Entwurf von digitalen Schaltungen und Systemen mit HDLs und FPGAs

Einführung mit VHDL und SystemC

von
Prof. Dr.-Ing. Frank Kesel und
Dr. Ruben Bartholomä

2., korrigierte Auflage

Oldenbourg Verlag München

Prof. Dr. Frank Kesel ist seit 1999 Professor für Integrierte Schaltungstechnik an der Hochschule Pforzheim. Er leitet zudem das Steinbeis-Transferzentrum an der Hochschule Pforzheim sowie den Studiengang "Technische Informatik".

Dr. Ruben Bartholomä studierte Nachrichtentechnik an der Fachhochschule Köln und Elektrotechnik/Informationstechnik an der Hochschule Pforzheim. Nach dem Studium arbeitete Herr Bartholomä als wissenschaftlicher Mitarbeiter am Institut für Angewandte Forschung der Hochschule Pforzheim und promovierte 2008 am Lehrstuhl für Technische Informatik der Universität Tübingen. Er ist bei der Firma Robert Bosch GmbH in Reutlingen tätig.

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Rosenheimer Straße 145, D-81671 München
Telefon: (089) 45051-0
oldenbourg.de

Lektorat: Anton Schmid
Herstellung: Anna Grosser
Coverentwurf: Kochan & Partner, München
Gedruckt auf säure- und chlorfreiem Papier
Druck: Grafik + Druck, München
Bindung: Thomas Buchbinderei GmbH, Augsburg

ISBN 978-3-486-58976-4

Begleitwort des Herausgebers

Dieses Buch behandelt den computergestützten Entwurf (CAD, Computer Aided Design) von mikroelektronischen Schaltkreisen. Diese Schaltungen sind Schlüsselelemente in der Informationsverarbeitung (Mikroprozessoren), in der Kommunikationstechnik (Internet, Mobil-Telefonie) und in Steuer- und Regelsystemen aller Art (Kfz-Elektronik, industrielle Prozesssteuerungen, Haushaltsgeräte). Sie verbreiten sich in immer weitere Gebiete der Elektrotechnik und werden dabei zunehmend komplexer. Durch Erhöhung des Integrationsgrads aufgrund der ständigen Verbesserung der Herstelltechniken werden Chips aber gleichzeitig auch immer preiswerter (Moore'sches Gesetz). Integrierte Schaltungen, auf denen mehrere Millionen logische Funktionen integriert sind, sind heute handelsüblich. Ohne rechnerbasierte Techniken, die sich in den letzten Jahrzehnten entwickelt und zunehmend an Bedeutung gewonnen haben, wäre das Design von solchen höchstintegrierten Schaltungen technisch und wirtschaftlich nicht durchführbar.

Die Fortschritte bei den Herstelltechniken und den Entwurfsmethoden versetzen zunehmend auch kleine und mittelständische Firmen in die Lage, ihre eigenen anwendungsspezifischen Designs auf Silizium zu bringen. Eine wesentliche Rolle spielen dabei feldprogrammierbare Logikbausteine (FPGAs), die sich beim Anwender, ohne teure und zeitintensive technologische Maßnahmen, per Datenübertrag von einem PC in ihrer Funktion festlegen lassen. Nicht nur die Designmethodiken wurden ständig weiter entwickelt, sondern auch der interne Aufbau der programmierbaren Logikbausteine. Heute können mehrere Millionen Gatterfunktionen in einem FPGA-Chip realisiert werden. Interne Datenpfad-Architekturen unterstützen die Implementierung digitaler Signalverarbeitungsalgorithmen (DSP), gleichzeitig stehen ohne nennenswerte Zusatzkosten ausgetestete Mikroprozessoren als Intellectual-Property-Komponenten zur Verfügung. Deshalb wird das Erlernen der nötigen Designmethodik für diese SOPCs (System on a Programmable Chip) zunehmend in der breiten studentischen Ausbildung in den Fächern Elektrotechnik und Technische Informatik wichtig.

Dieses ausgezeichnet geschriebene und praxisorientierte Buch „Entwurf von digitalen Schaltungen und Systemen mit HDLs und FPGAs“ bietet dazu eine zielgerichtete Einführung, beginnend bei MOS-Transistoren und FPGA-Technologien bis hin zu aktuellsten Entwicklungen in der Synthese (High-Level-Synthese) und den Hardwarebeschreibungssprachen (SystemC). In diesem Buch werden als Kernpunkte des CAD-Flows die Modellierung mit Hardwarebeschreibungssprachen (HDLs), die Simulation solcher HDL-Modelle und deren Umsetzung und Optimierung (Logiksynthese) von der Architektur- oder Verhaltensebene ausgehend bis zur Implementierung in der ausgewählten Zieltechnologie behandelt. In dieser modernen Darstellung des Themengebiets werden standardisierte

Beschreibungen und flexible Implementierungsverfahren in den Vordergrund gestellt, die auf objektorientierten Sprachen und den komplexen programmierbaren Logik-Bausteinen (FPGAs) beruhen. In vielen typischen Anwendungsbeispielen lernt der Leser die Besonderheiten der synthesefähigen Hardwarebschreibung systematisch kennen. Waren bisher spezielle HDL-Sprachen wie VHDL oder Verilog als Eingabeformen einer Designspezifikation üblich, ist es mittlerweile mit der High-Level-Synthese möglich, direkt algorithmische C ++-Beschreibungen aus der Systementwicklung (bspw. mit MATLAB) zu übernehmen und so zu transformieren, dass die Logiksynthese und damit die Abbildung auf eine Zieltechnologie möglich werden. So werden alle fachlichen Vorrausetzungen von den Grundlagen bis zu praxiserprobten Verfahrensweisen präsentiert, die Studierende oder Industriepraktiker benötigen, um heute und auch in Zukunft in diesem rasch veränderlichen technischen Gebiet erfolgreich tätig sein zu können.

Für mich als Herausgeber war nicht nur die Zusammenarbeit mit den Autoren die reinste Freude, sondern ich konnte auch Vieles dazulernen oder vorhandene Kenntnisse in neuem Licht sehen. Ich freue mich, dass nun auch in deutscher Sprache ein Lehrbuch über dieses wichtige Gebiet zur Verfügung steht, das mit den angelsächsischen Vorbildern keinen Vergleich zu scheuen braucht, und wünsche allen Lesern viele lehrreiche Stunden bei der Lektüre und viel Spaß bei der Umsetzung der Erkenntnisse in der Praxis, denn nach Goethe genügt es nicht etwas „nur zu wissen, sondern man muss es auch anwenden“.

Prof. Dr. Bernhard Hoppe
Darmstadt

Vorwort

Digitale Systeme durchdringen heute viele Bereiche des täglichen Lebens, ohne dass uns dies vielleicht überhaupt bewusst ist. Man denke hier beispielsweise an Mobilfunktelefone, Navigationsgeräte oder die Automobilelektronik. Der Entwurf solcher Systeme stellt daher für viele Unternehmen eine Schlüsselkompetenz dar. Während die Software einen wachsenden Anteil an digitalen Systemen hat, so muss doch häufig auch die Hardware in Form von integrierten Schaltungen entwickelt werden. Nicht nur für große sondern auch für kleine und mittelständische Firmen ergibt sich die Notwendigkeit digitale Hardware als ASICs oder mit programmierbaren Bausteinen zu entwickeln.

Der Entwurf digitaler Hardware beruht heute im Wesentlichen auf so genannten Hardwarebeschreibungssprachen, wie VHDL oder Verilog, welche eine ähnliche Funktion für die Hardwareentwicklung haben wie Programmiersprachen für die Softwareentwicklung. Mehr als fünfzehn Jahre Erfahrung in der Entwicklung von digitalen Schaltungen in der Industrie und an der Hochschule lehren jedoch, dass es nicht nur Kenntnisse einer Hardwarebeschreibungssprache alleine sind, welche für den erfolgreichen Entwurf wichtig sind, sondern auch Kenntnisse der digitalen Schaltungstechnik – dies ist heute hauptsächlich die CMOS-Schaltungstechnik – sowie der rechnergestützten Entwurfswerkzeuge, welche für den Entwurf benutzt werden. Ein Entwickler einer Schaltung sollte eine Vorstellung vom Aufbau und der Funktionsweise der von ihm in VHDL oder Verilog beschriebenen Hardware haben, um mögliche Fehler bei der automatischen Umsetzung der VHDL- oder Verilog-Beschreibung durch die Entwurfswerkzeuge entdecken zu können und um die Qualität der generierten Schaltung beurteilen zu können. Aus diesen Überlegungen heraus entstand die Idee, ein Buch zu schreiben, welches die Zusammenhänge und Wechselwirkungen zwischen den einzelnen Themengebieten des digitalen Hardwareentwurfs darstellt. Es gibt zwar viele Bücher über VHDL und andere Hardwarebeschreibungssprachen, über digitale Schaltungstechnik und über die Funktionsweise von Entwurfswerkzeugen, aber nur wenige umfassende und zusammenhängende Darstellungen.

Das vorliegende Buch ist in erster Linie ein Lehrbuch über den Entwurf von digitalen Schaltungen und Systemen. Die geschilderten Verfahren und Vorgehensweisen werden derzeit in der Industrie angewendet. Es ist auch ein Buch über VHDL, aber eben nicht nur. Ungewöhnlich ist in diesem Zusammenhang vermutlich die detaillierte Darstellung der CMOS-Schaltungstechnik, welche, angefangen bei den MOS-Transistoren, über digitale CMOS-Schaltungen und Speicherschaltungen zu den Technologien von programmierbaren Schaltungen führt. Letztere dienen in Form von FPGAs im weiteren Verlauf als Beispiel, um die Umsetzung von VHDL-Beschreibungen in Hardware und die damit verbundenen Probleme zu zeigen. Obgleich die von uns für das Buch als Beispiel gewählte

FPGA-Technologie naturgemäß schnell veraltet sein wird, so glauben wir doch, dass die daran aufgezeigten Probleme und Lösungsmöglichkeiten auch auf neuere Technologien anwendbar sein werden. Auf den Zusammenhang zwischen der abstrakten VHDL-Beschreibung und der Funktionsweise der daraus resultierenden mikroelektronischen Hardware kommt es uns insbesondere an. Nebenbei bekommt der Leser so auch einen Überblick über wesentliche Implementierungsformen von digitalen integrierten Schaltungen und deren Funktionsweise. Die Prinzipien des digitalen Hardwareentwurfs zeigen wir anhand von vielen überschaubaren Beispielen und leiten daraus Verallgemeinerungen ab – eine Vorgehensweise, die sich in der Lehre bewährt hat. So wird dem Lernenden auch ein Repertoire von VHDL-Musterbeschreibungen an die Hand gegeben, welches viele Anwendungsfälle abdeckt. Weil das Buch mehrere Themengebiete – Hardwarebeschreibungssprachen, digitale Schaltungstechnik und Entwurfswerkzeuge – abdeckt, kann natürlich nicht jedes Gebiet für sich erschöpfend behandelt werden. Dennoch ist das Buch als eine in sich abgeschlossene Abhandlung gedacht, so dass weitere Literatur für das Verständnis zunächst nicht benötigt wird. Für den interessierten Leser, der sich in einzelne Themen vertiefen möchte, haben wir weiterführende Literatur an den entsprechenden Stellen angegeben. Das Buch zeichnet sich ferner durch eine praxisorientierte Einführung in den modernen Entwurf von digitalen Schaltungen und Systemen aus, wobei die theoretischen Aspekte und Zusammenhänge nicht zu kurz kommen.

Dieses Buch wäre nicht ohne die tatkräftige Mithilfe und Unterstützung einiger Personen entstanden. Mein Dank gebührt zunächst meinem Koautor und Mitarbeiter an der Hochschule Pforzheim, Herrn Dipl.-Ing. (FH) Ruben Bartholomä. Er hat das sechste Kapitel geschrieben und dort einen Ausblick auf fortgeschrittene Entwurfsverfahren mit SystemC und High-Level-Synthese gegeben und auch durch zahlreiche Kommentare zu den restlichen Kapiteln des Buches zum Gelingen beigetragen. Einen besonderen Dank möchte ich meinem Kollegen an der FH Darmstadt, Herrn Prof. Dr. Bernhard Hoppe, aussprechen – ohne ihn wäre dieses Buch vermutlich nicht entstanden. Er hat als Herausgeber einer Buchreihe im Oldenbourg-Verlag den Anstoß zu diesem Buch gegeben und es von Anfang bis Ende begleitet. Ihm sei auch insbesondere für die mühevolle fachliche Überprüfung des Manuskripts gedankt. Ebenfalls für die fachliche Überprüfung des Manuskripts sei meinem Kollegen an der Hochschule Pforzheim, Prof. Dr. Frank Thuselt, und meinem Kollegen aus der MPC-Gruppe, Prof. Ermenfried Prochaska, FH Heilbronn, gedankt. Selbstverständlich gilt auch dem Oldenbourg Wissenschaftsverlag mein besonderer Dank für die Möglichkeit, dieses Buch zu veröffentlichen. Auch den Firmen Altera, Mentor Graphics und Xilinx sei für die Überlassung von Material und die Genehmigung zum Abdruck gedankt. Bedanken möchte ich mich insbesondere auch bei den Studierenden, Mitarbeitern und Kollegen an der Hochschule Pforzheim für die vielen Fragen, Anregungen und Diskussionen, welche mir über die letzten Jahre viele neue Einsichten gebracht und damit auch zum Buch beigetragen haben. Nicht zuletzt gilt meine Dankbarkeit meiner Familie, die viel Verständnis für die Arbeiten an diesem Buch aufgebracht und somit ebenfalls zum Gelingen beigetragen hat.

Trotz aller Sorgfalt und Überprüfungen durch Fachkollegen und das Lektorat des Verlags können sich bei der ersten Auflage eines Buches dennoch Fehler einschleichen. Ich möchte mich hierfür bei den Leserinnen und Lesern des Buches schon im Voraus entschuldigen. Für Fehlermeldungen und auch für weitere Anregungen bin ich jederzeit dankbar; richten Sie diese am besten per E-Mail (frank.kesel@hs-pforzheim.de) an mich. Nun verbleibt mir nur noch, Ihnen viel Freude bei der Lektüre und der Arbeit mit diesem Buch zu wünschen – mindestens so viel Freude wie wir bei der Erstellung des Buches hatten.

Frank Kesel
Pforzheim

Vorwort zur zweiten Auflage

Erfreulicherweise wurde für das vorliegende Buch nach relativ kurzer Zeit eine zweite Auflage erforderlich. Dies zeigt, dass das Buch erfolgreich angenommen wurde, was auch durch zahlreiche positive Rückmeldungen von Rezensenten und Lesern bestätigt wird. Das Buch hat auch seinen Praxistest beim Einsatz in der Lehre an unserer und an anderen Hochschulen bestanden und wird von den Studierenden gerne als Vorlesungsbegleiter benutzt. Auch die Rückmeldungen aus der Industrie und der erfolgreiche Einsatz des Buches in Schulungen für die Industrie zeigen, dass das Buch ebenfalls für den Praktiker im industriellen Umfeld geeignet ist.

Die didaktischen Ziele des Buches konnten somit für die intendierten Zielgruppen erreicht werden. Es war daher aus unserer Sicht nicht erforderlich, das Buch grundlegend zu überarbeiten und so wurden nur einige wenige Textpassagen überarbeitet und aktualisiert. Ferner wurden Fehler korrigiert und einige Abbildungen überarbeitet. Zwar sind in den vergangenen zwei Jahren wieder einige Fortschritte im Bereich der FPGAs zu verzeichnen, so dass die im Buch verwendete Beispieltechnologie der Virtex-2-FPGAs von Xilinx nun schon durch die übernächste Generation der Virtex-5-FPGAs abgelöst wurde, jedoch hat sich an der grundlegenden Vorgehensweise im physikalischen und im logischen Entwurf mit VHDL nichts geändert. Aus diesem Grund sehen wir für die neue Auflage des Buches noch keine Notwendigkeit, die Beispiele, insbesondere aus den Kapiteln 4 und 5, auf die neueste FPGA-Technologie umzustellen. Die anhand der Beispiele gezeigte Vorgehensweise lässt sich unserer Ansicht nach auch auf die neuesten Bausteingenerationen übertragen. Ähnliches gilt auch für die Entwurfswerkzeuge: Auch hier sind in den vergangenen Jahren seit der ersten Auflage des Buches einige neue Versionen der im Buch verwendeten Werkzeuge auf dem Markt erschienen. Beispielsweise ist das von uns für die Logiksynthese verwendete Werkzeug „LeonardoSpectrum“ von MentorGraphics durch das Nachfolgeprodukt „Precision“ abgelöst worden. Wenn sich dadurch auch Details der Werkzeuge oder deren Bedienung verändert haben oder neue Funktionen hinzugekommen sind, so ist doch die im Buch geschilderte grundsätzliche Vorgehensweise im Wesentlichen gleich geblieben.

Einige Leser äußerten den Wunsch, die zahlreichen VHDL-Beispiele aus dem Buch doch als Quellcode zum Download zur Verfügung zu haben. Wir kommen diesem Wunsch gerne nach und stellen die Quellcodes für die VHDL- und SystemC-Beispiele auf der Internetseite des Buches beim Oldenbourg Verlag (www.oldenbourg-wissenschaftsverlag.de) zur Verfügung. Wie wir im Buch ausgeführt haben, ist es notwendig, dass der Leser auch eigene Beispiele und Projekte bearbeitet, um das Wissen zu vertiefen, hierfür können die Beispiele des Buches als Startpunkt dienen. Wir empfehlen als Werkzeuge für den VHDL-

FPGA-Entwurf beispielsweise das „ISE WebPACK“ von Xilinx, welches kostenlos von der Xilinx-Homepage (www.xilinx.com) heruntergeladen werden kann. Darin enthalten sind sämtliche Werkzeuge für den Enwurf, z.B. für Simulation, Synthese, physikalischer Entwurf oder Timing-Analyse. Im Vergleich zur kostenpflichtigen ISE-Vollversion gibt es zwar einige Einschränkungen, hauptsächlich bezüglich der Schaltungskomplexität, jedoch ist das „WebPACK“ für kleinere Projekte gut einsetzbar. Ähnliche Angebote sind auch von anderen Herstellern, wie beispielsweise Altera, verfügbar. Die VHDL-Simulationen im Buch wurden durchgängig mit dem „Modelsim“-Simulator von MentorGraphics durchgeführt, welcher als Industrie-Standard-Simulator sehr weit verbreitet ist und z.B. auch in der „ISE“ von Xilinx enthalten ist.

Für die Simulationen zur MOS-Technologie im dritten Kapitel haben wir das OrCAD-Werkzeug und den darin enthaltenen PSpice-Simulator benutzt. Eine kostenfreie Demo-Version des Werkzeugs kann von der Firma Cadence bezogen werden (www.cadence.com). Für diejenigen, die die PSpice-Simulationen nachvollziehen möchten, ist eine Bibliothek für die im Buch benutzten MOSFET-Symbole und die dazugehörigen Simulationsmodelle für OrCAD/PSpice ebenfalls zum Download von der Internetseite des Buches verfügbar.

Mein Dank gilt Herrn Anton Schmid vom Oldenbourg Verlag für die Betreuung und Durchsicht der zweiten Auflage. Für Rückmeldungen zu dieser Auflage des Buches bin ich natürlich jederzeit dankbar. Richten Sie diese entweder an den Oldenbourg Verlag oder per Email an mich (frank.kesel@hs-pforzheim.de). Ich wünsche allen Lesern ein vergnügliches und erfolgreiches Arbeiten mit dem Buch.

