauptquellen: Sub-Threshold Leakage, Gate Leakage, Gate Induced Drain Leakage, Reverse Bias Junction Leakage) die gemeinsam betrachtet zu einem ungewünschten Stromfluss, dem Leckstrom, führen. [3]

2.2 Power Profiling

Man unterscheidet beim Power-Profiling prinzipiell 2 Ansätze:

- auf Messung basierend
- auf Abschätzung basierend = Power Estimation

Physikalische Messungen liefern sehr genaue Ergebnisse, haben jedoch den Nachteil, dass zusätzliche Messeinrichtungen benötigt werden.
Abschätzungsbasierte Verfahren bieten ein hohes Maß an Flexibilität, sind jedoch in der Genauigkeit eingeschränkt. Man kann hier wiederum zwischen zwei Gruppen unterscheiden, das wären zum einen simulationsbasierende und zum anderen von Hardware unterstützte Ansätze. Beim Hardwareansatz erhält man die Leistungsinformationen aus sogenannten Powermodellen, die als Hardware implementiert sind. [1]

2.2.1 Ebenen der Leistungsabschätzung

Die Leistungsabschätzung kann auf mehreren Ebenen durchgeführt werden, die niedrigste ist dabei die Transistorebene. Hierbei ist der Berechnungsaufwand und damit die notwendige Simulationszeit am größten (da jeder einzelne Transistor des Designs in die Berechnung eingeht), dafür liegt das Ergebnis ziemlich nahe am realen Wert. Je höher die Abstraktionsebene (Transistor - Gate - RTL - Architecture) ist, umso schneller können Simulationen durchgeführt werden, da die Modelle in höheren Ebenen durch die Abstraktion vereinfacht werden. Gleichzeitig ergibt sich im Gegensatz dazu das Problem, dass durch zunehmende Abstraktion die Leistungsbestimmung immer mehr von der Realität abweichen kann. Von allen Ebenen bietet die Applikationsebene die meisten Vorteile im Bezug auf Simulationsaufwand und Zeit.
Power Estimation wird prinzipiell auf Systemebene angewendet, wenn genauere Ergebnisse gefordert sind, werden einzelne Komponenten parallel auf niedrigeren Ebenen co-simuliert. Letztendlich versucht man, einen Kompromiss zwischen Aufwand und Genauigkeit zu treffen. [1]

Power Model

Auf höheren Ebenen werden Methoden der linearen Regression angewendet.

$$y = \sum_{i=0}^{n-1} c_i \cdot x_i + \epsilon$$

x ist dabei ein Vektor aus Modelparametern, x_i zeigt Systemzustände an (wie zum Beispiel: idle, run); c stellt die zugehörigen Modelkoeffizienten dar, c_i beinhaltet Leistungsinformationen und muss durch einen vorhergehenden Modellierungsprozess bestimmt werden; ϵ (Schätzfehler) ist der Fehler, der sich zwischen realer und abgeschätzter Leistung ergibt [1]

Power - Modellierungsprozess

Hierzu sind 3 Schritte notwendig: [1]

1. Auswahl der Modellparameter
   Hier geht es um eine sinnvolle Auswahl der richtigen Parameter, wovon letztlich die Genauigkeit der gesamten Leistungsabschätzung abhängt

2. Auswahl von Übungsdaten
   Dabei werden einige Messungen durchgeführt woraus man einen Vektor y (Leistungswerte) und eine Matrix X erhält (X beinhaltet die Modellparameter [pro Messung - ein Vektor])
   Nach dem linearen Regressionsmodell ist folgender Zusammenhang gewünscht:

$$y = Xc$$

3. Methode der kleinsten Quadrate
   um die Koeffizienten c_i zu bestimmen (anhand von y und X)

2.3 Power Emulation

Power Emulation arbeitet mit dem Ansatz, Design-Komponenten und deren Module um Zusatzhardware (auf RTL[1]-Ebene) zu erweitern, mit deren Hilfe es möglich ist, die Aktivität in Hinblick auf Signalwechsel am Modul zu registrieren. Dabei trägt jedes Modul unterschiedlich stark zum Leistungsumsatz bei, was durch entsprechende Variablen, die in einer Look-Up-Table gespeichert sind, definiert wird. Der Nachteil auf RTL-Ebene ist hierbei der gewaltige Overhead, der durch diese Hardwareerweiterung entsteht (bis zu Faktor 15). Daher kommen unterschiedliche Optimierungsansätze zum Einsatz, die allesamt das Ziel verfolgen, den Hardwareaufwand soweit zu reduzieren, dass der erhaltene Fehler aus der Leistungsabschätzung durch Power-Emulation möglichst klein bleibt. Realistische Ergebnisse liefern für Fehler kleiner als 10% einen Overhead vom Faktor 3 zum ursprünglichen Design. [4]

Auf System-Ebene spielen ressourcenintensive Prozesse eine Hauptrolle bei der Leistungsbilanz, und es gilt, ebendiese aufzuspüren. Durch Optimierungen hinsichtlich der Verwendung von an die Applikation angepasster Hardware, lassen sich sowohl Kosten sparen als auch Verbesserungen Richtung niedrigerer Leistungsaufnahme realisieren. [2]

Ein auf Systemebene implementiertes Konzept der Power Emulation eines 8051-Designs zeigt, dass bei einem FPGA-Overhead von gerade einmal 1,5% der verfügbaren Ressourcen sich der mittlere Schätzfehler mit 10% in Grenzen hält. [4] & [5]

2.3.1 Power Sensoren

Sie dienen zur Aufnahme von Zustandsinformationen aus den verschiedensten Systemmodulen. Der Zustandsvektor und die Leistungskoeffizienten sind in einer Tabelle abgespeichert. Die Werte der Koeffizienten können während der Programmlaufzeit verändert werden, was unter anderem eine Maskierung erlaubt um z.B. nur die Leistungsaufnahme von bestimmten Sub-Modulen zu bestimmen. [1]

[1] Register Transfer Level

2.4 Automatisierte VHDL-Analyse

2.4.1 Parsergenerator

Ein Parsergenerator ist ein Computerprogramm, das mittels bestimmter Spezifikationen einen Parser erzeugt. Ein Parser wiederum ist ein Unterprogramm, das ein zu weiterverabeitendes Format erzeugt. Meist dient der Parser dazu, die Semantik der Eingabe zu verstehen.
Bei der Analyse von Quellcode kommt zunächst ein Lexer zum Einsatz, der Zeichen für Zeichen einliest und die Daten in sogenannte Token (lexikalische Grundeinheiten; „Wörter“) zerlegt. Token dienen zur Weiterverabeitung im Parser. [6] & [7]

Als Parsergeneratoren bieten sich 2 Tools an, die im Folgenden kurz erwähnt werden:

