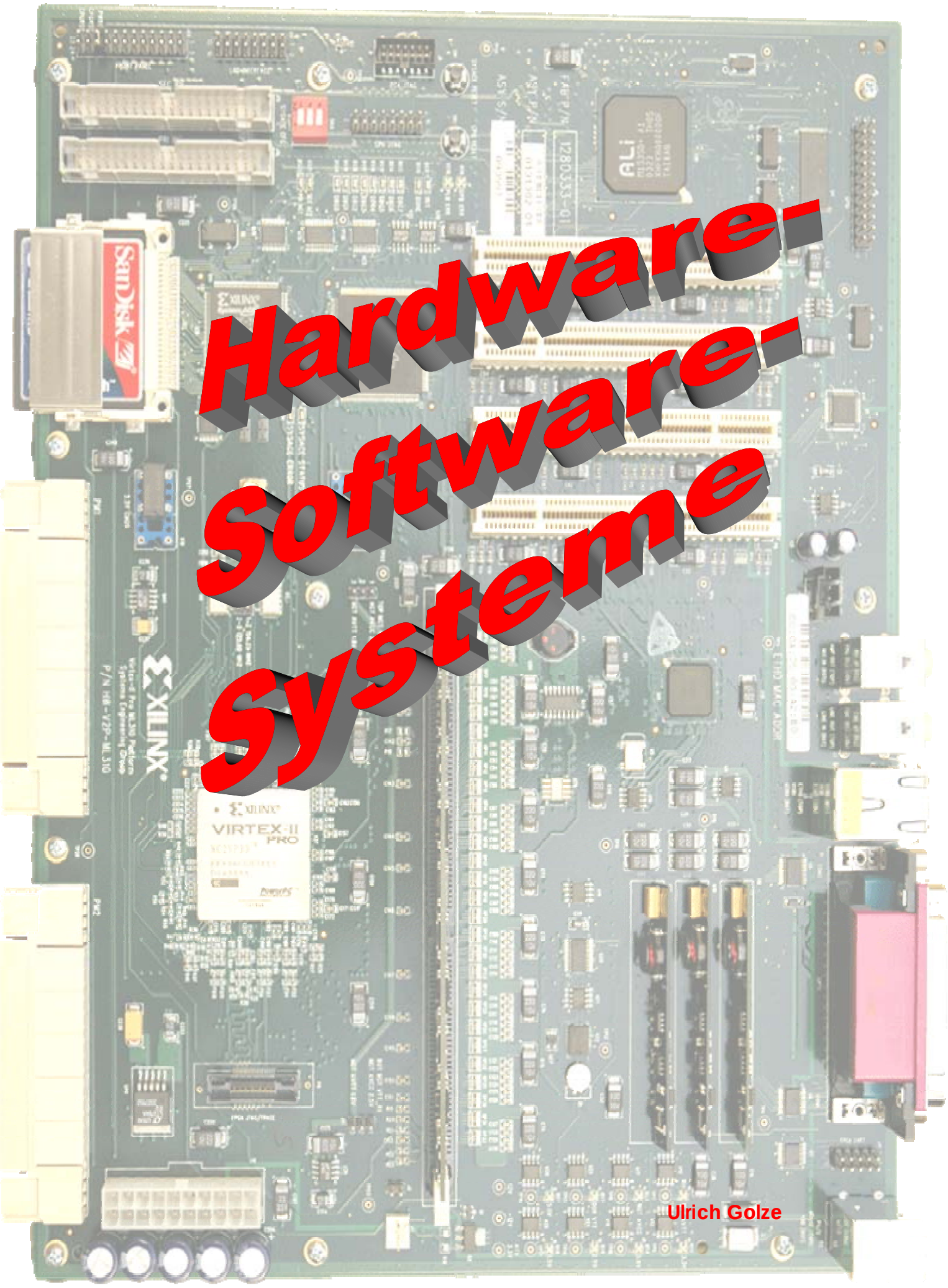

Hardware-
Software-
Systeme
Ulrich Golze

Hardware-Software-Systeme

Teil 1: Vorlesung

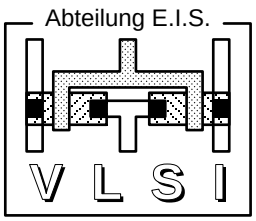
Ulrich Golze

Technische Universität Braunschweig

Abteilung Entwurf integrierter

Schaltungen (E.I.S.)

Oktober 2010



Vorwort

Willkommen bei den Hardware-Software-Systemen! Worum geht es? Diese Frage kann das Vorwort natürlich nicht beantworten, dazu gibt es dieses Skript, dazu treffen wir uns ein Semester lang in der Vorlesung und vor allem den praktischen Übungen an modernen Entwicklungsplattformen mit spannenden CAD-Werkzeugen.

Aber doch so viel: Sie entwerfen Ihre eigene Hardware, ohne zu löten oder Transistoren zu verstehen. Stattdessen formulieren Sie Ihre Absichten in einer speziellen komfortablen Programmiersprache, einer Hardware-Beschreibungssprache. Und Sie müssen auch nicht wochenlang warten, bis Ihre Hardware maßgefertigt aus einer Fabrik geliefert wird. Nein, nach Sekunden läuft Ihre Schaltung auf einer Entwicklungsplattform.

Und wenn Sie näher hinschauen, enthält diese Plattform auch gleich noch einen Standardprozessor für Ihre normale Software, die sich dann mit Ihrer Hardware unterhält. Und fertig ist ein *eingebettetes System*, das Sie im Navi wiederfinden, in der Ski-Bindung, im intelligenten Kühlschrank und (hoffentlich nicht) in Ihrer Neuroprothese.

Für zahlreiche Fehler dieses Skriptes bin ich allein verantwortlich und bitte um Ihre Verbesserungsvorschläge. Dagegen ist das Vorhandene nicht nur mein Verdienst, vielmehr schulde ich besonderen Dank meinen Mitarbeitern Susanne Neugebauer, Hagen Gädke, Robert Günzel, Christian Schröder, Gerrit Telkamp und Jürgen Hannken-Illjes, und trotz regelmäßiger Beziehungskrisen auch MS Word...

Ein erfolgreiches Semester und Spaß an Spiel und Ernst des Hardware-Software-Designs wünscht

Ulrich Golze im Oktober 2010

Inhalt

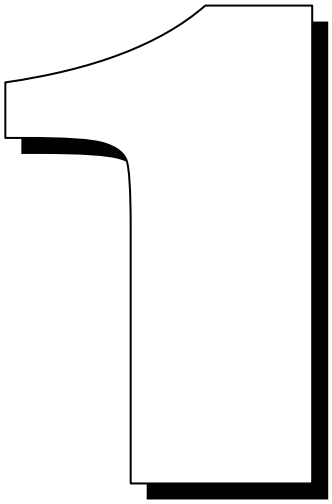
Teil 1: Vorlesung

Teil 2: Übungen

A Die drei Labs zu den HW-SW-Systemen

B Lab 1: VERILOG und seine Tools

C Lab 2: Logiksynthese mit VERILOG



-leitung

Ein Personal-Computer enthält Chips — das weiß jeder. Dass ein Pkw der Oberklasse durch etwa einhundert Computer-Chips gesteuert wird, ist schon weniger bekannt. Ein Besucher an der Haustürklingel erwartet wohl nicht, dass chip-basierte *Home-Automation* ein Bild des Besuchers speichert, innen das Licht anschaltet, ein Ferngespräch zwischen dem abwesenden Besitzer und der Türsprechanlage herstellt und so dessen Anwesenheit vortäuscht.

Als beinahe selbstverständlich gelten Herzschrittmacher, deren Chips das Leben verlängern. Eine elektronische Netzhaut für Blinde wird zwar erforscht, ist aber noch nicht Realität. In die Kleidung integrierte biegsame Chips (*Wearable Computing*) überwachen Körperfunktionen des fortschrittsbewussten Trägers, ersetzen beim Einsteigen in seinen Pkw den Zündschlüssel oder beim Betreten eines Ladens die Kreditkarte. Ob jedoch eine Verpflanzung der gleichen Chip-Funktionen unter die Haut einen Markt erwarten lässt, soll nicht diskutiert werden — einige Discos treten hier als Vorreiter auf.

1.1 Überall Chips

Chips — auch „höchstintegrierte Schaltungen“ genannt oder wissenschaftlich „VLSI-Chips“ (Very Large Scale Integration) oder zukunftsweisend „Schlüssel zum Informationszeitalter“ oder romantisch „Herzen der Technik“ — haben in wenigen Jahrzehnten die Welt verändert. Im Jahr 2004 wurden am Weltmarkt Chips im Wert von 213 Milliarden US-Dollar umgesetzt, davon geschätzt 9 Milliarden Euro in Deutschland.

Den größten Anteil am Chip-Weltmarkt hat die Informations- und Kommunikationstechnik (IuK). Neben der klassischen Informationsverarbeitung vom PC (der jedoch viel mehr kann als „rechnen“) über adaptive Rechner bis hin zum „number-crunchenden“ Supercomputer spielt die Kommunikation eine immer größere Rolle:

mit der steigenden Qualität des Internets wachsen Telefon, Email, Mobilfunk, Radio, TV und Multimedia zusammen und unterstützen Telearbeit, verteilten Unterricht (Distant Learning), verbesserten Service in Verwaltung und Gesundheit, E-Commerce, Business-to-Business, Fun und Freizeit etc. .

Viel weniger offensichtlich finden sich Chips in „eingebetteten Systemen“, die im industriellen Bereich messen, steuern und regeln. Bei eingebetteten Systemen sind die enthaltenen Chips von außen nicht direkt erkennbar. In diesem Bereich gibt es kaum noch ein Gerät, das ohne digitale Komponenten arbeitet. Durch programmierbare Mikrokontroller und sogar programmierbare Hardware-Schaltungen in Geräten, Automaten, Robotern und Fließbändern wird die Automatisierung wesentlich flexibler als früher. Darüber hinaus lassen sich die Chips der Mikroelektronik immer häufiger mit winzigen Aktoren und Sensoren integrieren, man spricht dann von Mikrosystemtechnik.

In der Autoelektronik steuern die besagten einhundert Computer-Chips nicht nur die elementaren Antriebsfunktionen, Treibstoffverbrauch und Abgasreduktion, sondern erhöhen auch Sicherheit und Komfort durch Airbag, ABS, ESP, ABC, Abstandsradar (nicht: Radarwarnung), Autopilot, Bordcomputer, Sprachsteuerung etc..

Sport: Ski-Bindung, Klima-Jacke, Schiedsrichter im Fußball, Crosstrainer, Skipass, Fahrrad-Navi, **Mobile Geräte:** GPS-Navi-Handy, PDA-Handy, Funkkopfhörer, Ipod, James-Bond-Armbanduhr, Schwiegermutter-Radar, Scotty-Brille, **Haus:** Regensburger Cocktailmixer, fühlende Kaffeetasse, adaptiver Staubsauger, Staub-Weg-Laser, Haustürklingel, Radiowecker, Pantoffelkino, Toaster, Nähmaschine, Kaffeemaschine, Spülmaschine, Kühlschrank, Waschmaschine, Alarmanlage, Jalousien, Heizungssteuerung, Biometrisches Türschloss, Garagentor, Backofen mit Kochbuch, Schlüssel-Transponder, Solarkollektor, Gewächshaussteuerung, Aquarium, **Medizin:** Blutbahn-U-Boot, Künstliches Ohr, Künstliches Auge, Bionischer Arm, Neuroprothese, Herzschrittmacher, Chip-Implantat, Atemassistent, Lab-on-a-Chip, DNA-Chip, Biochip, ID-Chip, Gensequenzer, OP-Roboter, Teleoperation, Kernspintomograph, EKG, EEG, **Auto:** Keyless Go, Abstandsradar, Sekundenschlaf-Detektor, Vibrationslenkrad, Wegfahrsperre, Tempomat, Airbag, ABS, ESP, ABC, Chip-Tuning, Radarfalle, Sprach-Navi, Automatische Türverriegelung, Verkehrsleitsystem, **Musik / Unterhaltung:** Disco-Eintrittschip, Mixer, Keyboard, E-Piano, E-Gitarre, Effektgerät, Heimkino, **Displays:** E-Zeitung, Leuchtteppich, Head-Up-Display , **Reisen:** E-Flugticket, Fahrkartenautomat, RFID-Gepäckbanderole, Reisepass, Zoll-Durchleuchter, Live-Fahrplan, Eintrittskarten, Autopilot, **Justiz, Überwachung:** Fußfessel, Lügendetektor, GPS-Wanze, Telefonüberwachung, Überwachungskamera, Gesichts-Identifikation, Toll-Collect, Nummernschildverfolgung, Email-Überwachung, WM-Tickets, **Militär:** Marschflugkörper, Nachtsichtgerät, **Einkauf:** Diebstahlschutz-Etikett, Pfandrückgabeautomat, Paketausgabe, **Unfall- / Katastrophenschutz:** Brücken-Stahlermüdungs-Sensor, Erdbebensicherer Hochhaus-Schwingungsdämpfer, Wassertiefen-Sonar, Tsunami-Sensor, **Baustelle:** Lasermaßband, Theodolit, **Naturwissenschaft:** Animal-Tracking-GPS-Halsband, Tauchroboter, Spiegelteleskop, Digital-Fernglas

Sammlung 1.1 Weitere eingebettete Systeme

Schließlich sei der dramatisch wachsende Bereich von Bioinformatik und Medizintechnik erwähnt, der sich so dynamisch und vielfältig entwickelt, dass wir hier

gar nicht erst mit seiner Diskussion beginnen (minimal invasive Chirurgie, Kernspin-Tomographie, Bio-Chip, Neuroprothese uvam.). Eine weitere Auswahl eingebetteter Systeme zeigt Sammlung 1.1 – allerdings ohne Garantie für Bedeutung oder Markterfolg.

Auf etlichen der genannten Gebiete arbeitet die Abteilung E.I.S. der TU Braunschweig; „E.I.S." steht dabei für den „Entwurf integrierter Schaltungen" oder auch „Entwurf integrierter Systeme". So werden *adaptive Rechner* entwickelt, die zur Laufzeit mit dynamisch angepassten Chips den klassischen Universalrechner wesentlich effizienter machen.

Im Grenzbereich zwischen IuK und Messen, Steuern, Regeln liegt die *Home-Automation*, die von der Abteilung E.I.S. seit Jahren nicht nur erforscht, sondern auch produktreif entwickelt wird. So wurden zur EXPO 2000 mehrere Wohnungen in Gifhorn seniorenfreundlich ausgestattet. Zur Steigerung von Komfort, Sicherheit und Energieersparnis werden im *intelligenten Haus* potenziell alle Geräte vernetzt und automatisch oder interaktiv gesteuert.

Mit der System-Beschreibungssprache SystemC erforscht E.I.S. im *Hardware-Software-Codesign* die komfortable Modellierung ganzer *eingebetteter Systeme* aus einem Guss, bevor sie in komplexe Subsysteme zerlegt werden aus Standard-Software, die mit angepassten Spezial-Chips kommuniziert.

Für die Entwicklung komplexer Autoradios hat E.I.S. in Kooperation mit der Fa. Blaupunkt eine statechart-basierte Entwurfsmethodik entwickelt und getestet.

Chips überall — daraus ergibt sich die *wirtschaftliche Bedeutung* fast von selbst. Ist schon der deutsche Chip-Markt mit den erwähnten 9 Milliarden Euro Umsatz im Jahr 2004 beeindruckend, so ist er doch noch viel bedeutender als Basis- oder Schlüsseltechnologie für einen fast hundert mal so großen Markt an Produkten: allein in den Branchen Büro- und Datentechnik, Elektrotechnik, Maschinenbau, Fahrzeugbau sowie Feinmechanik und Optik sorgen die Chips der Informatiker und Ingenieure für 500 Milliarden Euro Umsatz und Jobs für etwa 3 Millionen Menschen [Bark02].

1.2 Das exponenzielle Wachstum der Mikroelektronik

Wie konnte es so weit kommen? — 1960 gelang es Jack Kilby erstmals, zwei Transistoren und ihre Verbindungsleitungen auf einer einzigen Halbleiterscheibe unterzubringen — die erste integrierte Schaltung. Heute dagegen lassen sich mehrere *Milliarden* Transistoren als aktive Schaltelemente auf einem einzigen Chip integrieren. Schon 1965 prophezeite der Intel-Mitbegründer Gordon Moore das wichtigste Wachstumsgesetz der Mikroelektronik: *nach dem Moore'schen Gesetz sollte sich die Komplexität integrierter Schaltungen jedes Jahr verdoppeln*.

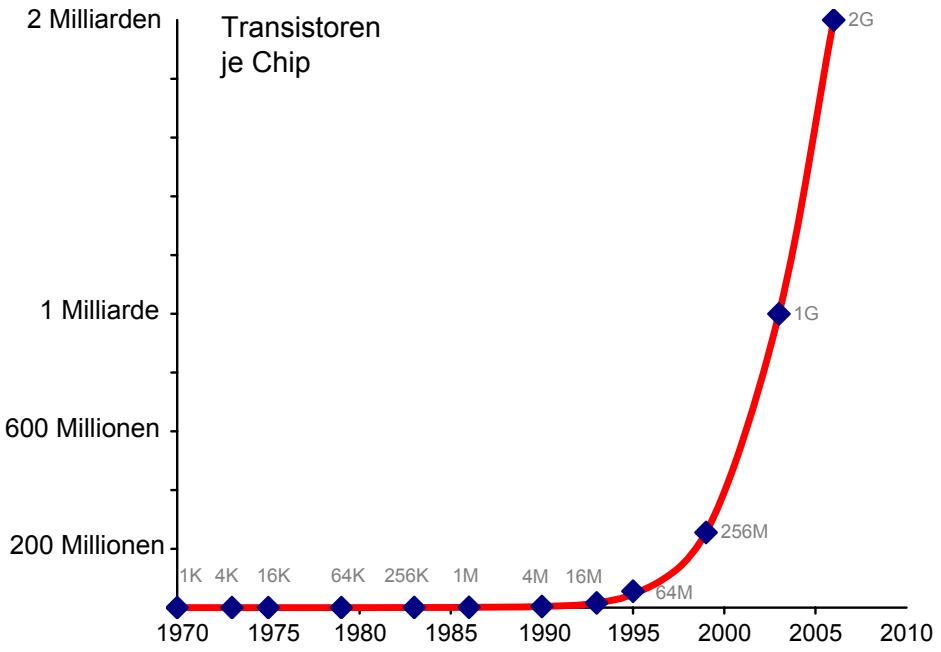


Bild 1.2 Exponenzielles Wachstum

Dieses Wachstum veranschaulicht Bild 1.2 anhand der Entwicklung der Speicherkapazitäten. Betreiben wir zunächst etwas Schulmathematik. Ganz harmlos klingt ein Wachstum von „plus 53%“. Statt dem harmlosen „plus“ handelt es sich jedoch um eine Multiplikation mit dem Faktor 1,53. Noch dramatischer wird es, wenn die Zunahme um „plus 53%“ jedes Jahr stattfindet: bereits nach zwei Jahren hat sich der Anfangswert weit mehr als verdoppelt ($1{,}53^2 \approx 2{,}3$), nach drei Jahren fast vervierfacht und schon nach fünf bis sechs Jahren verzehnfacht. Bei $1{,}53^n$ handelt sich offensichtlich um ein *exponenzielles Wachstum*[1].