Frank Kesel
Pforzheim, im August 2008

Inhaltsverzeichnis

1 Einleitung

1.1 Digitaltechnik und die mikroelektronische Revolution

Obgleich es aus dem alltäglichen Umgang mit technischen Geräten vielleicht nicht erkennbar ist, so hat die mikroelektronische Realisierung von digitalen Schaltungen die technische Entwicklung in den letzten fünfzig Jahren in dramatischer Weise vorangetrieben und ein Ende dieser Entwicklung ist derzeit nicht zu sehen. Das Beispiel Internet zeigt, dass die *Digitaltechnik* und die *Mikroelektronik* – Technologien die das Internet erst ermöglichten – auch erhebliche Einflüsse auf unsere Gesellschaft haben. Von einer revolutionären Entwicklung durch die Mikroelektronik zu sprechen ist daher nicht übertrieben. Wir wollen im Folgenden anhand eines kurzen historischen Exkurses aufzeigen, dass die enorme Weiterentwicklung der Leistungsfähigkeit von digitalen Computern immer geprägt war durch die technischen Realisierungsmöglichkeiten – von mechanischen Lösungen über Relais, Vakuumröhren und Transistoren hin zu mikroelektronischen Realisierungen.

Der Begriff „digital“ („digitus“ <lat.>: der Finger) bedeutet, dass ein Signal oder ein Zeichen nur endlich viele *diskrete* Werte annehmen kann, im Unterschied zu *analogen* Signalen. Kann das Signal oder das Zeichen nur zwei Werte (z. B. 0 und 1) annehmen, so spricht man von einem *binären*, digitalen Signal [83] oder von einem Binärzeichen (engl.: Bit, Binary Digit). Die Entwicklung der Digitaltechnik ist eng mit der Entwicklung von Rechenmaschinen – im Englischen als „Computer“ bezeichnet (to compute = berechnen) – verknüpft. Erste mechanische Rechenmaschinen wurden schon im 17. Jahrhundert von Blaise Pascal, Wilhelm Schickart oder Gottfried Wilhelm von Leibniz entwickelt. Charles Babbage machte im frühen 19. Jahrhundert mit der so genannten „Analytical Engine“ einen ersten Vorschlag für einen frei programmierbaren Computer, welcher wesentliche Bestandteile enthielt die auch in modernen Computern vorhanden sind. Zu diesem Zeitpunkt mussten Rechenmaschinen mechanisch realisiert werden, da die Elektrotechnik noch am Anfang ihrer Entwicklung stand. Babbage war seiner Zeit voraus: Die Analytical Engine war aufgrund ihrer Komplexität mechanisch nicht realisierbar.

Erst durch die Entwicklung der *Relaistechnik* in den zwanziger Jahren des vergangenen Jahrhunderts konnte Konrad Zuse 1936 in Deutschland eine erste elektromechanische Realisierung eines Rechners vorstellen. Auch in den USA wurde 1937 von Howard Aiken an der Harvard University eine elektromechanische Version eines Rechners (Harvard Mark I) realisiert. Während die Harvard Mark I noch im Dezimalsystem rechnete, so wurden die Rechner ab den vierziger Jahren weitgehend im dualen Zahlensystem implementiert. Rech-

nen im *Dualsystem* bedeutet, dass für die Implementierung der Arithmetik nur binäre Zeichen benutzt werden und es sich um ein Stellenwertsystem zur Basis 2 – das Dezimalsystem verwendet die Basis 10 – handelt. Einer der wesentlichen Gründe, warum man zum Dualsystem überging, lag in der Tatsache begründet, dass die Arithmetik im Dualsystem einfacher zu realisieren ist.

Die vierziger Jahre waren auch gekennzeichnet durch das Ersetzen der Relais durch elektronische *Vakuumröhren*, welche ein schnelleres Schalten ermöglichten und damit eine höhere Leistungsfähigkeit der Rechner. Nicht nur die Arithmetik, sondern auch die Informationsspeicherung und die Steuerschaltungen wurden digital mit einer zweiwertigen Logik – also binär – realisiert. Die digitale, binäre Implementierung hat gegenüber einer analogen Realisierung den wesentlichen Vorteil, dass die elektronischen Schaltungen sehr viel störunempfindlicher werden. Die Störunempfindlichkeit beruht auf der Tatsache, dass die elektronische Schaltung nur zwei diskrete Schaltzustände, nämlich „0“ und „1“, realisieren muss. Die mathematische Grundlage der Digitaltechnik ist die von George Boole 1847 eingeführte „Boole'sche Algebra“. Sie wurde 1937 von Claude Shannon in die so genannte „Schaltalgebra“ umgesetzt, welche noch heute die Grundlage für die Realisierung von hochkomplexen Schaltungen mit Milliarden von Transistoren ist.

Problematisch an der Röhrentechnik in den vierziger Jahren war allerdings die Tatsache, dass die Rechner für heutige Verhältnisse gigantische Ausmaße und einen enormen Energiebedarf hatten. Der von Eckert und Mauchley [57] 1945 an der Universität von Pennsylvania entwickelte ENIAC (Electronic Numerical Integrator and Computer) benötigte beispielsweise 18.000 Röhren und hatte einen Platzbedarf von 1.400 qm. Die benötigte elektrische Leistung betrug 140 kW (!) und ergab dabei eine nach heutigen Maßstäben äußerst geringe Rechenleistung von 5.000 Additionen pro Sekunde, was man heute als 0,005 MIPS (MIPS: Million Instructions per Second) bezeichnen würde. Während heutige Mikroprozessoren aus einem Watt zugeführter elektrischer Leistung Rechenleistungen von mehr als tausend MIPS gewinnen (1.000 MIPS/Watt), so brachte es die ENIAC nur auf umgerechnet etwa $3{,}6 \cdot 10^{-8}$ MIPS/Watt. Dieser enorme Unterschied in der Energieeffizienz macht vielleicht schon die gewaltigen Fortschritte der Computertechnik in den letzten sechzig Jahren deutlich.

Ein wesentlicher Schritt hin zur Mikroelektronik begann mit der Erfindung des *Bipolar-Transistors* durch Bardeen, Brattain und Shockley im Jahre 1948 [80]. Der Begriff „Transistor“ ist ein Kunstwort aus den beiden englischen Begriffen „Transfer“ und „Resistor“. Aus einzelnen Transistoren wurden dann Mitte der fünfziger Jahre digitale Logikgatter entwickelt, die zur „Transistorisierung“ der Computer von Firmen wie IBM oder DEC eingesetzt wurden. Wie schon zuvor durch die Einführung der Röhrentechnik konnten hierdurch die Leistungsparameter der Rechner, wie Rechenleistung, Baugröße und Energiebedarf, weiter verbessert werden. Die Entwicklung der Computertechnik ist gekennzeichnet durch das beständige Streben, die Leistungsparameter – dies betraf in erster Linie die Rechenleistung – verbessern zu können. Neben vielen Verbesserungen in der Architektur der Rechner sind die immensen Fortschritte in den letzten 80 Jahren jedoch insbesondere der Mikroelektronik zuzuschreiben.

Unter einer mikroelektronischen Realisierung versteht man die Integration von Transistoren sowie weiteren Bauelementen wie Dioden, Kapazitäten und Widerständen auf einem halbleitenden Substrat, was man auch als integrierte Schaltung (engl.: integrated circuit, IC) oder „Chip“ bezeichnet. Das erste IC wurde 1958 von Jack Kilby bei Texas Instruments als Oszillatorschaltung in einem Germanium-Substrat entwickelt [57]. Robert Noyce entwickelte 1959 bei Fairchild ebenfalls eine integrierte Schaltung, allerdings auf Silizium-Basis [57]. Fairchild konnte ein spezielles Fertigungsverfahren mit Hilfe der *Photolithographie* entwickeln, mit welchem ICs mit einer ebenen oder „planen“ Oberfläche gefertigt werden konnten. Diese *Silizium-Planar-Technik* war in der Folge auch der grundlegende Prozess für MOS-Schaltungen und wird in weiterentwickelter Form noch heute verwendet. Die ersten digitalen integrierten Schaltungen wurden ab 1962 zunächst in bipolarer Technik als so genannte TTL-Gatter (Transistor-Transistor-Logik) von Firmen wie Texas Instruments und Fairchild auf den Markt gebracht. In der Folge konnten wiederum die Computer, von Firmen wie IBM oder DEC, durch den Einsatz dieser Technik verbessert werden. Die TTL-Technik war gekennzeichnet durch einen geringen Integrationsgrad, so dass in einem TTL-Baustein einige Gatterfunktionen, Speicherfunktionen oder etwas komplexere Funktionen, wie Zähler, implementiert waren. Dies wurde als SSI (Small Scale Integration: < 100 Transistoren pro Chip) bezeichnet. Zum Aufbau eines größeren Systems waren jedoch immer noch einige Platinen voll mit TTL-Bausteinen notwendig. Neben der Anwendung in Computern brachte die Elektronik in Form von Transistoren und ICs auch in anderen Branchen, wie der Unterhaltungselektronik, der Investitionsgüter oder der Luft- und Raumfahrt, erhebliche Innovationsschübe.

Ein wesentlicher Nachteil der bipolaren TTL-Technik ist es, dass die Schaltungen im Ruhezustand eine nicht unerhebliche Stromaufnahme aufweisen. Dies resultiert aus dem Funktionsprinzip des Bipolartransistors durch die *Stromsteuerung* und stand einer weiteren Erhöhung der Integrationsdichte entgegen. Auf einem anderen Funktionsprinzip beruht der (unipolare) *Feldeffekttransistor*, bei dem die Leitfähigkeit des Kanals nicht durch einen Strom sondern durch ein elektrisches Feld gesteuert wird, welches von einer am – vom Kanal isolierten – *Gate* angelegten Spannung erzeugt wird, und somit eine nahezu leistungslose Steuerung ermöglicht. Obwohl das Prinzip schon 1931 durch Lilienfeld entdeckt wurde (IGFET: Insulated Gate Field Effect Transistor) [80], verhinderte die schwierige Herstellung des isolierenden Gateoxids zwischen Gate und Kanal jedoch lange Zeit die Einführung dieses Transistors. Erst in den sechziger Jahren konnten die technischen Probleme gelöst werden und führten zur Einführung der *MOS-Technologie* (Metal-Oxide-Semiconductor). Die ersten kommerziellen digitalen MOS-ICs wurden in PMOS-Technik – das P bezeichnet den auf positiven Ladungen oder „Löchern“ beruhenden Leitungsmechanismus im Kanal – beispielsweise in Taschenrechnern eingesetzt.

Die weitere Entwicklung der Mikroelektronik lässt sich am besten an den Mikroprozessoren der Firma Intel nachvollziehen. Einige Mitarbeiter der Firma Fairchild, darunter Gordon Moore und Robert Noyce, gründeten in den sechziger Jahren die Firma Intel. Während Intel anfänglich Speicherbausteine entwickelte und herstellte, bekam die Firma Ende der sechziger Jahre von der japanischen Firma Busicom den Auftrag, ein IC für einen

Tischrechner zu entwickeln. Im Laufe der Entwicklung wurde dieses IC als programmierbarer Rechner oder Prozessor implementiert und dies war die Geburtsstunde des so genannten „Mikroprozessors“. Intel kaufte die Lizenzen von Busicom zurück und verkaufte den ersten Mikroprozessor 1971 als „Intel 4004“. Der „4004“ wurde in einem PMOS-Prozess produziert, wobei die kleinste Strukturgröße und damit die Kanallänge der Transistoren 10 μm betrug; zum Vergleich weist ein menschliches Haar eine Größe von etwa 50 μm auf. Damit war man in der Lage 2.250 Transistoren zu integrieren (MSI: Medium Scale Integration, 100-3000 Transistoren pro Chip) und der Prozessor konnte mit 740 kHz getaktet werden, wobei er eine Rechenleistung von 0,06 MIPS erreichte.

Im Jahr 1974 stellte Intel den ersten 8-Bit-Mikroprozessor „8080“ in einer 6 μm-PMOS-Technologie vor und 1978 wurde mit dem „8086“ der erste 16-Bit-Prozessor eingeführt, welcher die Basis für die von IBM entwickelten und 1981 eingeführten „Personal Computer“ (PC) war. Der „8086“ wurde in einer NMOS-Technologie entwickelt, welche im FET-Kanal auf Elektronenleitung beruhte und damit schneller als PMOS war. Der „80386“ im Jahr 1985 markiert den Einstieg in die 32-Bit-Prozessoren und auch den Übergang zu einer komplementären Schaltungstechnik, welche als CMOS (Complementary MOS) bezeichnet wird und zu einer weiteren Reduktion der Ruhestromaufnahme und zu einer noch höheren Störunempfindlichkeit führte. Die CMOS-Technik ist bis heute die wesentliche Technologie für die Implementierung von hochkomplexen digitalen ICs. Mit jeder neuen Prozessgeneration verringern sich die Abmessungen oder Strukturgrößen der Transistoren, so dass man die Prozesse nach der minimal möglichen *Kanallänge* der Transistoren charakterisiert. Nach dem „80486“ im Jahr 1989 wurde 1993 der „Pentium“-Prozessor beispielsweise in einer 0,6 μm-CMOS-Technologie eingeführt, die also eine um den Faktor 17 kleinere Strukturgröße als der „4004“ aufweist und damit etwa um den Faktor 83 kleiner als ein Haar ist. Diese Entwicklung führt zu zwei Effekten: Zum einen verringert sich der Platzbedarf für die Transistoren, so dass mehr Transistoren auf den Chips untergebracht werden können (Integrationsdichte) und damit mehr Funktionen auf der gleichen Chipfläche implementiert werden können. Zum anderen schalten die Transistoren schneller, so dass die Schaltung mit einer höheren Taktfrequenz betrieben werden kann, was wiederum zu einer höheren Rechenleistung führt. Die verschiedenen Integrationsgrade können weiter in etwa – da nicht immer einheitlich dargestellt – wie folgt klassifiziert werden: LSI (Large Scale Integration, 3.000 bis 100.000 Transistoren pro Chip), VLSI (Very Large Scale Integration, 100.000 bis 1.000.000 Transistoren pro Chip) und ULSI (Ultra Large Scale Integration, $>$ 1.000.000 Transistoren pro Chip).

Heute stellen die „Core-2“-Doppelkern-Prozessoren die letzte Stufe der Prozessorentwicklung bei Intel für Desktop-Systeme dar. Die aktuellen Prozessoren werden in 45 nm-CMOS-Technologie implementiert, also mit Strukturgrößen, die mehr als 1000 mal kleiner als ein menschliches Haar sind! Dabei werden mehr als 400 Millionen Transistoren auf einem Chip von etwa 100 mm^2 Größe integriert und der Prozessor mit mehr als 3 GHz getaktet. Damit ist der Integrationsgrad, also die Anzahl der Transistoren pro Chip, in 35 Jahren um mehr als den Faktor 170.000 erhöht worden und die Taktfrequenz um mehr als den Faktor 4.000 angewachsen.

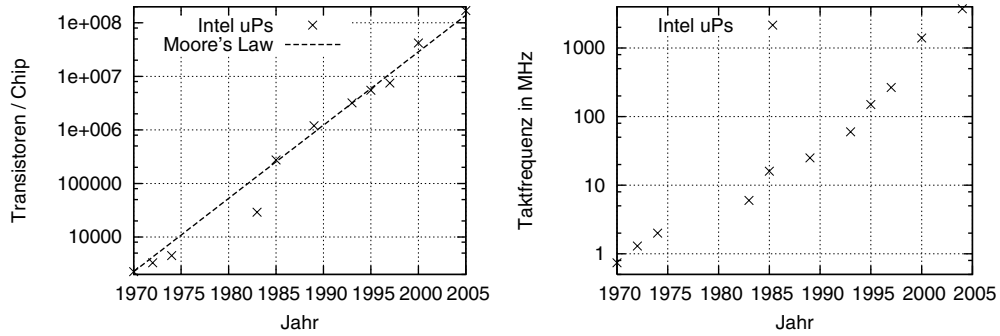

Abb. 1.1: *Entwicklung der Transistorzahlen bei Intel-Mikroprozessoren und Moore'sches Gesetz: Die linke Abbildung zeigt die Anzahl der Transistoren pro Chip für verschiedene im Text erwähnte Intel-Mikroprozessoren. Die Kurve „Moore's Law" geht von einer Verdoppelung der Anzahl der Transistoren pro Chip alle 2,2 Jahre aus. Die rechte Abbildung zeigt die Taktfrequenzen für die Intel-Prozessoren aus der linken Abbildung. Zu beachten ist, dass in beiden Abbildungen die Ordinate logarithmisch dargestellt ist, so dass jeweils ein exponentielles Wachstum vorliegt.*

Die exponentielle Entwicklung der Transistorzahlen ist in Abbildung 1.1 dargestellt. Betrachtet man die Anzahl der Transistoren pro Chip, so ergibt sich bei den Intel-Prozessoren in Abbildung 1.1 eine Verdoppelung der Anzahl etwa alle 2,2 Jahre, so dass der Vervielfachungsfaktor m für die Erhöhung der Anzahl der Transistoren in einer bestimmten Zeitspanne nach $m \approx 2^{Jahre/2{,}2}$ berechnet werden kann. Dieses exponentielle Wachstum wurde schon von Gordon Moore 1965 [51] beobachtet und ist seitdem als „Moore'sches Gesetz" (engl.: Moore's Law) bekannt. Es handelt sich allerdings weniger um ein Naturgesetz sondern um einen empirisch gewonnenen Zusammenhang. Bei den Speicherbausteinen – wo heute schon einige Gigabit Speicherkapazität vorliegen und damit einige Milliarden Transistoren integriert werden – ist das Wachstum noch etwas höher, so dass man dort von einer Verdoppelung etwa alle 18 Monate ausgeht oder $m \approx 2^{Jahre/1{,}5}$.

Nun stellt sich die Frage, wie Moore 1965 schon so hellseherisch sein konnte? Moore beobachtete die bis 1965 eingeführten Prozessgenerationen und konnte daraus diesen Zusammenhang gewinnen. Etwa alle zwei Jahre führen Halbleiterhersteller neue Prozesse ein. Jede neue Prozessgeneration weist – hauptsächlich durch Verbesserungen in der Lithographie – eine etwa um den Faktor $s \approx 0{,}7$ kleinere minimale Strukturgröße auf. Da ein Transistor damit um diesen Faktor sowohl in der Breite als auch in der Länge schrumpft, verringert sich der Flächenbedarf eines Transistors quadratisch mit $s^2 \approx 0{,}5$ und die Anzahl der Transistoren auf der gleichen Fläche erhöht sich damit um den Faktor $1/s^2 \approx 2$. Das „Moore'sche Gesetz" hat sich zu einer selbsterfüllenden Prophezeihung entwickelt, da sich die Halbleiterindustrie um „Gesetzestreue" bemüht. Letzten Endes treibt der Konkurrenzdruck und der Erwartungsdruck der Konsumenten, welche ständig höhere Rechenleistungen erwarten, die Hersteller dazu, das „Moore'sche Gesetz" zu erfüllen.

Die vielen technischen Errungenschaften, die wir als Anwender genießen können, ergeben sich aus dieser rasanten Entwicklung der Mikroelektronik, welche durch das „Moore'sche

Gesetz“ charakterisiert ist. Neben der Technologie ist insbesondere auch die Tatsache entscheidend, dass die Funktionen hauptsächlich digital realisiert werden. Man denke hier an digitale Festnetz- oder Mobiltelefone oder an digitales Fernsehen – Funktionen, die zunächst analog realisiert wurden. Viele mobile Anwendungen, wie Pocket-PCs, Navigationsgeräte oder MP3-Player sind nur durch die Digitaltechnik realisierbar. Digitale Mikroelektronik ist ein wesentlicher Wirtschaftsfaktor, wie das Beispiel Automobilelektronik zeigt: Derzeit beträgt der Anteil der Elektrik und Elektronik an den Herstellkosten eines Pkws zwischen 20% und 30% und man geht davon aus, dass sich dieser Anteil bis zum Jahr 2010 auf 40% vergrößern wird. Ferner werden 90% der zukünftigen Innovationen im Auto durch die Elektronik geprägt sein. Digitale Mikroelektronik steckt nicht nur in Computern sondern ist mittlerweile in sehr vielen technischen Geräten zu finden. Sie verrichten oft unbemerkt vom Benutzer ihren Dienst. Solche elektronischen Systeme werden daher auch häufig als „eingebettete Systeme“ oder im Englischen als „Embedded Systems“ bezeichnet.