ANTLR

ANTLR (ANother Tool for Language Recognition) ist ein objektorientierter Parsergenerator. Der Übersetzer ist in JAVA geschrieben und als freie Software verfügbar. Er unterstützt mehrere Zielsprachen wie C, Java oder Python.
Allgemein stellt ANTLR dem Benutzer eine Vielzahl an Funktionen bereit sodass es auch möglich ist, komplexere Parser für Programmiersprachen wie z.B. C# zu entwickeln, er ist daher verglichen mit JavaCC mächtiger. Ein weiterer Vorteil in Hinblick auf die Nutzung mit vMAGIC (siehe Kapitel 2.4.2) besteht in der Korrektur von etwaigen einfachen Fehlern im Syntax. [8]

JavaCC

JavaCC (Java Compiler Compiler) ist ein Parsergenerator zur Erstellung von Parsern für eine Vielzahl von Programmiersprachen wie Java, JavaScript, C, C++, HTML usw. und wurde ursprünglich von Sun Microsystems entwickelt. Die Konvertierung in eine Baumstruktur wie AST ist im Gegensatz zu ANTLR nur mittels Erweiterungen wie zum Beispiel dem Präprozessor JJTree möglich. Die Dokumentation zu JavaCC fällt eher dürftig aus. [9]

2.4.2 vMAGIC-Bibliothek

vMAGIC (**V**HDL **Ma**nipulation and **G**eneration **I**nterfa**c**e) wurde von Mitarbeitern des Heinz-Nixdorf-Instituts der Universität Paderborn entwickelt. Wie aus der Abkürzung schon ersichtlich, ermöglicht vMAGIC dem Entwickler einerseits das Auslesen und Modifizieren von bestehendem VHDL[2]-Code, andererseits auch die Generierung von neuem Code.

Die wesentlichsten Bestandteile dieser Bibliothek sind der Parser, der den VHDL-Code in ein AST[3]-Format konvertiert und dabei Redundanzen entfernt. Diese AST-Struktur

[2] Very High Integrated Circuit Hardware Description Language
[3] Abstract Syntax Tree

bildet eine Baumstruktur nach, die für den Anwender nicht gut lesbar ist. Für den benutzerfreundlichen Umgang wurden daher sogenannte Meta-Klassen definiert, welche über Member Functions aufgerufen werden können. Meta-Klassen selbst lassen sich wiederum grob in 2 Kategorien unterteilen: Man unterscheidet zum einen eine Low-Level- und zum anderen eine High-Level-Klassen. Während Low-Level-Klassen grundlegende VHDL-Konstrukte wie Prozesse beinhalten, werden in den High-Level-Klassen mehrere Basiskonstrukte zusammengeführt um komplexere Funktionen (z.B. Zustandsautomaten) zu verwirklichen.
Zum Schreiben von VHDL-Code greift vMAGIC auf das StringTemplate-System von ANTLR zu, um aus der AST-Struktur leserlichen Code zu generieren. [10]

Die Hilfe zu den einzelnen Klassen und Methoden der vMAGIC-Bibliothek kann der API-Dokumentation entnommen werden. [11]
Die vMAGIC-Bibliothek selbst steht kostenlos sowohl als Bibliothek-JAR-Datei als auch als Open-Source zur eigenständigen Erweiterung zum Download zur Verfügung. [12]

Kapitel 3

Problemstellung

3.1 Formulierung

Die Aufgabe besteht darin, Signale aus mit VHDL[1] erstellten Designs automatisiert (Software) für sogenannte Power-Sensoren zugänglich zu machen. Dabei gilt es, die entsprechenden Signale an die Top-Level-Entity des Designs zu leiten. Im Folgenden wird dieser Vorgang als „Routing“ bezeichnet. Zum Test ist dieser Automatisierungsschritt anhand eines Designs des Mikrocontrollers 8051 von Oregano Systems [13], welches unter der LGPL (Lesser General Public License) verfügbar ist, zu verifizieren und die gesamte Prozedur der Power-Emulation soll letztendlich auf eine Zielhardware (FPGA[2]) portiert und getestet werden.

3.2 Mögliche Lösungswege

Für die Umsetzung der Aufgabenstellung fiel die Entscheidung auf die vMAGIC + ANTLR Lösung. Der Hauptgrund dafür ist die bereits vorhandene Grammatik, welche durch vMAGIC schon zur Verfügung steht. Somit entfällt die Programierung einer eigenen VHDL-Grammatik, was in erster Linie viel Zeit spart und umfangreichen Verifikationsaufwand einer selbst-erstellten Grammatik vermeidet. Da der vMAGIC + ANTLR Ansatz auf der Programmiersprache Java basiert, wurde die Software auch in dieser Sprache implementiert.
Das Nios 2-Entwicklungsboard beinhaltet ein FPGA der Firma Altera, von daher wird die Altera-spezifische Entwicklungsumgebung Quartus zur Synthese verwendet, insbesondere in Hinblick darauf, dass Design auch auf das Altera-FPGA zu mappen. Es kann selbstverständlich jede andere Entwicklungsumgebung (z.B. Xilinx ISE) auch verwendet werden.
Als Simulator wird ModelSim verwendet, wobei es auch hier keine Rolle spielt, welche VHDL-Simulatoren verwendet werden. Es gibt eine eigene Quartus-Edition dieses Simulators und er beinhaltet bereits die FPGA-spezifischen Bibliotheken.

[1] Very High Integrated Circuit Hardware Description Language
[2] Field Programmable Gate Array

Kapitel 4

Praktische Umsetzung

4.1 Konzept

4.1.1 Software

Zunächst soll spezifiziert werden, wie der Analyse und Adaptierungs - Algorithmus aufgebaut werden soll.

Die wesentlichen Schritte dabei sind (siehe Abbildung 4.1):

- Einlesen sämtlicher benötigter *.vhd-Dateien laut Pfadangabe
- Einlesen der Konfiguration (XML[1])
- Auslesen relevanter Informationen aus diesen Dateien
- Erstellung einer internen Baumstruktur mit den Entity-Abhängigkeiten
- Auslesen einer Signalliste und Suchen der Signale im Design
- Signale auf den Top-Level routen
- Modifikation des VHDL-Codes und Speichern der veränderten Dateien

Bei der Modifikation des VHDL-Codes soll darauf geachtet werden, den ursprünglichen Code so wenig als möglich zu verändern.
Weiters muss auf spezifische Sonderfälle geachtet werden, die sich beim Routen der Signale ergeben. So kann beispielsweise eine Entity mehrfach im Design instanziert sein, d.h. entsprechende zu routende Signale in der Architecture müssen mit einer Nummer versehen werden.
Vom Designer eigens zusammengesetzte Datentypen stellen ebenfalls einen Sonderfall dar, die entsprechenden Daten müssen nämlich an höhere Design-Ebenen automatisch weitergeleitet werden. Eventuell müssen auch zusätzlich definierte Packages analysiert werden, was entsprechende Definitionen bedeutet usw.