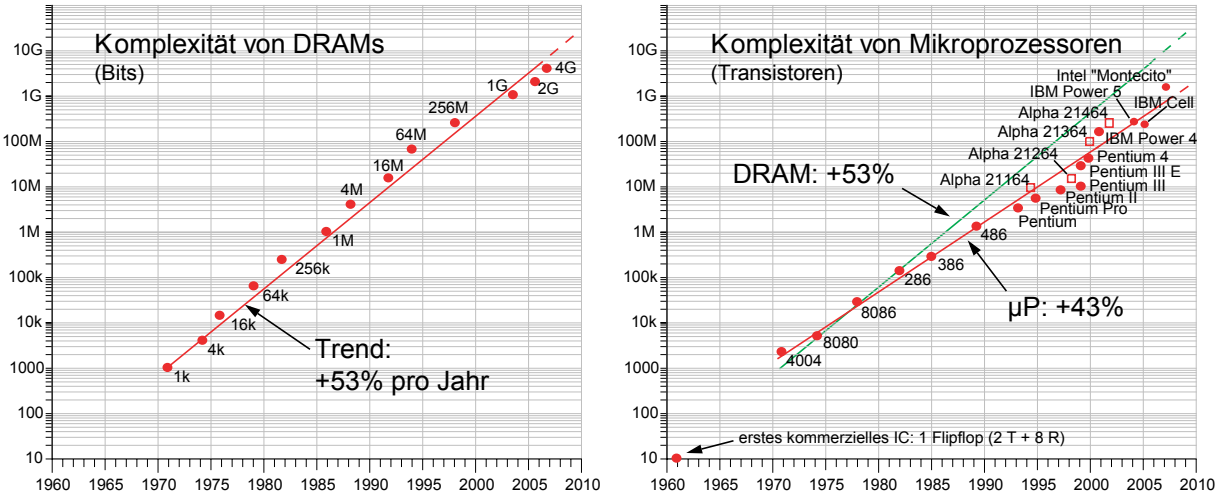


Bild 1.3 Chip-Komplexität

[1] Nebenbei bemerkt scheint die Kurve in Bild 1.2 erst ab 1995 zu „explodieren“, aber bei anderen Maßstäben der y-Achse würde die e-Funktion entsprechend früher „abheben“. Daher ist der logarithmische Maßstab in Bild 1.3 zwar weniger eindrucksvoll, aber brauchbarer.

Statt des Fachbegriffs „Very Large Scale Integration“ oder VLSI für den *Stand* der Technologie ist daher der *Trend* der Mikroelektronik noch charakteristischer, den man durch ALSI bezeichnen könnte: „Always Larger Scale Integration“.

Zwar hat Gordon Moore den jährlichen Zuwachs etwas überschätzt, aber ganz bestimmt hat er es sich nicht träumen lassen, dass sein Gesetz wie in Bild 1.3 beim Wachstum der DRAM-Speicher[2] 45 Jahre lang gelten würde und von einem Flipflop zu einem zwei Milliarden mal so großen 2G-Speicher führen würde. Während in Bild 1.3 das Speicherwachstum in Transistoren mit dem Wachsen in Bits praktisch übereinstimmt, sind die Mikroprozessoren „nur“ um jährlich durchschnittlich 43% gewachsen. Dieses Wachstum hat in letzter Zeit wieder zugenommen, da der größte Anteil eines modernen Prozessors ohnehin aus Cache-Speichern besteht.

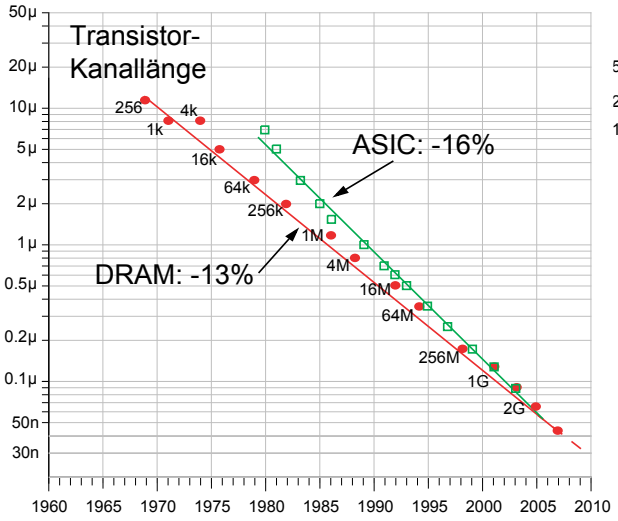


Bild 1.4 Transistorgröße

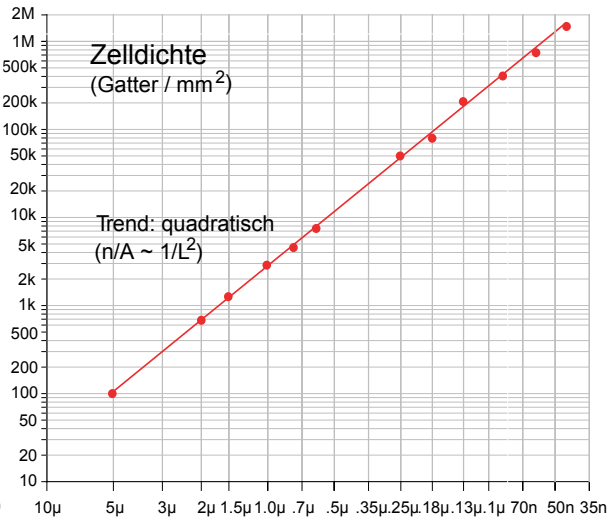


Bild 1.5 Gatterdichte

Ermöglicht wurde das exponenzielle Wachstum der Chip-Komplexität in erster Linie durch ständige atemberaubende Verbesserungen der Herstellungsprozesse, die insbesondere eine Verkleinerung der Transistorlängen L um jährlich 13% ermöglicht haben: in weniger als fünf Jahren ließen sich die minimalen Transistorlängen jeweils halbieren (Bild 1.4). Da auch die übrigen Abmessungen wie die der Metallverbindungen zwischen den Transistoren in erster Näherung gleich schnell geschrumpft sind, wurden die Strukturen in der Fläche jährlich kleiner um $0{,}87^2 = 0{,}75$, d.h. es passten jährlich $1/0{,}75 = 1{,}33$ oder 33% mehr Bauelemente auf eine Einheitsfläche. Zu einem sehr ähnlichen (theoretisch gleichen) Ergebnis kommt indirekt die Integrationsdichte in Bild 1.5 in Abhängigkeit von der Strukturgröße, nämlich einem jährlichen Trend von 35%. Verwendet werden heute Strukturen *weit im Sub-µ-Bereich (deep submicron)*, die in der Größenordnung von 45 Nanometern sind ($1\ \text{nm} = 10^{-9}\ \text{m}$).

---

[2] DRAM steht für Dynamic Random Access Memory, das sind Speicher, die ihre Information kontinuierlich (dynamisch) auffrischen müssen.

Während Speicher und Mikroprozessoren in höchsten Stückzahlen vermarktet werden, interessieren wir uns auch für den Entwurf maßgeschneiderter oder kundenspezifischer Schaltungen (Application-Specific Integrated Circuits, ASICs). Bild 1.4 zeigt, dass solche ASICs früher auf älteren Technologien gefertigt wurden, während sie heute auch in jeweils modernster Technologie zur Verfügung stehen.

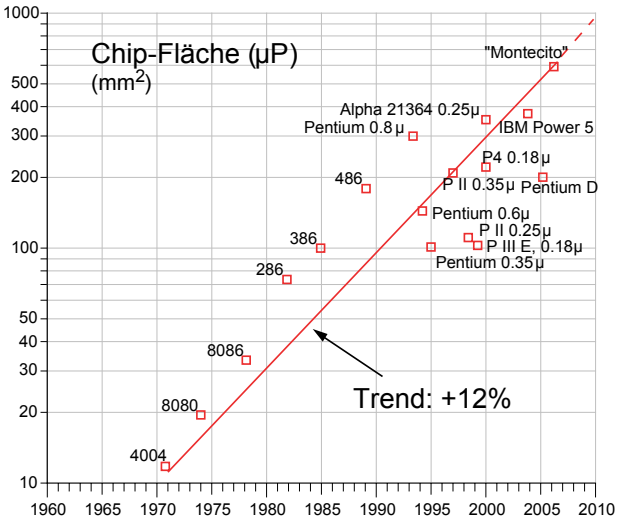


Bild 1.6 Chip-Fläche

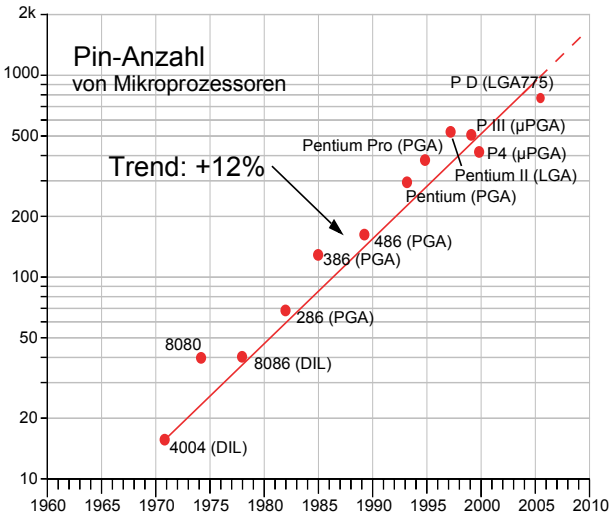


Bild 1.7 Anschlüsse

Ein weiterer wichtiger Faktor war die Vergrößerung der Chip-Flächen bei Mikroprozessoren (Bild 1.6), die trotz der höheren Dichte pro Fläche von 10 auf heute etwa 600 mm^2 angestiegen ist — ein Meisterwerk fehlerfreier Fertigungstechnologie!

Der jährliche Anstieg der Dichte um 35% in Bild 1.5 ergibt zusammen mit diesem 12%igen Flächenwachstum fast die ursprünglich genannte Komplexitätszunahme um 53%. (Es sind hier noch andere nicht genannte Faktoren im Spiel.)

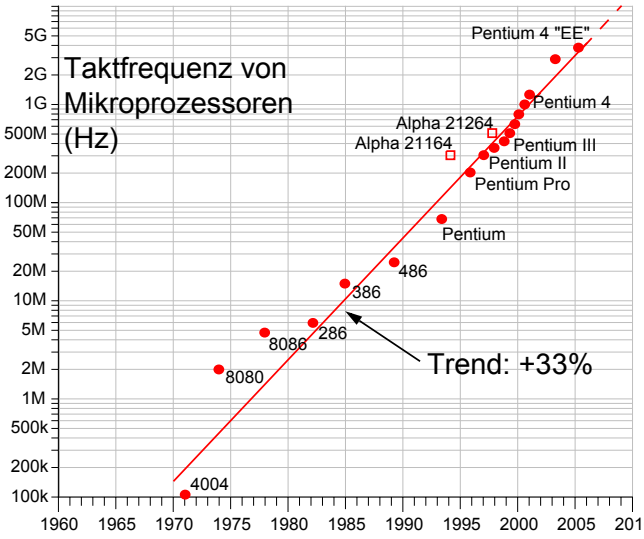


Bild 1.8 Taktung

Auch die Anzahl der Chip-Anschlüsse oder Pin-Anzahl ist bisher exponenziell gewachsen: bei Mikroprozessoren auf heutzutage bewundernswerte etwa 1.000 Pins (Bild 1.7), bei beliebigen Schaltungen sogar noch stärker. Da dieses Wachstum jedoch jährlich nur 12% betrug, während der Chip-Inhalt mit 53% wuchs, entstand der Flaschenhals, dass heute immer weniger Informationen im Chip von außen zugänglich sind.

Die immer kleineren Transistoren in Bild 1.4 führen übrigens nicht nur zu einer höheren Packungsdichte, sondern sogar zu quadratisch kürzeren Laufzeiten des einzelnen Transistors und damit des gesamten Systems. Dies führt in Bild 1.8 zum bekannten Wettlauf der Taktfrequenzen (auch wenn Rechnerarchitekten nachweisen, dass diese Frequenzen allein noch nicht zu entsprechend schnelleren Rechenleistungen führen müssen).

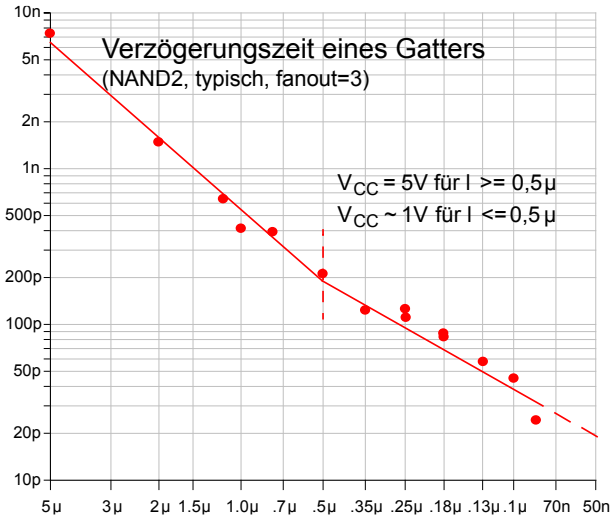


Bild 1.9 Gatter-Schaltzeit

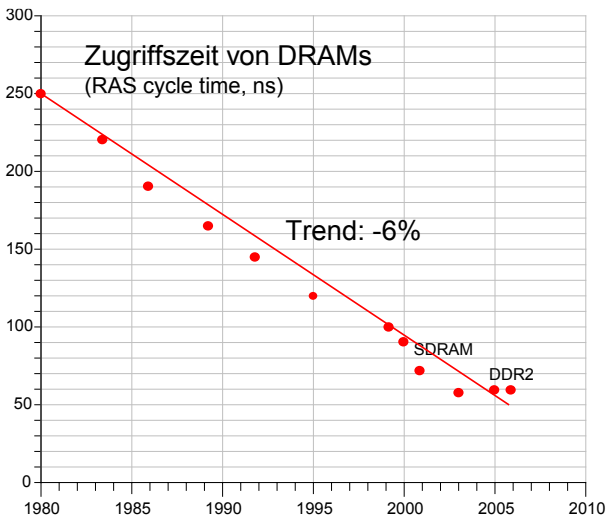


Bild 1.10 Speicher-Zugriffszeit

Aus den kürzeren Transistoren resultieren unmittelbar die schnelleren Gatter aus Bild 1.9, während sich die Speicher-Zugriffszeiten in Bild 1.10 im Wesentlichen nur um müde 6% jährlich verbessert haben — ein weiterer Flaschenhals zwischen Chip und Speicher wird sichtbar.

Schließlich werfen wir noch einen Blick auf die wirtschaftliche Seite. Die Kosten einer modernen Chip-Fabrik sind nach Bild 1.11 auf astronomische 5 Milliarden Dollar gestiegen (Faktor 9 alle 10 Jahre), was sich auch in steigenden Chip-Herstellungskosten in Bild 1.12 niederschlägt. Ein Kunde wird heute abgeschreckt durch zwei Millionen Dollar, die für den ersten Prototypen in Maximalgröße fällig sind. (Natürlich werden auch kleinere und ältere Technologien wesentlich kostengünstiger angeboten.) Trotz steigender Gesamtpreise sind jedoch in Bild 1.13 die Speicherkosten pro gespeichertes Bit dramatisch gefallen. Glücklicherweise werden wir in den programmierbaren Schaltungen eine kostengünstige Alternative für die Prototypenfertigung kennenlernen.

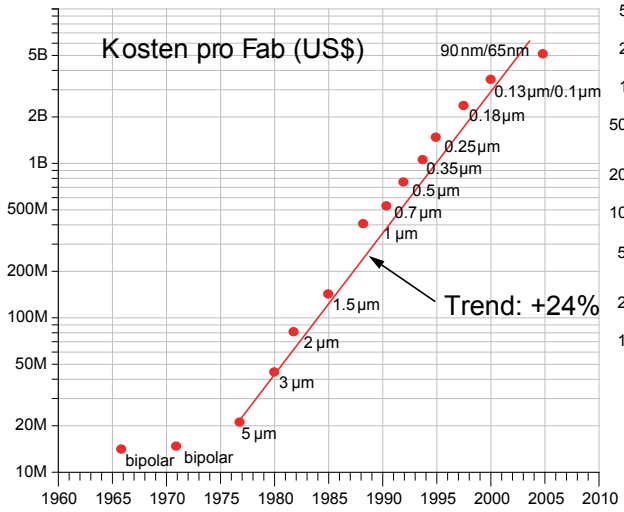


Bild 1.11 Kosten einer Chip-Fabrik

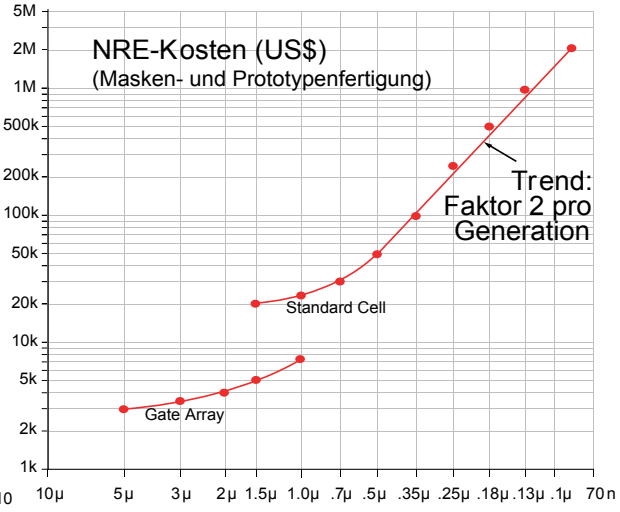


Bild 1.12 Kosten eines Chip-Musters

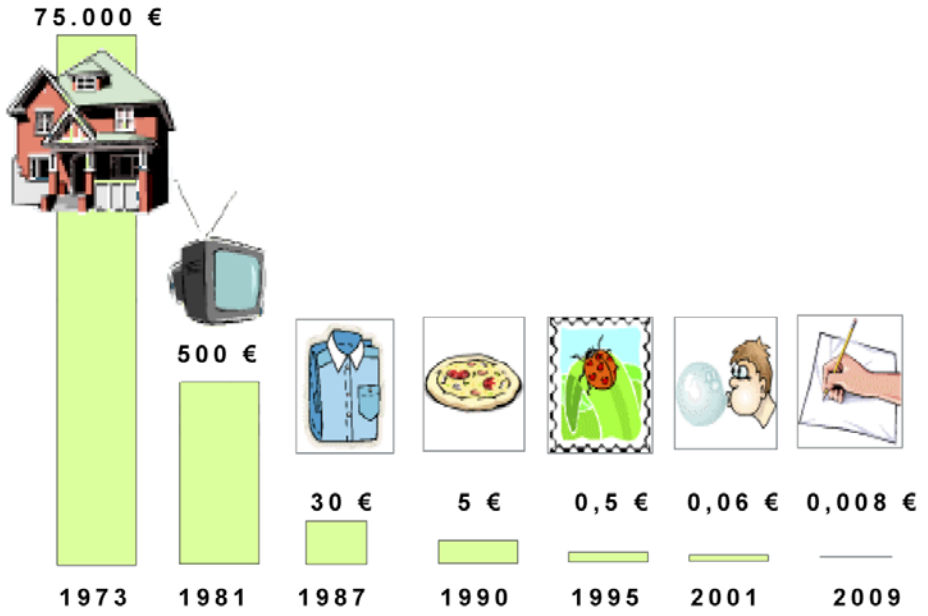


Bild 1.13 Kosten für 1 MBit DRAM[3]

Übrigens ist Bild 1.13 [Bark02] entnommen, während sich die übrigen Bilder und Daten bis 2001 in [Baue02] finden und dann durch eigene Recherchen fortgeschrieben wurden.