Es sind daher nicht nur die großen Halbleiterhersteller wie Intel oder Texas Instruments, sondern auch Systemhersteller wie Bosch oder Nokia, die integrierte Schaltungen für ihre Zwecke – wobei es sich häufig um „Embedded Systems“ handelt – entwickeln und diese bei Halbleiterherstellern fertigen lassen. Diese Schaltungen werden als *ASICs* bezeichnet (Application-Specific ICs), da sie zumeist für eine bestimmte Anwendung zugeschnitten sind. Zwar können ASICs bei der Leistungsfähigkeit und der Integrationsdichte nicht ganz mit den Hochleistungsprozessoren von Intel, AMD und anderen Herstellern mithalten, jedoch gilt das „Moore’sche Gesetz“ auch bei den ASICs, die heute einige zehn Millionen Transistoren integrieren können, so dass mittlerweile komplette Systeme auf einem Chip Platz haben („SOC: System-On-A-Chip“).

ASICs sind dadurch gekennzeichnet, dass die *Masken* für die Photolithographie speziell für einen ASIC hergestellt werden müssen und dass die Erstellung des *Layouts* – das sind die Konstruktionszeichnungen für die Masken – ein aufwändiger Entwurfsprozess ist. Mit jeder neuen Prozessgeneration werden mehr Masken erforderlich und die Layouterstellung aufwändiger. Masken- und Entwicklungskosten können zu hohen Fixkosten von einigen hundertausend Euro bis zu einigen Millionen Euro für ein ASIC führen, die in der Regel nur durch eine entsprechend hohe Stückzahl für ein Unternehmen tragbar sind. Schon in den sechziger Jahren wurden daher aus Speicherschaltungen so genannte „programmierbare Bausteine“ (engl.: Programmable Logic Device, *PLD*) zur Realisierung von digitalen Funktionen entwickelt. Diese Bausteine ermöglichen, je nach verwendeter Speichertechnologie, eine beliebig häufige *Reprogrammierbarkeit* der Schaltung. Während ein ASIC nach der Herstellung nicht mehr verändert werden kann, bietet ein PLD daher gerade in der Entwicklungsphase eines ICs, wo sich manche Fehler häufig erst bei der Erprobung in der Anwendung herausstellen, sehr große Vorteile.

Während die PLDs in der Anfangszeit wenig komplex waren und daher nicht als Ersatz für ASICs taugten, konnten die so genannten *FPGAs* (Field-Programmable Gate-Array) in den letzten zehn Jahren den ASICs zunehmend Marktanteile abnehmen. Dies hat zwei Gründe: Zum einen macht die Masken- und Layoutproblematik die ASICs ökonomisch zunehmend uninteressant – gerade für kleinere oder mittelständische Firmen – und zum

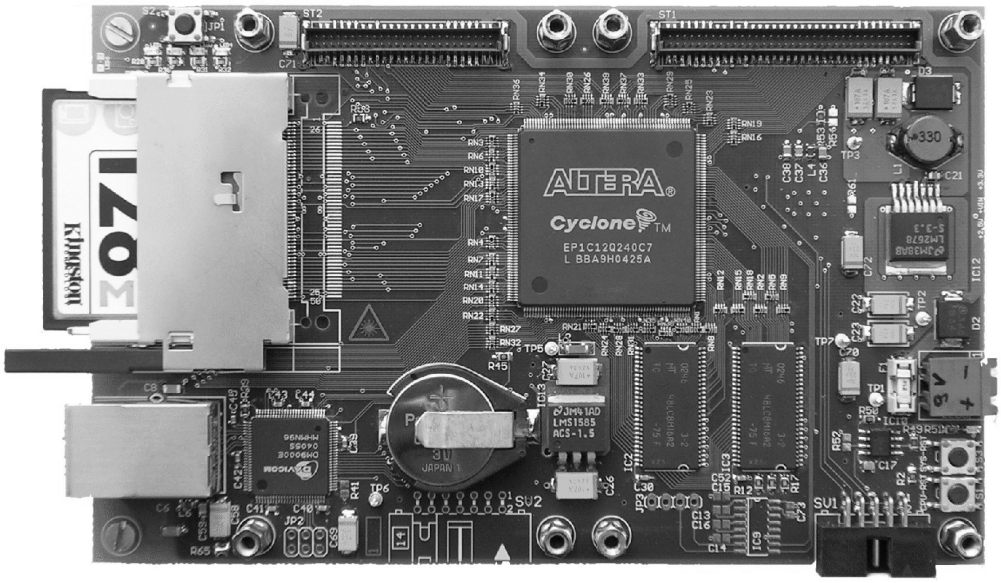

Abb. 1.2: *Beispiel für eine SOPC-Entwicklung: Es handelt sich um eine Steuerung für Lasershows [7]. Die wesentlichen Funktionen des Systems werden durch den „Cyclone“-FPGA der Firma Altera realisiert. Er beinhaltet einen NiosII-Mikroprozessor und in VHDL entwickelte Hardware für die geometrische Bildkorrektur. Auf der linken Seite ist eine CompactFlash-Karten-Schnittstelle und eine Ethernet-Schnittstelle für die Internetanbindung zu sehen, welche durch den NiosII-Prozessor bedient werden.*

anderen erlauben FPGAs, die heute ebenfalls in modernen 65 nm-CMOS-Technologien gefertigt werden, mittlerweile eine für viele Anwendungen ausreichende Designkomplexität und Rechenleistung. Auch die PLDs/FPGAs profitieren vom „Moore'schen Gesetz“. Bei PLDs und FPGAs handelt es sich um *Standard-ICs*, die in großen Stückzahlen hergestellt werden und erst vom Anwender durch Programmierung von Speicherzellen ihre Funktion erhalten. Es fallen keine Maskenkosten an und die Entwicklungskosten sind auch erheblich niedriger. Nachteilig an FPGAs ist, dass die Stückkosten, je nach Baustein, hoch sein können und dass FPGAs in der Regel nicht die gleiche Leistungsfähigkeit und Designkomplexitäten erreichen, wie ein ASIC im gleichen Prozess. Auch in FPGAs können heute komplette (eingebettete) Systeme integriert werden, was als „System-On-A-Programmable-Chip“ (SOPC) bezeichnet wird, wie das Beispiel in Abbildung 1.2 zeigt.

Waren in den sechziger und siebziger Jahren die realisierbaren Funktionen auf einem Chip durch die Anzahl der Transistoren begrenzt, so tritt seit einigen Jahren ein anderes Problem zu Tage: Die Komplexität der realisierbaren Funktionen ist heute durch die Entwurfsproduktivität der Entwickler begrenzt. Die Anzahl der Transistoren, die für Logikfunktionen zur Verfügung stehen, beträgt heute bei ASICs und FPGAs einige Millionen bis einige zehn Millionen. Mit den heute verfügbaren Entwurfsmethodiken, die im vorliegenden Buch beschrieben werden, ist ein Entwickler in der Lage einige tausend bis zehntausend Transisto-

ren pro Monat zu entwickeln [88]. Gehen wir davon aus, dass heute eine Schaltung in etwa einem Jahr fertig entwickelt sein muss, so kann ein Entwickler etwa 100.000 Transistoren pro Jahr entwickeln. Für eine Schaltung mit einer Komplexität von 1 Million Transistoren benötigen wir daher schon ein Team von zehn Entwicklern. Während die Anzahl der Transistoren, und damit die theoretisch mögliche Entwurfskomplexität, nach dem Moore'schen Gesetz mit einer Rate von etwa 50% pro Jahr wächst, entwickelt sich die Produktivität der Entwickler nur mit etwa 20% pro Jahr [88]. Diese Lücke zwischen der Anzahl der Transistoren, die die Technologie zur Verfügung stellt, und der Anzahl der Transistoren, die in einer vernünftigen Zeit entworfen werden können, wird als „Produktivitätslücke" oder im Englischen als „Design Gap" bezeichnet.

Wie kann die Produktivitätslücke geschlossen werden? Eine beliebige Vergrößerung von Design-Teams ist nicht sinnvoll. Die einzige Möglichkeit besteht somit darin, die Entwurfsproduktivität zu erhöhen. Eine wesentliche Methode hierfür ist heute die Verwendung von vorentwickelten Teilen. Hierbei kann es sich beispielsweise um einen, wie in Abbildung 1.3 gezeigten, Mikroprozessor handeln. So wie man früher ICs kaufte, um damit eine Leiterplatte zu bestücken, so kann man heute Designinformationen kaufen, die eine bestimmte Funktionalität in einem ASIC oder FPGA realisieren, wie eben einen Mikroprozessor. Da es sich nicht mehr um physikalisch vorhandene Bauelemente handelt, sondern nur noch um Informationen in Dateien auf dem Computer, spricht man auch von „geistigem Eigentum" oder im Englischen von „Intellectual Property" (IP). Ein zugekaufter Block wird dann zumeist als „IP-Core" bezeichnet. „IP-Cores" werden von darauf spezialisierten Firmen entwickelt und vertrieben. Insbesondere die Hersteller von PLDs/FPGAs bieten für

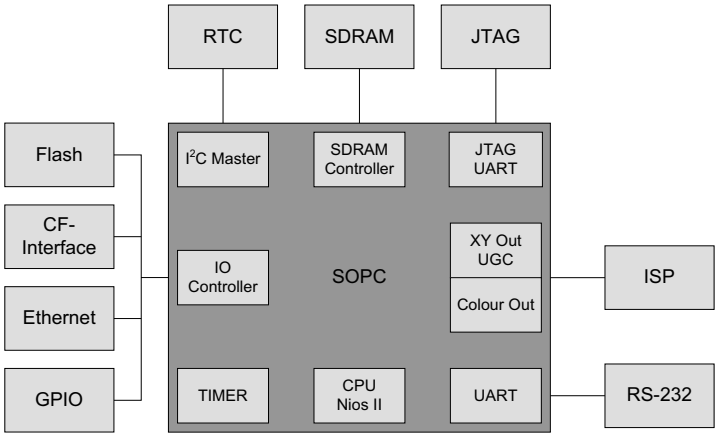

Abb. 1.3: *Nutzung von vorentworfenen Schaltungsteilen: Die Abbildung zeigt ein Blockschaltbild für die Lasersteuerung aus Abbildung 1.2 [7]. Die gesamte Komplexität des in der Abbildung mit SOPC bezeichneten FPGA-Designs beträgt ungefähr 80.000 Transistoren. Etwa die Hälfte davon wurde durch Nutzung des von Altera vorentwickelten NiosII-Prozessors und weiteren Peripherieblöcken realisiert. Nur der als UGC bezeichnete Block, welcher die andere Hälfte der benutzten Ressourcen belegt, musste selbst entwickelt werden. Das SOPC wurde von einem Entwickler in einem halben Jahr erfolgreich fertiggestellt.*

ihre Bausteine eine reiche Auswahl an „IP-Cores“ für verschiedenste Anwendungen an. Ein damit verwandter Ansatz ist es, darauf zu achten, dass Blöcke, die in früheren Designs benutzt wurden, wiederverwendet werden können (engl.: Design Reuse). Dies gelingt nur, wenn eine gewisse Standardisierung bezüglich der Schnittstellen vorgenommen wird. Ein gutes Beispiel hierfür ist wiederum ein Mikroprozessor mit einem Bussystem: Über die standardisierte Busschnittstelle können schon vorhandene Blöcke oder zugekaufte Blöcke angeschlossen werden.

Trotz „IP“ und „Design Reuse“ verbleibt immer noch ein gewisser Anteil, der in einem Design-Projekt neu entwickelt werden muss. Zur Entwicklung von digitalen integrierten Schaltungen werden heute in der Regel so genannte *Hardwarebeschreibungssprachen* (engl.: Hardware Description Language, HDL) wie VHDL oder Verilog eingesetzt. Es handelt sich dabei um Programmiersprachen, mit denen allerdings keine Software entwickelt wird, sondern eben Hardware. Untrennbar damit verbunden ist der Einsatz von *Softwarewerkzeugen* (engl.: tools), die auf einem PC unter Windows oder Linux oder einer so genannten „Workstation“ unter UNIX benutzt werden können, um die HDL-Beschreibung in ein Layout für ein ASIC oder in Programmierinformationen für ein PLD umzusetzen. Der Einsatz dieser Werkzeuge wird im Englischen häufig als „Electronic Design Automation“ (EDA) bezeichnet [30]. Das Ziel dabei ist es, die Produktivität zu erhöhen, indem man nicht mehr die Logikgatter von Hand verschaltet, sondern die Funktion der Schaltung auf einer höheren *Abstraktionsebene* mit einer HDL beschreibt und eine Realisierung der Schaltung in einem ASIC oder PLD durch Anwendung der EDA-Werkzeuge gewinnt. Diese Vorgehensweise kann mit der Softwareentwicklung verglichen werden: Dort wird ebenfalls die Software mit einer höheren Programmiersprache wie C oder Java beschrieben und man gewinnt den Maschinencode, den der Prozessor ausführen soll, durch Anwenden von Werkzeugen wie Compiler und Assembler.

Um die Entwicklung von digitalen integrierten Schaltungen am Beispiel von FPGAs mit HDLs und EDA-Werkzeugen soll es im vorliegenden Buch gehen. Bevor wir in den folgenden Kapiteln in die Details gehen, möchten wir im anschließenden Abschnitt zunächst eine Übersicht über die Abstraktionsebenen und die dazugehörigen EDA-Werkzeuge geben. Für denjenigen, der sich zum ersten Mal mit HDLs und EDA-Werkzeugen beschäftigt, mag die Vorgehensweise – insbesondere die Transformationen der Schaltung zwischen den Abstraktionsebenen – zunächst tatsächlich sehr „abstrakt“ erscheinen. Der Einsatz von HDLs und EDA birgt viele potentielle Fehlerquellen in sich, die HDLs für einen Anfänger auch zu einem frustrierenden Erlebnis machen können. HDLs und EDA erfordern für einen erfolgreichen Einsatz daher unbedingt eine methodische Vorgehensweise. Es muss an dieser Stelle aber auch betont werden, dass der Einsatz von HDLs und EDA-Werkzeugen sich über die letzten zehn Jahre zu einem produktiven Standardentwurfsverfahren in der Industrie entwickelt hat, so dass heute ein Entwickler von digitaler Hardware die Beschäftigung mit diesen Entwurfsverfahren nicht vermeiden kann. Das vorliegende Buch möchte insbesondere durch die Beschreibung von erprobten Methodiken dem Lernenden helfen, den Einstieg in die Entwicklung von digitalen integrierten Schaltungen mit HDLs zu einem erfolgreichen und motivierenden Erlebnis zu machen.

1.2 Abstraktionsebenen und EDA-Werkzeuge

Die Vorgehensweise beim Entwurf einer integrierten Digitalschaltung und die verschiedenen Abstraktionsebenen lassen sich am besten mit Hilfe des so genannten „Y-Diagramms“ nach Gajski [15, 88] in Abbildung 1.4 beschreiben. In diesem Diagramm zeigen die Arme die drei möglichen Darstellungsformen oder *Sichtweisen* eines Designs: Struktur, Verhalten, Geometrie bzw. physikalische Implementierung. Die konzentrischen Kreise zeigen die verschiedenen Abstraktionsebenen eines Designs. Je näher eine Abstraktionsebene zum Zentrum ist, desto detaillierter ist die Beschreibung des Designs auf dieser Ebene.

Ein *Modell* einer Schaltung beschreibt das Verhalten und die Struktur der Schaltung auf einer bestimmten Abstraktionsebene und dient hauptsächlich auch zur *Simulation*: Hierbei wird ein *Simulationsmodell* des Designs, also eine Nachbildung der realen Schaltung durch Software, auf einem Entwicklungsrechner ausgeführt, so dass man die Funktion

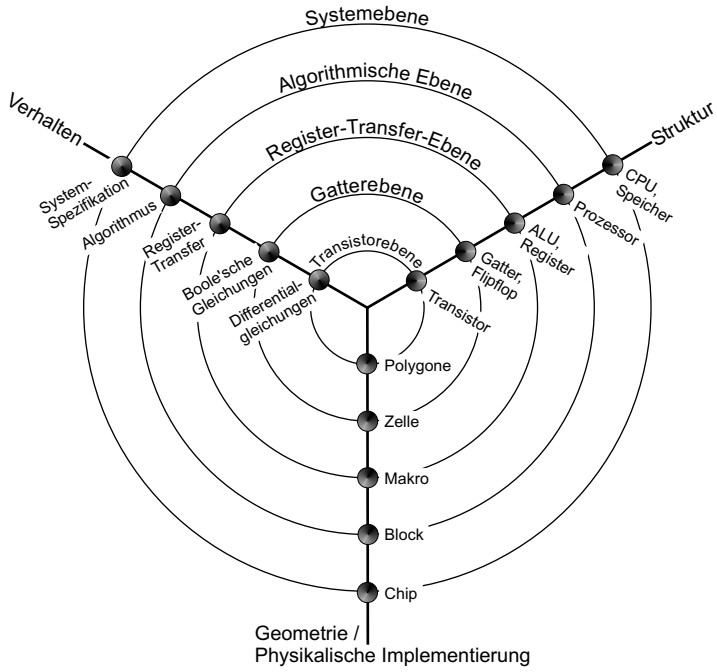

Abb. 1.4: *Y-Diagramm (engl.: Y-Chart) nach Gajski: Der Geometrie-Arm beschreibt hauptsächlich die Aspekte der physikalischen Realisierung eines ASICs, d. h. der Layouterstellung. Da diese Aspekte bei der PLD-Entwicklung im Prinzip keine Rolle spielen, werden wir die Geometrie im Folgenden vernachlässigen. Weitere Informationen hierzu können beispielsweise [30] entnommen werden. Am Struktur-Arm sind jeweils Beispiele für mögliche Komponenten auf den einzelnen Abstraktionsebenen angegeben. Diese sind beispielhaft zu verstehen. Am Verhaltens-Arm sind die typischen Beschreibungsformen angegeben. Die Ebenen Register-Transfer und Gatter werden in den folgenden Kapiteln noch sehr viel detaillierter und anhand von zahlreichen Beispielen erklärt; wir fassen uns daher in diesem Abschnitt relativ kurz.*

des Designs überprüfen und gegebenenfalls Fehler korrigieren kann. Ohne Simulation müsste man viele Prototypen herstellen und diese so lange verbessern, bis das gewünschte Verhalten erreicht ist. Während Simulationen des realen Geschehens mittlerweile auch verstärkt in anderen technischen Disziplinen benutzt werden, waren sie in der integrierten Schaltungstechnik schon recht früh ein Standardentwurfswerkzeug, da die schnell ansteigende Komplexität der Schaltungen eine Entwicklung von Schaltungen ohne Simulation unmöglich machten.