[1] Extensible Markup Language

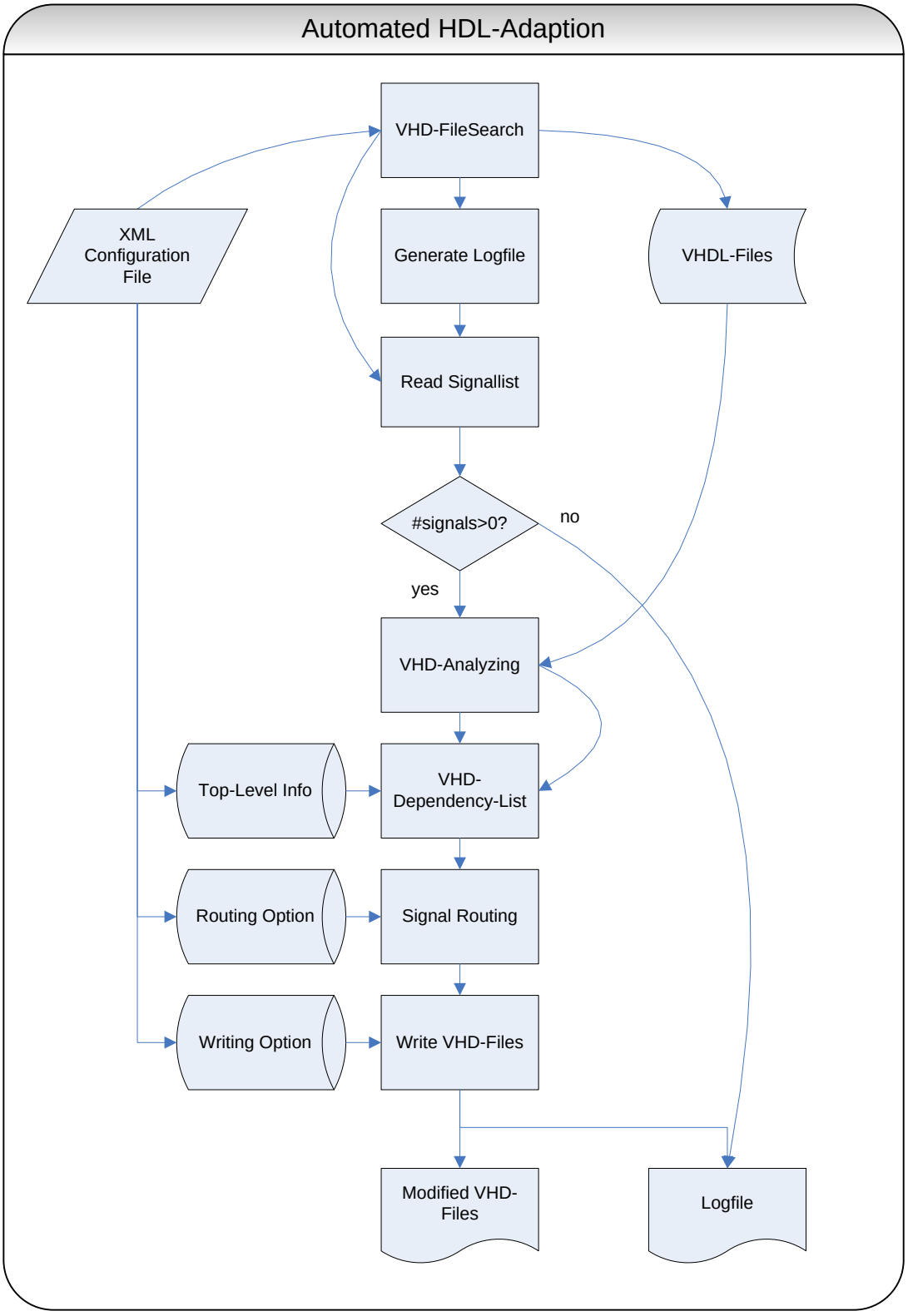

Abbildung 4.1: Flussdiagramm zum Adaptions-Algorithmus

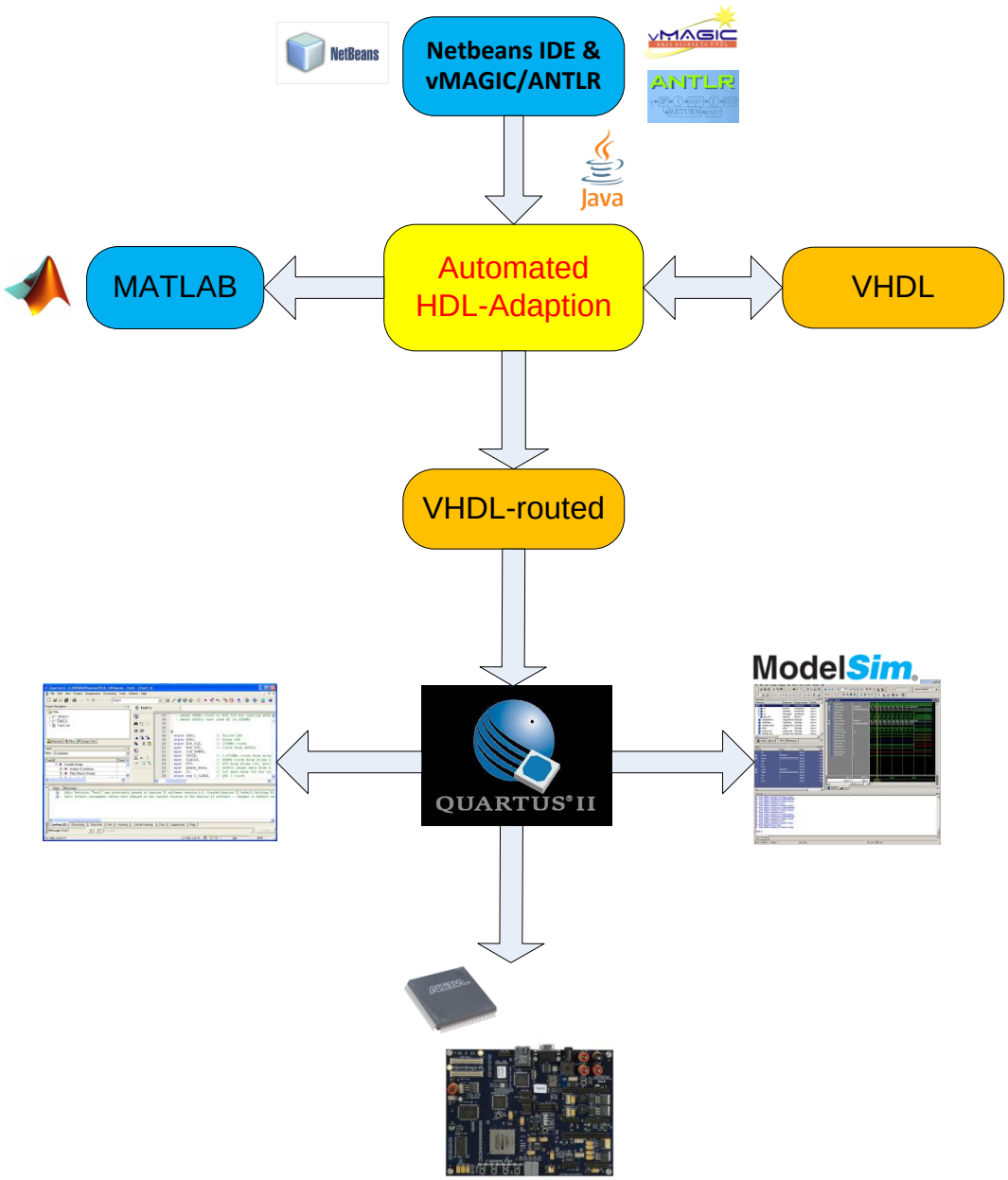

Abbildung 4.2: Blockschaltbild

Abbildung 4.2 zeigt eine Übersicht des umgesetzten Projekts mit den verwendeten Software-Tools als Blockschaltbild.

4.2 Implementierung

4.2.1 Verwendete Tools

Folgende Tools wurden zur Umsetzung verwendet:

- NetBeans 6.7 & Java JDK 6 Runtime
- vMAGIC 0.2.1 & ANTLR 3.1.1
- Quartus II 9.0 (32-Bit)
- ModelSim-Altera 6.4a (Quartus II 9.0) Starter Edition

Die Entwicklungsumgebung NetBeans steht kostenlos zum Download zur Verfügung. [14]

4.2.2 8051-Testsystem

Die Zielplattform für das Testsystem ist das Entwicklungsboard *Nios II Development Kit, Cyclone II Edition* der Firma Altera.

Es ist mit einem Cyclone II EP2C35F672 FPGA bestückt und hat zahlreiche Schnittstellen und Peripherieeinheiten sowie zusätzliche Speicher (FLASH-ROM, DDR-SDRAM, SRAM).

4.2.3 Softwareimplementierung (Klassen)

Nachfolgend wird die Implementierung der im Konzept erläuterten Schritte erklärt.

Klasse: Main

Zunächst erfolgt die Definition der Top-Level VHD-Datei, weiters der Ordner, indem die VHD-Dateien abgelegt sind und zuletzt der Name der Signalliste, die die zu routenden Signale beinhaltet.
Beginnend mit dem Erstellen der Log-Datei wird danach die Konfiguration eingelesen. Anschließend wird im angegebenen Ordner rekursiv nach VHD-Dateien gesucht und die gefundenen Dateien werden eingelesen. Danach erfolgt das Analysieren (VHDLFileParsing & Referenzierung), die Generierung einer internen Dependency-List, die die Abhängigkeiten der Entitäten im Design nachbildet, und schließlich die Datenmanipulation und die Speicherung der modifizierten VHD-Dateien.

Die Signalliste kann dabei drei verschiedene Formate annehmen. Entweder erfolgt eine Pfadangabe inklusive dem zu routenden Signal (Entity-Pfad bzw. Instanzen-Pfad) oder aber es wird nur der Signalname angegeben. In diesem Fall wird das gesamte Design nach Signalen, die mit dem angegebenen Namen korrelieren, durchsucht. Die Auswahl erfolgt hierbei über einen Parameter, der der Methode übergeben wird.

Klasse: FileInfo