Im Folgenden betrachten wir zunächst den klassischen Hardware-Entwurf (Abschnitt 1.3) und dann im Hardware-Software-Codesign die Koppelung mit normaler Software auf Standardrechnern sowie die moderne Möglichkeit, alles wiederum auf einem einzigen System-on-Chip unterzubringen (Abschnitt 1.4).

[3] Die Eigenheimkosten wären heutzutage wesentlich höher und der Preisverfall damit noch krasser.

1.3 Der Hardware-Entwurf

Wir wenden uns jetzt dem Entwurf oder Design von derart mächtigen Schaltungen zu. Während kein vernünftiger Mensch Puzzles mit mehr als 1.000 Teilen lösen wird, steht der heutige Designer vor der Aufgabe, ein Puzzle mit bis zu einer Milliarde Teilen zusammenzusetzen, wobei jeder noch so kleine Fehler zu einem Schaden von einer Million Dollar führt. Sind härtere Spielregeln denkbar?

1.3.1 Chip-Entwurf leicht(er) gemacht

Bevor jedoch auch der letzte Interessent für unser Gebiet deprimiert aufgibt, wollen wir in einer wesentlich optimistischeren Sicht plausibel machen, warum der Entwurf eines normalen digitalen Chips gar nicht so schwierig und schnell erlernbar ist. Wir wollen uns also Mut machen, dass der Chip-Entwurf trotz explodierender Komplexitäten relativ leicht und ohne allzu viele technische Spezialkenntnisse erfolgreich durchgeführt werden kann.

Entkoppelung von Entwurf und technischer Realisierung

Der jahrzehntelange exponenzielle Anstieg der Integrationsdichte war keineswegs nur eine Last. Denn ähnlich wie bei der Entwicklung der Rechner oder Programmiersprachen entstanden unter dem Druck steigender Komplexität zunächst in der Forschung und dann in der Praxis allgemeinere, abstraktere und höhere Denkweisen und Prinzipien, die letztlich zu besseren und einfacheren Lösungsstrategien geführt haben. So ermöglichten schnellere und größere Rechner und Speicher den an sich verschwenderischen, aber komfortablen und fehlerarmen Einsatz höherer Programmiersprachen und ihrer Compiler, deren Existenzberechtigung niemand mehr bezweifelt.

Beim Chip-Entwurf fand eine verwandte und ähnlich aufregende Entwicklung statt, indem sich die *Schnittstelle zwischen Entwurf und technischer Realisierung*, also dem Chip-Designer und dem Halbleiterhersteller, der die Schaltung des Designers ohne Kenntnis ihrer Logik in Silizium umsetzt, immer mehr „nach oben“ verschob und immer einfacher wurde. Zu Beginn der Entwicklung lag der Chip-Entwurf in der Hand sehr erfahrener Ingenieure, die vor allem die grundlegende Halbleitertechnologie mit einer Fülle von Regeln beherrschen mussten. Auch heute noch werden diese Fachleute bei der Entwicklung immer leistungsfähigerer Technologien oder beim Entwurf analoger Schaltungen benötigt. Gleichwohl ist es seit längerem problemlos möglich, leistungsfähige digitale Chips zu entwerfen, ohne die Transistoreigenschaften überhaupt zu kennen. Statt dessen wird eine Lösung ziemlich abstrakt und verständlich in einer Hardware-Beschreibungssprache formuliert. Den Rest erledigen mächtige

CAD-Werkzeuge

Extrem große Zahlen von Bauelementen, mehr oder weniger abstrakt dargestellt, und ihre dynamischen Abhängigkeiten können nur durch den massiven Einsatz von CAD-Werkzeugen („CAD-Tools“) menschenwürdig verarbeitet werden. Wir unterscheiden statische CAD-Software zur Beschreibung und Verwaltung der Schaltungen und dynamische CAD-Software zur Simulation ihres zeitlichen Verhaltens. Die Fülle heute üblicher CAD-Werkzeuge kann hier kaum angedeutet werden.

Es beginnt mit graphischen oder textuellen *Editoren*, die die Eingabe, Verwaltung und Änderung der Schaltungsdaten ermöglichen. Diese Editoren sind mit anderen Werkzeugen integriert. So gibt es Werkzeuge zur *Überprüfung* von elektrischen oder logischen Entwurfsregeln. Es gibt *Übersetzer* von höheren auf tiefere Entwurfsebenen (*Synthese*, *Silicon-Compiler*). Es gibt Werkzeuge, die einen Teil der Entwurfsarbeit automatisieren wie beispielsweise zur *Platzierung und Verdrahtung*.

Im Bereich der dynamischen Werkzeuge sind vor allen Dingen *Simulatoren* zu nennen, die das zeitliche Verhalten einer Schaltung berechnen. Durch diese ziemlich exakten Vorhersagen *vor* der Fertigung kann ein relativ teures Redesign vermieden werden. „First time right“ ist das Ziel, bei dem ein Entwurf nur einmal gefertigt werden muss.

Im Zentrum des modernen Hardware-Entwurfs aber stehen zweifellos *Hardware-Beschreibungssprachen* als Grundlage für Simulation und Synthese auf vielen Entwurfsebenen. Sie werden in den Kapiteln 2 bis 4 eingeführt.

Weitere Erleichterungen

Weitere Erleichterungen beim Entwurf großer Schaltungen und Systeme sind ein hierarchisches Vorgehen (Entwurfshierarchie und Zerlegungshierarchie) und das Geschenk programmierbarer Logikbausteine, die unter anderem einen frühen Versuchsaufbau (Rapid Prototyping) unterstützen. Diese Faktoren sind so wichtig, dass sie jetzt in eigenen Abschnitten eingeführt werden.

1.3.2 Hierarchische Vorgehensweise

Ein weiteres wichtiges Hilfsmittel zur Bewältigung der enormen Chip-Komplexität ist wie so oft in der Informatik die hierarchische Vorgehensweise. Hier unterscheiden wir die *Entwurfshierarchie* und die *Zerlegungshierarchie*. Die Entwurfshierarchie[4] besteht aus mehreren *Abstraktionsebenen* oder *Sichten*. Im Software-Engineering etwa haben wir Ebenen wie Spezifikation, graphische Flussdiagramme, höhere Programmiersprachen, Assembler usw. . Im Bereich des Chip-Entwurfs bestehen die Sichten darin, den Entwurf funktional durch sein Verhalten „nach außen“ darzustellen oder sehr grob

[4] Der Begriff „Hierarchie“ ist nicht besonders glücklich gewählt, da keine baumartige Struktur vorliegt.

auf einer Systemebene oder auf einer Register-Transfer-Ebene oder mit Hilfe von Logikgattern und Registern oder durch Transistoren oder durch ein Layout als maßstabsgetreue Vorlage für die tatsächlichen Chip-Strukturen (Bild 1.14).

Entwurfsebene	**Beschreibung**	**typische Komponenten**
funktionale oder Verhaltensebene	*Was* wird geleistet? Nicht: *Wie* wird es erreicht?	—
Systemebene	grobe Aufteilung des Systems (grobe Struktur, Zeit, Daten, Kommunikation)	CPU, FPGA, DRAM, Bus

Register-Transfer-Ebene (RTL)	synchrone Datenverarbeitung zwischen Registern
	logischer Takt
		Register
		kombinatorische Logik
		`always @(posedge CLOCK)`
		`  R2 <= f1(R1);`

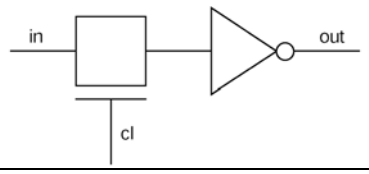
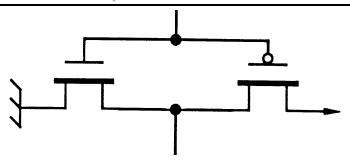
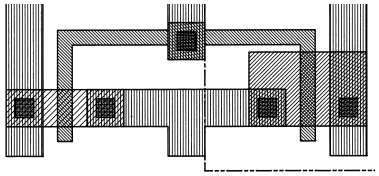
Logik- oder Gatterebene	logische Netzliste aus Gattern, Flipflops etc. geschätzter realer Takt	
Transistorebene	elektrischer Schaltplan genauer Takt, jedoch ohne Leitungsverzögerungen	
Layout-Ebene	maßstabsgetreues Abbild des späteren Chips auf allen Ebenen genauer Takt mit Leitungsverzögerungen	

Bild 1.14 Entwurfsebenen der Abstraktionshierarchie

Auf jeder Entwurfsebene kann es dann eine Zerlegungshierarchie im Sinne der schrittweisen Verfeinerung geben, wo ein Problem ähnlich wie bei der modularen Programmierung zunächst in Teilprobleme zerlegt wird, die wieder verfeinert werden usw. (siehe unten).

Um in dieser Einleitung ein erstes Gefühl für den Entwurfsprozess zu bekommen, wollen wir uns die in Bild 1.14 erwähnten Abstraktionsebenen oder Sichten der Entwurfshierarchie etwas genauer ansehen.

Verhaltensebene

Wie im Software-Engineering geht es zunächst einmal darum, die *Anforderungen* oder die *Spezifikation* oder das *Verhalten* eines Systems oder einer Schaltung oder eines Teils davon nach außen zu beschreiben. Dabei interessiert nur das *Was* und kaum das *Wie*. Bild 1.15 zeigt die funktionale Beschreibung eines einfachen dynamischen

Schieberegisters in einer Hardware-Beschreibungssprache. Es besteht eine offensichtliche Ähnlichkeit zu höheren Programmiersprachen. Der wesentliche Unterschied besteht darin, dass *paralleles* Arbeiten sehr intensiv unterstützt wird und dass hardware-nahe Strukturen wie „Wires“ oder Register besonders berücksichtigt sind. Ohne das Verhalten in Bild 1.15 im einzelnen analysieren zu wollen, sei hier nur auf die Anweisung `always` hingewiesen, die einen Prozess zur Beobachtung des Taktes `CLOCK` startet, der dann parallel zu anderen Prozessen ständig ausgeführt wird.

```
module SHIFT_REGISTER (     // invertiert und gibt es verzoegert aus
  input  wire IN, CLOCK,    // Eingangssignal und Takt (Draehte)
  output reg  OUT           // Ausgangssignal (speicherndes Register)
);

always @(posedge CLOCK)     // immer, wenn CLOCK von 0 auf 1 wechselt
  OUT <= #9 ~IN ;           // invertieren und 9 Zeiteinheiten verzoegern

endmodule
```

Bild 1.15 Funktionale Beschreibung eines Schieberegisters in der Hardware-Beschreibungssprache VERILOG

Natürlich rechtfertigt ein Schieberegister meist keine funktionale Beschreibung. Diese wird vor allem bei großen Problemen bedeutsam, deren Anforderungen und Lösungen zunächst einmal nur durch normale Algorithmen beschrieben werden. Obwohl im strengen Sinne einer Verhaltensbeschreibung keine internen Strukturen zugelassen sind, werden bei funktionalen Algorithmen natürlich auch Unterprogramme, Variablen und ähnliche strukturelle Hilfsmittel benutzt; diese brauchen jedoch noch in keiner Weise die spätere Hardware-Struktur wiederzuspiegeln. Natürlich lassen sich solche Verhaltensbeschreibungen auch *simulieren*, was der Ausführung eines normalen Programms entspricht. Außerdem stellen sie eine präzise *Dokumentation* dar.

Systemebene

Auf dieser Ebene werden die großen Komponenten eines Systems wie Steuerwerk, Speicher oder Bus beschrieben (Bild 1.16), im Hardware-Software-Codesign auch ganze Einheiten wie ein JPEG-Encoder, Kartenschreiber und Ähnliches (Bild 1.17). Die meisten Details bleiben auf der Systemebene verborgen bzw. sind ja zu Beginn des Entwurfs noch gar nicht bekannt. Ausgewählte Eigenschaften werden recht grob modelliert: Struktur, Kommunikation, Zeit, Daten oder Algorithmen. Für Systemmodelle gibt es besonders geeignete System-Beschreibungssprachen wie die objektorientierte C++-Erweiterung SystemC, die wir in späteren Vorlesungen einsetzen werden.

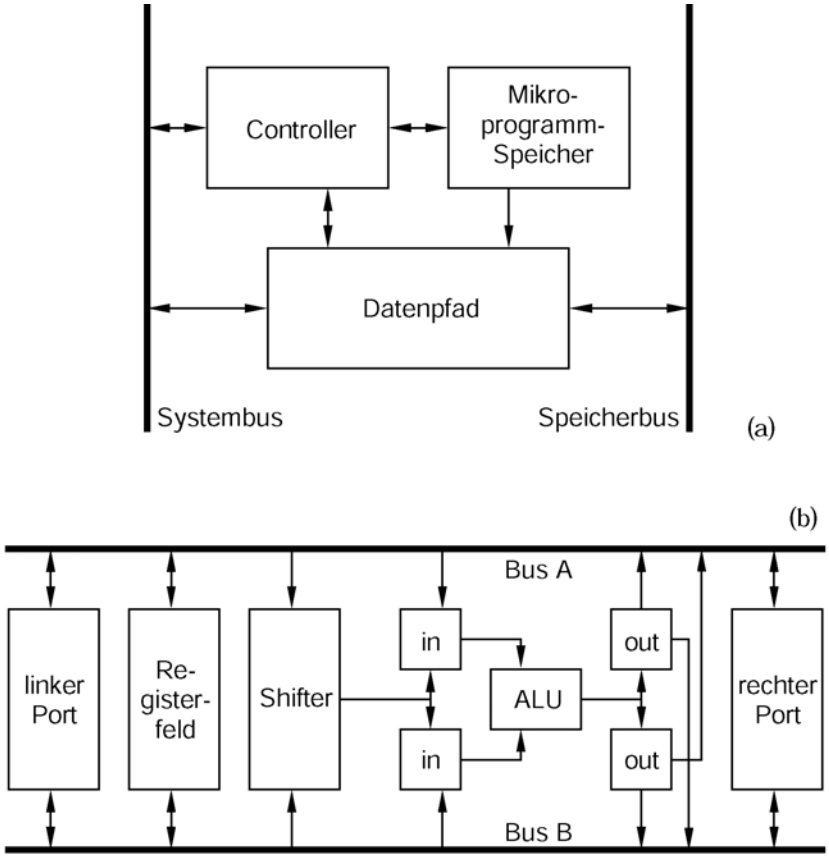


Bild 1.16
Systemebene:
(a) Grobstruktur eines Systems;
(b) Grobstruktur des Datenpfades aus (a)

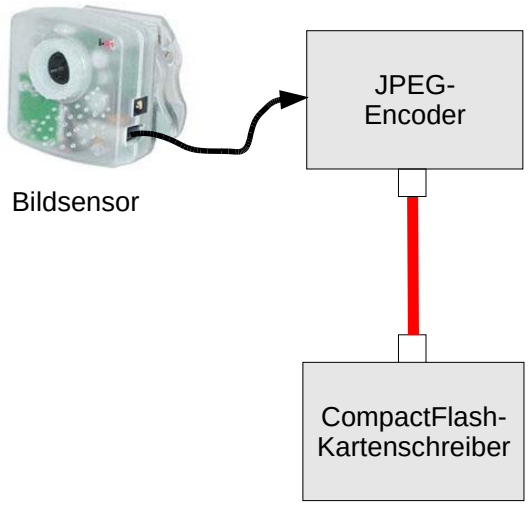


Bild 1.17
Einfaches System-Modell einer Digitalkamera

Register-Transfer-Ebene (RTL-Ebene)

Auf dieser Ebene findet der klassische Hardware-Entwurf statt. Wie der Name bereits andeutet und Bild 1.18 skizziert, werden Register R_1,… betrachtet, zwischen denen eine Datenverarbeitung durch kombinatorische Netze f_1,… erfolgt. Die Register werden durch einen *Takt* `CLOCK` synchron ausgelesen, so dass die Information wie in einer *Pipeline* schrittweise und parallel weiterverarbeitet wird. Bekanntlich arbeiten

Pipelines parallel, indem hier alle kombinatorischen Netze gleichzeitig aktiv sind, was den Durchsatz erheblich erhöht. Auch hier ist eine Hardware-Beschreibung wie in Bild 1.19 praktisch.

Übrigens lassen sich RTL-Strukturen wie in Bild 1.18 durch Rückkopplungen erweitern zu beliebigen Automatennetzen.