Das (Simulations-)Modell einer integrierten Schaltung besteht aus der Verschaltung von Komponenten – dies ist die Struktur oder Netzliste der Schaltung – und aus den Verhaltensmodellen der Komponenten. Der Detaillierungsgrad der Verhaltensmodelle hängt von der Abstraktionsebene ab. Die tiefste Ebene in Abbildung 1.4 wird üblicherweise als *Transistor-* oder *Schaltkreisebene* bezeichnet. Die *strukturelle Beschreibung* (Struktur-Arm) des Designs auf dieser Ebene besteht in der Verschaltung von entsprechenden Komponenten, wie Transistoren, Kapazitäten, Induktivitäten, Widerständen oder Dioden. Das *Verhalten* dieser Bauelemente kann durch Differentialgleichungen beschrieben werden, welche von einem Simulationswerkzeug (Netzwerk- oder Schaltkreissimulator) gelöst werden müssen, um die Schaltung auf dieser Abstraktionsebene simulieren zu können. Während in den Anfängen der integrierten Schaltungstechnik ein Design aus einer relativ geringen Anzahl von Transistoren bestand, umfasst ein modernes Design einige Millionen von Transistoren. Um eine solch große Netzliste mit einem Schaltkreissimulator simulieren zu können, wären sehr leistungsfähige Computer und sehr viel Rechenzeit erforderlich. Die Simulation eines kompletten Designs auf Transistorebene wird daher in der Entwicklung von digitalen Schaltungen nicht mehr durchgeführt.

Um die enorme Komplexität heutiger Schaltungen im Entwurfsprozess handhaben zu können, verfolgt man zwei Prinzipien: Erstens wird die Schaltung in mehrere, kleinere Teilschaltungen zerlegt, welche separat von mehreren Entwicklern bearbeitet werden können. Dieses Prinzip wird als „Teile-und-Herrsche“ bezeichnet (lat.: divide et impera, engl.: divide-and-conquer). Zweitens erstellt man am Anfang ein simulierbares Modell der Schaltung auf einer möglichst hohen Abstraktionsebene. Hierzu kann man eine Programmiersprache, wie z. B. C/C^{++}, benutzen oder eine Hardwarebeschreibungssprache, wie beispielsweise VHDL. Mit steigendem Abstraktionsgrad wird das Modell immer weniger komplex: Wenige Zeilen C- oder VHDL-Code können einige hunderte bis tausende von Transistoren beschreiben. Man gewinnt durch die Abstraktion im Wesentlichen zwei Vorteile: Die Schaltungsbeschreibung kann erheblich schneller erstellt werden und zeigt auch sehr viel übersichtlicher die Funktion der Schaltung. Wer schon einmal versucht hat, anhand einer Transistorschaltung die Funktion derselben zu verstehen, kann diesen Punkt sicher gut nachvollziehen. Durch diese beiden Vorteile ist der Entwickler sehr viel schneller in der Lage, Fehler zu entdecken und diese zu verbessern. Insgesamt erhöht sich die Entwurfsproduktivität durch die Beschreibung der Schaltung auf einer höheren Abstraktionsebene erheblich.

Üblicherweise wird zu Beginn einer IC-Entwicklung eine *Spezifikation* erstellt, was auch als „Pflichtenheft“ oder „Lastenheft“ bezeichnet wird. Die Spezifikation ist *die* wesentli-

che Schnittstelle zwischen Entwickler und Kunde und beschreibt die Eigenschaften und Funktionen der Schaltung. Zumeist begnügt man sich damit, die Spezifikation schriftlich in einem Dokument festzuhalten. Das Problem mit solchen Dokumenten ist, dass sie häufig fehlerhaft, mehrdeutig oder unvollständig sind. Die Erfahrung zeigt, dass viele Probleme dadurch entstehen, dass die Spezifikationen von Entwicklern anders umgesetzt werden, als sie vom Kunden gedacht waren. Erkennt man diese Fehler erst zum Schluss einer Entwicklung, sind damit in der Regel hohe Kosten verbunden. Es empfiehlt sich daher, einerseits eine Spezifikation mit äußerster Sorgfalt aufzustellen und andererseits möglichst früh ein Simulationsmodell („ausführbare Spezifikation“) auf einer hohen Abstraktionsebene zu erstellen, das der Kunde selbst simulieren kann oder das dem Kunden vorgeführt werden kann, um mögliche Verständnisfehler zu entdecken.

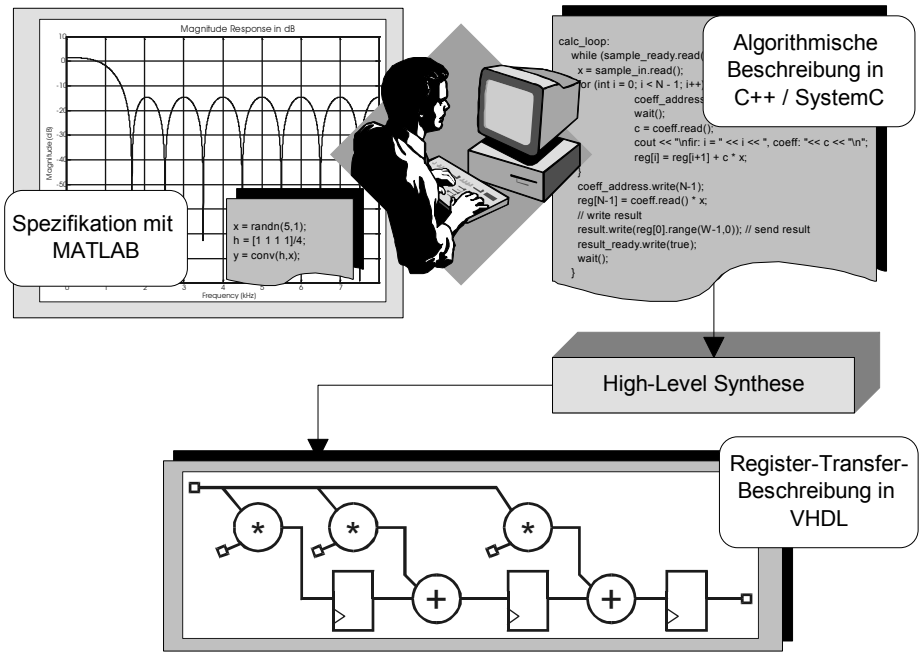

Abb. 1.5: *Entwicklung einer Schaltung von der Spezifikation bis zur Register-Transfer-Ebene am Beispiel eines digitalen Filters.*

Diese Vorgehensweise ist in Abbildung 1.5 an einem Beispiel aus der Signalverarbeitung gezeigt. Ein erstes Modell auf der Systemebene oder der algorithmischen Ebene kann durch einen Entwickler beispielsweise mit der bekannten MATLAB/Simulink-Entwicklungsumgebung der Firma MathWorks [47] erstellt werden; eine weitere sehr häufig verwendete Möglichkeit besteht in der Beschreibung des Systems oder der Schaltung mit der Programmiersprache C^{++}. Das Verhalten des Modells kann untersucht werden und es kann optimiert werden. So können beispielsweise für das Filter aus Abbildung 1.5 mit einem Filterentwurfswerkzeug in MATLAB verschiedene Realisierungsvarianten untersucht werden.

Bis vor kurzem mussten solche in MATLAB oder C^{++} erstellten Systemmodelle manuell in entsprechende RTL-Beschreibungen (RTL: Register-Transfer-Level) in VHDL oder Verilog umgesetzt werden. Häufig stellt sich dabei auch das Problem, dass die Systemmodelle von Systementwicklern beschrieben werden, die wenig Kenntnisse von VHDL und der Hardwarerealisierung haben, und die RTL-Modelle von Hardwareentwicklern beschrieben werden, die wenig Kenntnisse des Systems haben. An dieser Schnittstelle ergibt sich daher Mehrarbeit, die zu zusätzlichen Kosten führt.

Deshalb ist man daran interessiert, Systemmodelle möglichst automatisch in eine Hardwarerealisierung umsetzen zu können. Das Umsetzen von einer höheren Beschreibungsebene in eine tiefere Beschreibungsebene wird als *Transformation* bezeichnet. Für die Transformation eines Modells von der algorithmischen Ebene auf die Register-Transferebene (RT-Ebene) können so genannte „High-Level-Synthese"-Werkzeuge (HLS) eingesetzt werden, die im Englischen auch als „Behavioral Synthesis" bezeichnet werden [39, 17]. Mit HLS-Werkzeugen von Mentor Graphics [48] oder Synopsys [73] können beispielsweise algorithmische C^{++}/SystemC-Beschreibungen in RTL-Beschreibungen umgesetzt werden. SystemC ist eine Klassenbibliothek für C^{++}, die viele Konstruktionen für die Hardwaremodellierung und einen Simulationskern beinhaltet.

In der Regel verlangen HLS-Werkzeuge, dass ganzzahlige Datentypen oder *Fixpunktdatentypen* (engl.: fixed-point) verwendet werden. Da man MATLAB-Modelle für die Signalverarbeitung häufig zunächst mit *Gleitpunktdaten* (engl.: floating-point) modelliert, muss das Modell mit Fixpunktdaten mit einer bestimmten *Wortbreite* beschrieben und die daraus resultierenden Effekte untersucht werden; dies kann noch in MATLAB erfolgen. Anschließend wird ein algorithmisches Modell mit Fixpunktdaten und der festgelegten Wortbreite in C^{++} erstellt, welches dann mit der HLS in ein RTL-Modell umgesetzt werden kann. Da nun die Wortbreite festgelegt ist, werden diese algorithmischen Beschreibungen auch als „bitrichtig" bezeichnet. Um sich das manuelle Erstellen eines C^{++}-Modells zu ersparen, werden mittlerweile auch Werkzeuge angeboten, die MATLAB-Modelle direkt in RTL-Beschreibungen synthetisieren können, wie beispielsweise von AccelChip [1] oder Synplicity [74]. Xilinx [99] bietet mit dem „System Generator" IP-Cores an, mit welchen MATLAB/Simulink-Modelle erstellt werden können, die ein direktes Umsetzen der Modelle in Xilinx-FPGAs ermöglichen.

Bei der Umsetzung eines algorithmischen Modells in ein RTL-Modell wird festgelegt, mit welcher Zahl von Hardwareressourcen der Algorithmus implementiert werden soll und in welcher Anzahl von Taktschritten der Algorithmus ausgeführt wird. Im Beispiel aus Abbildung 1.5 wird das Filter mit drei Multiplizierern, zwei Addierern und drei Registern implementiert. Im Gegensatz zur algorithmischen Ebene ist in einer RTL-Realisierung die Anzahl von Taktschritten für die Abarbeitung der Funktion festgelegt, eine RTL-Beschreibung ist daher „bitrichtig" und „taktrichtig". Die HLS generiert auf RT-Ebene Rechenwerke sowie eventuell benötigte Steuerwerke und Speicher, um den Algorithmus zu implementieren. Es entsteht dabei also gewissermaßen ein „Spezialprozessor" für den zu implementierenden Algorithmus.

Da die automatische Umsetzung von der Systemebene oder der algorithmischen Ebene auf die RT-Ebene noch nicht leistungsfähig genug ist oder sich auf bestimmte Anwendungen, wie die Signalverarbeitung, beschränkt, erfolgt die Hardwaremodellierung durch einen Entwickler heute noch häufig auf RT-Ebene in VHDL oder Verilog, wie Abbildung 1.6 zeigt. Neben der Bitrichtigkeit und der Taktrichtigkeit ist eine RT-Beschreibung *technologieunabhängig* – im Gegensatz zu einer Beschreibung auf Gatterebene, welche *technologieabhängig* ist. Es werden die Register mit der entsprechenden Bitbreite modelliert und weitere Registerfunktionen wie Flanken-/Pegelsteuerung oder Set und Reset werden beschrieben. Bei den zwischen den Registern liegenden Transferfunktionen handelt es sich um kombinatorische Funktionen oder Schaltnetze ohne speicherndes Verhalten. Die Beschreibung der Register- und Transferfunktionen erfolgt mit Konstruktionen, wie sie aus Programmiersprachen bekannt sind (Auswahl, Verzweigung, Schleife: CASE, IF, LOOP). Technologieunabhängigkeit bedeutet, dass keine Gatter, Flipflops oder Makros aus speziellen ASIC- oder FPGA-Technologien verwendet werden.

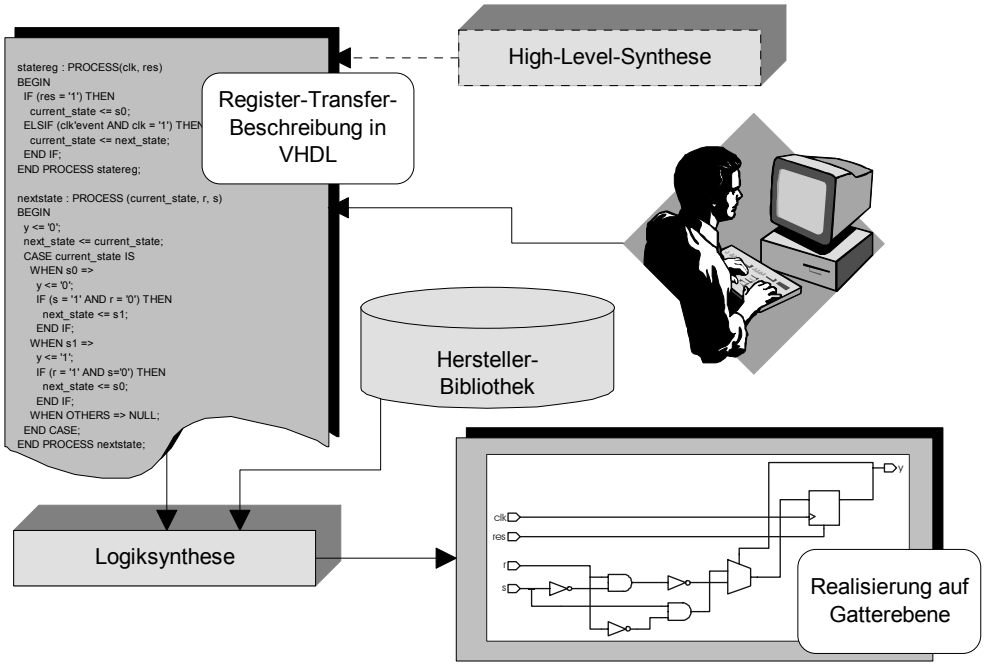

Abb. 1.6: *Entwicklung einer Schaltung von der RT-Ebene zur Gatterebene.*

Eine RTL-Beschreibung wird durch die *Logiksynthese* in eine technologieabhängige Realisierung auf Gatterebene umgesetzt, wie in Abbildung 1.6 gezeigt. Als Zieltechnologien können dabei ASIC-Technologien oder PLD/FPGA-Technologien benutzt werden. Die von der Logiksynthese erzeugte Verschaltung von Komponenten realisiert die gleiche Funktion wie das RT-Modell. Die Komponenten, wie Gatter oder Flipflops, sind in einer Bibliothek

beschrieben, die vom ASIC- oder PLD-Hersteller geliefert wird. Aus der Verhaltensbeschreibung auf RT-Ebene entsteht bei der Logiksynthese eine Strukturbeschreibung auf Gatterebene. Die einzelnen Komponenten sind wiederum in der Bibliothek durch Verhaltensmodelle auf Gatterebene beschrieben. Sie beschreiben im Wesentlichen die Boole'schen Funktionen und das Zeitverhalten der Komponenten. Letzteres ist technologieabhängig und ein wesentlicher Unterschied zur RT-Ebene: Auf der Gatterebene kann man bestimmen, welche Verzögerungszeit ein Schaltnetz zwischen zwei Registern benötigt und welche minimale Periodendauer der Takt daher nicht unterschreiten darf. Auf RT-Ebene wird diese Verzögerungszeit überhaupt nicht modelliert, da dies ein technologieabhängiges Merkmal ist. Ein RTL-Modell kann daher gewissermaßen mit einer beliebig hohen Taktrate getaktet werden.

Logiksynthesewerkzeuge werden von einer Reihe von EDA-Firmen angeboten, wie Synopsys [73], Mentor Graphics [48], Cadence [12] oder Synplicity [74], und haben über die letzten fünfzehn Jahre einen hohen Reifegrad und eine hohe Leistungsfähigkeit erreicht; Logiksynthese kann daher als Standardentwurfsverfahren für das ASIC- und PLD-Design angesehen werden. Einer der wesentlichen Vorteile der Beschreibung auf RT-Ebene besteht in der Tatsache, dass technologieunabhängige RTL-Beschreibungen via Logiksynthese auf beliebige Zieltechnologien umgesetzt werden können. So kann man zunächst einen FPGA-Prototypen entwickeln und aus dem gleichen VHDL-Code später ein ASIC realisieren.

Im Entwurf von digitalen Schaltungen wird die Transistorebene nur indirekt benutzt. Sie steckt im Grunde in den Komponentenbibliotheken, da die Hersteller das Zeitverhalten der Gatter aus Simulationen auf Transistorebene gewinnen und dieses dann in einem Verhaltensmodell der Komponenten auf Gatterebene ablegen. Abbildung 1.7 zeigt in einer Übersicht den wesentlichen Ablauf beim Entwurf einer digitalen Schaltung: Zunächst wird ein Modell der Schaltung auf algorithmischer Ebene oder Systemebene entwickelt und kann durch Simulation bezüglich seiner Korrektheit überprüft werden. Dieses Modell wird nun entweder automatisch oder manuell in eine RTL-Beschreibung beispielsweise mit VHDL umgesetzt und kann wiederum simuliert werden. Üblicherweise wird das RTL-Simulationsergebnis mit dem Ergebnis der Simulation auf der Systemebene oder der algorithmischen Ebene verglichen. Die Logiksynthese generiert aus dem RTL-Modell ein Gattermodell. Dieses wird dann mit weiteren Werkzeugen in eine physikalische Realisierung umgesetzt: Beim ASIC entsteht das Layout und bei einem PLD werden die Programmierinformationen erzeugt.

Nach der physikalischen Realisierung kann ein zweites Gattermodell der Schaltung als VHDL-Beschreibung automatisch generiert werden, wobei nun insbesondere die genauen Verzögerungszeiten (Timing-Daten, üblicherweise im SDF-Format) der Gatter bei der Simulation berücksichtigt werden können. Die Timing-Daten können auch ohne Simulation mit der so genannten „Timing-Analyse“ untersucht werden. Dieses Gattermodell ist nun „bitrichtig“, „taktrichtig“ und „verzögerungszeitrichtig“ – es zeigt das Verhalten der Schaltung für den Entwickler eines ASICs oder FPGAs mit dem maximalen Detaillierungsgrad. Wie wir im Verlauf des Buches noch sehen werden, erreichen diese Gattermodelle aber auch den maximalen Grad an Unübersichtlichkeit. Weil eine Fehlersuche auf Gatterebene

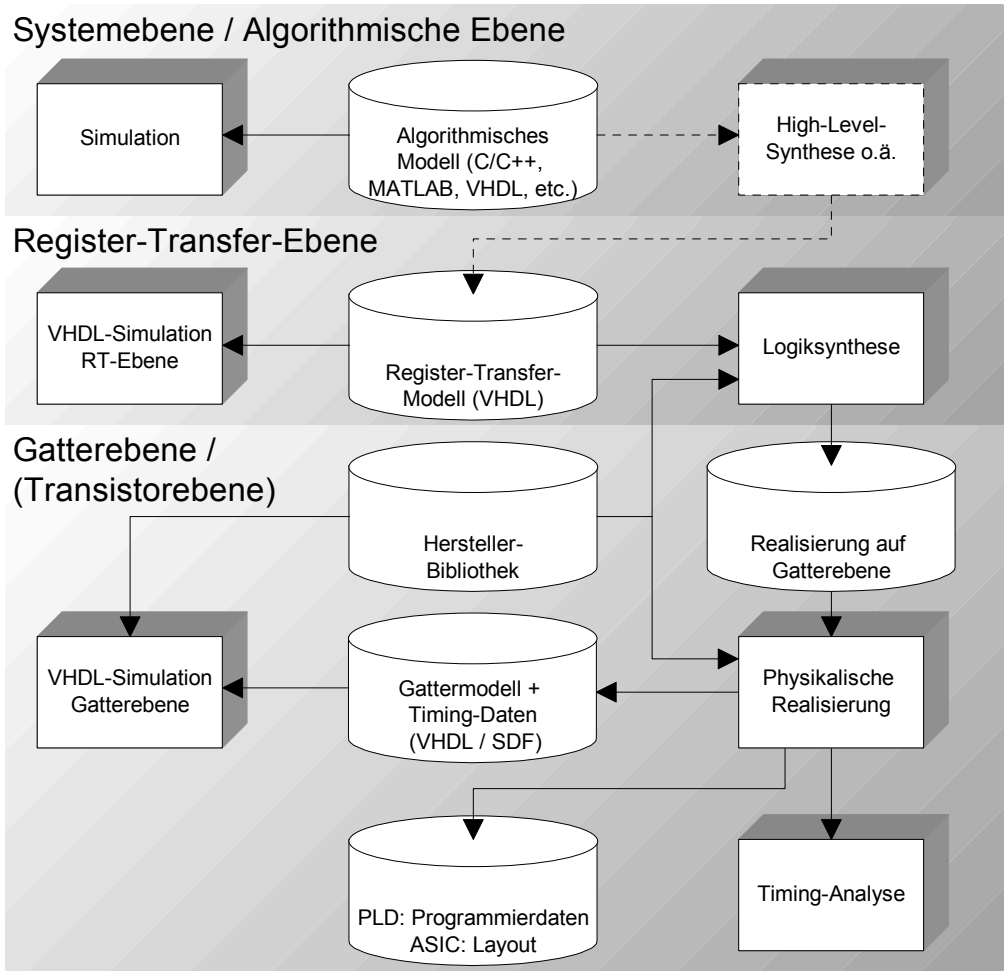

Abb. 1.7: *Entwurfsablauf*

sehr zeitaufwändig ist, ist es das Ziel, ein funktional richtiges RTL-Modell auch in eine fehlerfreie Gatterbeschreibung umzusetzen. Dies gelingt, wenn einige Grundsätze bei der Entwicklung von RTL-Modellen berücksichtigt werden und ein prinzipielles Verständnis der Logiksynthese vorhanden ist. Dies zu vermitteln ist ein hauptsächliches Anliegen des vorliegenden Buches.