Für dateibasierende Anwendungen wurde die Klasse FileInfo erstellt, die aus zwei Methoden besteht. Das Auslesen erfolgt mit der Methode

public File[ ] listFilesAsArray(File directory, String filter, boolean recurse)

Übergabeparameter: Startordner, Dateiendung, Rekursive Suche - ein/aus
Rückgabeparameter: Feld aus gefundenen vhd-Dateien

Diese Methode sucht per Option rekursiv ausgehend von der Ebene des angegebenen Startordners nach Dateien mit der angegebenen Endung (in diesem Fall *.vhd) und gibt diese als Feld zurück.

.deldir(File)

Die zweite Methode ist dazu da, um Ordner (und etwaige Unterordner bzw. Dateien) zu löschen.

Klasse: myvhdl

Diese Klasse beinhaltet sämtliche Methoden, die für das Analysieren, Modifizieren und Schreiben der *.vhd-Dateien benötigt werden.

.vhdanalyzer(File[ ])

erstellt Felder für geparste Entities, Architectures sowie allgemeine VHD-Informationen zu jeder Datei und übergibt es der Methode *.vhdparser(File, int)*
Danach werden mit Hilfe der Methode *.vhdreference()* Referenzen zwischen den Dateien (Entity und Architecture) hergestellt.

Im Feld *vhdinf[ ]* werden sämtliche Informationen zu jeder VHD-Datei gespeichert
(Feld aus Objekten wird aus Klasse VHDLInfo erstellt):

```
    public File file;
    public String entity;
    public String archid;
    public String archentity;
    public int myentindex;
    public int myarchindex;
    public boolean modified;
    public boolean ent;
    public boolean arch;
    public Vector<Signal> signal;
    public Vector<Component> component;
    public Vector<String> comp_entity;
    public Vector<String> comp_name;
    public Vector<ComponentInstantiation> compinst;
    public int[] comp_elements;
```

.vhdparser(File, int)

Diese Methode führt zuerst die wichtige Funktion *VhdlFile.parse(File)* aus, mit der die VHD-Datei übersetzt und ein AST generiert wird. Im Anschluss daran wird die Entity eingelesen, und Informationen, wie der Entity-Identifier im vhdinf[ ]-Array abgespeichert.

Beim Auslesen der Architecture werden zusätzlich Signal-Deklarationen und Komponenten-Deklarationen separat wieder im vhdinf[ ]-Array gespeichert, dasselbe wird bei den Komponenten-Instanzierungen gemacht, da diese Informationen später für die Erweiterungsmaßnahmen benötigt werden.