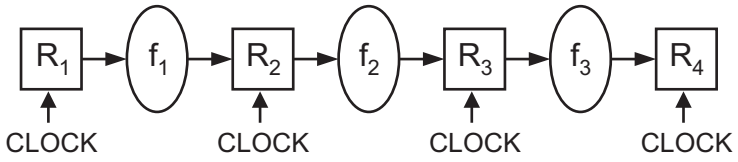


Bild 1.18 Register-Transfer-Logik

```
• • •

always @(posedge CLOCK)  // mit jeder steigenden Taktflanke
begin
  R2 <= f1(R1);          // Register-Transfer von Ri durch fi nach Ri+1
  R3 <= f2(R2);
  R4 <= f3(R3);
end

• • •
```

Bild 1.19 Register-Transfer in der Hardware-Beschreibungssprache VERILOG

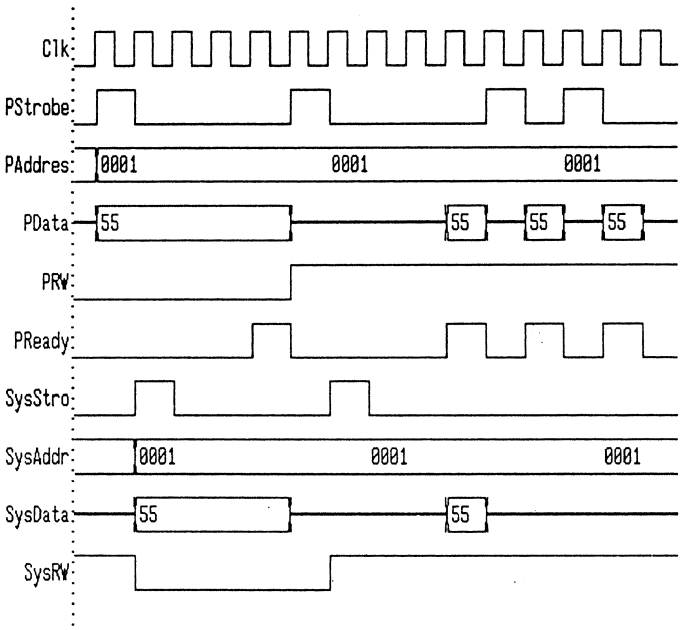


Bild 1.20 RTL-Simulation

Bild 1.20 zeigt eine Möglichkeit, Simulationsergebnisse zu einem RTL-Modell grafisch darzustellen. Das derart getestete Modell wird anschließend meist mit einer Logiksynthese umgesetzt auf die

Logik- oder Gatterebene

Auf dieser Ebene werden Schaltungen durch Netze wie in Bild 1.21 aus den bekannten Gattern AND, OR, Inverter usw. beschrieben. Auch solche Gattermodelle können mit einer Hardware-Beschreibungssprache wie VERILOG simuliert und verifiziert werden. Gattermodelle fassen wir meist jedoch nur mit spitzen Fingern an, denn normalerweise werden sie automatisch aus unserem RTL-Modell durch einen Logiksynthese-Compiler erzeugt.

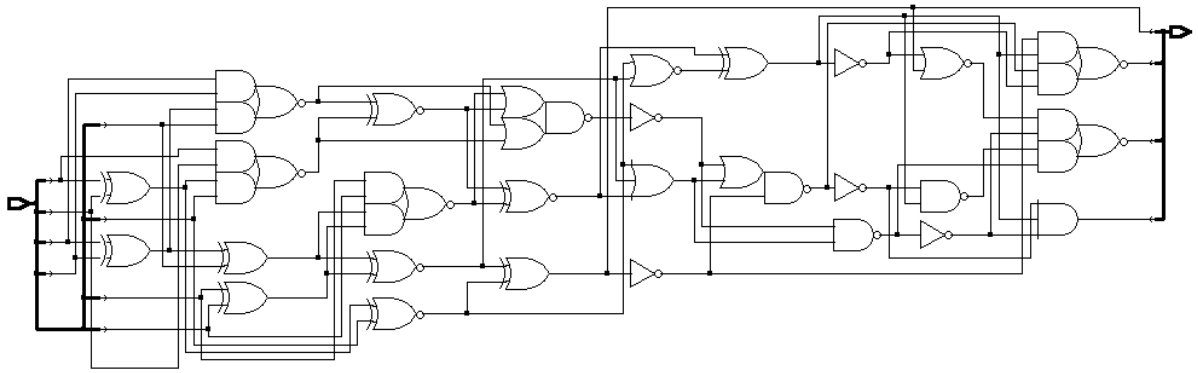


Bild 1.21 Gattermodell

Gattermodelle werden im Entwurfsprozess auf drei Arten weiterverarbeitet. Im *halbkundenspezifischen* oder *Semi-Custom-Entwurf* wird entweder das Gattermodell an den Halbleiterhersteller übergeben, der die Gatter durch (meist geheim gehaltene) Transistorschaltungen seiner Technologie-Bibliothek ersetzt, also ein Transistormodell der nachfolgenden Schaltkreisebene anfertigt, und dieses dann platziert und verdrahtet. Oder wir platzieren und verdrahten selbst, um das Layout für eine programmierbare Schaltung zu erhalten (Abschnitt 1.3.3); auch das wird noch dem Semi-Custom-Entwurf zugerechnet. Nur bei Standardprodukten mit hohen Stückzahlen lohnt es dagegen für den Designer, im *vollkundenspezifischen* oder *Full-Custom-Entwurf* bis hinunter zur Layout-Ebene alles selbst zu optimieren und erst dann den Halbleiterhersteller zu rufen.

Schaltkreisebene

Hier finden sich die seit jeher bekannten Schaltpläne aus Transistoren, Widerständen und Kondensatoren, wobei die Transistoren mit Abstand die häufigsten Bausteine sind. Ob man nun den Transistor und seine partiellen Differentialgleichungen liebt oder

lieber einen Bogen darum macht – wir kommen hier überhaupt nicht mit den Transistoren in Berührung, schon weil der Lieferant einer Technologie-Bibliothek die Transistor-Realisierungen seiner Bausteine in aller Regel geheim hält. Auf der Schaltkreisebene versagen auch VERILOG-Simulatoren, statt dessen produzieren Analog-Simulatoren Ergebnisse wie in Bild 1.22

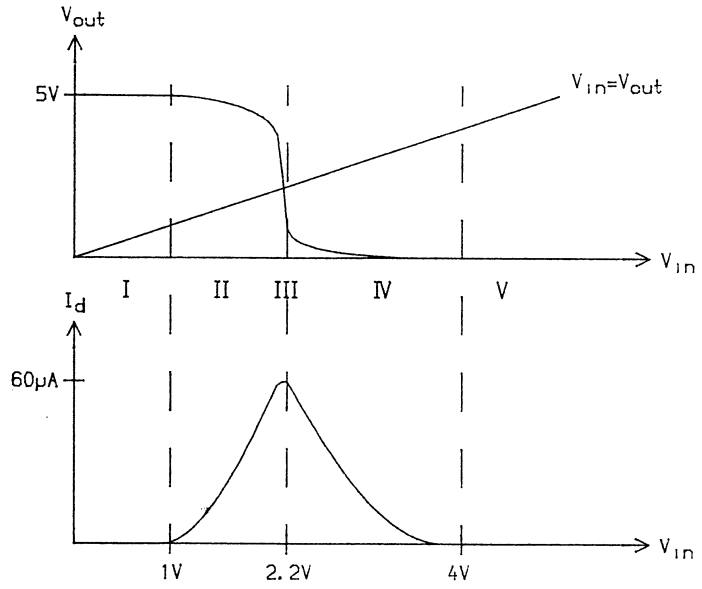


Bild 1.22
Analoge Simulation eines Inverters

Layout-Ebene

Ein Chip enthält übereinander mehrere elektrisch leitende und im Wesentlichen voneinander isolierte Schichten. Auf jeder Schicht können im *Layout* Leitungen gewisser Breite entworfen werden, die elektrische Signale leiten. In Bild 1.23 sind in Blau Leitungsstücke einer Metallebene dargestellt (in Wirklichkeit gibt es zahlreiche Metallebenen übereinander). Neben Metall gibt es rote Leitungen der Schicht *Poly(silizium)* und in Hellgrün bzw. Grün n- bzw. p-leitende Strukturen der Schicht *Diffusion*. Weiter gibt es durch schwarze Quadrate angedeutete *Kontakte*, die beispielsweise einen blauen Metallleiter mit einem grünen Diffusionsleiter verbinden.

Die größte Bedeutung haben jedoch rote Leiter, die einen grünen Leiter kreuzen. An dieser Kreuzungsstelle entsteht ein *n*- bzw. *p-Transistor*. Wie ein Transistor funktioniert und wieso die sehr schlichte Darstellung einer *rot-grünen Kreuzung* für einen Transistor sinnvoll ist, brauchen wir hier nicht zu verstehen.

Ein Layout ist nichts anderes als eine maßstabsgetreue Vergrößerung der schichtenweisen Strukturen im fertigen Chip. Bis auf Fertigungstoleranzen und einen wohldefinierten Verkleinerungsmaßstab wird aus einem blauen Rechteck der Größe 3x12 eine rechteckige Metallstruktur beispielsweise der Größe 90 nm x 360 nm.

Solche Layouts sind das Ergebnis von Platzierungs- und Verdrahtungswerkzeugen, die nach den Vorgaben eines Gatter- oder Schaltkreismodells arbeiten,

dabei Transistoren und Leitungen durch passende geometrische Abmessungen elektrisch sinnvoll dimensionieren, zahlreiche elektrische und geometrische Regeln einhalten und vor allem für eine Chip-Fläche und kurze Verbindungsleitungen sorgen.

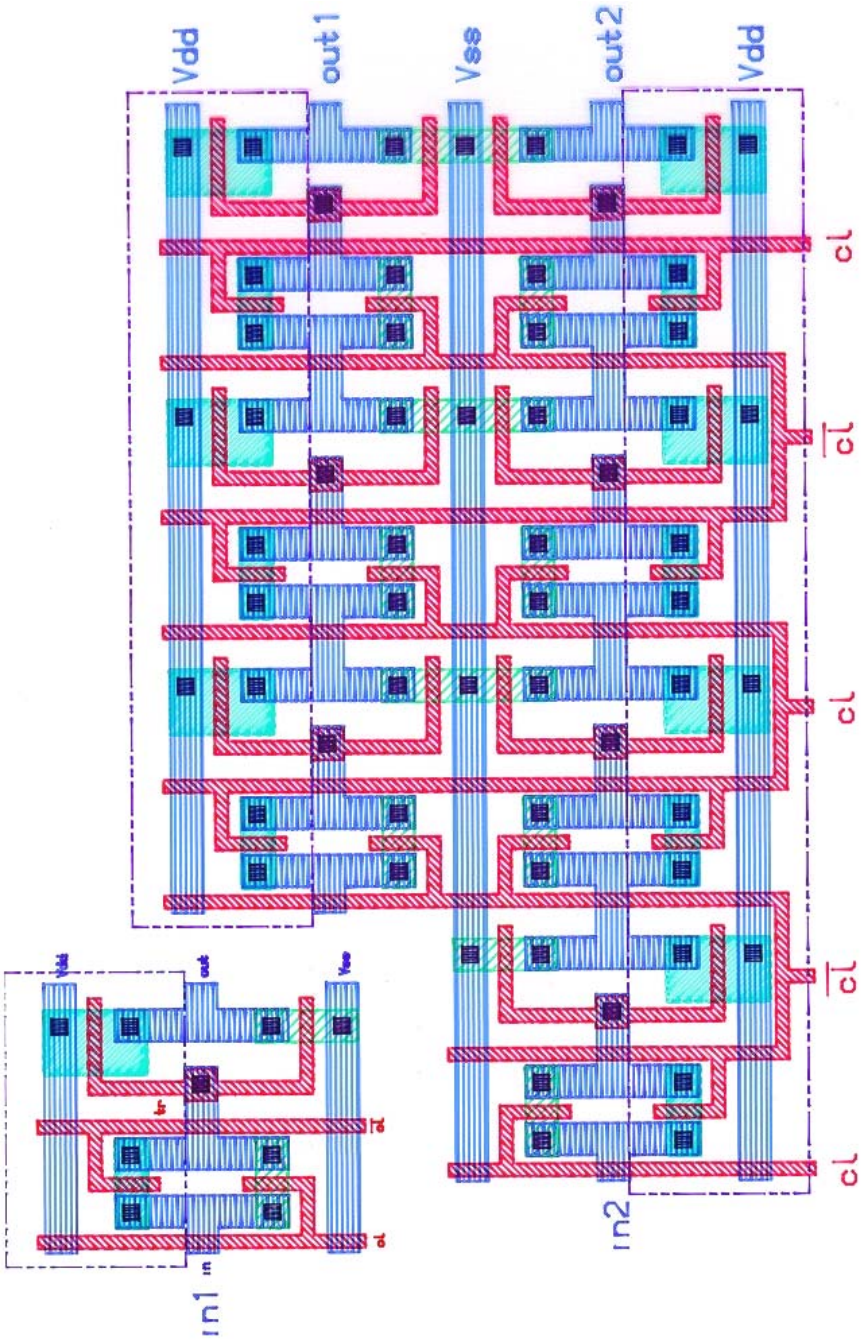


Bild 1.23
Layout eines Schieberegisters

Das geometrische Layout wird aus der graphischen Beschreibungsform in eine textuelle Form übersetzt. Daraus wird das so genannte Maskenband erstellt, das die Herstellung von Masken steuert, die mit Foto-Negativen vergleichbar sind. Diese Masken steuern dann die eigentliche Chip-Fertigung durch photolithographische und andere Methoden. Diese Fertigung gehört zu den präzisesten Produktionsprozessen überhaupt.

Anschließend werden die Chips in ein Gehäuse verpackt und mit Außenanschlüssen verdrahtet.

Zerlegungshierarchie

Wie bereits angedeutet, ist es auf jeder Entwurfsebene sinnvoll, größere Strukturen hierarchisch zu organisieren, indem sie schrittweise in Unterstrukturen zerlegt werden. Auf der funktionalen Ebene ähnelt diese Zerlegung sehr der hierarchischen Gliederung eines Programms durch beispielsweise Klassen oder abstrakte Datentypen. Auf der funktionalen Ebene interessiert jedoch nur das Verhalten, also die Antworten einer Komponente am Ausgang auf Änderungen am Eingang. Hierarchische Unterstrukturen sind nur praktische Hilfsmittel und brauchen dabei nicht den tatsächlichen Hardware-Strukturen zu entsprechen.

Auf der Systemebene lässt sich der Flurplan in Bild 1.16a verfeinern, indem beispielsweise der Datenpfad in Teilbild b in die Komponenten Register, ALU und Shifter sowie Ports zerlegt wird. Register-Transfer-Modelle lassen sich durch Untermodule hierarchisch gliedern mit der Absicht, diese Gliederung in einer späteren physikalischen Realisierung zu übernehmen.

Netzlisten aus Gattern oder Transistoren können Millionen von Strukturelementen enthalten. Hier ist es zwangsläufig notwendig, durch eine hierarchische Gliederung die Komplexität in den Griff zu bekommen. Dasselbe gilt für die Layout-Ebene, wo größere Zellen in Unterzellen zerlegt werden.

Ein interessanter Spezialfall dieser Zerlegung in Teilzellen ist die *reguläre Vervielfältigung*. Allgemein bekannt ist die geometrische Regularität von Speicherstrukturen, wo in erster Näherung nur eine Speicherzelle wirklich entworfen wird und dann in beiden Richtungen regulär vervielfältigt wird. Interessanterweise beschränken sich reguläre Strukturen aber nicht auf den Speicherentwurf, sondern es lassen sich viele Aufgaben ganz oder teilweise regularisieren.

In Bild 1.23 ist das Layout einer 1-Bit-Schieberegisterzelle in beiden Richtungen vervielfältigt, nämlich einmal in y-Richtung und zweimal in x-Richtung. So entsteht ohne wesentlichen zusätzlichen Design-Aufwand ein Schieberegisterfeld, bei dem zwei Bit breite Worte über mehrere Stufen hinweg getaktet verschoben werden können. (Offensichtlich wird damit noch keine mathematisch interessante Funktion realisiert, aber solche Schieberegister sind die Grundlage für beliebige RTL-Strukturen)

Natürlich sollten alle CAD-Werkzeuge die Hierarchie der bearbeiteten Strukturen unterstützen und ausnützen.

1.3.3 Programmierbare Hardware und Rapid Prototyping

Unter Rapid-Prototyping versteht man die Möglichkeit, während der Entwicklung komplexer Systeme frühzeitig die Funktion des Systems praktisch testen zu können. Während Simulatoren gerade bei Schaltungen meist nur einen kleinen zeitlichen Ausschnitt des Verhaltens testen, gibt es beim praktischen Test zwischen dem einen Extrem der teuren Fertigung eines Chips und dem früher eingesetzten anderen Extrem eines Versuchsaufbaus mit diskreten Komponenten (TTL-Gatter etc.) heute einen glücklichen Kompromiss in Form von FPGAs (Field-Programmable Gate-Arrays).

In diese physikalisch fertigen Logikbausteine lässt sich beliebig oft und sofort eine völlig individuelle Schaltung „hineinprogrammieren“. Zwar sind sie bei großen Stückzahlen weniger wirtschaftlich und an Geschwindigkeit und Komplexität gefertigten Chips unterlegen, für viele mittlere Anwendungen reichen sie aber völlig aus und sind gerade beim schnellen Ausprobieren oder Rapid Prototyping unentbehrlich geworden.

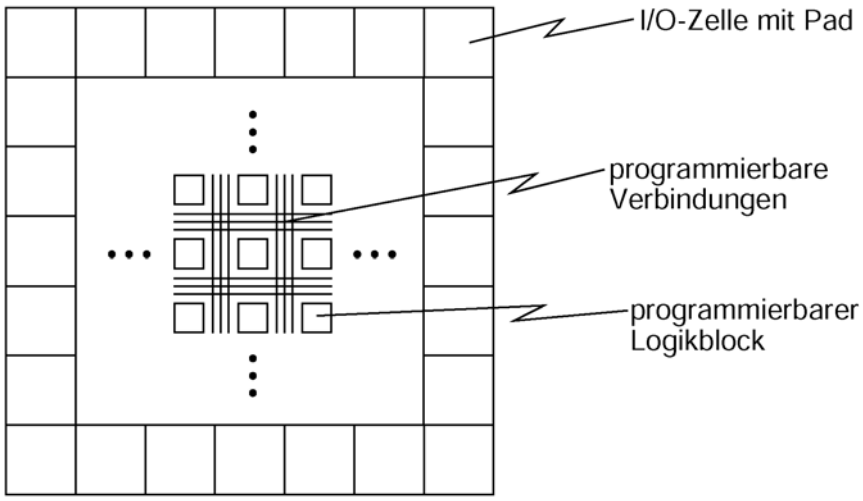


Bild 1.24 Typischer Aufbau eines FPGA

Bei FPGAs sind wie in Bild 1.24 ein Feld von beispielsweise 100.000 relativ einfachen Logikblöcken und ein Netz von Verbindungsbussen vorgegeben; beide können durch das Laden von 0-1-Mustern programmiert werden, so dass man eine individuelle Verschaltung aus individuellen Logikbausteinen erhält.