Zusammenfassend besteht der Entwurf von digitalen ICs also aus folgenden Aufgaben:

- Modellierung der Schaltung auf Systemebene, algorithmischer Ebene, RT-Ebene oder Gatterebene mit Systementwicklungswerkzeugen, Programmiersprachen oder Hardwarebeschreibungssprachen
- Simulation der Modelle

- Transformationen zwischen den Abstraktionsebenen mit Hilfe von Synthesewerkzeugen
- Physikalische Realisierung

1.3 Ziele und Aufbau des Buches

Der Entwurf von digitalen integrierten Schaltungen erfordert heute Kenntnisse der Hardwarebeschreibungssprachen, Kenntnisse der verwendeten EDA-Werkzeuge und Kenntnisse der ASIC- oder PLD-Technologien, in denen die Schaltungen implementiert werden sollen. Gerade der Einsatz von HDLs verleitet oft zu der Annahme, dass es sich im Prinzip „nur“ um Softwareentwicklung handelt. Leider ist das Schreiben des HDL-Codes nur *ein* Teil der Aufgabe und ein Entwickler von digitaler Hardware muss sich intensiv auch mit den EDA-Werkzeugen und der verwendeten Technologie auseinandersetzen. Langjährige Erfahrung im Entwurf von digitalen integrierten Schaltungen lehrt, dass solche Schaltungen nur dann mit guten Ergebnissen entwickelt werden können, wenn die in Abbildung 1.8 gezeigten unterschiedlichen Aspekte des digitalen Entwurfs verstanden werden.

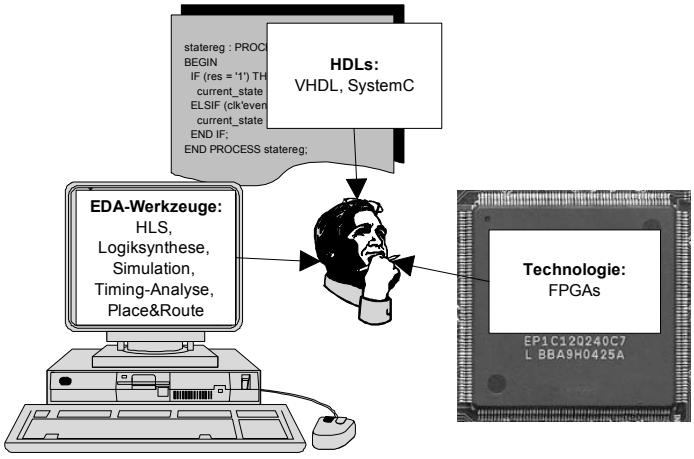

Abb. 1.8: *Aspekte des digitalen Entwurfs.*

Es gibt zwar viele, auch deutlich umfassendere, Bücher zu den einzelnen Themen HDL, EDA-Werkzeuge oder Technologie, aber nur wenige Bücher weisen eine zusammenhängende Sicht der Themengebiete auf. Zentrales Anliegen des Buches ist es daher, die Zusammenhänge und Wechselwirkungen zwischen der HDL-Beschreibung, den EDA-Werkzeugen und der Technologie in einem Lehrbuch darzustellen. Dies beinhaltet hauptsächlich folgende Themen:

- Synthesegerechter Hardwareentwurf auf Register-Transfer-Ebene mit Hardwarebeschreibungssprachen am Beispiel von VHDL.

- Aspekte der physikalischen Realisierung auf hochintegrierten CMOS-ICs am Beispiel von FPGAs.
- Verwendung von EDA-Werkzeugen für die automatisierte Realisierung von HDL-Beschreibungen auf FPGAs, wie Simulation, Synthese, Timing Analyse sowie Platzieren und Verdrahten der Schaltung auf dem FPGA.

Das Buch behandelt in erster Linie VHDL, da dies den zumindest in Europa vorherrschenden industriellen Standard darstellt. Auf eine Darstellung der Sprache Verilog, welche ebenfalls häufig verwendet wird, haben wir verzichtet, weil das gleichzeitige Erlernen von zwei HDLs nicht sinnvoll erscheint. Jedoch erleichtert das Verstehen der Prinzipien von HDLs am Beispiel von VHDL auch einen späteren Einstieg in Verilog. Als Ausblick auf neuere Trends in der Entwicklung von digitalen Schaltungen werden wir auch die Sprache SystemC vorstellen, weil dies den derzeit von vielen Firmen unterstützten Versuch darstellt, die Programmiersprache C^{++} für das Hardwaredesign zu verwenden. Wir werden hier allerdings nur eine kurze Einführung in SystemC geben und die Unterschiede zu VHDL darstellen. Ferner werden wir kurz auf die algorithmische Beschreibung und die High-Level-Synthese als Ausblick auf Entwurfsverfahren auf höherem Abstraktionsniveau eingehen.

Das Buch behandelt im Wesentlichen die Modellierung von Schaltungen auf Register-Transfer-Ebene, da digitale Schaltungen heute in der Industrie hauptsächlich auf dieser Ebene entwickelt werden. Bei der Darstellung von VHDL werden nur die Konstruktionen vorgestellt, die zum Hardwareentwurf benötigt werden und auch von Logiksynthesewerkzeugen tatsächlich automatisch umgesetzt werden können. Neben der Logiksynthese und der Simulation von VHDL-Modellen werden wir auch insbesondere die Problematik des physikalischen Entwurfs von FPGAs behandeln. Dieser besteht hauptsächlich aus der Platzierung und Verdrahtung der Schaltung, der Analyse des Zeitverhaltens mit Hilfe der Timing-Analyse und der Simulation des Zeitverhaltens. Als Realisierungsmöglichkeit haben wir uns für FPGAs entschieden, da sie aus verschiedenen Gründen derzeit bei kleinen und großen Firmen für die Hardwarerealisierung in ständig wachsendem Maße eingesetzt werden. Aufgrund der mehrfachen Reprogrammierbarkeit bieten sie insbesondere für die Ausbildung im Hardwaredesign erhebliche Kostenvorteile gegenüber maskenprogrammierten ASICs und es lassen sich trotzdem die wesentlichen Implementierungsprobleme aufzeigen, wie sie auch bei ASICs auftreten. Von den FPGA-Herstellern werden für die Ausbildung häufig so genannte „Starter-Kits“ mit FPGAs und EDA-Werkzeugen preisgünstig angeboten. Daher sollte das Buch nicht nur gelesen werden, sondern auch als Anregung für eigene Experimente mit VHDL und FPGAs dienen. Wie auch eine Programmiersprache, so kann man eine HDL nur durch die Anwendung richtig lernen. Als Demonstrationsvehikel benutzen wir im Buch – stellvertretend für die wichtige Klasse der SRAM-FPGAs – hauptsächlich die Virtex-II-FPGAs von Xilinx.

Das Buch wendet sich als Lehrbuch in erster Linie an Studierende der Elektrotechnik/Informationstechnik oder der technischen Informatik im Hauptstudium, da die Entwicklung von digitalen Schaltungen mit HDLs heute häufig Bestandteil von Lehrplänen an Fachhoch-

schulen und Universitäten ist; es ist aber ebenso geeignet für den Praktiker in der Industrie, der den Einstieg in die Entwicklung von FPGAs mit HDLs sucht. Es sollten Grundkenntnisse in digitaler Schaltungstechnik und im Programmieren in C/C^{++} vorhanden sein. Für das Verständnis der technologischen Zusammenhänge sind zwar Grundkenntnisse der Halbleiterphysik sinnvoll, wir werden jedoch die für das weitere Verständnis wesentlichen Fakten der MOS-Technologie zusammenfassend darstellen. Aufgrund der zusammenhängenden Darstellung von HDLs, EDA-Werkzeugen und technologischen Aspekten ist eine tiefergehende Darstellung von einzelnen Themengebieten nicht möglich. Dies betrifft insbesondere die EDA-Werkzeuge: Eine genaue Beschreibung der in den EDA-Werkzeugen verwendeten Algorithmen, beispielsweise der Logiksynthese oder des physikalischen Entwurfs, bleibt spezieller Literatur vorbehalten und erscheint aus unserer Sicht auch nur für denjenigen sinnvoll, der an der Entwicklung von EDA-Werkzeugen interessiert ist; dies ist thematisch eher in einem Informatik-Studium zu finden. Wir begreifen die EDA-Werkzeuge in diesem Sinne tatsächlich als „Werkzeuge“ und beschränken uns auf die Erläuterung der prinzipiellen Funktionsweise und ihrer Wirkungsweise auf die zu entwickelnde Schaltung. Für den interessierten Leser haben wir für ein weitergehendes Studium einzelner Themengebiete auf entsprechende Literatur verwiesen. Wir geben im Folgenden eine kurze Übersicht über die nachfolgenden Kapitel des Buches.

Das zweite Kapitel führt in die Sprache VHDL ein und konzentriert sich auf wesentliche Sprachkonstruktionen, die für eine synthesefähige Beschreibung von digitaler Hardware notwendig sind, sowie auf wesentliche Eigenschaften der Sprache im Hinblick auf Simulation und Synthese. Da man eine neue Sprache am besten anhand von Beispielen lernt, werden die Sprachkonstruktionen und damit auch die Syntax von VHDL hauptsächlich anhand von Beispielen eingeführt.

Im dritten Kapitel wird zunächst eine Übersicht und Klassifizierung der heute verwendeten digitalen integrierten Schaltungen gegeben; diese beruhen hauptsächlich auf Transistorschaltungen in CMOS-Technologie, so dass für das Verständnis der Vorgänge auf den ICs eine Erläuterung der Eigenschaften der MOS-Feldeffekttransistoren und der daraus aufgebauten digitalen Schaltungen wichtig ist. Ausgehend von einer Beschreibung der MOS-Speichertechnologien werden dann die Programmierungstechnologien und Architekturen von PLDs und FPGAs erläutert.

Im vierten Kapitel wird zunächst die Funktionsweise der Logiksynthese und der Timing-Analyse übersichtsmäßig erläutert. In den darauf folgenden Abschnitten wird anhand eines Beispiel-Prozessors die RTL-Beschreibung und Synthese von typischen Komponenten, wie sie häufig in digitalen Entwürfen auftreten, dargestellt. Behandelt werden hierbei Schaltwerke und Zähler, arithmetische Einheiten, Speicherblöcke, Bussysteme und I/O-Blöcke. Abschließend werden einige Gesichtspunkte für effizientes RTL-Design erläutert.

Das fünfte Kapitel beschäftigt sich mit dem physikalischen Entwurf von FPGAs. Die Platzierung und Verdrahtung der Schaltung im FPGA wird anhand von Beispielen dargestellt. Die Auswirkungen der Verdrahtung im FPGA auf das Zeitverhalten der Schaltung werden anschließend untersucht. Im Zusammenhang mit der Verdrahtung wird auch die Problema-

tik der Taktverteilung im FPGA und die Sicherstellung der Synchronität besprochen. Abschließend geht das Kapitel noch auf die Simulation des Zeitverhaltens mit einem VHDL-Modell der Schaltung auf Gatterebene ein.

Das sechste und letzte Kapitel geht als Ausblick auf die Sprache SystemC zur Hardwaremodellierung ein und stellt die Funktionsweise der „High-Level Synthese“ in einer Übersicht dar. Ausgangspunkt ist hier die Beschreibung der Schaltung auf algorithmischer Ebene mit SystemC.

Das zweite Kapitel ist eine Einführung in die Grundlagen von VHDL und kann daher als Basis für eine entsprechende einführende Lehrveranstaltung dienen. Das dritte Kapitel legt ebenfalls wesentliche Grundlagen der Mikroelektronik sowie der digitalen CMOS-Technik und ihrer Anwendung in Speicherschaltungen und PLDs. Das vierte und fünfte Kapitel führen die beiden einführenden Kapitel weiter, indem der Entwurf von Schaltungen und Systemen mit VHDL auf FPGAs und die damit verbundenen Probleme detailliert beschrieben werden. Diese beiden Kapitel können die Basis für eine vertiefende Lehrveranstaltung sein, wobei das sechste Kapitel der Abrundung und dem Ausblick dienen kann. Die vielen Beispiele, insbesondere der im vierten Kapitel beschriebene Mikroprozessor, können für begleitende Laborübungen benutzt werden. Wir haben bewusst darauf verzichtet, die Bedienung der von uns für das Buch verwendeten EDA-Werkzeuge zu erläutern und uns auf die Erläuterung der prinzipiellen Funktionsweise beschränkt. Da die EDA-Werkzeuge und die an die Benutzer ausgelieferten Versionen sich sehr schnell verändern, wäre ein „Bedienungshandbuch“ bald veraltet. Wir gehen davon aus, dass der Leser sich anhand der von den EDA-Herstellern gelieferten Dokumentation selbst in die Werkzeuge einarbeitet oder dies im Rahmen von Laborübungen stattfindet.

2 Modellierung von digitalen Schaltungen mit VHDL

Dieses Kapitel führt in die Sprache VHDL ein. Da VHDL seit mehr als zehn Jahren in der Entwicklung von digitalen Schaltungen eingesetzt wird, liegen eine Reihe von Büchern zu diesem Thema vor [8, 13, 14, 24, 59, 84, 77]. Dieses Kapitel erhebt daher nicht den Anspruch auf eine vollständige und umfassende Darstellung der Sprache, wie sie beispielsweise [8] entnommen werden kann. Eine Eigenschaft von vielen Hardwarebeschreibungssprachen ist es, dass nur eine Untermenge der Sprachkonstruktionen auch in Hardware durch ein Synthesewerkzeug umgesetzt werden kann. Wenn von Synthesefähigkeit in diesem Kapitel gesprochen wird, so ist damit die Logiksynthese gemeint; die Besonderheiten der High-Level-Synthese werden in einem späteren Kapitel am Beispiel von SystemC dargestellt. Dieses Kapitel konzentriert sich auf wesentliche Sprachkonstruktionen, die für eine synthesefähige Beschreibung von digitaler Hardware notwendig sind, und auf wesentliche Eigenschaften der Sprache im Hinblick auf Simulation und Synthese. Da man eine neue Programmiersprache am besten anhand von Beispielen lernt, werden die Sprachkonstruktionen – und damit auch die Syntax von VHDL – hauptsächlich anhand von Beispielen eingeführt. Auf die formale Syntaxdarstellung wird nur an einigen Stellen eingegangen; eine Übersicht über die Syntax findet sich im Anhang. Nachteilig an dieser Vorgehensweise ist, dass der komplette Sprachumfang nicht dargestellt werden kann. Von Vorteil für den Lernenden ist jedoch der schnellere Zugang zur Sprache VHDL über die Beispiele und die vergleichende Darstellung der aus der VHDL-Beschreibung resultierenden digitalen Hardware. Ferner werden wir einige Konstruktionen in den ersten Beispielen erst in späteren Abschnitten genauer erläutern. Dies entspricht nicht der klassischen pädagogischen Vorgehensweise, zunächst alle erforderlichen Grundlagen zu erläutern, führt jedoch erfahrungsgemäß beim Lernenden zu einer höheren Motivation, sich mit der Materie zu beschäftigen.

Aus Platzgründen wird auch nicht im größeren Umfang auf die Erstellung der Testumgebung, der so genannten *Testbench*, eingegangen, welche für die Simulation der Schaltung benötigt wird und die Systemumgebung der Schaltung in VHDL modelliert. Testbench ist ein Begriff aus dem Englischen und kann anschaulich mit einem *Prüfstand* für die Schaltung verglichen werden. Gleichwohl muss bemerkt werden, dass eine gute Testbench sehr wichtig ist, um das Verhalten der Schaltung möglichst umfassend simulieren und damit die Funktion verifizieren zu können. Mit der steigenden Funktionskomplexität der Schaltungen steigt auch der Aufwand für die Entwicklung der Testbenches, um möglichst alle Funktionen einer Schaltung verifizieren zu können. Testbenches dienen nur der Simulation

und müssen nicht in Hardware umgesetzt werden. Somit kann der komplette Sprachumfang für die Modellierung der Testbench eingesetzt werden. Eine Literaturquelle, die sich mit dem Entwickeln von Testbenches befasst, findet sich beispielsweise in [9].

2.1 Historische Entwicklung von VHDL

Das Akronym *VHDL* steht für *VHSIC Hardware Description Language*, wobei *VHSIC* selbst wiederum ein Akronym ist und *Very High Speed Integrated Circuit* bedeutet. Der Begriff *Hardware Description Language* (HDL) kann übersetzt werden mit Hardwarebeschreibungssprache. VHDL entstand in den achtziger Jahren aus dem VHSIC-Programm des amerikanischen Verteidigungsministeriums (engl.: Department of Defense, DoD) heraus. Das 1980 gestartete VHSIC-Programm hatte zum Ziel, die Entwicklungszeiten von Hardware zu verkürzen. Das DoD hatte insbesondere das Problem, dass der Ersatz einer elektronischen Schaltung sehr teuer war, da die Schaltungen nur unzureichend dokumentiert waren (so genannte „Hardware Lifecycle Crisis“). Die Schaltungen kamen von einer Reihe von Zulieferern und jeder Zulieferer hatte andere EDA-Werkzeuge und andere Methoden und Sprachen, die zum Teil firmenspezifisch und inkompatibel waren, um die Schaltungen zu entwickeln. Das DoD wollte eine einheitliche Sprache zur Beschreibung oder Dokumentation der Hardware, wobei man die Beschreibungen dann auch *simulieren* können sollte. Die Ziele von VHSIC/VHDL wurden 1981 in einem Workshop in Woods Hole in Massachusetts unter Beteiligung von Vertretern der Regierung, der Industrie und der Hochschulen diskutiert. Die Firmen Intermetrics, IBM und Texas Instruments bekamen 1983 den Auftrag die Sprache VHDL zu entwickeln. Das Ziel war, eine genormte Beschreibungssprache für die Funktion und Struktur von komplexen Schaltungen zu entwickeln. Die Sprache sollte an die Programmiersprache *ADA* angelehnt sein. ADA wurde ebenfalls für das DoD entwickelt und 1983 als *Ada 83* von *ANSI* (American National Standards Institute) standardisiert. Der Name „Ada“ leitet sich übrigens von Lady Ada Lovelace ab – die Mitarbeiterin von Charles Babbage –, welche als erste „Programmiererin“ betrachtet werden kann. Ada 83 [6] ist eine „objekt-basierte“ Sprache, da wesentliche Merkmale „objektorientierter“ Sprachen wie *Vererbung* und *Polymorphie* nicht implementiert wurden, siehe z. B. C^{++} [72]. Erst die neuere Version *Ada 95* implementiert objektorientierte Konzepte. Der Begriff des Objektes kommt daher auch in VHDL vor, wobei VHDL ebenfalls nicht objektorientiert sondern wie Ada 83 nur objekt-basiert ist. Auch VHDL implementiert keine Vererbung und Polymorphie. Im Jahre 1985 wurde eine erste VHDL-Version 7.2 veröffentlicht.