.vhddependencylist(String vhdtop, File[ ] files)

Die Aufgabe dieser Methode besteht darin, die Abhängigkeiten zwischen den Entities herzustellen. Dazu wird eine Datenstruktur vom Typ *DefaultMutableTreeNode* als Variable *root* verwendet.
Innerhalb wird die Methode *buildTree(..)* aufgerufen, die rekursiv eine Baumstruktur erstellt, in dem die Variable *root* um TreeNodes erweitert wird. Die Baumstruktur lässt sich auf Wunsch auch graphisch darstellen mit *showTree(root)*, diese Funktion wird aber im regulären Programmablauf nicht verwendet und dient lediglich zur Kontrolle (Debugging).

```
private static DefaultMutableTreeNode buildTree(String ent, DefaultMutableTreeNode parent,
    int depth)
{
    int comp_index=vhdinf[getvhdindexfromentity(ent)].myarchindex;
    if(comp_index>=0)
    if(vhdinf[comp_index].comp_entity != null && depth < 100)
    {
        DefaultMutableTreeNode node;
        for(int i=0;i<vhdinf[comp_index].comp_entity.size();i++)
        {
            node=new DefaultMutableTreeNode(vhdinf[comp_index].comp_entity.get(i));
            parent.add(node);
            int index2=getvhdindexfromentity(vhdinf[comp_index].comp_entity.get(i));
            if(index2>=0)
            if(vhdinf[vhdinf[index2].myarchindex].comp_entity != null)
            {
                buildTree(vhdinf[comp_index].comp_entity.get(i), node, ++depth);
            }
        }
    }
    return parent;
}
```

.vhdsignalstotop(char pathoption, String writeoption)

Zunächst kann zwischen drei Pfadoptionen (Aufbau der Signalliste) unterschieden werden:

1. pathoption:
   (a) Signalliste enthält nur Signalnamen (Architecture oder Entity-Port-Signal)
   (b) Signalliste enthält Entity-Path & Signalnamen
   (c) Signalliste enthält Entity-Instance-Path & Signalnamen

Bei der Schreiboption kann zwischen In-Place-Modification der VHD-Dateien bzw. Anlegen eines eigenen Ordners mit dem Namen „pe_vhd“ ausgewählt werden.
In-Place-Modification erstellt von den zu modifizierbaren Dateien jeweils eine Backup-Datei mit der Zusatzendung *_original* und verändert anschließend die ursprünglichen Dateien, danach werden diese am selben Ort abgelegt.

writeoption:

- „backup“; In-Place-Modification

- „standard“; Ausgabeordner pe_vhd

Top-Level-Architecture-Modification

Während des Routingvorgangs werden die gerouteten Signale in eine eigene Vektorstruktur abgespeichert. In der Top-Level-Architecture wird nach dem Routing ein *pe_state*-Vektor eingefügt, dessen Bitbreite sich aus der Breite aller gerouteten Signale zusammensetzt und mit dem in weiterer Folge alle Routingsignale verbunden werden.
Das Problem ist, dass vMAGIC noch keine Möglichkeit besitzt, die Bitbreite eines Signals einfach über eine Methode abzurufen. Deshalb musste eigens nach einer Lösung gesucht werden. Dabei wird der Signal-Subtype aufgesplittet und die einzelnen Ausdrücke untersucht. Zur Bestimmung der Bitbreite wird JEP (Math Expression Parser) zu Hilfe genommen, da die angegebene Bitbreite in einer z.B. „std_logic_vector(x downto y)“-Struktur für die Variablen x und y nicht generell eine Konstante sein muss.

.writevhdfiles(String option)

Es erfolgt eine Überprüfung der Modified-Flags (Anzeige einer etwaigen Änderung während des Routingvorgangs) für jede VHD-Datei. Wurde die Datei modifiziert, wird als nächstes eruiert, welche Option („standard“ oder „backup“) für das Schreiben der modifizierten Dateien ausgewählt wurde und dementsprechende Schreiboptionen werden angepasst.
Beim Schreiben der Architecture werden entsprechende Komponenten-Deklarationen im Declarative-Part durch die modifizierten ersetzt, dasselbe gilt ebenso für Komponenten-Instanzen im Statement-Part.

Ein Problem ergab sich im Umgang mit Generic-Statements. Das Parsen von VHD-Dateien mit diesen Ausdrücken ergibt zusätzliche redundante Erweiterungen im AST. Daher musste eine Möglichkeit geschaffen werden, die unnötigen Statements zu löschen. Zum Ersetzen einer Component-Instantiation innerhalb eines Generic-Statements dient die Methode *resetcomps(Tree parent, String id, int fileindex, int compindex)*

Log-Datei

Die Log-Datei *log.txt* wird parallel bei der Programmausführung erstellt. Sie beinhaltet sämtliche Schritte des Programmablaufs und dient in erster Linie zur Kontrolle und zur Feststellung von Fehlern. Bei der Modifikation von Entities bzw. Component-Instantiations oder beim Hinzufügen zusätzlicher Ausdrücke werden die in den jeweiligen Dateien vorgenommenen Änderungen ebenfalls mitprotokolliert.

4.2.4 Adaptions-Algorithmen

Algorithmus der Signal- und Pfad- Suche, Modifikation und Speicherung

Für jedes Element aus der Signalliste wird in ähnlicher Weise (je nach Pfadoption) eine Prozedur durchlaufen bestehend aus:

1. *searchforsignal(signal, file_counter)*
   Sucht das entsprechende Signal aus der Signalliste im angegebenen Datei-Index einer Architecture und gibt wiederum einen Index (Signalposition) zurück. Im Normalfall

sollte nur ein Signal pro Datei den Namen aufweisen, andernfalls würde es sich um einen Fehler handeln.

2. *searchforportsignal(signal, file_counter)*
   Sucht das entsprechende Signal aus der Signalliste im angegebenen Datei-Index einer Entity und gibt wiederum einen Index (Signalposition) zurück.

3. Abspeichern von Datei-Index und Signal-Index

4. *getpaths(DefaultMutableTreeNode parent, String name, Vector path)*
   Mithilfe der Dependency-List (Variable *root*) wird rekursiv nach dem Pfad, ausgehend von der Entity, indem das Signal deklariert wurde, gesucht und die einzelnen Ebenen des Pfades (= Entities) werden in der Variable *path* abgespeichert.

5. *modifyvhdfiles_x(...)*
   Hier findet die eigentliche Manipulation des VHDL-Codes statt, welche im nachfolgenden Abschnitt näher erläutert werden.

6. Speichern der modifizierten VHD-Dateien je nach Schreiboption *writevhdfiles()*

Algorithmus der VHD-Manipulation (Routingverfahren)

modifyvhdfiles_x(...)

Es gibt insgesamt drei verschiedene Modifikationsmethoden (daher das x, das als Platzhalter für 1, 2 bzw. 3 steht). Der Grund liegt in geringfügigen Unterschieden beim Routingvorgang je nach Pfadoption.
Im Folgenden wird nur eine allgemeine Darstellung des Prinzips der Verarbeitungsschritte dieser Methode dargestellt, genaue Details können aus den Kommentaren innerhalb der Klasse/Methode entnommen werden.

1. Im ersten Schritt werden Überprüfungen und Modifikationen in der Entity sowie Architecture des zu routenden Signals (Im Folgenden als *signal* bezeichnet) vorgenommen.