Der Einsatz solcher FPGAs beschränkt sich bei weitem nicht auf das Rapid Prototyping. Vielmehr sind FPGAs auch immer häufiger als Endprodukt in eingebetteten Systemen zu finden. Beispielhaft genannt seien Controller aller Art für Speicher, FIFOs, Schnittstellen, Grafik, Peripherie, viele Anwendungen in der Kommunikation uvam. . Dann ist es von Vorteil, dass sich die FPGA-Schaltungen wie normale Software updaten lassen, etwa wenn der Standard sich geändert hat oder auch nur im nie endenden Marathon von Versionen, Fehlerbeseitigung und neuen Fehlern…

In *adaptiven Rechnern* kann die momentane FPGA-Schaltung jederzeit durch eine völlig andere ersetzt werden. So kann sich jedes Anwendungsprogramm in Sekundenbruchteilen seinen maßgeschneiderten Coprozessor wünschen (und bekommen).

Auch in finanzieller Hinsicht sind FPGAs fast ideal, da die Entwicklungszeit nicht die Fertigungszeit von einigen Wochen bis Monaten enthält und da die Anfangsinvestition nur aus den Entwurfskosten besteht, während die Materialkosten je Prototyp zwischen einhundert und einigen tausend Euro liegen im Gegensatz zu 10.000 bis einer Million Euro für die erste Fertigung von Bausteinen vergleichbarer Komplexität.

Schließlich macht die sofortige und wiederholbare Programmierbarkeit FPGAs zum geradezu idealen Trainingsobjekt in der Lehre. Erfreulicherweise wird dabei nicht nur der spezielle FPGA-Entwurf, sondern der Chip-Entwurf schlechthin geübt.

1.4 Hardware-Software-Codesign, eingebettete Systeme und Systems-on-Chip

Wir haben bisher eine Vision erhalten, dass es uns relativ einfach gelingen könnte, Chips und damit Hardware maßgeschneidert zu entwickeln. Schön und gut, aber brauchen wir diese Speziallösungen überhaupt? Tut es nicht auch eine Software-Lösung auf einem effizienten Standardrechner? Die Antwort ist ein klares „jein“.

Für viele Anwendungen im Bereich eingebetteter Systeme reicht in der Tat ein kleiner eingebetteter Mikrocontroller mit passender Software oder ein Signalprozessor. Für andere Zwecke kommt man mit einer einfachen Schaltung aus. Immer häufiger aber paaren sich die Intelligenz von Software mit der Effizienz von Hardware.

In diesem *Hardware-Software-Codesign* werden die Software auf einem Standardrechner und die maßgeschneiderte Hardware auf einem FPGA *gleichzeitig* entworfen. Es gilt eine schwierige Entscheidung zu fällen, die Aufteilung oder *Partitionierung* der Aufgaben zwischen Soft- und Hardware, und schließlich muss eine geeignete *Kommunikation* über passende *Schnittstellen* zwischen beiden entwickelt werden.

Die Entscheidungen werden durch eine Vielzahl von Aspekten beeinflusst: neben der Aufgabe selbst spielt die nötige Effizienz eine Rolle (Energieverbrauch, Speicherbedarf, Geschwindigkeit, Realzeit-Anforderungen, Baugröße, Entwicklungs- und Fertigungskosten), es geht um Zuverlässigkeit (Ausfallsicherheit, Wartbarkeit, Unschädlichkeit und Datensicherheit) und schließlich ist die Kommunikation mit der Umgebung (Benutzer, Sensoren, Aktoren) zu berücksichtigen.

Anwendungsbeispiele finden sich in Vermittlungssystemen und Endgeräten der Telekommunikation, vor allem auch der Mobilkommunikation, in der Unterhaltungselektronik, in den Bereichen Messen, Prüfen und Regeln, im Kraftfahrzeug, in der Automatisierung oder bei verteilten Systemen.

Während man früher Standardrechner und Spezial-Hardware einzeln auf einem Board zusammengefasst hat, geht der Trend zum *System-on-Chip* (*SoC*), wo ein einzelnes Chip sowohl mehrere klassische Software-Prozessoren wie PowerPCs enthalten kann als auch frei programmierbare Schaltungslogik im Sinne von FPGAs als auch Busse mit hocheffizienten Kommunikationsprotokollen.

Solche SoCs zeichnen sich natürlich durch geringes Gewicht, geringe Leistungsaufnahme und hohe Zuverlässigkeit aus und sind damit prädestiniert für Mobilsysteme und viele Consumer-Geräte.

Als komplexes SoC-Beispiel zeigt Bild 1.25 einen Chip von IBM, das auf 1cm^2 einen kompletten digitalen DVB-Fernsehempfänger inklusive Video-on-Demand-Funktion und interaktivem Programmführer enthält.

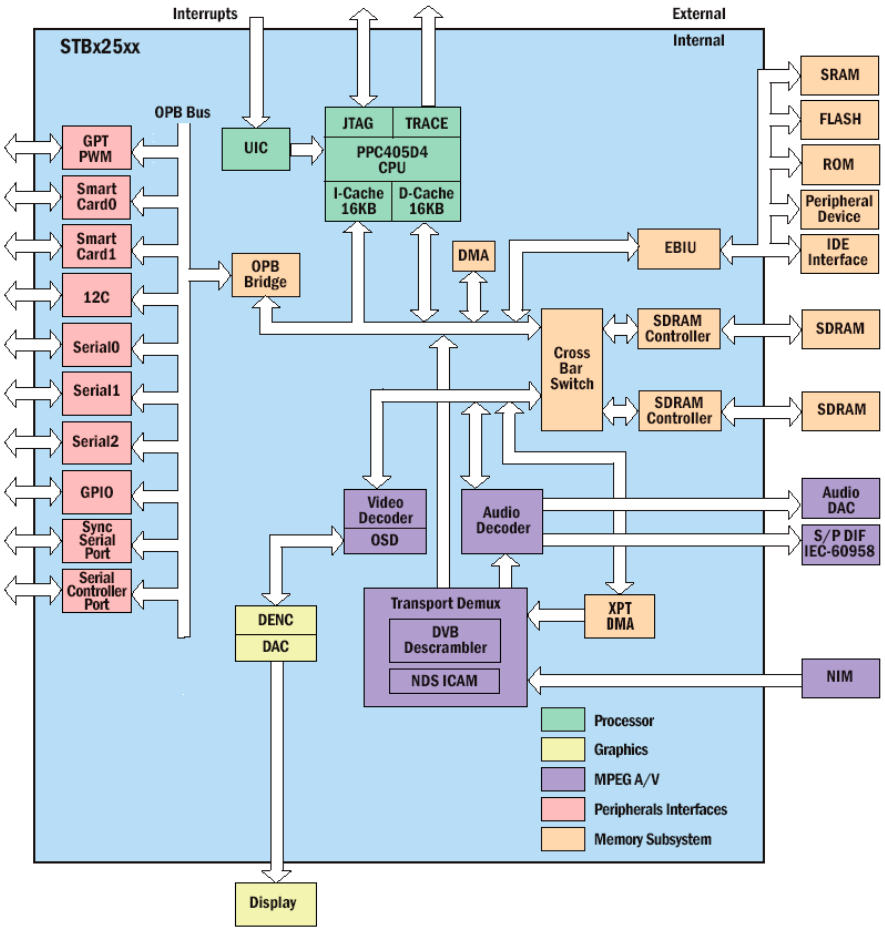


Bild 1.25
System-On-Chip
IBM STBx25

Entworfen werden solche komplexen Systems-on-Chip auf mächtigen Entwicklungsplattformen, die viel Entwicklungskomfort bieten. Anschließend werden nur die tatsächlich benötigten Teile des Prototypen realisiert und in die Anwendung eingebettet. Da dabei das Systemdenken immer wichtiger wird, reichen Hardware-Beschreibungssprachen nicht mehr aus, und es lohnen sich komplexe objektorientierte System-Beschreibungssprachen wie SystemC. Unterstützt wird dabei der Einsatz von

Intellectual Property oder IP, wohinter sich fertige wiederverwendbare Teillösungen verbergen, etwa ein Ethernet-Controller als Fertigbaustein.

Insgesamt ist der Hardware-Software-Entwurf zu einem *interdisziplinären* Gebiet geworden zwischen Hardware, Software und natürlich den Anwendungen der Praxis.

1.5 Ausblick

… auf die Vorlesung

Das weitere Skript gliedert sich in einen Vorlesungsteil (Kapitel 2 bis 7) und einen Übungsteil (Kapitel A bis E).

In den Kapiteln 2 bis 4 führen wir zunächst die für den praktischen Schaltungsentwurf zentral wichtige Hardware-Beschreibungssprache VERILOG ein. Eine neue Sprache zu erlernen ist nicht in jedem Moment erregend, hier lernen wir aber grundlegende Konzepte praktisch kennen wie Parallelität und Zeit, synchrone Logik, hardware-nahe Datenstrukturen, Verhalten und Struktur, Pipeline, Testrahmen und vieles andere mehr. In Kapitel 5 werfen wir einen kurzen Blick über die Schulter der Logiksynthese.

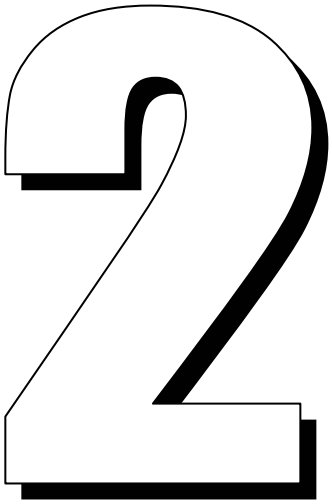
Praxisnahe und technisch spannend wird es in den nächsten und zugleich letzten beiden Vorlesungskapiteln. Das Geheimnis programmierbarer Logikbausteine und von FPGAs lüften wir in Kapitel 6, auch anhand kommerzieller Bausteine bis hin zum mächtigen Virtex-II Pro, das Teil der universellen Entwicklungs-Plattform ML310 in Kapitel 7 ist – man stelle sich nur mal vor: auf einem Chip gibt es zwei ausgewachsene Standardprozessoren, drumherum können wir mithilfe von VERILOG beliebige eigene Schaltungen synthetisieren, und um dieses Superchip herum gibt es jede Menge Anschlüsse und Adapter: USB, PCI, Ethernet, DDR, Audio, Video, Kartenleser, schnelle optische Anschlüsse etc. . In Kapitel 7 geht es aber auch generell um Hardware-Software-Codesign.

… auf die Übungen

„Übungen“ klingt trocken, deswegen nennen wir sie Labs. Diese sind eng auf die Vorlesung abgestimmt. Die Kapitel B und C enthalten Aufgaben und Werkzeug-Leitfäden für die praktische Arbeit am Rechner zur Sprache VERILOG und zur Logiksynthese einschließlich der Profi-Entwicklungsumgebung ModelSim/ISE. Multimediale Lernprogramme und weiteres Übungsmaterial werden separat eingesetzt und verteilt werden.

Während die VERILOG- und Synthese-Labs eher konventionell und brav sind, werden Sie staunen, wie einfach Sie eigene Hardware bauen und mit Ihrer eigenen Software kommunizieren lassen können. Dies Wunder passiert im ML310-Lab in

Kapitel D. Dort werden auch noch ein Remote-Debugger (Fehlersuche aus der Ferne) und ein „Oszilloskop“ auf dem zu untersuchenden Chip gleich mit integriert.



Überblick zur Hardware-Beschreibungssprache VERILOG

Hardware-Beschreibungssprachen (Hardware Description Languages, HDL) stehen im Zentrum des Schaltungsentwurfs. Designer beschreiben ihren Entwurf zunächst auf einer funktionalen Verhaltensebene, wobei die Einzelheiten der Implementierung auf ein späteres Entwurfsstadium verschoben werden. Eine abstrakte algorithmische Darstellung unterstützt den Designer bei der Erforschung von Alternativen der Schaltungsarchitektur und bei der Entdeckung von Entwurfsengpässen, bevor der Entwurf detailliert wird.

Es gibt mehrere Gründe für den Einsatz von HDLs im Gegensatz zum früher üblichen Entwurf mit Schaltplänen. Um Komplexitäten von Millionen von Gattern je Chip für den Menschen zugänglich zu machen (Abschnitt 1.2), muss die Funktionalität auf einer hohen Sprachebene ausgedrückt werden, die die Einzelheiten der Implementierung verbirgt. Aus einem ähnlichen Grund haben ja auch die höheren Programmiersprachen die Assemblersprachen bei den meisten großen Programmen verdrängt.

Der Wettbewerb auf dem Gebiet der Mikroelektronik ist groß, neue Firmen auf dem Markt erzeugen einen hohen Druck, die Entwurfseffizienz zu erhöhen, die Entwurfskosten zu senken und vor allem die „time to market“ zu verringern. Ausgiebige Simulationen können Entwurfsfehler vor der Chip-Fertigung entdecken und so die Zahl der Entwurfsiterationen verringern. Eine effiziente Hardware-Beschreibungssprache hilft dabei.

Die Beschreibung eines Entwurfs in einer Hardware-Beschreibungssprache bringt mehrere Vorteile. Eine HDL kann erstens zur *Spezifikation* verwendet werden. Der Vorteil einer formalen Sprache wie VERILOG HDL besteht darin, dass die Spezifikation vollständig und eindeutig ist. Trotzdem ist die Spezifikation durch eine formale Sprache hinsichtlich einer Realisierung zunächst „weich“ im Vergleich zu einem „harten“ Schaltplan.

Ein zweiter bedeutsamer Vorteil einer formalen Hardware-Beschreibungssprache besteht in der Möglichkeit, eine *Synthese* anzuschließen. Wir werden in Kapitel 5 Synthesewerkzeuge einsetzen, die eine HDL-Beschreibung in eine Gatter-Implementierung mit Bibliothekskomponenten übersetzen. Diese Werkzeuge versuchen, den Entwurf bezüglich Geschwindigkeit, Chip-Größe und anderen Kostenfunktionen zu optimieren. Heutige Synthesewerkzeuge sind allerdings noch erheblich eingeschränkt. Beispielsweise verwenden sie nur eine Untermenge der jeweiligen HDL, und der synthetisierte Schaltkreis ist oft nicht so effizient wie ein Expertenentwurf. Trotzdem spart die Synthese oft viel Zeit und Geld.

Drittens ist eine HDL besonders gut zur *Dokumentation* geeignet. Eine gut kommentierte HDL-Beschreibung ergibt meist eine viel bessere und verständlichere Dokumentation als ein Schaltplan auf der Gatterebene.

Der vierte Vorteil ist die Möglichkeit zur *Simulation*. Eine HDL-Simulation kann Fehler entdecken, die sonst erst später festgestellt werden, im schlimmsten Fall erst nach der Fertigung. Je später ein Fehler entdeckt wird, umso teurer sind die Entwurfskosten. Eine Simulation kann auf verschiedenen Ebenen durchgeführt werden. Auf der funktionalen Ebene wird das System durch algorithmische Konstrukte beschrieben. Auf der Logikebene wird das System zwar immer noch hierarchisch beschrieben, wobei am unteren Ende jedoch einfache Bausteine wie Gatter stehen. Auf dieser Ebene können Timing-Analysen durchgeführt werden.

Dabei sind *Mixed-Mode-Simulationen* möglich, indem Teile einer Schaltung oder eines ganzen Systems auf einer hohen Verhaltensebene, andere Teile dagegen beispielsweise auf der Gatterebene simuliert werden. Da die Simulationszeit auf den unteren Ebenen bei großen Entwürfen eine wesentliche Beschränkung darstellt, können jeweils ausgewählte Module verfeinert, getestet und anschließend wieder durch ihre höhere Darstellung ersetzt werden. Dies unterstützt die *vertikale Verifikation*.

Die Sprache VERILOG HDL ist einfach aufgebaut. Sie stellt lesbare Konstrukte für die Beschreibung von Hardware bereit. Eine vergleichbare Beschreibung in VHDL kann doppelt so lang werden wie in VERILOG. Einerseits ist diese prominente Rivalin von VERILOG sprachlich mächtiger, andererseits ist sie umständlicher zu erlernen und einzusetzen, und die Simulationszeiten sind oft deutlich länger.

In VERILOG braucht ein Designer nur eine Sprache für alle Ebenen oberhalb der Analog-Ebene zu lernen. VERILOG ist der Programmiersprache C ähnlich und nicht schwer zu lernen.

VERILOG unterstützt sowohl Strukturbeschreibungen als Netze von Gattern, Komponenten und Modulen als auch algorithmische Verhaltensmodelle mit Variablen, Fallunterscheidungen (`if`, `case`), Schleifen (`for`, `while`) und Prozeduren (`function`, `task`) sowie Mischformen davon (Abschnitt 3.1).

Der Rest dieses Kapitels soll anhand mehrerer kleiner Beispiele ein erstes Gefühl für die Hardware-Beschreibungssprache VERILOG verschaffen. Im nächsten Kapitel werden die wichtigsten Befehle dann genauer behandelt.

In Kapitel 4 werden typische Modellierungskonzepte angeboten wie Parallelität, Zeit, Pipelining und Register-Transfer-Logik und dabei durch Beispiele mittlerer Größe unterstützt.

Wir verwenden mit dem Simulator ModelSim getestete Beispiele. Wir beabsichtigen keine vollständige Zusammenfassung der Sprache, wie sie im Benutzerhandbuch enthalten ist, sondern eine pragmatische Einführung, die naturgemäß unvollständig und teilweise unpräzise bleiben wird.

Mit „VERILOG“ bezeichnen wir die Hardware-Beschreibungssprache VERILOG HDL. Alle Schlüsselworte beginnend mit einem „$“ bezeichnen Systemkommandos des Simulators und gehören nicht zur Sprache selbst.

VERILOG unterscheidet Groß- und Kleinbuchstaben, wobei alle Schlüsselworte klein geschrieben werden. Leerzeichen, Tabulatoren, Zeilenvorschübe und Kommentare können zur Erhöhung der Lesbarkeit verwendet werden.