Durch die Mitarbeit und das Interesse von weiteren Firmen, insbesondere Herstellern von EDA-Werkzeugen, bekam VHDL einen weiteren Impuls. 1987 wurde die Sprache vom amerikanischen *IEEE* (Institute of Electrical and Electronics Engineers) standardisiert unter der Bezeichnung *IEEE 1076-1987*. Im Jahre 1988 wurde VHDL auch als ANSI-Standard veröffentlicht. Seit 1988 müssen alle Zulieferer des DoD ihre Komponenten in VHDL dokumentieren. Der IEEE-Standard wurde 1993 überarbeitet unter der Bezeichnung *IEEE 1076-1993*, wobei einige Erweiterungen hinzu kamen. Dies ist der derzeit gültige Standard;

er ist im so genannten *Language Reference Manual* (LRM) dokumentiert, das vom IEEE bezogen werden kann [32]. Die Beispiele im vorliegenden Buch beziehen sich auf den *IEEE 1076-1993*-Standard. Der anfängliche Fokus von VHDL war allerdings nur die Beschreibung von Schaltungen und Systemen und die Simulation dieser Beschreibungen. Es lässt sich damit auch vieles beschreiben, was man nicht in digitale Hardware umsetzen kann. Die automatische Umsetzung von HDL-Beschreibungen auf Register-Transfer-Ebene in eine Gatterrealisierung wurde zu Ende der achtziger Jahre zunehmend interessanter. In der Folge wurden daher so genannte „Synthesewerkzeuge“ entwickelt, insbesondere von der Firma Synopsys, die diese automatisierte Umsetzung ermöglichen. Die (Logik-) Synthesewerkzeuge können allerdings nur eine Untermenge aller möglichen VHDL-Konstruktionen in Hardware umsetzen. Diese Untermenge, die zunächst von den Werkzeugherstellern – zum Teil auch unterschiedlich – definiert wurde, wurde 1999 in den Zusatz *IEEE 1076.6* des VHDL-1076-Standards gefasst. Erarbeitet wurde dieser Standard von der so genannten „VHDL Synthesis Interoperability Working Group“, in welcher namhafte Firmen wie Mentor Graphics, IBM, Cadence und Synopsys vertreten sind. Schreibt ein Entwickler ein VHDL-Modell nach diesem Standard, dann wird auch die Synthesefähigkeit auf Synthesewerkzeugen, die ebenfalls den Standard unterstützen, garantiert. Für die Beschreibung synthesefähiger Hardware ist insbesondere der Standard *IEEE 1164* wichtig, welcher Datentypen für eine *mehrwertige* Logik und zugehörige Operatoren und Funktionen definiert; dieser wird in der Regel als einbindbares VHDL-*Package* mit den Entwurfswerkzeugen mitgeliefert. Ebenso ist ein weiterer Standard *IEEE 1076.3* für die Synthese definiert worden, in dem Datentypen und Funktionen für vorzeichenlose und vorzeichenbehaftete ganzzahlige Arithmetik beschrieben sind. Für die Simulation des Zeitverhaltens auf Gatterebene ist der Standard *IEEE 1076.4* wichtig, welcher auch als *VITAL*-Standard bekannt ist (engl.: *V*HDL *I*nitiative *T*owards *A*SIC *L*ibraries). Die meisten PLD- oder ASIC-Hersteller bieten ihre Bibliotheken als VHDL-Modelle nach dem VITAL-Standard an, so dass eine genaue Simulation des Zeitverhalten auf Gatterebene mit einem VHDL-Simulator möglich ist. Weitere Arbeitsgruppen arbeiten an anderen Erweiterungen des IEEE 1076-Standards, so wird beispielsweise unter IEEE 1076.1 eine Erweiterung für die Beschreibung von analogen Schaltungen erarbeitet (VHDL-AMS) oder unter IEEE 1076.2 an einem mathematischen Package für Gleitpunktzahlen und komplexe Zahlen; bei beiden Arbeitsgruppen geht es allerdings nur um die Simulation, nicht um Synthesefähigkeit. Nach zwei kleineren Revisionen in den Jahren 2000 und 2002 wird derzeit an einer umfangreicheren Überarbeitung der VHDL-Standards unter dem Namen VHDL-200X gearbeitet. Unter anderem soll hierbei auch ein synthesefähiger Standard für Gleitpunktzahlen definiert werden, was ein Novum darstellen würde, da bislang nur die Synthese von ganzzahligen Datentypen möglich ist.

Neben VHDL existieren einige andere HDLs, die aber vielfach firmenspezifische Lösungen sind. Zu nennen wären hier beispielsweise die Sprache ABEL (Advanced Boolean Expression Language), die hauptsächlich zur Programmierung von PLDs eingesetzt wurde, oder die Sprache AHDL (Altera Hardware Description Language) der Firma Altera. Diese Sprachen weisen jedoch im Vergleich zu VHDL zumeist einen geringeren Funktionsumfang aus.

Die Sprache *Verilog* (*Veri*fying *Log*ic), die eine ähnliche große Bedeutung und Verbreitung wie VHDL erfahren hat, war zunächst auch eine proprietäre Sprache. Verilog wurde 1984 und 1985 von der Firma *Gateway Design Automation* entwickelt, wobei die Firma auch gleichzeitig einen Simulator für Verilog entwickelte. Dieser Simulator wurde durch den „XL-Algorithmus“ (Verilog-XL) bekannt, welcher zu einer schnellen Simulation der Schaltungsbeschreibungen auf Gatterebene führte. Wie bei VHDL konnte auch bei Verilog eine Schaltung auf der Gatterebene beschrieben werden, aber auch in Form von Beschreibungen auf Register-Transfer-Ebene. Dies führte hauptsächlich in den USA ab 1986 dazu, dass viele Entwickler ihre Schaltungen mit Verilog beschrieben. 1988 lieferte die Firma *Synopsys* das erste Logiksynthesewerkzeug aus, welches Verilog als Eingabesprache verwendete und eine RTL-Beschreibung in eine Gatterrealisierung überführen konnte. Dieser so genannte „Design Compiler“, der auch heute noch von vielen ASIC-Entwicklern verwendet wird, führte zu einem weiteren Schub für Verilog.

Ab 1989 verwendeten viele ASIC-Anbieter, also Firmen, die aus einer vom Kunden gelieferten Netzliste auf Gatterebene eine applikationsspezifische integrierte Schaltung herstellten, den Verilog-XL-Simulator als so genannten „Sign-Off-Simulator“, mit dem die Funktion und das Zeitverhalten der Schaltung auf Gatterebene simuliert wurde. Der englische Begriff „Sign-Off“ bedeutet dabei soviel wie *Abnahme*: Wenn der Kunde diesen Simulator und die *Simulationsmodelle* des ASIC-Herstellers für die Gatter verwendet, dann garantiert der ASIC-Hersteller, dass die Funktion und das Zeitverhalten des gefertigten ASICs den Simulationen entspricht.

1989 wurde Gateway von der Firma *Cadence Design Systems* aufgekauft, die noch heute zu den Marktführern im EDA-Bereich zählt. Doch noch immer war Verilog eine proprietäre Lösung und nur Cadence durfte einen Verilog-Simulator verkaufen. Andere EDA-Hersteller setzten daher stärker auf VHDL, da dies ein offener Standard war. Dies ist vielleicht vergleichbar mit der Situation bei den PC-Betriebssystemen zwischen *Microsoft Windows* und den offenen *Linux*-Systemen. Cadence erkannte das Problem und rief die „Open Verilog Initiative“ (OVI) ins Leben, mit dem Ziel auch aus Verilog einen offenen IEEE-Standard zu machen. Die „IEEE 1364“-Arbeitsgruppe wurde gegründet und führte 1995 zur IEEE-Standardisierung von Verilog als *IEEE 1364-1995*. In der Folge konnten andere Firmen Simulatoren für Verilog entwickeln. Die meisten relevanten HDL-Simulatoren heutzutage, wie beispielsweise „Modelsim“ von der Firma *Model Technology*, können sowohl VHDL als auch Verilog verarbeiten. Neben den Dokumenten vom IEEE zum Verilog-Standard [34] findet sich beispielsweise in [92] eine Beschreibung der Sprache.

Verilog bietet vergleichbare Konstruktionen wie VHDL an, beispielsweise für die Beschreibung von Schleifen und Auswahl. Diese Konstruktionen finden sich ebenso in Programmiersprachen für Software, so dass Softwareentwickler HDLs wie Verilog oder VHDL schnell erlernen können. VHDL und Verilog sind heute die beiden beherrschenden Hardwarebeschreibungssprachen. Während Entwickler in den USA eher in Verilog entwickeln, wird in europäischen Firmen hauptsächlich auf VHDL gesetzt. Häufig werden beim Entwurf einer Schaltung beide Sprachen verwendet, beispielsweise wenn bei global arbeitenden Firmen Blöcke von verschiedenen Entwicklungsgruppen in einen Entwurf einfließen.

VHDL besitzt gegenüber Verilog ein fortschrittlicheres *Datentypensystem* und implementiert auch eine sehr strenge *Typprüfung*. Des Weiteren bietet VHDL mehr Konstruktionen an, um große Systeme zu strukturieren. Während Verilog eher an der Programmiersprache C [87] orientiert ist, ist VHDL eine sehr *wortreiche* Sprache, da die Programmiersprache ADA die Grundlage für VHDL ist. Der Benutzer muss daher in VHDL für die gleiche Funktion etwas mehr schreiben; auf der anderen Seite führt dies aber auch dazu, dass VHDL-Beschreibungen gut verständlich sind.

2.2 Grundlegende Konzepte von VHDL

Dieser Abschnitt wird anhand von Beispielen in grundlegende Konzepte von VHDL einführen. Viele Schwierigkeiten und daraus resultierende Frustrationen bei einem VHDL-Einsteiger ergeben sich zumeist daraus, dass diese wesentlichen Konzepte nicht oder nur unvollständig verstanden wurden. Häufig ergeben sich diese Probleme, wenn man zunächst eine Programmiersprache kennengelernt hat, die keine *Nebenläufigkeit* implementiert, wie beispielsweise C [87]. In solchen Programmiersprachen geht man von einem sequentiellen Programmiermodell aus: Der Code wird sequentiell in der Reihenfolge abgearbeitet, die der Entwickler durch das Schreiben des Codes vorgibt. Dies ist in VHDL anders, da VHDL keine Sprache zur Softwareentwicklung sondern zur Hardwaremodellierung ist. Eines der wesentlichen Modellierungselemente in VHDL ist ein *Prozess*. Alle Prozesse sind nebenläufig bezüglich einer *Modellzeit*, wobei die Modellzeit den tatsächlichen zeitlichen Verlauf der *Signale* später in der Hardware nachbildet. Nur der Code *innerhalb* eines Prozesses wird sequentiell ausgeführt, aber alle Prozesse sind nebenläufig; das heißt die Reihenfolge der Prozesse innerhalb einer *Architecture* ist beliebig und führt in Simulation und Synthese zum gleichen Ergebnis. Dies und einige weitere wichtige Konzepte, die für die *Beschreibung von Hardware* wichtig sind, sollen in den nachfolgenden Unterabschnitten dargestellt werden.

2.2.1 Entity und Architecture

Die enorme Komplexität der heute entwickelten digitalen Schaltungen von einigen Millionen Gatterfunktionen macht eine *hierarchische Aufteilung* der Schaltung notwendig; sie besteht daher in der Regel aus einer Vielzahl von Komponenten, die zur Gesamtschaltung verbunden werden. Damit eine Komponente auf der nächsten *Hierarchieebene* verschaltet werden kann, müssen Anschlüsse vorhanden sein. Diese Anschlüsse können Eingangssignale (engl.: input) oder Ausgangssignale (engl.: output) oder bidirektionale Signale sein, die in beide Richtungen betrieben werden können (engl.: bidirectional). Die Anschlüsse einer Schaltung oder die so genannte *Schnittstelle* werden in VHDL über die „Entity“ beschrieben. Abbildung 2.1 zeigt das Symbol eines Beispiels. Der gewohnte Weg für einen Hardwareentwickler ist, die Symbole von Komponenten aus einer Bibliothek mit Hilfe eines graphischen „Schema-Editors“ (engl.: schematic editor) zu verschalten. Auch für VHDL existieren von verschiedenen Herstellern Schema-Editoren. Nach der graphischen

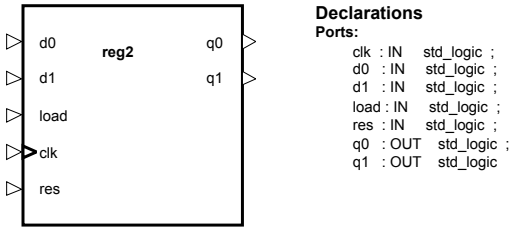

Abb. 2.1: *Symbol eines 2-Bit-Registers*

Eingabe der Schaltung können diese Werkzeuge eine *Netzliste* der Schaltung im VHDL-Format erzeugen, eine so genannte *Strukturbeschreibung*.

Die Entity in der einfachsten Form ist zunächst nichts weiter als eine textuelle Beschreibung eines solchen Symbols. Der zugehörige VHDL-Quellcode zum Symbol aus Abbildung 2.1 ist in Listing 2.1 gezeigt. Der Name der Komponente in Abbildung 2.1 ist im Beispiel `reg2` und dies ist auch der Name der Entity. Die Deklaration der Entity wird im Quelltext durch das Schlüsselwort `ENTITY` eingeleitet und mit einem eindeutigen Namen bezeichnet. Mit `END` (Zeile 13) wird die Entity-Deklaration abgeschlossen. Rechts vom Symbol in Abbildung 2.1 sind die Deklarationen der Ein- und Ausgänge (engl.: port) für VHDL zu sehen, die sich im Quellcode-Listing 2.1 zwischen den Zeilen 4 und 12 nach dem Schlüsselwort `PORT` wiederfinden. Die Deklarationen der Ports werden mit der „(“-Klammer geöffnet und mit „);“ wieder geschlossen. Deklarationen und Anweisungen werden in VHDL durch ein Semikolon „;“ abgeschlossen, mit Ausnahme von Zeile 11, da dies die letzte Port-Deklaration ist. Die Deklaration jedes Ports wird mit dem Namen des Ports eingeleitet und nach dem Doppelpunkt wird die Richtung (Modus) des Ports angegeben. In VHDL werden Inputs mit dem Schlüsselwort `IN` bezeichnet, Outputs mit `OUT` und Bidirectionals mit `INOUT`. Der `BUFFER`-Port ist eine spezielle Form eines bidirektionalen Ports, von dessen Benutzung jedoch abgeraten wird [14]. Die jeweils letzte Angabe in der

Listing 2.1: *Entity des 2-Bit-Registers*

```
0   LIBRARY ieee;
1   USE ieee.std_logic_1164.all;
2
3   ENTITY reg2 IS
4      PORT(
5         clk  : IN     std_logic;
6         d0   : IN     std_logic;
7         d1   : IN     std_logic;
8         load : IN     std_logic;
9         res  : IN     std_logic;
10        q0   : OUT    std_logic;
11        q1   : OUT    std_logic
12     );
13  END reg2 ;
```

Deklaration der Ports (Zeile 5–11) bezieht sich auf den Datentyp des Ports, da in VHDL für jedes Objekt ein Datentyp deklariert werden muss. Datentypen und insbesondere die IEEE-Datentypen werden noch in einem späteren Abschnitt gesondert dargestellt. Der Datentyp `std_logic` kommt aus einem *Package* der Bibliothek `ieee`. Die Bibliothek muss dem Compiler bekannt gemacht werden, dies erfolgt durch einen „logischen“ Namen, hier `ieee`, siehe auch Abschnitt 2.2.6. Die Verwendung von Bibliothekselementen, im Beispiel das Package `std_logic_1164`, erfolgt dann über `USE`-Anweisungen (engl.: use clause). Zumeist werden Datentypen, aber auch Funktionen und Prozeduren, die von mehreren Entities und Architectures benötigt werden, in Packages nur einmal definiert. Dies kann in etwa verglichen werden mit den so genannten „Header“-Dateien in C [87]. Auf Packages wird in Abschnitt 2.6 näher eingegangen.

Zu einer Entity muss mindestens *eine* „Architecture“ gehören. Eine Architecture beschreibt entweder die innere Funktion, dies wird zumeist als *Verhalten* bezeichnet, oder die Struktur der Komponente, das heißt die Verschaltung von (Sub-)Komponenten. Im ersten Fall spricht man von einer *Verhaltensbeschreibung* und im zweiten Fall von einer *Strukturbeschreibung*. Wie schon erwähnt können Strukturbeschreibungen, also Netzlisten, graphisch mit Hilfe von Schema-Editoren eingegeben werden. Dies sollte man gerade als VHDL-Neuling nutzen, da man hierdurch weniger Fehler machen kann und damit insgesamt produktiver wird. Selbstverständlich können VHDL-Netzlisten auch mit einem Texteditor geschrieben werden. Für Verhaltensbeschreibungen ist die Eingabe mit einem Texteditor die Regel, wobei auch hier teilweise graphische Werkzeuge angeboten werden, um zum Beispiel Schaltwerksgraphen zu zeichnen, aus denen wiederum VHDL-Code generiert werden kann.

Zu einer Entity können aber auch beliebig viele Architectures gehören. Hier zeigt sich die große Flexibilität von VHDL: Die einzelnen Architectures *einer* Entity können für verschiedene Simulationsläufe ausgetauscht werden. Welche Architecture zu simulieren ist, wird dem Simulator in der Regel über eine *Configuration* mitgeteilt. Die Configuration wird in Abschnitt 2.8.3 erläutert. Für den VHDL-Neuling mag die Tatsache, dass zu einer Entity mehrere Architectures gehören können, zunächst kompliziert und überflüssig erscheinen. In der Praxis hat dies jedoch eine große Bedeutung: Auf der nächst höheren Netzlisten-Ebene, in der die Komponente „eingebaut“ oder *instanziert* wird, ist keine Veränderung notwendig wenn die Architecture ausgetauscht wird, da die Entity und damit die Schnittstelle nicht verändert wird. Somit können verschiedene Implementierungen, in der Gestalt von verschiedenen Architectures, für ein und diesselbe Entity in der gleichen Umgebung getestet werden – vorausgesetzt, dass keine zusätzlichen Ports benötigt werden. Eine wesentliche Anwendung dieses Mechanismus ist beispielsweise folgende: Normalerweise schreibt man ein technologieunabhängiges RTL-Modell einer Schaltung, dieses wird von einem Synthesewerkzeug in eine technologiespezifische Gatterrealisierung überführt. Das Gattermodell der Schaltung soll dann – genauso wie das RTL-Modell – simuliert werden. Dies erfolgt dadurch, dass man die *gleiche* Testbench, die ursprünglich für die RTL-Simulation geschrieben wurde, verwendet und nun unter die *gleiche* Entity der Schaltung die Architecture des Gattermodells legt, wobei das Gattermodell der Schaltung im VHDL-Format vom Synthesewerkzeug geliefert wird, wie wir später noch sehen

werden. Im Idealfall erfordert dieser Schritt nur eine neue Configuration und ansonsten keinen weiteren Aufwand. Als weiterer Vorteil ist auch keine erneute Kompilation der nächst höheren Schaltungsebene (Strukturbeschreibung) und der Entity notwendig.