   - Port-Signal auf Output überprüfen; Signal muss vom Untertyp std_logic bzw. std_logic_vector (auch std_ulogic möglich) sein
   - Hinzufügen des erweiterten Power Emulation-Signals in die Start-Entity;

     ```
     signal_pe : OUT std_logic;
     ```

   - Hinzufügen eines ConcurrentSignalAssignments in die Architecture;

     ```
     signal_pe <= signal;
     ```

   - Auslesen der Konstanten der Start-Entity-Generic und mit zu routendem Signal vergleichen; bei Verwendung einer Konstante im Signal muss die entsprechende Konstante ebenso zum Top-Level geroutet werden, in diesem Fall werden diese im *generic_array* abgespeichert.
   - Überprüfen auf Konstanten-Definition und dementsprechend Anpassung des Signaltyps (Namenserweiterung für Konstante)

```
//(e.g. DWIDTH -> DWIDTH_entity_identifier)
String root_signal_type_replace=root_signal.getType().toVhdlString();
for(int generic_index=0;generic_index<generic_array.size();generic_index++)
    root_signal_type_replace=root_signal_type_replace.replace(..);
```

- Erstellen des PE-Signals mit entsprechendem Typ aus Root-Signal bzw. *root_signal_type_replace* und eventuelles Hinzufügen eines Index bei mehreren Pfaden

```
//decide, whether indices are needed, if there is more than one instantiation
if(paths.size()>1)
pe_signal=new Signal(root_signal.getIdentifier()+"_pe_"+String.valueOf(i+1),
    root_signal.getType());
else
pe_signal=new Signal(root_signal.getIdentifier()+"_pe",root_signal.getType());

//signal-setting for entity: Output
pe_signal.setMode(Signal.Mode.OUT);
```

2. Nun werden in einer Schleife für jede Entity-Ebene folgende Schritte vollzogen:

- Suchen eines Index bei Übergabe des Entity-Namens

```
//get index from current path-entity
entindex=getvhdindexfromentity(entitypath[j].toString());
```

- Erweiterung der Entity-Generics mit den Konstanten, deren Namen den Zusatz des Entity-Identifiers der Base-Entity erhält

```
//create constant aus base_ent_generic with name: old-name+entity-identifier
Constant myconst=new Constant(base_ent_generic.getConstants().get((Integer)
    generic_array.get(generic_index)).getIdentifier().concat("_"+paths.get(i)[paths.
    get(i).length-1].toString()));
//copy type to new constant
myconst.setType(base_ent_generic.getConstants().get((Integer)generic_array.get(
    generic_index)).getType());
//copy default value to new constant
myconst.setDefaultValue(base_ent_generic.getConstants().get((Integer)generic_array.
    get(generic_index)).getDefaultValue());
```

- Erstellen einer neuen ComponentInstantiation

```
//create a component from previous entity (one level down)
Component comp=new Component(parsed_vhd_ent[previous_ent_index].getIdentifier(),
    parsed_vhd_ent[previous_ent_index].getGeneric(),parsed_vhd_ent[
    previous_ent_index].getPort());
//read old component instantiation
ComponentInstantiation compinst_old = vhdinf[vhdinf[entindex].myarchindex].comp.get(
    index);
//create new component instantiation with created component from above
ComponentInstantiation compinst_new = new ComponentInstantiation(vhdinf[vhdinf[
    entindex].myarchindex].comp.get(index).getIdentifier(),comp);
```

- Auslesen der alten PortMap und Erweiterung um das PE-Signal

```
compinst_new.getPortMap().addAssociation(pe_signal.getIdentifier(), pe_signal);
```

- Zum Schluss wird einmal das geroutete pe_signal in einem Vektor gespeichert um in weiterer Folge in der Top-Level-Architecture den pe_state-Vektor zu erstellen.

```
pe_signals.add(new Signal(root_signal.getIdentifier()+"_pe",
    root_signal_constant_replace));
```

4.2.5 Erweiterung der vMAGIC-Bibliothek

Es ergaben sich beim Verwendung der vMAGIC-Bibliothek einige Probleme, die meist umständlich gelöst werden mussten, einige dieser „Workarounds“ konnten durch Erweiterung bestimmter Klassen vereinfacht bzw. eliminiert werden. Nachfolgend werden diese beschrieben.

Klasse PortMap

Das Problem beim Hinzufügen einer neuen Assoziierung ist, dass dieses nur über die Methode *.connect* möglich ist und dabei sämtliche unveränderten Assoziierungen aus der quasi alten PortMap ausgelesen werden müssen um die erweiterte PortMap zu erstellen. Das hat allerdings zusätzliche Schwierigkeiten mit sich gebracht. So wurden zum Beispiel Signale, die mit einem Klammerausdruck versehen waren (um beispielsweise nur bestimmte Bits eines Signals zu verwenden), generell auf den Namen reduziert und der Klammerausdruck war im Code nach der Erweiterung nicht mehr zu finden.
Daher wurde eine einfache Erweiterung im Sinne des Hinzufügens weiterer Assoziierungen geschaffen:

```
public void addAssociation(String signalName, Signal signal) {
  list.addAssociation(signalName, new Name(signal.getIdentifier()));
}
```

Es gibt leider keine Methode, die einfach eine Liste der verknüpften Signale der PortMap zurückliefert, daher wurde eine get-Methode für die Assoziierungsliste eingefügt:

```
public AssociationList getAssociationList()
{
  return list;
}
```

Klasse GenericMap

Diese Klasse ist mit dem selben Problem behaftet, wie die Klasse PortMap, daher sieht die Erweiterung gleich aus:

```
public void addAssociation(String genericName, Constant constant) {
  list.addAssociation(genericName, new Name(constant.getIdentifier()));
}

public AssociationList getAssociationList()
{
  return list;
}
```

4.2.6 Synthese

Als Entwicklungsumgebung dient Quartus von der Firma Altera.
Ein wichtiger Hinweis beim Umgang mit Quartus ist, dass der Name der Top-Level-Datei mit dem Namen der Top-Level-Entity korrelieren sollte.

Einstellung - FPGA-Typ: Cyclone II EP2C35F672C6

Es wurden keine speziellen Einstellungen betreffend Synthese oder Place & Route getroffen (Standardwerte)

4.3 Resultate

4.3.1 Algorithmus-Laufzeit

Als Referenz wurde das Programm auf einem Computer mit folgenden Eckdaten ausgeführt:

Intel Core 2 Duo CPU T7100 @1.8 GHz; 2,00 GB RAM

Das 8051-Design von Oregano-Systems [13], welches aus insgesamt 49 VHD-Dateien besteht, wird zum Testen verwendet. Es werden 10 beliebige Signale auf den Top-Level geroutet. Durch den Routingvorgang werden insgesamt 12 Dateien modifiziert.