In VERILOG sind *Module* die grundlegenden Einheiten für Gatter, Zähler, CPUs oder ganze Rechner. Sie lassen sich hierarchisch schachteln. Das Modul `count` in Beispiel 2.1, begrenzt von `module` und `endmodule`, gibt die Zahlen `1` bis `3` auf dem Bildschirm aus. In Zeile 1 wird die Zählvariable `I` definiert. Die (zusammengesetzte) Anweisung nach `initial` wird genau einmal ausgeführt. Die Anweisungen `$display` geben jeweils eine Zeichenkette aus. `I` wird mit `%d` ganzzahlig ausgegeben. `//` startet einen Kommentar bis zum Zeilenende. `begin` ist wie alle reservierten Wörter klein geschrieben und verschieden von `Begin` und `BEGIN`. Die Anweisungen innerhalb des `initial`-Blockes von `begin` bis `end` werden sequenziell, also der Reihe nach ausgeführt.

```
module count;                          // dies ist ein Kommentar    // 00
integer I;                                                       // 01
                                                                 // 02
initial                                                          // 03
begin                                                            // 04
  $display ("Beginn der Simulation...");                         // 05
  for (I=1; I <= 3; I=I+1)                                       // 06
    $display ("Durchlauf %d", I);                                // 07
  $display ("Ende der Simulation");                              // 08
end                                                              // 09
endmodule                                                        // 10
```

Beispiel 2.1 Ein einfaches Modul

Der Simulator ModelSim liefert Beispiel 2.2 als Ergebnis. Zu sehen ist auch der zugehörige Dialog, den wir bei künftigen Ergebnissen ebenso weglassen werden wie das # zu Beginn jeder Ausgabenzeile.

```
ModelSim> vsim beispiel_2_01_count
# vsim beispiel_2_01_count
# Loading beispiel_2_01_count
VSIM 9> run
# Beginn der Simulation...
# Durchlauf           1
# Durchlauf           2
# Durchlauf           3
# Ende der Simulation

VSIM 10>quit -sim
```

Beispiel 2.2 Ausgabe zu Beispiel 2.1

Das Modul `maximum` in Beispiel 2.3 berechnet das Maximum zweier Zahlen `A` und `B` und übergibt das Ergebnis in `MAX`. Die *Parameterliste* hinter dem Modulnamen enthält die Variablen, die das Modul importiert (`A` und `B`) bzw. an andere Module exportiert (`MAX`). Zusätzlich muss für jede Variable ihre Richtung `input` oder `output` deklariert werden; bei `inout` ist beides der Fall. Außerdem wird der Variablentyp deklariert (`integer`, `wire`, `reg`, `time` und andere).

```
// Bestimmung des Maximums                                    // 00
module maximum (                                              // 01
input  wire [31:0] A,                                         // 02
                   B,                                         // 03
output reg  [31:0] MAX                                        // 04
);                                                            // 05
                                                              // 06
always @(A, B)                                                // 07
begin                                                         // 08
  if (A > B)                                                  // 09
    MAX = A;                                                  // 10
  else                                                        // 11
    MAX = B;                                                  // 12
  $display ("new maximum is %d", MAX);                        // 13
end                                                           // 14
endmodule                                                     // 15
```

Beispiel 2.3 Schnittstelle, Variable, `always` und `@`

Wir gehen hier nur auf den Unterschied zwischen `wire` und `reg` ein, die übrigen werden später erläutert. Ein *Wire* vom Typ `wire` stellt eine Verbindung zwischen mehreren Punkten einer Schaltung dar und entspricht in erster Näherung einer Leitung. Ein *Register* des Typs `reg` entspricht einer Speicherzelle zum Schreiben und Lesen von Binärwerten. Dies ist bei einem Wire nicht möglich, der nur bidirektional zwischen Enden verbindet.

`wire` und `reg` sind *hardware-nahe* Typen; beide vertreten Worte aus mehreren Bits, wobei jedes Bit nicht nur die Werte `0` und `1`, sondern auch `z` und `x` annehmen kann: `z` bedeutet den hochohmigen Zustand, und `x` steht für unbestimmt; treiben mehrere Quellen einen Wire unterschiedlich, so ist das Ergebnis `x`. Für beide Typen

kann ihre Bit-Breite angegeben werden durch zwei Intervallgrenzen in eckigen Klammern vor der Variablendefinition. `A[31]` greift auf Bit 31 von `A` zu.

Die Anweisung `always` führt den nachfolgenden Block immer wieder aus. Erreicht der Simulator die Bedingung

```
@ (A, B),
```

so wird erst dann ein neuer Durchlauf des `always`-Blockes gestartet, wenn sich mindestens eine der Variablen `A` oder `B` geändert hat seit dem Zeitpunkt, als der Simulator die Bedingung erreichte.

Beispiel 2.4 zeigt das Modul `parallel_blocks`. Im Gegensatz zu üblichen Programmiersprachen mit sequenziellem Programmablauf arbeiten die Komponenten einer realen Schaltung alle parallel. Hierfür stellt VERILOG etliche Konstrukte bereit.

```
// zwei parallele Bloecke                                    // 00
module parallel_blocks;                                      // 01
                                                             // 02
initial                                                      // 03
begin                                                        // 04
  $display ("Ja");                                           // 05
  $display ("Ja");                                           // 06
end                                                          // 07
                                                             // 08
initial                                                      // 09
begin                                                        // 10
  $display ("Nein");                                         // 11
  $display ("Nein");                                         // 12
end                                                          // 13
                                                             // 14
endmodule                                                    // 15
```

Beispiel 2.4 Zwei parallele Blöcke

Die beiden `initial`-Blöcke in `parallel_blocks` werden parallel ausgeführt. Dies bedeutet, dass sie *in beliebiger Reihenfolge*, dann aber nur *nacheinander*, ausgeführt werden. Der Programmierer darf sich nicht auf eine bestimmte Reihenfolge verlassen. Bei mehrfacher Ausführung des gleichen Programms wird jedoch stets die gleiche Reihenfolge gewählt. Weiterhin ist garantiert, dass die Anweisungen innerhalb eines Blockes sequenziell bearbeitet werden, ohne dass in der Zwischenzeit ein anderer Block bearbeitet wird (es sei denn, es wird eine *Zeitkontrolle* erreicht: `@` (*Bedingung*), # (*Verzögerung*) und `wait` (*warten auf*)).

Daher sind als Simulationsergebnisse möglich

```
Ja Ja Nein Nein
```

oder

```
Nein Nein Ja Ja ,
```

nicht aber

```
Ja Nein Nein Ja ,
Ja Nein Ja Nein
```

und ähnliche Mischformen. Zur Vertiefung des überaus grundlegenden Parallelitäts-begriffes wird auf Abschnitt 4.1 verwiesen.

Eine Konstante beginnt optional mit einem Vorzeichen und einer Bit-Breite, für welche default-mäßig der kleinste notwendige Wert genommen wird. Für die optionale Basis sind `'b` (Basis 2), `'o` (Basis 8), `'d` (Basis 10) und `'h` (Basis 16) erlaubt, der Default ist `10`. `'h12` etwa erzeugt eine 5 Bit breite Konstante, `2'b1` die binäre 2-Bit-Konstante `01`. Weitere Möglichkeiten zeigt Beispiel 2.6, Zeilen 8-13.

Die Operationen in Tabelle 2.5 sind offensichtlich, auf den Unterschied zwischen `==` und `===` sowie deren Negation wird später eingegangen.

Operationsgruppe	Bedeutung
`+ - * / %`	Arithmetik
`< <= > >=`	Vergleich
`== != === !==`	Gleichheit
`! && ||`	logische Operatoren
`~ & | ^`	bit-weise Operatoren
`?:`	Auswahl
`<< >>`	Shift

Tabelle 2.5 Operationen

Beispiel 2.6 zeigt eine einfache ALU (Arithmetic Logical Unit) für die Operationen Addition, Multiplikation, UND, logisches UND (das logische UND ist wahr, wenn sowohl in `A` als auch in `B` mindestens ein Bit `1` ist), Modulo und Links-Shift. Die Eingaben für die ALU sind der Opcode und die beiden Operanden, die Ausgabe ist natürlich das Ergebnis. Nach der Variablendeklaration werden die Operationsnamen mit einem `` `define `` an eine Bit-Kombination gebunden. Solche Ersetzungen gelten ab der Deklaration bis zum Ende des Quelltextes, also auch über Modulgrenzen hinaus.

Sobald sich `A`, `B` oder `OPCODE` ändern, ruft der `always`-Block die Funktion `calculate` auf. Die Fallunterscheidung `case` wählt dort mit dem 3 Bit breiten `OPCODE` die zu berechnende Funktion aus. Das Ergebnis liefert die Funktion über ihren Namen. Dabei kann die Anzahl der Ergebnis-Bits vor dem Funktionsnamen angegeben werden.

```
// ALU-Verhaltensmodell                                          // 00
module alu (                                                     // 01
input  wire  [2:0] OPCODE,                                       // 02
       wire [31:0] A,                                            // 03
                   B,                                            // 04
output reg  [31:0] RESULT                                        // 05
);                                                               // 06
                                                                 // 07
`define ADD    3'b000   // 0          // nur zur Uebung:         // 08
`define MUL     'b001   // 1          // Konstanten auf          // 09
`define AND    3'o2     // 2          // verschiedene Arten      // 10
```

```
`define LOGAND 3'h3     // 3                                    // 11
`define MOD    4        // 4                                    // 12
`define SHL    3'b101   // 5                                    // 13
                                                                // 14
                                                                // 15
function [31:0] calculate(                                      // 16
input     [2:0] OPCODE,                                         // 17
          [31:0] A,                                             // 18
                 B                                              // 19
);                                                              // 20
                                                                // 21
case (OPCODE)                                                   // 22
   `ADD:    calculate = A +  B;                                 // 23
   `MUL:    calculate = A *  B;                                 // 24
   `AND:    calculate = A &  B;                                 // 25
   `LOGAND: calculate = A && B;                                 // 26
   `MOD:    calculate = A %  B;                                 // 27
   `SHL:    calculate = A << B;                                 // 28
  default: $display ("Unimplemented Opcode: %d!", OPCODE);      // 29
endcase                                                         // 30
endfunction                                                     // 31
                                                                // 32
                                                                // 33
always  @ (OPCODE, A, B)                                       // 34
  RESULT = calculate(OPCODE, A, B);                             // 35
                                                                // 36
endmodule                                                       // 37
```

Beispiel 2.6 Eine einfache ALU

Wie jedes Stück Software oder Hardware muss die ALU getestet werden. Dies geschieht oft durch einen *Testrahmen* wie in Bild 2.7, der geeignete Test-Inputs (Test-Stimuli oder Testmuster) anlegt und die Testantworten oder Test-Outputs beobachtet und prüft.

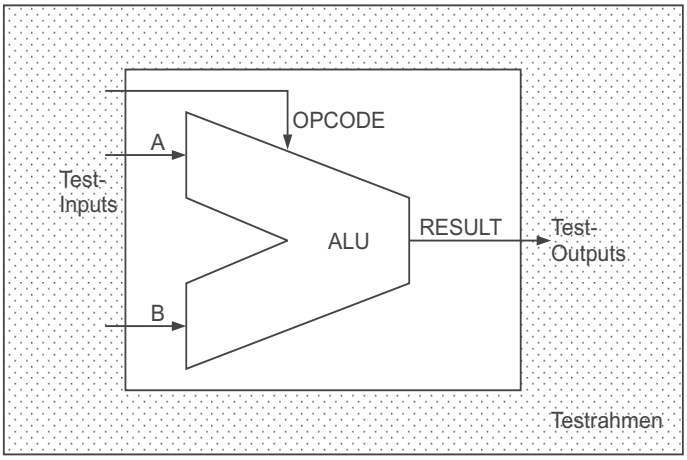


Bild 2.7
Testrahmen für die ALU

Zunächst könnte man auch den Testrahmen in das Programm von Beispiel 2.6 packen. Es empfiehlt sich jedoch sehr, die entwickelte Hard- oder Software und den Test sauber zu trennen. Deshalb gibt es in Beispiel 2.8 ein eigenes Testmodul, in dem die ALU als Untermodul instanziert wird: von der Vorlage `alu` aus Beispiel 2.6 wird eine Kopie oder Instanz namens `Alu` platziert. Allerdings müssen wir jetzt die

Variablen `A`, `B` usw. der Alu-Schnittstelle auch im Testrahmen noch einmal deklarieren. Dagegen hat das Modul `test` eine nach außen abgeschlossene, leere Schnittstelle.

Die Testmuster in den Zeilen 19 und 20 sind durch je eine Verzögerung `#1` abgeschlossen. Sonst würde der `initial`-Block ohne Unterbrechung bearbeitet. Insbesondere würde das erste Testmuster gleich durch das zweite überschrieben, ohne dass die parallel laufende ALU-Instanz und der Test-Beobachter ab Zeile 25 Gelegenheit hätten, darauf zu reagieren.

Das offensichtlich vernünftige Simulationsergebnis zeigt Bild 2.9.

```
// ALU-Testrahmen                                                    // 00
module test;                                                         // 01
                                                                     // 02
reg    [2:0] OPCODE;                                                 // 03
reg   [31:0] A,                                                      // 04
             B;                                                      // 05
wire  [31:0] RESULT;                                                 // 06
                                                                     // 07
`define ADD     0                                                    // 08
`define MUL     1                                                    // 09
`define AND     2                                                    // 10
`define LOGAND  3                                                    // 11
`define MOD     4                                                    // 12
`define SHL     5                                                    // 13
                                                                     // 14
alu Alu (OPCODE, A, B, RESULT);      // ALU-Instanz                  // 15
                                                                     // 16
initial begin                        // Test-Inputs                  // 17
  $display ("Simulation beginnt...");                                // 18
  OPCODE = `ADD; A = 3; B = 2; #1;                                   // 19
  OPCODE = `SHL; A = 3; B = 2; #1;                                   // 20
  $display ("Simulation endet.");                                    // 21
  $finish;                                                           // 22
end                                                                  // 23
                                                                     // 24
always @ (RESULT)                    // Test-Outputs                 // 25
  $display ("OPCODE = %d, A = %d, B = %d: RESULT = %d",              // 26
    OPCODE, A[5:0], B[5:0], RESULT[5:0]);                            // 27
                                                                     // 28
endmodule                                                            // 29
```

Beispiel 2.8 Testrahmen für die ALU aus Beispiel 2.6

```
Simulation beginnt...
OPCODE = 0, A =  3, B =  2: RESULT =  5
OPCODE = 5, A =  3, B =  2: RESULT = 12
Simulation endet.
```

Bild 2.9 Simulationsergebnis zum ALU-Test

Zu Beginn einer Hardware-Entwicklung geht es zunächst mal um das *Verhalten*, bei dem der spätere Baustein als *Black-Box* aufgefasst wird und nur das Wechselspiel

zwischen Ein- und Ausgaben interessiert. Ein Verhaltensmodell ist irgendwie realisiert, oft durch ein „gewöhnliches“ sequenzielles Programm. Für sequenzielle Programme reicht in VERILOG ein einziger `initial`- oder `always`-Block aus, dessen Inneres wie oben erklärt sequenziell ausgeführt wird.

Wollen wir uns dagegen in Richtung einer späteren Hardware-Realisierung bewegen, werden wir statt einer Black-Box ein *Strukturmodell* formulieren, bei dem das Ganze in Teile zerlegt ist, die miteinander verdrahtet sind. Für die Teile werden wiederum nur Verhaltensbeschreibungen angegeben, bis auch sie möglicherweise durch noch feinere Unterstrukturen ersetzt werden.

Für die Teile eines Strukturmodells eignen sich in VERILOG Untermodule besonders gut; der Verdrahtung der Teile entspricht die Angabe von Wires oder Registern, die gemeinsam in den Parameter-Schnittstellen mehrerer Untermodule vorkommen.

Für das Beispiel eines 4-Bit-Addierers, der in kleine 1-Bit-Addierer zerlegt wird, zeigt Bild 2.10 zunächst das Verdrahtungsprinzip, während Beispiel 2.11 die tatsächliche VERILOG-Modellierung durchführt.