Listing 2.2: *Architecture des 2-Bit-Registers*

```
0   ARCHITECTURE beh OF reg2 IS
1     SIGNAL q0_s, q0_ns, q1_s, q1_ns : std_logic;
2   BEGIN
3
4     reg: PROCESS (clk, res)
5     BEGIN
6       IF res = '1' THEN
7         q0_s <= '0';
8         q1_s <= '0';
9       ELSIF clk'event AND clk = '1' THEN
10        q0_s <= q0_ns;
11        q1_s <= q1_ns;
12      END IF;
13    END PROCESS reg;
14
15    q0 <= q0_s AFTER 2 ns;
16    q1 <= q1_s AFTER 2 ns;
17
18    mux: PROCESS (load, q0_s, q1_s, d0, d1)
19    BEGIN
20      IF load = '1' THEN
21        q0_ns <= d0 AFTER 3 ns;
22        q1_ns <= d1 AFTER 3 ns;
23      ELSE
24        q0_ns <= q0_s AFTER 4 ns;
25        q1_ns <= q1_s AFTER 4 ns;
26      END IF;
27    END PROCESS mux;
28
29  END beh;
```

Üblicherweise speichert man die Entity und eine oder mehrere Architectures, die zur Entity gehören, in einer gemeinsamen Quelldatei ab. In Listing 2.2 ist die Architecture gezeigt, welche zur Entity in Listing 2.1 gehört. Der Bezug zur Entity ist in der ersten Zeile zu sehen: Jede Architecture muss einen *eindeutigen* Bezeichner bekommen, im Beispiel `beh`, und eine Referenz zu einer Entity haben, im Beispiel zur Entity `reg2`. Der Rest dieser Zeile besteht aus Schlüsselworten. VHDL macht im Übrigen keinen Unterschied zwischen Groß- und Kleinschreibung, das heißt Bezeichner und auch Schlüsselworte können nicht durch Groß- und Kleinschreibung unterschieden werden! In den Beispielen des Buches werden die VHDL-Schlüsselworte zur Verdeutlichung groß geschrieben. Vor dem `BEGIN`-Schlüsselwort in Zeile 2 befindet sich ein *Deklarationsteil*. Hier muss alles deklariert werden, was nicht in anderen VHDL-Einheiten, wie beispielsweise der Entity oder einem Package, schon deklariert wurde und vor Übersetzen der Architecture daher bekannt ist. In der Regel sind dies interne *Signale*, die nach dem `BEGIN`-Schlüsselwort benötigt werden. Signale dienen hauptsächlich der Verbindung von Komponenten und Prozessen; ihre

Eigenschaften werden in den Abschnitten 2.2.6 und 2.2.8 näher erläutert. Im Beispiel-Listing 2.2 sind zwei so genannte *Prozesse* enthalten. Die Funktionsweise von Prozessen soll im nächsten Abschnitt erläutert werden.

2.2.2 Verhaltensbeschreibungen und Prozesse

Eine Verhaltensbeschreibung besteht typischerweise aus mehreren Prozessen innerhalb einer Architecture und es werden keine weiteren Komponenten instanziert. Stellt man sich die Schaltungshierarchie als Baum vor, wobei die oberste Ebene der Schaltung die Wurzel des Baumes ist, dann ist eine Verhaltensbeschreibung ein Blatt des Baumes und die Äste und die Wurzel sind Strukturbeschreibungen.

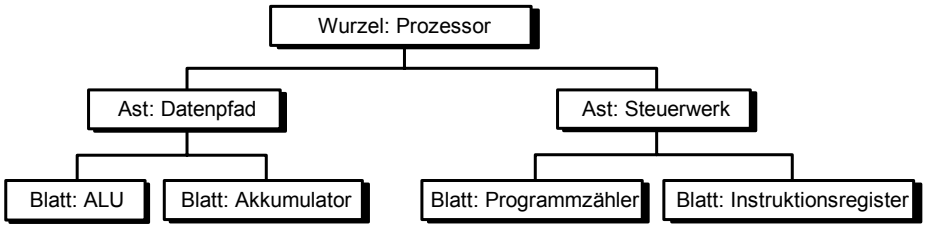

Abb. 2.2: *Schaltungshierarchie am Beispiel eines Prozessors: In der Strukturbeschreibung „Datenpfad“ werden die Verhaltensbeschreibungen „ALU“ und „Akkumulator“ verschaltet. Die Strukturbeschreibung „Steuerwerk“ verschaltet die Verhaltensbeschreibungen „Programmzähler“ und „Instruktionsregister“ und die Strukturbeschreibung „Prozessor“ verschaltet wiederum „Datenpfad“ und „Steuerwerk“.*

Ein Beispiel für diese Sichtweise ist in Abbildung 2.2 gezeigt: *Prozessor*, *Datenpfad* und *Steuerwerk* sind reine Strukturbeschreibungen, die Komponenten der untersten Ebene sind reine Verhaltensbeschreibungen. Diese Vorgehensweise wird allerdings nicht durch VHDL vorgegeben, sondern ist eine Frage der *Entwurfsmethodik*. VHDL lässt es nämlich zu, in einer Architecture Struktur- und Verhaltensbeschreibungen zu mischen; in Testbench-Beschreibungen findet man diese Mischung auch häufig, wie an einem Beispiel noch zu sehen sein wird. Solche gemischten Beschreibungen können auch synthetisiert werden. Es hat sich in der Praxis jedoch als günstig herausgestellt, in der hierarchischen Beschreibung einer *Schaltung*, im Gegensatz zur Vorgehensweise bei einer Testbench, eine Architecture entweder als reine Strukturbeschreibung oder als reine Verhaltensbeschreibung auszulegen. Neben der besseren Übersicht ergeben sich auch Vorteile in der Handhabung während der Synthese, dies gilt insbesondere dann, wenn sehr große Schaltungen entwickelt werden sollen.

Innerhalb einer Verhaltensbeschreibung erfolgt auch eine gewisse Strukturierung durch die Verwendung von Prozessen. Im Beispiel von Listing 2.2 aus dem letzten Abschnitt sind zwei Prozesse enthalten: `reg` und `mux`. Die Prozesse sind sowohl untereinander über *Signale* verbunden als auch zu den Ports. Die vier internen Signale müssen zunächst in der Zeile 1 der Architecture deklariert werden. `SIGNAL` ist wieder ein Schlüsselwort, gefolgt von den Bezeichnern für die vier Signale und einer nachfolgenden Angabe des

Datentyps. Es können beliebig viele Signal-Deklarationen angegeben und in jeder Deklaration beliebig viele Signale des gleichen Datentyps deklariert werden. Nach dem Deklarationsteil folgt die eigentliche Beschreibung der Funktionalität der Architecture, die mit dem Schlüsselwort `BEGIN` eingeleitet wird, auf welches dann die Beschreibung der Prozesse folgt. Eine Prozess-Beschreibung beginnt mit dem Namen des Prozesses und dem `PROCESS`-Schlüsselwort. Optional kann danach eine *Sensitivitätsliste* folgen. Die Sensitivitätsliste spielt eine wesentliche Rolle: Ein Prozess wird dann „ausgeführt“, wenn sich der Wert eines Signales in der Sensitivitätsliste ändert. Ausführen bedeutet, dass die Anweisungen im Körper des Prozesses, zwischen `BEGIN` und `END PROCESS`, *sequentiell*, also aufeinanderfolgend, ausgeführt werden, wie von prozeduralen Programmiersprachen wie C oder Pascal gewohnt. Daher werden die Anweisungen im Körper eines Prozesses auch als „sequentielle Anweisungen“ bezeichnet. Wenn keine Sensitivitätsliste vorhanden ist, muss der Prozess über `WAIT`-Anweisungen gesteuert werden, wobei sich der Gebrauch von Sensitivitätsliste und `WAIT`-Anweisungen gegenseitig ausschließen. Die Prozessbeschreibungen innerhalb der eigentlichen Schaltungsbeschreibung werden typischerweise über Sensitivitätslisten gesteuert, wohingegegen in Testbenches auch `WAIT` häufig eingesetzt wird. Zwischen der Sensitivitätsliste und dem `BEGIN`-Schlüsselwort befindet sich noch ein Deklarationsteil, der zumeist genutzt wird, um lokale *Variablen* zu deklarieren. Der Unterschied zwischen Signalen und Variablen wird später erklärt.

Es sei an dieser Stelle vermerkt, dass nicht alle Signale, die im Körper des Prozesses als *Quelle* dienen, in der Sensitivitätsliste aufgeführt werden *müssen*. Allerdings wird der Prozess auf die fehlenden Signale auch nicht mehr reagieren! Dies zählt erfahrungsgemäß zu den häufigsten Fehlern und führt zuweilen in der Simulation zu nur schwer zu entdeckendem Fehlverhalten. Dem aufmerksamen Leser ist vielleicht nicht entgangen, dass im Beispiel-Listing 2.2 genau dieser „Fehler“ vorliegt: Die Signale `q0_ns` und `q1_ns` werden in der Sensitivitätsliste von Prozess `reg` nicht aufgelistet. Das ist in diesem Fall allerdings kein Fehler, da es sich hier um die Beschreibung von zwei taktflankengesteuerten Flipflops mit asynchronem Reset handelt. Der Prozess ist daher nur sensitiv auf den Takt `clk` und den Reset `res`. Ist `res` logisch „1“ so werden die Flipflops asynchron, das heißt unabhängig vom Takt, auf ihren Rücksetzwert gesetzt; dies wird mit der `IF`-Anweisung von Zeile 6 bis 8 beschrieben. Im `ELSIF`-Teil wird die steigende Taktflanke mit `clk'event AND clk = '1'` abgefragt. Hierzu muss ein Ereignis (engl.: event) auf dem Signal `clk` vorhanden sein, welches dazu führt, dass `clk` gleich logisch „1“ wird. Ein Ereignis ist ein Wechsel des Signalwertes. In einem binären Logiksystem sind zwei Werte möglich: „0“ und „1“. Daher war der Logikwert des Signals vor dem Ereignis „0“ und ist nach dem Ereignis „1“; damit wird eine steigende Taktflanke beschrieben. Sollte also diese Bedingung `clk'event AND clk = '1'` „wahr“ (engl.: true) sein, so werden die Werte von `q0_ns` und `q1_ns` in `q0_s` und `q1_s` übernommen. In der `IF`-Konstruktion fehlt nun ein `ELSE`-Teil, was folgende Bedeutung hat: Der `ELSE`-Fall ist *implizit* kodiert, das heißt, wenn die `IF`- und `ELSIF`-Bedingungen nicht wahr sind – also nicht zutreffen – wird den Signalen `q0_s` und `q1_s` nichts Neues zugewiesen, was in VHDL bedeutet, dass der alte Wert gehalten werden soll. Die Signale `q0_s` und `q1_s` modellieren daher jeweils *ein Flipflop*. Die Signale `q0_ns` und `q1_ns` beeinflussen das

Verhalten der Flipflops nicht, sondern sind lediglich die zur steigenden Flanke zu übernehmenden Eingangssignale der Flipflops und erscheinen demzufolge nicht in der Sensitivitätsliste. Dies entspricht der aus den Grundlagen der Digitaltechnik [44, 11, 83] bekannten Funktionsweise eines flankengesteuerten Flipflops. Der Prozess `reg` beschreibt somit eine *speichernde* Schaltung.

Beim zweiten Prozess `mux` handelt es sich um eine so genannte *kombinatorische* Schaltung oder ein *Schaltnetz*. Dies ist eine Schaltung ohne speicherndes Verhalten. Hier sollten nun *alle* Quell- oder Eingangssignale in der Sensitivitätsliste vorhanden sein, da eine kombinatorische Schaltung auf alle Eingangssignale reagiert. Im Beispiel sind dies die Signale `load`, `q0_s`, `q1_s`, `d0` und `d1`. Ein Fehlen von Signalen in der Sensitivitätsliste wird von einem VHDL-Compiler oder VHDL-Simulator nicht als Fehler gemeldet, da es zulässig ist und wie oben gesehen bei sequentiellen Teilen der Schaltung auch benötigt wird. Ein Synthesewerkzeug wird allerdings eine Warnung an dieser Stelle bei kombinatorischen Schaltungen ausgeben, daher empfiehlt es sich, auch frühzeitig Syntheseläufe durchzuführen. Reagiert ein Prozess nicht wie gewünscht in einer Simulation, sollten als Erstes immer die Sensitivitätslisten überprüft werden. Der Prozess `mux` kodiert einen Multiplexer mit der Breite 2 Bit. Ist `load` logisch „1“, so werden die Eingangssignale `d0` und `d1` auf die Ausgänge des Multiplexers `q0_ns` und `q1_ns` geschaltet, welche wiederum Eingänge der Flipflops sind. Anderenfalls werden die Signale `q0_s` und `q1_s` auf die Ausgänge geschaltet; das heißt wenn `load` logisch „0“ ist, werden die Daten der Flipflops in einer Rückführungsschleife gehalten. Beide Prozesse zusammen kodieren also ein ladbares 2 Bit breites Register. Hinter dieser Aufteilung in einen kombinatorischen Prozess und einen Prozess mit Speicherverhalten steckt im Übrigen die aus der Digitaltechnik bekannte Darstellung von Schaltwerken, siehe Abbildung 2.5 und [44, 11, 83], durch Überführungsfunktion oder Überführungsschaltnetz zur Berechnung der neuen Werte für das Zustandsregister, im Beispiel die Signale `q0_ns` und `q1_ns`, das Zustandsregister, im Beispiel `q0_s` und `q1_s`, und ein Ausgabeschaltnetz, welches im Beispiel nicht notwendig war. In Zeile 15 und 16 in Listing 2.2 sind zwei so genannte *nebenläufige Anweisungen* zu sehen. In diesem Fall dienen sie nur der Verbindung der internen Signale zu den Ports. Sie sind wie „kleine“ Prozesse oder *implizite* Prozesse zu verstehen, das heißt sie sind ebenfalls nebenläufig zu allen Prozessen und allen anderen nebenläufigen Anweisungen. Nebenläufige Anweisungen werden später noch genauer erläutert. Einigen Anweisungen sind Verzögerungszeiten zugeordnet, ausgedrückt durch `AFTER`-Anweisungen, welche dazu führen, dass die Signalzuweisungen um die angegebene Zeit verzögert ausgeführt werden. Die Einheit `ns` steht für Nanosekunden; die Angabe von Verzögerungszeiten ist optional.

Eine digitale Hardwarelösung mit zwei Flipflops und zwei Multiplexern, welche die gleiche Funktion wie der VHDL-Code aus Listing 2.2 realisiert, ist in Abbildung 2.3 gezeigt. An dieser Sichtweise kann der Sinn der Prozesse in VHDL gut demonstriert werden: Prozesse sind Modelle von Hardwareblöcken. Ein Hardwareblock, im Beispiel ein Multiplexer oder ein Flipflop, reagiert auf Signalwechsel an den Eingängen – im VHDL-Modell ist das die Sensitivitätsliste – und gibt das Funktionsergebnis mit einer gewissen Verzöge-

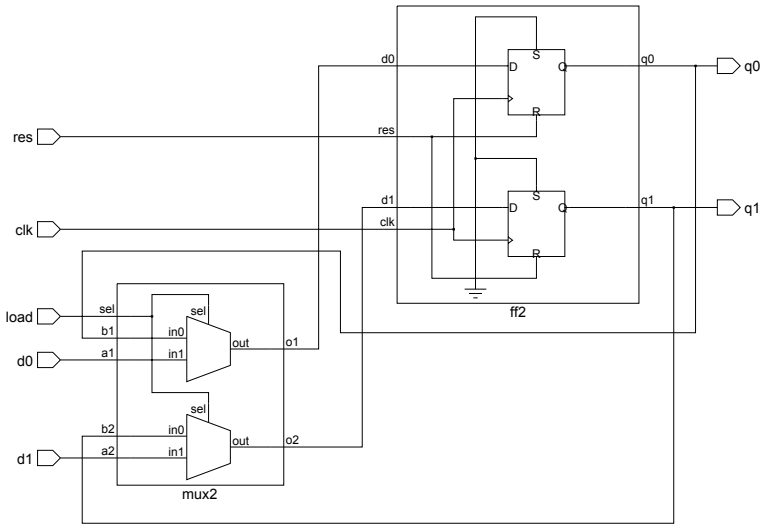

Abb. 2.3: *Hardwarelösung des 2-Bit-Registers*

rung an den Ausgang weiter. Die Hardwareblöcke, also die Prozesse, sind über Signale verbunden, welche die Ereignisse am Ausgang eines Blockes zum Eingang von weiteren Blöcken führen. Ein Block kann ein einzelnes Flipflop sein, oder aber etwas sehr viel komplexeres, wie beispielsweise eine komplette ALU (engl.: Arithmetic Logic Unit) eines Mikroprozessors, wie in Kapitel 4 noch zu sehen sein wird. Wie in unserem Beispiel gezeigt, können auch mehrere voneinander unabhängige Hardwareblöcke in einem Prozess zusammengefasst werden, dies sind im Beispiel jeweils zwei Multiplexer und zwei Flipflops. Damit lassen sich sehr kompakte Beschreibungen der Hardware kodieren.

Eine inhärente Eigenschaft von Hardwarerealisierungen ist die *Parallelität* oder *Nebenläufigkeit*: Voneinander unabhängige Hardwareblöcke können *gleichzeitig* arbeiten; beispielsweise können mehrere ALUs in einem Mikroprozessor vorhanden sein und damit die Leistungsfähigkeit erhöhen. Parallelisierung ist *die* leistungssteigernde Maßnahme in der Digitaltechnik. Um die inhärente Parallelität der Hardware modellieren zu können, wird die *Nebenläufigkeit* (engl.: concurrency) von Prozessen und deren *Reaktivität* auf Signale (Sensitivitätsliste oder `WAIT`-Anweisungen) benötigt. Dies ist einer der wesentlichen Unterschiede zu einer Programmiersprache für Software wie C oder Pascal. Nochmals sei erwähnt, dass bei der Diskussion von VHDL immer die Nebenläufigkeit bezüglich der Modellzeit gemeint ist. Sind in einem Prozess keine `WAIT`-Anweisungen vorhanden, wie im Beispiel dieses Abschnitts, so wird alles sequentiell ausgeführt, was in einem Prozess kodiert ist, bis zur `END PROCESS`-Zeile; der Prozess ist „ausführend“ (engl.: executing). Danach wartet der Prozess wieder auf erneute Ereignisse, also *Signalwechsel*, in seiner Sensitivitätsliste; er ist somit unterbrochen in seiner Ausführung (engl.: suspended). Solange sich keines der Signale in der Sensitivitätsliste ändert, bleibt der Prozess unterbro-

chen. Die Steuerungsmöglichkeiten mit einer WAIT-Anweisung sind in Abschnitt 2.4.4 beschrieben. Ein Prozess wird also *nie* beendet, er kann allerdings mit einer speziellen WAIT-Anweisung für immer suspendiert werden. Die Beendigung der gesamten Simulation wird in der Regel durch den Simulator selbst oder in einer Testbench gesteuert, so dass die Simulation bei einem bestimmten Zeitpunkt in der Modellzeit abbricht.