Anzumerken ist hierbei, dass jedes dieser Signale geroutet wurde, d.h. etwaige Fehlangaben (Input-Signale bzw. benutzerdefinierte Datentypen) wurden von vornhinein vermieden, da ein nicht-geroutetes Signal sich natürlich begünstigend auf eine kürzere Gesamtausführungszeit auswirkt.

Da beim mehrmaligen Ausführen des Programms sich jeweils unterschiedliche Werte für die effektive Ausführungszeit ergaben (Die zeitliche Toleranzbreite liegt hierbei im Bereich von ca. einer Sekunde), wurde einfach der Mittelwert über mehrere Durchläufe bestimmt.

signal	**port/arch**	**depth**	**path**
q	p	1	/i_mc8051_ram/q
s_dvsor	a	2	/i_mc8051_core/i_mc8051_alu/s_dvsor
ram_wr_o	p	3	/i_mc8051_core/i_mc8051_control/i_control_mem/ram_wr_o
acc_o	p	3	/i_mc8051_core/i_mc8051_control/i_control_mem/acc_o
s_inthigh_d	a	3	/i_mc8051_core/i_mc8051_control/i_control_fsm/s_inthigh_d
s_ov	a	1	/i_mc8051_core/s_ov
psw	a	3	/i_mc8051_core/i_mc8051_control/i_control_mem/psw
alu_cmd_o	p	3	/i_mc8051_core/i_mc8051_control/i_control_fsm/alu_cmd_o
cy_o	p	3	/i_mc8051_core/i_mc8051_alu/i_alucore/cy_o
opa_o	p	3	/i_mc8051_core/i_mc8051_alu/i_alumux/opa_o

Tabelle 4.1: Routing-Signale

Tabelle 4.1 zeigt die zufällig gewählten Signale und des weiteren die Signalart (Port-Signal bzw. Architecture-Signal) sowie den Pfad an. Aus dem Pfad lässt sich auch die entsprechende Tiefe (vom Top-Level aus gesehen) ableiten.

Beim ersten Durchgang wurde die Pfadoption „c“ (entspricht Instanzen-Pfad) und beim zweiten Durchgang die Pfadoption „a“ (entspricht Signalname) ausgewählt!
(Schreiboption: „standard“ = pe_folder)

Wie man erkennen kann, hat sich die Zeit für das Routing um den Faktor 2,4 erhöht. Der Grund liegt darin, dass insgesamt 28 Signale (!) an den pe_state-Vektor im Top-Level zugewiesen wurden, das sind beinahe dreimal soviele Signale wie mit der Pfadangabe. Da Signalnamen des öfteren im Design vorkommen können, erhöht sich die Anzahl dementsprechend.

Part	**Time (sec)**
Analysis	2,8
Dependency-List	0,1
Routing&Writing	8,9
Sum:	**11,8**

Tabelle 4.2: Ausführungszeit (instance path; 10 Signale)

Part	**Time (sec)**
Analysis	2,8
Dependency-List	0,1
Routing&Writing	21,1
Sum:	**24**

Tabelle 4.3: Ausführungszeit (signal name; 28 Signale)

Durch das Hinzufügen zusätzlicher Signale in die Signalliste erkennt man beim verwendeten Design entsprechende Unterschiede. Der Hauptteil der Programmausführungszeit besteht demnach im Routing der Signale. Für das verwendete Testsystem lässt sich grob als Mittelwert annehmen, dass pro zu routendem Signal sich die Programmlaufzeit um ca. 800 ms erhöht.

4.3.2 Overhead nach Synthese

Zunächst wird das ursprüngliche mc8051-Design auf das Ziel-FPGA (Cyclone II EP2C35) portiert, anschließend wird das Design um die gerouteten Signale erweitert und beide Syntheseergebnisse miteinander verglichen:

Parameter	base	adapted	overhead from base
Total logic elements	3929	4073	3,67%
Total combinational functions	3888	4044	4,01%
Total registers	581	581	0,00%
Total memory bits	132096	132096	0,00%
Timing Analyse/Clock Setup	**23,09 MHz**	**23,84 MHz**	

Tabelle 4.4: Ressourcen-Vergleich

Wie man Tabelle 4.4 entnehmen kann, werden erwartungsgemäß keine zusätzlichen Register nach dem Routingvorgang verwendet. Die Timing-Analyse zeigt eine höhere maximale Taktfrequenz für das erweiterte Design. Es wäre zu erwarten, dass die maximale Frequenz durch das Hinzufügen zusätzlicher Signale geringer wird. Das dem nicht so ist, begründet sich aus der Tatsache, dass keine anspruchsvollen Timing Constraints verwendet wurden.

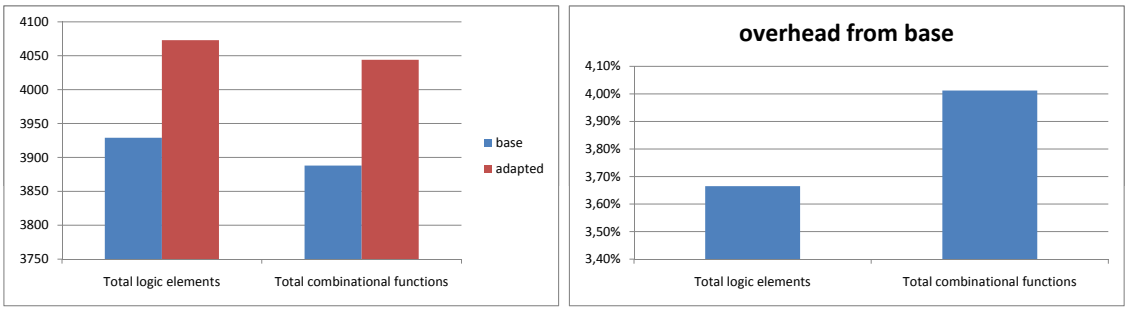

Abbildung 4.3: Ressourcen-Vergleich (links: absolut; rechts: prozentual)

Abbildung 4.3 stellt eine graphische Darstellung von Tabelle 4.4 dar.