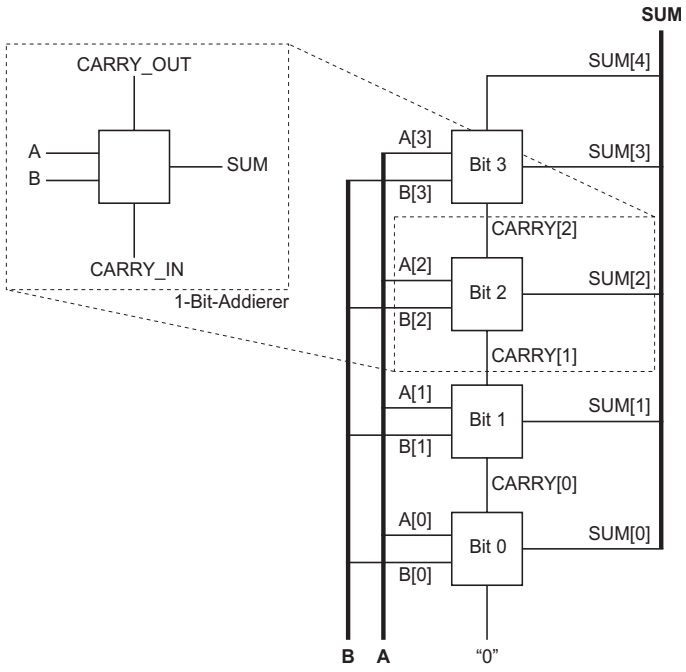


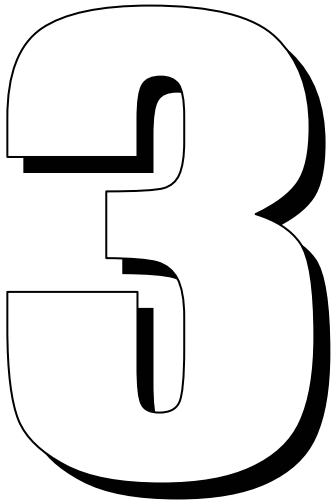
Bild 2.10 Struktur eines 4-Bit-Addierers

Beispiel 2.11 instanziert die Untermodule `Bit0` bis `Bit3`. Es entsteht eine hierarchische Struktur von Modulen. Es gibt zwei Modultypen `one_bit_adder` und `four_bit_adder`. Die Signale `CARRY_OUT` und `SUM` werden mit `{` und `}` zu einer neuen, 2 Bit breiten Variablen zusammengeklammert. Dieser Konkatenation wird in einem `always`-Block die Summe von `A`, `B` und `CARRY_IN` zugewiesen, sobald sich eine der drei Variablen ändert. Der `four_bit_adder` implementiert die Addition zweier 4 Bit breiter Variablen durch vierfache Instanzierung des `one_bit_adder`. Dabei werden ein 5 Bit breites Ergebnis erzeugt und ein Flag

gesetzt, falls das Ergebnis 0 ist. In den Zeilen 28 und 29 wird ein weiterer, in VERILOG bereits vordefiniertes Modul `nor` instanziert. `CARRY` reicht Überträge weiter.

```
module one_bit_adder(                                                    // 00
input       A,                        // 1-Bit-Wires per Default      // 01
            B,                                                         // 02
            CARRY_IN,                                                  // 03
output reg  SUM,                                                       // 04
            CARRY_OUT                                                  // 05
);                                                                     // 06
                                                                       // 07
// Verhalten des one_bit_adder                                         // 08
always @ (A, B, CARRY_IN)                                              // 09
  {CARRY_OUT, SUM} = A + B + CARRY_IN;                                 // 10
endmodule  // one_bit_adder                                           // 11
                                                                       // 12
module four_bit_adder(                                                 // 13
input  wire [3:0] A,                                                   // 14
                  B,                                                   // 15
output wire [4:0] SUM,                                                 // 16
output wire       NULL_FLAG                                            // 17
);                                                                     // 18
                                                                       // 19
wire [2:0] CARRY;                                                      // 20
                                                                       // 21
// Struktur des four_bit_adder                                         // 22
one_bit_adder Bit0 (A[0], B[0],     1'b0, SUM[0], CARRY[0]);           // 23
one_bit_adder Bit1 (A[1], B[1], CARRY[0], SUM[1], CARRY[1]);           // 24
one_bit_adder Bit2 (A[2], B[2], CARRY[1], SUM[2], CARRY[2]);           // 25
one_bit_adder Bit3 (A[3], B[3], CARRY[2], SUM[3], SUM[4] );            // 26
                                                                       // 27
nor Zeroflag (NULL_FLAG,                                               // 28
  SUM[0], SUM[1], SUM[2], SUM[3], SUM[4]);                             // 29
                                                                       // 30
endmodule // four_bit_adder                                            // 31
```

Beispiel 2.11 4-Bit-Addierer mit instanzierten Untermodulen



Die wichtigsten Befehle von VERILOG

Dieser Abschnitt ist sowohl zum Nachschlagen als auch zum Lernen der Befehle gedacht. Neben der Einteilung in Befehle zur Modulstruktur (z.B. `module`), Zeitkontrolle (wie `wait`), Programmsteuerung (`case`), Variablen (`wire`), Operationen (+), Zuweisungen (`assign`) und sonstige Befehle ist jeder Befehl bzw. eine kleine Gruppe eng verwandter Befehle nach folgendem Schema dargestellt.

1. Name oder Bedeutung der Befehlsgruppe, nummeriert mit V1, V2, ...;
2. verbale Beschreibung;
3. kurzes Beispiel.

Es gibt folgende Gruppen.

Modulstruktur:
V1 `module, endmodule`
V2 `input, output, inout`
V3 `parameter, defparam`
V4 `always`
V5 `initial`

Zeitkontrollen:
V6 Warten `#`
V7 `at @`
V8 `wait`

klassische Programmsteuerung:
V9 `task, endtask`
V10 `function, endfunction`
V11 `if, else`
V12 `?, :`
V13 `case, casez`
V14 `while`
V15 `for`

Variablen und Konstanten:
V16 `integer`
V17 `wire`
V18 `reg`
V19 Felder von Variablen
V20 Konstante

Operationen:
V21 Arithmetik `+, -, *, /, %`
V22 Vergleich `==, !=, ===, !==, <, >, <=, >=`
V23 Logik `!, &&, ||`
V24 Bit-weise Logik `~, &, |`
V25 Konkatenation `{}`
V26 Shift `<<, >>`

Zuweisungen:
V27 Blockende Zuweisung `=`
V28 Nichtblockende Zuweisung `<=`
V29 `assign` (Ständige Zuweisung, Continuous Assignment)

sonstige Befehle:
V30 `` `define ``
V31 `$display, $write`
V32 `$finish`
V33 `$readmemh, $readmemb`

3.1 Modulstruktur

V1 module, endmodule

Die Module stellen die grundlegenden Bausteine von VERILOG dar. Sie repräsentieren kleine oder große Hardware-Komponenten wie AND-Gatter, Zähler, CPU oder ein ganzes Rechnernetz.

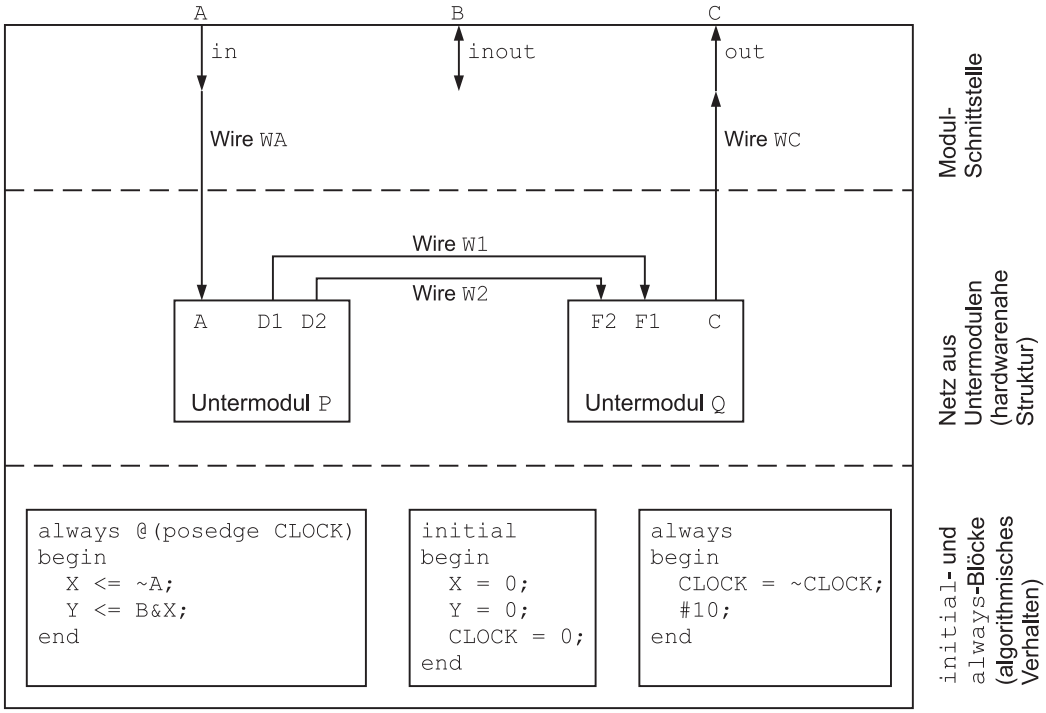


Bild 3.1 Wesentliche mögliche Teile eines Moduls

Bild 3.1 deutet schematisch an, welche wesentlichen Teile ein Modul enthalten kann (nicht muss). `module` und `endmodule` umschließen das Modul. Nach `module` und dem Modulnamen folgt als Schnittstelle oder Interface eine optionale Parameterliste, die die Verbindung mit anderen Modulen herstellt. Zwar können andere Module auch direkt, d.h. an der Schnittstelle vorbei, auf Variable des Moduls zugreifen; dies führt jedoch zu einer schlecht überprüfbaren Struktur und sollte daher vermieden werden. Außerdem können lokale Variablen definiert werden. Für sie muss der Typ (z.B. `reg`) angegeben werden. Falls die Variable Teil der Parameterliste ist, wird angegeben, ob sie Eingang, Ausgang oder beides ist.

Ein Modul kann mit Untermodulen hierarchisch zerlegt werden (vgl. Beispiel 3.5 und V3), die durch Wires (V17) ständig verbunden sind. Dies stellt in der Regel eine hardware-nahe strukturelle Realisierung des Moduls dar.

Eine algorithmische, mehr abstrakte Realiserung ergibt sich durch `initial`- und `always`-Blöcke, die zueinander parallel ausgeführt werden (Abschnitt 4.1), deren

Inneres aber in der Regel sequenziell wie bei „normalen“ Programmen ausgeführt wird mit `if-else`-Alternativen, Schleifen usw. Sie werden erst später in Strukturen aus Untermodulen, Gattern und Wires überführt.

Das Modul in Beispiel 3.2 gibt in einem `initial`-Block, der einmal am Anfang ausgeführt wird, die Zahlen 1 bis 10 aus.

```
module counter;
integer R;

initial
  for (R=1; R <= 10; R = R + 1)
    $display ("R= %d", R);
endmodule
```

Beispiel 3.2 Ein Modul mit `initial`-Block

Beim Instanzieren von Modulen können die Ein- und Ausgänge übrigens nicht nur wie in Beispiel 2.11 in der richtigen Reihenfolge direkt übergeben werden, sondern auch „by name“ in beliebiger Reihenfolge:

```
one_bit_adder Bit0 (.SUM(SUM[0]), .B(B[0]),
.CARRY_IN(1'b0), .A(A[0]), .CARRY_OUT(CARRY[0]));
```

```
V2  input, output, inout
```

Durch `input` wird eine Variable als Eingang eines Moduls deklariert. Eine `input`-Variable kann gelesen, aber nicht geändert werden. Eine `output`-Variable kann nur

```
module maximum(
input  wire [7:0] A,
                  B,
output reg  [7:0] RESULT
);

always @(A, B)
  if (A > B)
    RESULT = A;
  else
    RESULT = B;

endmodule
```

Beispiel 3.3 Modulparameter mit Richtung

vom Modul beschrieben werden, dessen Ausgang sie darstellt. Andere Module können sie nur lesen. `inout`-Variablen sind les- und schreibbar. `input` kann auch bei Tasks und Funktionen benutzt werden (V9, V10).

Für Ein- und Ausgänge wird anschließend der Typ deklariert. Für einen Wire der Breite 1 kann diese Angabe entfallen. Beispiel 3.3 berechnet das Maxmimum zweier Eingangsvariablen und übergibt das Ergebnis im Register `RESULT`.

Das Schlüsselwort `inout` wird hauptsächlich für die Modellierung bidirektionaler Datenbusse benötigt und kann nur auf Wires angewendet werden (V17).

In älteren VERILOG-Programmen findet man übrigens auch eine ausführliche Darstellung der Modulparameter wie in Beispiel 3.4.

```
module maximum (A, B, RESULT);
input       A,
            B;
output      RESULT;

wire [7:0] A,
           B;
reg  [7:0] RESULT;

always @(A, B)
  if (A > B)
    RESULT = A;
  else
    RESULT = B;
endmodule
```

Beispiel 3.4 Ausführliche Modulparameter

V3 parameter, defparam

Während durch `input`, `output` und `inout` „hardware-nahe“ Schaltungseingänge und Ausgänge eines Moduls definiert werden, wird mit `parameter` eine „abstrakte“ Konstante an einen Modul übergeben. Sie kann vom Modul selbst nicht geändert, jedoch bei einer Modul-Instanzierung durch `defparam` (`Max=5` wie bei `Counter1` in Beispiel 3.5) oder direkt (`Max=10` wie bei `Counter2`) erneut festgelegt werden, sodass mehrere Instanzen eines Moduls sich in den Parametern unterscheiden können.

Im Beispiel 3.5 werden zwei Module `counter` instanziert; `Counter1` zählt bis 5, `Counter2` bis 10. Zunächst wird ein allgemeines Modul `counter` deklariert, welcher bis zum Parameter `Max` zählt (default-mäßig 0). Dieser wird bei der Instanzierung festgelegt.

```
module counter #(
parameter Max = 0
);
integer R;

initial
  for (R=1; R <= Max; R = R + 1)
    $display ("R= %d", R);
endmodule // counter


module main;
defparam Counter1.Max = 5;       // Parameter explizit definiert

counter Counter1;
counter #(10) Counter2;          // Parameter bei Instanzierung
endmodule // main
```

Beispiel 3.5 Parameter

V4 always

Ein `always`-Block führt die auf `always` folgende Anweisung immer wieder aus. Er wird gleichzeitig zu allen anderen `always`- und `initial`-Blöcken (V5) ausgeführt. Daher arbeiten alle diese Anweisungen *parallel* (Abschnitt 4.1).

Der algorithmische Teil eines Moduls wie in Bild 3.1 besteht aus `always`- und `initial`-Blöcken. Während `always`-Blöcke oft Schaltungsteile darstellen, werden `initial`-Blöcke hauptsächlich in Testmodulen verwendet.

Ein `always`-Block wird häufig durch eine Eintrittsbedingung oder Sensitivitätsliste gestartet: `always @(COUNTER)` (V7). Sollen alle Lesevariablen des Blockes zur Sensitivitätsliste gehören, kann man kurz `always @(*)` schreiben.

```
module always_test;
reg [3:0] COUNTER;

always @(COUNTER)
  $display ("               COUNTER-Aenderung");

always @(COUNTER) begin
  $display ("Zeit = %2.0f", $time);
  $display ("COUNTER = %d", COUNTER);
end

always begin
  if (COUNTER === 4'bx)            // erster Durchlauf: Initialisierung
    COUNTER = 0;
  else
    COUNTER = COUNTER + 1;
  if (COUNTER == 5)
    $finish;                       // Simulationsende nach 5 Laeufen
  #10;                             // 10 Zeiteinheiten warten
end

endmodule
```

```
                COUNTER-Aenderung
Zeit =  0
COUNTER =  0
                COUNTER-Aenderung
Zeit = 10
COUNTER =  1
                COUNTER-Aenderung
Zeit = 20
COUNTER =  2
                COUNTER-Aenderung
Zeit = 30
COUNTER =  3
                COUNTER-Aenderung
Zeit = 40
COUNTER =  4
```

Beispiel 3.6 Parallele `always`-Blöcke

Die Anweisungen innerhalb eines `always`-Blockes werden *sequenziell ohne Unterbrechung* bearbeitet, bis eine Zeitkontrolle erreicht wird (Abschnitt 3.2).

Im Beispiel 3.6 geben die ersten beiden `always`-Blöcke Meldungen aus, wenn `COUNTER` sich ändert. Der dritte ändert `COUNTER`. Dazu wird mit `if` abgefragt, ob `COUNTER` noch nicht initialisiert wurde (dies ist beim ersten Durchlauf der Fall). Ansonsten wird `COUNTER` erhöht, bis bei 5 die Simulation beendet wird.

Alle drei `always`-Blöcke arbeiten parallel in dem Sinne, dass zum gleichen Simulationszeitpunkt die Reihenfolge ihrer Ausführung nicht festgelegt ist. Die Anweisungen innerhalb eines `always`-Blockes werden wie oben erwähnt sequenziell ohne Unterbrechung bearbeitet bis zu einer Zeitkontrolle. Daher ist ein Simulationsergebnis der Art

```
Zeit = 10
                COUNTER-Aenderung
COUNTER =  1
```

nicht möglich, weil in diesem Fall der zweite `always`-Block durch den ersten unterbrochen würde, im zweiten Block jedoch zwischen den beiden `$display`-Anweisungen keine Zeitkontrolle ist. Würde am Ende des dritten `always`-Blockes die Zeitkontrolle

```
#10
```

fehlen, so würde der dritte Block fünf mal direkt hintereinander ablaufen, ohne die Kontrolle an einen der anderen Blöcke abzugeben. Einzelheiten finden sich in den Abschnitten 4.1 und 4.2.

V5 initial

Ein `initial`-Block führt die auf `initial` folgende Anweisung genau einmal aus. Er wird gleichzeitig zu allen anderen `initial`- und `always`-Blöcken (V4) ausgeführt. Daher arbeiten alle diese Anweisungen *parallel* (Abschnitt 4.1).

Der algorithmische Teil eines Moduls wie in Bild 3.1 besteht aus `always`- und `initial`-Blöcken. Während `always`-Blöcke oft Schaltungsteile darstellen, werden `initial`-Blöcke hauptsächlich in Testmodulen verwendet.

Ein `initial`-Block wird häufig durch eine Eintrittsbedingung oder Sensitivitätsliste gestartet: `initial @(COUNTER)` (V7). Sollen alle Lesevariablen des Blockes zur Sensitivitätsliste gehören, kann man kurz `initial @(*)` schreiben.

Die Anweisungen innerhalb eines `initial`-Blockes werden *sequenziell ohne Unterbrechung* bearbeitet, bis eine Zeitkontrolle erreicht wird (Abschnitt 3.2). Beispiel 3.7 zeigt zwei parallele `initial`-Blöcke. Ob der erste oder der zweite Block zuerst ausgeführt wird, ist nicht festgelegt.

```
module initial_test;
initial begin                          // initial-Block 1
  $display ("AAA");
  $display ("BBB");
  #10;
  $display ("CCC");
end

initial begin                          // initial-Block 2
  $display ("1");
  $display ("2");
end
endmodule
```

Beispiel 3.7 Parallele `initial`-Blöcke

Ein mögliches Simulationsergebnis ist

```
AAA
BBB
1
2
CCC
```

Hier wurde der erste Block zuerst ausgeführt, `#10` unterbrach diese Ausführung, und dann wurde der zweite Block bearbeitet. Ein anderes mögliches Simulationsergebnis wäre

```
1
2
AAA
BBB
CCC
```

3.2 Zeitkontrollen

Es existieren in VERILOG drei Zeitkontrollen:

- die Zeitverzögerung # (warten),

- die Ereignis-Kontrolle @ und
- die Anweisung `wait`.

Alle drei Zeitkontrollen können die sequenzielle und zusammenhängende Bearbeitung der Anweisungen in einem `initial`- oder `always`-Block unterbrechen. Bei der Zeitverzögerung und der Ereignis-Kontrolle ist dies stets der Fall, d.h. die unterbrochene Bearbeitung der Anweisungen wird zu einem späteren (oder im gleichen) Zeitpunkt wieder aufgenommen. Die `wait`-Anweisung unterbricht die Ausführung nur, falls die abzuwartende Bedingung nicht bereits erfüllt ist.

V6 Warten

Bei `#7` tritt eine Zeitverzögerung von `7` Zeiteinheiten ein, während der andere parallele Prozesse weiterarbeiten. Danach wird die Bearbeitung fortgesetzt. Statt `7` können auch variable Ausdrücke stehen.