Listing 2.3: *Alternative Architecture des 2-Bit-Registers*

```
0   ARCHITECTURE beh1 OF reg2 IS
1     SIGNAL q0_s, q1_s : std_logic;
2   BEGIN
3
4     reg: PROCESS (clk, res)
5     BEGIN
6       IF res = '1' THEN
7         q0_s <= '0';
8         q1_s <= '0';
9       ELSIF clk'event AND clk = '1' THEN
10        IF load = '1' THEN
11          q0_s <= d0;
12          q1_s <= d1;
13        END IF;
14      END IF;
15    END PROCESS reg;
16
17    q0 <= q0_s AFTER 2 ns;
18    q1 <= q1_s AFTER 2 ns;
19
20  END beh1;
```

Die Trennung in zwei Prozesse muss nicht unbedingt sein. Man könnte die gleiche Funktionalität auch kompakter in einem einzigen Prozess kodieren, wie in Listing 2.3 gezeigt. Da hier allerdings keine explizite Trennung in kombinatorischen und speichernden Teil erfolgt, sind solche Beschreibungen etwas anfälliger für Codierungsfehler, wir werden hierauf in Kapitel 4 näher eingehen. Zu beachten ist in diesem Zusammenhang, dass das Halten der Werte für `load = '0'` nicht explizit kodiert ist, das Fehlen des ELSE-Zweiges bedeutet das Halten der Daten im Register. Für kurze übersichtliche Registerbeschreibungen kann die einfachere Form in Listing 2.3 durchaus verwendet werden, für die Beschreibung von größeren Schaltwerken empfiehlt sich jedoch eine Trennung in mindestens zwei Prozesse. Für die Umsetzung in Hardware mit Hilfe von Synthesewerkzeugen sind beide Formen gleich gut geeignet und führen zum gleichen Ergebnis. Erwähnt werden muss allerdings, dass eine größere Anzahl von Prozessen zu einer verminderten Simulationsgeschwindigkeit führt, so dass man auch nicht zu viele Prozesse verwenden sollte.

Für unser Beispiel ist das Ergebnis der Synthese für beide Architectures `beh` und `beh1` in Abbildung 2.4 gezeigt: Das Synthesewerkzeug kann den VHDL-Code jeweils in zwei ladbare Flipflops mit asynchronem Reset umsetzen, die Halte-Funktion des Multiplexers ist durch den CE-Eingang des Flipflops realisiert. Dieses Ergebnis setzt voraus, dass die Bibliothek der Zielhardware solche Flipflops anbietet. Zu beachten ist im Übrigen, dass

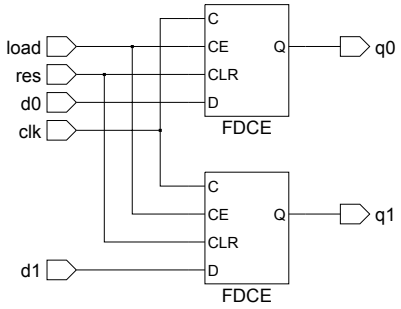

Abb. 2.4: *Ergebnis der Synthese des ladbaren 2-Bit-Registers*

man sich an gewisse *Muster* bei VHDL-Beschreibungen halten muss, damit die Synthesewerkzeuge den Code auch richtig „verstehen“. Für die Beschreibung der Taktflanke darf beispielsweise nur ein Signal (das Taktsignal) verwendet werden; es wäre für die Synthese falsch, den Code in Listing 2.3 in Zeile 9 und 10 durch folgenden „kürzeren“ Code zu ersetzen:

```
ELSIF clk'event AND clk = '1' AND load = '1' THEN
```

Obwohl es sich um korrekten VHDL-Code handelt und dies auch simuliert werden kann, wird ein Synthesewerkzeug beispielsweise Folgendes melden:

```
Error, clock expression should contain only one signal.
```

Welche Muster für RTL-Beschreibungen verwendet werden sollten, wird insbesondere in Kapitel 4 näher erörtert.

Aus der bisherigen Diskussion dürfte ausreichend klar geworden sein, dass für eine erfolgreiche Anwendung von VHDL Kenntnisse aus der Digitaltechnik unerlässlich sind. Eine „Software“-Sicht auf VHDL ist nicht zielführend und sollte vermieden werden: Ein VHDL-Entwickler sollte bei der Codierung eine ungefähre Vorstellung davon haben, wie der Code in Hardware umgesetzt werden wird. Die Entwicklung einer Schaltung in VHDL sollte vor der eigentlichen Codierung zunächst konzeptionell überlegt werden, wobei als Erstes eine hierarchische Aufteilung in Komponenten, die aus Entity und Architecture bestehen, im Sinne der Abbildung 2.2 vorgenommen wird. Für die „Blätter“ des Entwurfs, also die Verhaltensbeschreibungen, hält man sich am besten an die in diesem Abschnitt demonstrierte Vorgehensweise. Man zerlegt die Komponente in speichernde und kombinatorische Teile, die Codierung erfolgt dann in der Regel auf Register-Transfer-Ebene. Beschrieben werden daher die Funktionen der Register (speichernde Teile) und die Transferfunktionen (kombinatorische Teile, Schaltnetze) zwischen den Registern. Damit hält man sich auch an die schon angesprochene und in Abbildung 2.5 gezeigte Darstellungsweise für Schaltwerke aus der Digitaltechnik [44, 11, 83], so genannte „endliche Automaten“ (engl.: Finite-State Machine, FSM): Das Überführungsschaltnetz berechnet aus den Eingangsvariablen X und den Zustandsvariablen Z des (Zustands-)Registers den mit der

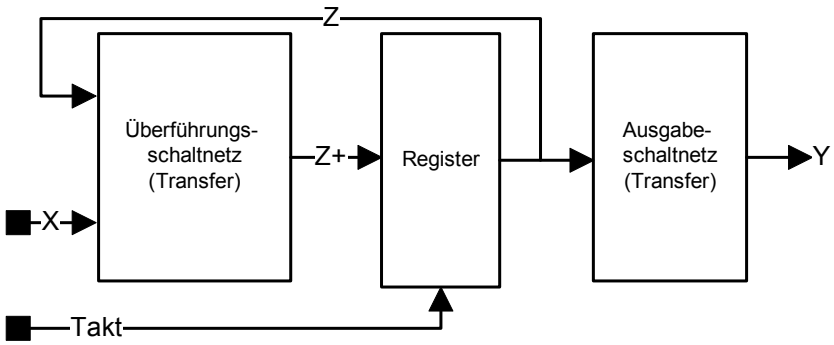

Abb. 2.5: *Modell des endlichen Automaten (Moore-Schaltwerk)*

nächsten Taktflanke zu übernehmenden, neuen Wert des Registers $Z+$ und ist somit eine Boole'sche Funktion $Z+ = f(X,Z)$. Das Ausgabeschaltnetz berechnet aus den Zustandsvariablen Z die Ausgabe des Schaltwerkes Y, ebenfalls als Boole'sche Funktion $Y = g(Z)$. Dies ist die Beschreibung eines so genannten „Moore"-Automaten. Bei einem „Mealy"-Automaten ist zusätzlich ein direkter Durchgriff des Eingangs auf den Ausgang möglich, so dass für das Ausgabeschaltnetz gilt: $Y = g(X,Z)$. Die Boole'schen Funktionen werden in einer VHDL-RTL-Beschreibung normalerweise mit sequentiellen Anweisungen kodiert; hierzu gehören insbesondere *Auswahlkonstruktionen* (if, case) oder *Schleifenkonstruktionen* (for, loop, while) wie sie aus prozeduralen Programmiersprachen wie beispielsweise C bekannt sind. Das Synthesewerkzeug hat dann die Aufgabe, daraus eine Boole'sche Funktion und letztlich eine Realisierung für eine Zieltechnologie zu generieren, wie in Kapitel 4 noch ausführlich erläutert wird.

2.2.3 Strukturbeschreibungen

Wie in den vorangegangenen Abschnitten erläutert, benötigt man für größere Schaltungen eine Möglichkeit diese hierarchisch zu gliedern. Hierfür müssen Komponenten in Strukturbeschreibungen – welche üblicherweise auch als Netzlisten bezeichnet werden – einfach oder mehrfach instanziert werden sowie die Ein- und Ausgänge der *Instanzen* der Komponenten über eindeutig bezeichnete *Netze*, daher der Ausdruck Netzliste, miteinander verbunden werden. Instanzierung bedeutet, dass man eine Komponente eines bestimmten Typs, dies ist der *Komponentenname*, auswählt, in die Schaltung „einbaut" und mit einem eindeutigen Bezeichner versieht, dies ist der *Instanzname*. Um Netzlisten beschreiben zu können, bietet VHDL Konstruktionen für die Instanzierung von Komponenten an, die Rolle der Netze übernehmen dabei wieder die schon aus den Verhaltensbeschreibungen bekannten Signale. Die Komponenten sind wiederum Entities mit zugeordneten Architectures.

Abbildung 2.6 zeigt eine weitere Variante für den Entwurf eines ladbaren 2-Bit-Registers. Hier handelt es sich um einen hierarchischen Entwurf, bestehend aus einem 2-Bit-Register

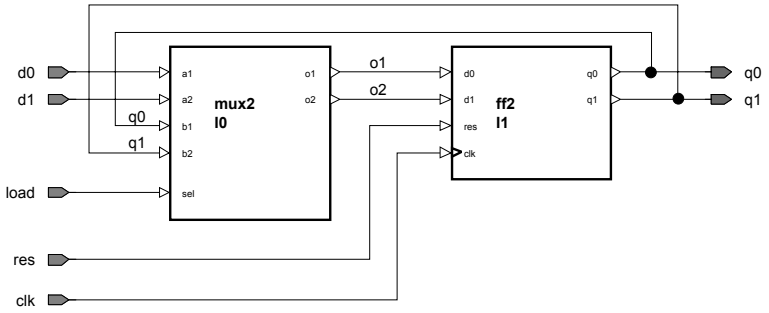

Abb. 2.6: *Schema-Darstellung des ladbaren 2-Bit-Registers*

und einem 2-Bit-Multiplexer. Diese graphische Darstellung wurde mit einem Schema-Editor erzeugt, der über die Möglichkeit verfügt, aus der graphischen Darstellung eine VHDL-Strukturbeschreibung zu erzeugen, so dass diese nicht mühsam von Hand eingegeben werden muss. Die automatisch generierte VHDL-Netzliste ist in Listing 2.4 gezeigt; es ist die dritte Architecture zur Entity `reg2` aus Listing 2.1. Es hat sich bei größeren Projekten übrigens als sinnvoll erwiesen, die Architectures je nach Typ einheitlich zu bezeichnen, bei den Strukturbeschreibungen beispielsweise mit `struct` und bei den Verhaltensbeschreibungen beispielsweise mit `beh`.

Listing 2.4: *Hierarchischer Entwurf des 2-Bit-Registers*

```
0   ARCHITECTURE struct OF reg2 IS
1      SIGNAL o1 : std_logic;
2      SIGNAL o2 : std_logic;
3      SIGNAL q0_internal : std_logic;
4      SIGNAL q1_internal : std_logic;
5
6      COMPONENT ff2
7      PORT (
8         clk : IN     std_logic ;
9         d0  : IN     std_logic ;
10        d1  : IN     std_logic ;
11        res : IN     std_logic ;
12        q0  : OUT    std_logic ;
13        q1  : OUT    std_logic
14     );
15     END COMPONENT;
16     COMPONENT mux2
17     PORT (
18        a1  : IN     std_logic ;
19        a2  : IN     std_logic ;
20        b1  : IN     std_logic ;
21        b2  : IN     std_logic ;
22        sel : IN     std_logic ;
23        o1  : OUT    std_logic ;
24        o2  : OUT    std_logic
25     );
26     END COMPONENT;
```

```
27
28  BEGIN
29
30     I1 : ff2
31        PORT MAP (
32           clk => clk, d0  => o1, d1  => o2,
33           res => res, q0  => q0_internal, q1  => q1_internal
34        );
35     I0 : mux2
36        PORT MAP (
37           a1  => d0, a2  => d1, b1  => q0_internal,
38           b2  => q1_internal, sel => load, o1  => o1, o2  => o2
39        );
40
41     q0 <= q0_internal;
42     q1 <= q1_internal;
43
44  END struct;
```

Vor dem `BEGIN`-Schlüsselwort in Zeile 28 muss zunächst wieder alles deklariert werden, was danach für die Beschreibung der Architecture verwendet wird und nicht schon bekannt ist. Neben den Signalen müssen hier die zu verschaltenden Komponenten deklariert werden. Die Deklaration einer Komponente beginnt mit dem `COMPONENT`-Schlüsselwort und beinhaltet neben dem Bezeichner der Komponente die schon von den Entities bekannte `PORT`-Deklaration. Die Ports der Komponente werden im Englischen als „local ports“ [24] bezeichnet. Die `COMPONENT`-Deklaration ist damit ähnlich zur Beschreibung einer Entity (deren Ports als „formal ports“ bezeichnet werden). Normalerweise erfolgt die Bindung (engl.: binding) zu einer Entity dadurch, dass in einer VHDL-Bibliothek eine passende Entity mit gleichem Namen vorhanden ist (so genanntes „default binding“). Wird eine Configuration verwendet, so kann auch eine Entity mit einem anderen Namen mit der Komponente gebunden werden, dies soll in Abschnitt 2.8.3 noch erläutert werden. In der Regel prüft ein VHDL-Compiler beim Kompilieren einer Strukturbeschreibung, ob schon eine passende Entity gleichen Namens übersetzt und damit in der Bibliothek vorhanden ist. Ist dies nicht der Fall kann es beispielsweise zu folgender Warnung kommen:

```
No default binding for component: "ff2".
(No entity named "ff2" was found)
```

In diesem Fall wurde die Entity `ff2` aus dem Beispiel in Listing 2.4 noch nicht kompiliert. Dies ist allerdings nur eine Warnung und die Architecture wird trotzdem übersetzt, es könnte ja später noch durch eine Configuration zu einer Bindung kommen. Gibt es keine Configuration wird der Simulator mit einem Bindungsfehler abbrechen.

In den Zeilen 30 bis 34 und 35 bis 39 von Listing 2.4 werden die Komponenten instanziert. Zunächst wird der Instanz der Komponente (hier: `ff2` und `mux2`) ein Instanzname (hier: `I1` und `I0`) zugewiesen. Anschließend werden die Ports der Komponente mit den Signalen der Architecture und den Ports der Entity `reg2` in der `PORT MAP` verbunden. Bei der in Listing 2.4 gezeigten Variante werden die Signale der Architecture und die Ports der Entity mit den lokalen Ports der Komponente über eine explizite Zuordnung zwischen Namen der

lokalen Ports der Komponente und Namen der aktuellen Ports der Entity verbunden (engl.: named association, vgl. [8]); dabei spielt die Reihenfolge der Zuordnungen keine Rolle.

Listing 2.5: *Verbindung der Ports durch die Position („positional association“)*

```
0       I1 : ff2
1          PORT MAP (clk, o1, o2, res, q0_internal, q1_internal);
2       I0 : mux2
3          PORT MAP (d0, d1, q0_internal, q1_internal, load, o1, o2);
```

Eine andere Möglichkeit zeigt Listing 2.5: Hier werden die lokalen Ports der Komponente dadurch angeschlossen, dass die Signale und aktuellen Ports über die *Position* den lokalen Ports zugeordnet werden, wobei nun die Reihenfolge wichtig ist und sich an die Reihenfolge der formalen Ports in der `COMPONENT`-Deklaration halten *muss*. Im Beispiel von Listing 2.5 ergeben sich die gleichen Verbindungen wie in Listing 2.4. Obwohl diese Schreibweise deutlich kürzer ist, soll davon abgeraten werden, da sie fehleranfälliger ist. Werden Anschlüsse einer Komponente nicht benötigt, so besteht die Möglichkeit, sie durch Angabe des Schlüsselwortes `open` nicht anzuschließen. Während dies bei Ausgängen (`OUT`) problemlos ist, sollten Eingänge (`IN`) immer angeschlossen werden; weitere Möglichkeiten sind z. B. in [14] beschrieben. Da man von den `OUT`-Ports einer Entity nicht lesen kann, müssen zusätzliche Signale für die interne Verdrahtung eingeführt werden, wie dies in den Zeilen 41 und 42 in Listing 2.4 der Fall ist.

Der Vollständigkeit halber sind in den Listings 2.6 und 2.7 noch die Verhaltensbeschreibungen von `ff2` und `mux2` gezeigt. Es ist ersichtlich, dass die beiden Prozesse aus dem anfänglichen Beispiel in Listing 2.2 offensichtlich jeweils separat „verpackt“ wurden. Obgleich auch diese Beschreibung in Simulation und Synthese zum gleichen Ergebnis wie die anderen beiden Architectures führt, sollte ein Entwurf jedoch nicht in zu kleine Einheiten verpackt werden, sonst entstünden bei einem komplexeren Entwurf eine hohe Anzahl von Entities und Architectures, was sich negativ auf die Handhabung des gesamten Entwurfs auswirkt. In einer Architecture können schon zehn oder mehr Prozesse angeordnet werden; bei einigen hundert Prozessen verliert man dann natürlich auch innerhalb der Architecture wieder die Übersicht, so dass hier ein gesundes Maß gefunden werden muss. Als Beispiel diene wieder die ALU eines Prozessors: Diese könnte man problemlos inklusive der notwendigen Register in eine Architecture als Verhaltensbeschreibung, also ein Blatt, verpacken. Zumeist ergibt sich aus dem Konzept der Schaltung auch eine sinnvolle Aufteilung in die Verhaltensbeschreibungen (Blätter) und daraus auch die notwendigen Strukturbeschreibungen (Äste) für den hierarchischen Aufbau der Schaltung. In Listing 2.6 ist auch ein Beispiel für einen Kommentar zu sehen: Ein Kommentar kann an beliebiger Stelle stehen und wird mit `--` eingeleitet, wobei dann der Rest der Zeile als Kommentar interpretiert wird.

Listing 2.6: *VHDL-Code des 2-Bit-Registers*

```
0   LIBRARY ieee;
1   USE ieee.std_logic_1164.all;
2   USE ieee.std_logic_arith.all;
```

```
3
4   ENTITY ff2 IS
5      PORT(
6         clk : IN     std_logic;
7         d0  : IN     std_logic;
8         d1  : IN     std_logic;
9         res : IN     std_logic;
10        q0  : OUT    std_logic;
11        q1  : OUT    std_logic
12     );
13  END ff2 ;
14
15  ARCHITECTURE beh OF ff2 IS
16    SIGNAL q0_s, q1_s : std_logic;
17  BEGIN
18
19    reg: PROCESS (clk, res)
20    BEGIN
21      IF res = '1' THEN                    -- Asynchroner Reset
22        q0_s <= '0';
23        q1_s <= '0';
24      ELSIF clk'event AND clk = '1' THEN   -- Steigende Taktflanke
25        q0_s <= d0;
26        q1_s <= d1;
27      END IF;
28    END PROCESS reg;
29
30    q0 <= q0_s AFTER 2 ns;
31    q1 <= q1_s AFTER 2 ns;
32
33  END beh;
```

Listing 2.7: *VHDL-Code des 2-Bit-Multiplexers*

```
0   LIBRARY ieee;
1   USE ieee.std_logic_1164.all;
2   USE ieee.std_logic_arith.all;
3
4   ENTITY mux2 IS
5      PORT(
6         a1  : IN     std_logic;
7         a2  : IN     std_logic;
8         b1  : IN     std_logic;
9         b2  : IN     std_logic;
10        sel : IN     std_logic;
11        o1  : OUT    std_logic;
12        o2  : OUT    std_logic
13     );
14  END mux2 ;
15
16  ARCHITECTURE beh OF mux2 IS
17  BEGIN
18
19    mux: PROCESS (a1, a2, b1, b2, sel)
20    BEGIN
21      IF sel = '1' THEN
```