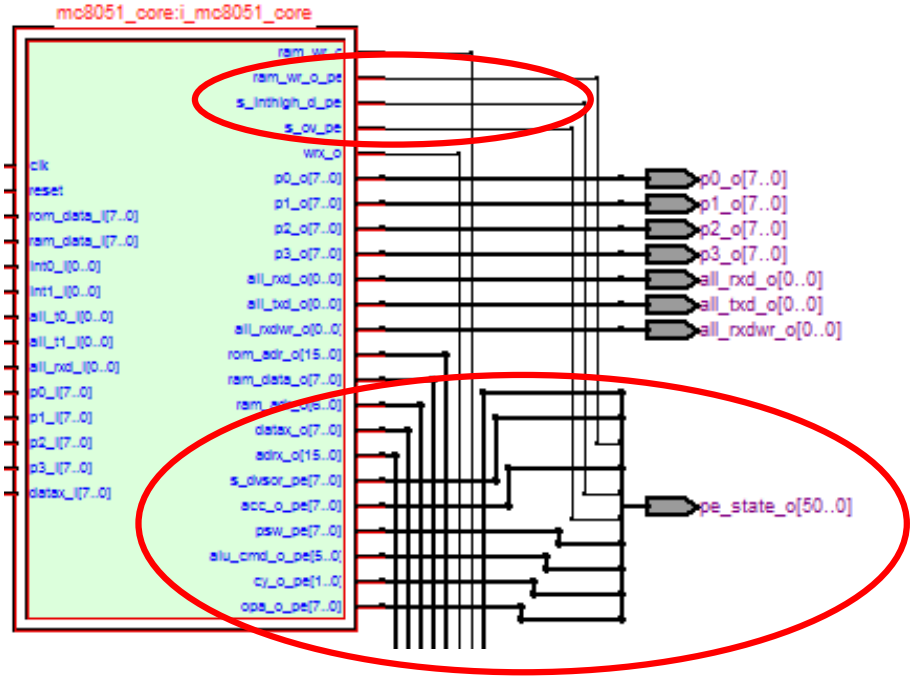

Abbildung 4.4: Netlist Viewer mit pe_state-Vektor

Abbildung 4.4 zeigt einen Screenshot aus dem RTL-Viewer von Quartus, der die Top-Level-Entity des Mikrocontroller-Designs zeigt. Man kann (rot umrandet) den *pe_state*-Vektor erkennen, der alle gerouteten Signale zusammenfasst.

4.4 Schlussfolgerung und Ausblick

Das rechtzeitige Erkennen von leistungsbestimmenden bzw. leistungskritischen Prozessen während des Designzyklus stellt einen wichtigen Schritt dar. Power Emulation bietet aufgrund von Hardware-Erweiterungen die Möglichkeit, zur Laufzeit entsprechende Informationen des Leistungsbedarfs einer Integrierten Schaltung zu erhalten.
Da digitale Schaltkreise heutzutage zumeist mittels Hardwarebeschreibungssprachen wie VHDL erstellt werden, und sich daraus eine hierarchische Struktur des Designs ergibt, ist es oft schwer, Signale in unteren Ebenen auf höheren Ebenen einfach verfügbar zu machen. Um die Signalinformation den in meist höheren Ebenen angesiedelten Power-Sensoren zugänglich zu machen, bedarf es eines Routingvorgangs der entsprechenden Signale. Dieser automatisierte Vorgang wurde in der vorliegenden Arbeit realisiert.

Die im Rahmen dieser Arbeit implementierte Software bietet zur Zeit die Möglichkeit, eine Dependency-List aus einem beliebigem, mit VHDL beschriebenen, Design zu erstellen, die gewünschten Signale aus der Signalliste auszulesen und entsprechend im Design zu finden sowie die notwendigen Schritte durchzuführen, um die Signale am Top-Level verfügbar zu machen.

Ein Problem, dass sich beim VHDL-File-Parsing zur Zeit ergibt, ist das Verlorengehen von etwaigen Kommentaren innerhalb von Entities und Architectures. Eventuell könnte dies zukünftig mitberücksichtigt werden.
Momentan bleiben Config-Files generell unberücksichtigt. Da man davon ausgehen kann, dass pro Entity jeweils eine Architecture definiert wird, kann die Konfiguration als unnötig angesehen werden.
Eine Erweiterungsmöglichkeit besteht im Modifizieren von benutzerdefinierten Signalen wie zum Beispiel *std_logic_vector* und *std_ulogic_vector*. Momentan werden diese Signale hochgeroutet ohne darauf zu achten, ob etwaige Packages dahingehend definiert wurden. Um dies umzusetzen, wäre eine Erweiterung im Sinne des Auslesens von zusätzlichen, definierten Packages notwendig. Leider ist laut Entwickler ein Package-Handling in der aktuellen Version der vMAGIC-Bibliothek nicht verfügbar.

Bei der Implementierung kam es zu unerwarteten Problemen beim Umgang mit der vMAGIC-Bibliothek, insbesondere bei Component-Instantiations. Eine einfache Erweiterung vom Port-Mapping (sowie auch dem Generic-Mapping) sucht man vergebens. Die einzige Möglichkeit besteht darin, eine neue Komponente anzulegen, der die erweiterte Entity bergeben wird und in weiterer Folge alle Signale aus der alten PortMap (GenericMap) ausgelesen und verknüpft werden und schließlich um das Power-Emulation-Signal erweitert wird.

Abschließend sei noch erwähnt, dass an der vMAGIC-Bibliothek seitens der Entwickler noch immer Erweiterungen und Verbesserungen vorgenommen werden. Somit ist es denkbar, dass sich obige Probleme zukünftig ziemlich einfach durch Verwendung einer neueren Version dieser Bibliothek beheben lassen.
Aus dem vMAGIC-Forum ist bekannt, dass die neue Bibliothek auch mit einer neuen API geliefert wird, von daher ist ein erweiterter Programmumbau wahrscheinlich unumgänglich.

Literaturverzeichnis

[1] A. Genser, Ch. Bachmann, J. Haid, Ch. Steger, and R. Weiss. An Emulation-Based Real-Time Power Profiling Unit for Embedded Software. In *SAMOS2009, International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, Samos, Greece*, July 2009.

[2] Christian Piguet. *Low Power Electronics Design.* CRC Press, 2005.

[3] M. Keating, D. Flynn, R. Aitken, A. Gibbons, and S. Kaijian. *Low Power Methodology Manual.* Springer, 2007.

[4] J. Coburn, S. Ravi, and A. Raghunathan. Power emulation: a new paradigm for power estimation. In *Proc. 42nd Design Automation Conference*, pages 700–705, 2005.

[5] C. Bachmann, A. Genser, C. Steger, R. Weiss, and J. Haid. Accelerating embedded software power profiling using run-time power emulation. In *Power and Timing Modeling, Optimization and Simulation, 19th International Workshop, PATMOS 2009*, 2009.

[6] http://de.wikipedia.org/wiki/parser. Abgerufen am 23.09.2009.

[7] http://de.wikipedia.org/wiki/parsergenerator. Abgerufen am 23.09.2009.

[8] http://www.antlr.org/.

[9] https://javacc.dev.java.net/.

[10] Christopher Pohl, Carlos Paiz, Mario Porrmann. vMAGIC - Automatic Code Generation for VHDL. Technical report, Heinz Nixdorf Institute, University of Paderborn, 2008.

[11] http://wwwhni.uni-paderborn.de/sct/extern/vmagic/documentation/.

[12] http://sourceforge.net/projects/vmagic/.

[13] http://www.oregano.at/ip/8051.htm.

[14] http://www.netbeans.org/.