Der `always`-Block im Beispiel 3.8 gibt Änderungen von `DATA` aus. Der `initial`-Block enthält drei Zeitverzögerungen; die mittlere befindet sich innerhalb einer Zuweisung. In diesem Fall verschiebt sie die Zuweisung von `1` an `DATA` um 10 Zeiteinheiten, sodass sich `DATA` erst zum Zeitpunkt 20 in `1` ändert. Man beachte die Unterschiede zwischen

```
#10; DATA = A;
DATA = #10 A;   und
DATA = A; #10;
```

```
module time_delay;
reg DATA;

always @(DATA)
  $display ("Zeit: %2.0f, DATA = %d", $time, DATA);

initial begin
  DATA = 0;
  #10;
  DATA = #10 1;
  #10;
end
endmodule
```

```
Zeit:  0, DATA = 0
Zeit: 20, DATA = 1
```

Beispiel 3.8 Warten mit #

Bei dem Block

```
IN1 = 3;
IN2 = 5;
```

```
RESULT = IN1 + IN2;
#1;
RESULT = 4 * RESULT;
```

werden die ersten drei Anweisungen immer direkt hintereinander ausgeführt, d.h. es wird mit Sicherheit keine Anweisung eines weiteren `always`- oder `initial`-Blockes oder eines sonstigen parallelen Prozesses eingeschoben. So wie man zu Recht erwartet, dass die Summe aus `IN1` und `IN2` und die Zuweisung an `RESULT` ununterbrochen und als Ganzes bearbeitet werden, werden auch „Nebenrechnungen“ aus mehreren Anweisungen ohne Zeitkontrolle als Einheit zu einem Simulationszeitpunkt bearbeitet.

Dagegen unterbricht die Zeitkontrolle `#1` die Ausführung des Blockes zum aktuellen Zeitpunkt t_1, und es wird ein Ereignis für $t_2=t_1+1$ erzeugt, dass nämlich dann die Anweisungen hinter der Zeitkontrolle ausgeführt werden. Andere parallele und für t_1 geplante Ereignisse können jetzt bearbeitet werden (auch z.B. `RESULT=0`!). Danach werden t_1 erhöht und die dann vorgesehenen Ereignisse bearbeitet.

V7 at @

Die Ereignis-Kontrolle `@` (ausgesprochen: „ät“) wartet auf eine Änderung des Ausdrucks hinter dem `@`.

Bei vorangestelltem `posedge` wird auf eine positive Flanke von `0`, `x` oder `z` auf `1` gewartet. Entsprechend wartet `negedge` auf eine negative Flanke von `1`, `x` oder `z` auf `0`.

Soll auf mehrere Änderungen gewartet werden, so können diese mit `or` oder Komma verknüpft werden, nicht zu verwechseln mit den Logikoperationen `|` und `||` (V23, V24).

Im Beispiel 3.9 wartet der erste `always`-Block auf eine positive Flanke von `CLOCK`, der zweite auf eine negative und der dritte auf `SIGNAL1` oder `SIGNAL2`.

Bereits zum Zeitpunkt 0 liegt eine negative Flanke für `CLOCK` vor, die sich von `x` auf `0` ändert. Zur Zeit 20 ändert sich `SIGNAL2`, die Bedingung `@(SIGNAL1, SIGNAL2)` ist daher erfüllt.

Liegt ein Block der Form

```
$display ("Start des Blockes");
RESULT = IN1 + IN2;
@(EVENT);
$display ("Ende des Blockes");
```

vor, so wird beim Erreichen der Ereignis-Kontrolle

```
@(EVENT);
```

die Ausführung gestoppt. Es wird ein Test von `EVENT` am Ende des aktuellen Simulationszeitpunktes t_1 vorgemerkt. Danach werden andere parallele Ereignisse bei t_1 bearbeitet. Bei der Überprüfung von `EVENT` am Ende von t_1 gibt es zwei Möglichkeiten: entweder ist `EVENT` jetzt schon eingetreten, dann wird noch bei t_1 hinter

EVENT weitergemacht; oder es wird für das nächste $t_2 > t_1$ eine erneute Überprüfung von EVENT vorgesehen. Dieses Verhalten wird durch Beispiel 3.10 verdeutlicht.

```
module aett;
reg   CLOCK,
      SIGNAL1,
      SIGNAL2;

always @(posedge CLOCK)
  $display ("Zeit: %2.0f: positive Flanke", $time);

always @(negedge CLOCK)
  $display ("Zeit: %2.0f: negative Flanke", $time);

always @(SIGNAL1, SIGNAL2)                // oder @(SIGNAL1 or SIGNAL2)
  $display ("Zeit: %2.0f: SIGNAL1 oder SIGNAL2",$time);

initial begin
  CLOCK   = 0; #10;
  CLOCK   = 1; #10;
  SIGNAL2 = 0; #10;
  CLOCK   = 0; #10;
  SIGNAL1 = 1; #10;
  $finish;
end
endmodule
```

```
Zeit:  0: negative Flanke
Zeit: 10: positive Flanke
Zeit: 20: SIGNAL1 oder SIGNAL2
Zeit: 30: negative Flanke
Zeit: 40: SIGNAL1 oder SIGNAL2
```

Beispiel 3.9 Warten mit @

```
module test_events;
reg EVENT;

initial begin                        // Start bei Zeitpunkt 0
  $display ("Start bei %d", $time);
  @(EVENT);
  $display ("EVENT bei %d", $time);
end

initial begin                        // Start bei Zeitpunkt 0
  #1;
  EVENT = 0;
end
endmodule
```

Beispiel 3.10 Zeitkontrollen und Event

Beide `initial`-Blöcke beginnen zum Zeitpunkt 0, von denen der Simulator einen Block als ersten auswählt. Man darf sich nicht darauf verlassen, dass dies der

erste Block ist. Wird jedoch beispielsweise der erste `initial`-Block zuerst gestartet, so gibt dieser den Text und den Simulationszeitpunkt 0 aus. Dann unterbricht

```
@(EVENT);
```

die Bearbeitung, und es wird der zweite `initial`-Block gestartet, welcher zunächst durch #1 die weitere Ausführung auf den Zeitpunkt 1 verschiebt, jedoch *vor* den `EVENT`-Test im ersten `initial`-Block. Es wird `EVENT` ausgelöst. Im ersten Block wird daher bei `t=1` erfolgreich `EVENT` überprüft, und die zweite `$display`-Anweisung wird ausgeführt. Die Simulationsausgabe in Bild 3.11 ist eindeutig.

```
Start bei 0
EVENT bei 1
```

Bild 3.11 Simulationsausgabe zum Beispiel 3.10

Die Verzögerung `#1` im zweiten Block ist bedeutsam. Ohne sie kommt es zu einem unerwünschten Nichtdeterminismus, indem die Simulationsausgabe davon abhängt, welchen `initial`-Block der Programmierer bzw. der Simulator zuerst zur Ausführung bringt! Wird (ohne `#1`) der zweite Block zuerst bearbeitet, ist die Ausgabe

```
Start bei 0
```

andernfalls

```
Start bei 0
EVENT bei 0
```

V8 wait

Ein `wait` wartet, bis der nachfolgende Ausdruck wahr ist. Ist dieser beim Erreichen des `wait` bereits erfüllt, wird ohne Unterbrechung fortgesetzt.

Im Beispiel 3.12 gibt der erste `always`-Block den aktuellen Simulationszeitpunkt aus, falls `ENABLE` den Wert `1` hat. Dabei ist #1 unbedingt notwendig, da die Schleife sonst endlos würde: das erste `always` würde die Kontrolle nicht mehr an die anderen `initial`- und `always`-Blöcke abgeben. Der zweite `always`-Block gibt jedes Mal `ENABLE` bei dessen Änderung aus. Der `initial`-Block erzeugt das Freigabesignal.

```
module wait_test;
reg ENABLE;

always begin
  wait (ENABLE);
  $display ("            Zeit: %2.0f", $time);
  #1;
end
```

```
always @(ENABLE)
  $display ("ENABLE = %d", ENABLE);

initial begin
  ENABLE = 1; #5;
  ENABLE = 0; #5;
  ENABLE = 1; #5;
  $finish;
end
endmodule
```

```
ENABLE = 1
            Zeit:  0
            Zeit:  1
            Zeit:  2
            Zeit:  3
            Zeit:  4
ENABLE = 0
ENABLE = 1
            Zeit: 10
            Zeit: 11
            Zeit: 12
            Zeit: 13
            Zeit: 14
```

Beispiel 3.12 Warten mit `wait`

3.3 Klassische Programmsteuerung

V9 task, endtask

```
module task_example;
reg   [7:0] RESULT;

task add(
input reg [7:0] A,
                B
);
RESULT = A + B;
endtask

task display_result;
$display ("Die Summe ist %d", RESULT);
endtask

initial begin
  add (1,2);
  display_result;
end
endmodule
```

Beispiel 3.13 Zwei Tasks

`task` und `endtask` umfassen eine *Task*, mit der logisch zusammenhängende Programmteile zusammengefasst werden können. Eine Task darf im Gegensatz zu einer Funktion (V10) Zeitkontrollen besitzen (Abschnitt 3.2).

Das Modul in Beispiel 3.13 besteht aus zwei Tasks sowie einem `initial`-Block, in dem zunächst die Task `add` mit den aktuellen Parametern `1` und `2` aufgerufen wird. Diese Task besitzt also zwei Parameter. Das Ergebnis der Addition wird in der globalen Variable `RESULT` abgelegt. Der Aufruf der parameterlosen Task `display_result` gibt das Ergebnis auf dem Bildschirm aus.

V10 function, endfunction

`function` und `endfunction` begrenzen eine *Funktion*. Auch hier wird ein logisch zusammenhängendes Programmstück gebildet. Eine Zeitkontrolle ist im Funktionscode unzulässig. Daher entspricht der Funktionscode einer kombinatorischen Logik, welche aus den Argumenten ein Ergebnis verzögerungsfrei berechnet. Eine Funktion gibt genau einen Wert über ihren Namen zurück und muss mindestens einen Parameter besitzen.

Die Funktion `maximum` im Beispiel 3.14 berechnet das Maximum zweier Werte, die beim Aufruf übergeben werden. Der `initial`-Block ruft die Funktion mehrfach mit Testwerten auf und gibt die Ergebnisse aus.

```
module function_example;

function maximum(
input reg A,
          B
);
if (A > B)
  maximum = A;
else
  maximum = B;
endfunction

initial begin
  $display ("maximum (0,0)=%d", maximum (0,0));
  $display ("maximum (1,0)=%d", maximum (1,0));
  $display ("maximum (0,1)=%d", maximum (0,1));
end
endmodule
```

```
maximum (0,0)=0
maximum (1,0)=1
maximum (0,1)=1
```

Beispiel 3.14 Funktion mit Simulationsausgabe

V11 if, else

Ist die Bedingung hinter `if` wahr (die Oder-Verknüpfung aller Bits ist `1`), wird die folgende (möglicherweise leere) Anweisung ausgeführt, andernfalls der `else`-Teil.

Im zweiten bis fünften Fall von Beispiel 3.15 ist die „Veroderung“ aller Bits `1`.

```
module if_test;
reg [7:0] DATA;                    // Variable mit 8 Bit

always @(DATA) begin
  $write ("Zeit: %2.0f, DATA = %b, ", $time, DATA);
  if (DATA)
    $display ("if-Zweig");
  else
    $display ("else-Zweig");
end

initial begin
  DATA = 0;            #10;
  DATA = 1;            #10;
  DATA = 100;          #10;
  DATA = 8'b0000_001x; #10;
  DATA = 8'b1111_111z; #10;
  DATA = 8'bxxxx_xxxx; #10;
  DATA = 8'bzzzz_zzzz; #10;
end
endmodule
```

```
Zeit:  0, DATA = 00000000, else-Zweig
Zeit: 10, DATA = 00000001, if-Zweig
Zeit: 20, DATA = 01100100, if-Zweig
Zeit: 30, DATA = 0000001x, if-Zweig
Zeit: 40, DATA = 1111111z, if-Zweig
Zeit: 50, DATA = xxxxxxxx, else-Zweig
Zeit: 60, DATA = zzzzzzzz, else-Zweig
```

Beispiel 3.15 Alternative `if-else`

V12 ?, :

Mit der Alternative `?` kann man in Zuweisungen ein `if-else` ersetzen. Im Beispiel 3.16 übernimmt `RESULT` den Wert von `SOURCE`, falls `ENABLE` wahr ist, sonst `8'bx`.

```
module selection;
reg ENABLE,
    RESULT;

always @(ENABLE) begin
  RESULT = ENABLE ? 0 : 1;
  $display ("Zeit: %2.0f, RESULT = %b", $time, RESULT);
end

initial begin
  ENABLE = 1;            #10;      // Aenderung von x auf 1
  ENABLE = 0;            #10;
  $finish;
end
endmodule
```

```
Zeit:  0, RESULT = 0
Zeit: 10, RESULT = 1
```

Beispiel 3.16 Alternative mit ?

V13 case, casez

Bei der Fallunterscheidung mit `case` wird der Auswahlausdruck in Klammern der Reihe nach mit jedem der Fälle bit-weise verglichen. Ist ein Vergleich erfüllt, d.h. stimmen beide Ausdrücke an allen Bit-Stellen wörtlich überein (Werte `0`, `1`, `x` bzw. `z`, V22), so wird die entsprechende Anweisung ausgeführt. Danach wird hinter dem `case` fortgefahren. Trifft kein Vergleich zu, wird, sofern vorhanden, die `default`-Anweisung ausgeführt.

Bei `casez` dagegen sind alle `z` in allen Ausdrücken „don't care“, d.h. ein Vergleich mit `z` ist automatisch wahr. Bei `case` wäre dies nur der Fall, wenn der andere Wert ebenfalls ein `z` ist.

Das Beispiel 3.17 verdeutlicht den Unterschied zwischen `case` und `casez`. Die `case`- und die `casez`-Anweisung geben jeweils aus, welchen Fall von `SELECT` sie erkannt haben. Der `initial`-Block erzeugt die Änderungen von `SELECT`. Ein `?` kann synonym zu `z` verwendet werden.

```
module case_and_casez;
reg [1:0] SELECT;

always @(SELECT) begin
  $write ("Zeit : %2.0f, SELECT = %b, ",
    $time, SELECT);

  $write ("case: ");
  case (SELECT)
    2'b00:   $write ("2'b00, ");
    2'b01:   $write ("2'b01, ");
    2'b0x:   $write ("2'b0x, ");
    2'b0z:   $write ("2'b0z, ");
    default: $write ("undef, ");
  endcase

  $write ("casez: ");
  casez (SELECT)
    2'b00:   $display ("2'b00");
    2'b01:   $display ("2'b01");
    2'b0x:   $display ("2'b0x");
    2'b0z:   $display ("2'b0z");
    default: $display ("undef");
  endcase

end

initial begin
  SELECT = 2'b00; #10;
  SELECT = 2'b0x; #10;
  SELECT = 2'b0z; #10;
  SELECT = 2'b??; #10;
end
endmodule
```

```
Zeit :  0, SELECT = 00, case: 2'b00, casez: 2'b00
Zeit : 10, SELECT = 0x, case: 2'b0x, casez: 2'b0x
Zeit : 20, SELECT = 0z, case: 2'b0z, casez: 2'b00
Zeit : 30, SELECT = zz, case: undef, casez: 2'b00
```

Beispiel 3.17 Fallunterscheidung mit `case` und `casez`

Zu den Zeitpunkten 0 und 10 stimmen die Ergebnisse überein. Zum Zeitpunkt 20 vergleicht die `casez`-Anweisung zuerst `SELECT` (mit dem Wert `2'b0z`) mit `2'b00`. Das Ergebnis ist positiv. Zum Zeitpunkt 30 passt bei `case` kein Wert, daher wird die `default`-Anweisung ausgeführt; bei `casez` passt bereits der erste Fall.

V14 while

Eine `while`-Schleife wird wie üblich bearbeitet. Beispiel 3.18 gibt die Zahlen 0 bis 10 aus.

```
module while_loop;
reg [3:0] COUNTER;

initial begin
  COUNTER = 0;
  while (COUNTER <= 10) begin
    $display ("%d", COUNTER);
    COUNTER = COUNTER + 1;
  end
end
endmodule
```

Beispiel 3.18 Schleife mit `while`

V15 `for`

Auch eine `for`-Schleife wird wie üblich bearbeitet. Beispiel 3.19 gibt die Zahlen von 0 bis 10 aus.

```
module for_loop;
reg [3:0] COUNTER;

initial
  for (COUNTER = 0; COUNTER <= 10; COUNTER = COUNTER + 1)
    $display ("%d", COUNTER);
endmodule
```

Beispiel 3.19 Schleife mit `for`

3.4 Variablen und Konstanten

V16 `integer`

`integer` definiert eine oder mehrere ganzzahlige Variablen. Die Breite ist in Abhängigkeit vom Simulator mindestens 32 Bit.

V17 `wire`

Variablen des Typs `wire` können pro Bit die Werte `0`, `1`, `x` (unbestimmt) und `z` (hochohmig) annehmen. Die Breite kann in Bit angegeben werden, indem sie der Liste

mit den Variablennamen vorangestellt wird. Sie ist dann für die ganze Liste gleich. Ein Wire stellt ein Verbindungsnetz zwischen mehreren Punkten dar. Ein Wire kann nicht speichern und daher keinen direkten Wert zugewiesen bekommen. Wohl aber kann er einen Wert durch eine „ständige Zuweisung“ (Continuous Assignment, V29) `assign` erhalten.

Durch die Instanzierung eines Untermoduls in einem Modul werden die Leitungen in der Parameterliste des Untermoduls mit denen des Moduls verbunden; dies entspricht ebenfalls einer ständigen Zuweisung. Wurde dem Wire ein Register durch eine der oben angeführten Zuweisungsalternativen zugewiesen, so kann der Wert des Wire durch eine Zuweisung an das zugehörige Register gesteuert werden.

Das Hauptmodul `main` im Beispiel 3.21 instanziert ein Modul `A` vom Typ `a` und entsprechend `B`. Die Instanz `A` generiert Rechteckimpulse der Periode 20 für die anderen Module. Da sie die Variable `CLOCK_A` beschreibt, muss diese einen zugewiesenen Wert speichern können und daher ein Register sein (V18). Durch die Instanzierung

```
a A (CLOCK_MAIN);
```

wird das Register `CLOCK_A` von `A` mit dem Wire `CLOCK_MAIN` des Hauptmoduls verbunden (Bild 3.20). `CLOCK_MAIN` darf nicht vom Typ `reg` sein, da bereits `CLOCK_A` eine Registervariable ist. Die Instanzierung

```
b B (CLOCK_MAIN, CLOCK_MAIN_2);
```

verbindet den Wire `CLOCK_B` mit dem Wire `CLOCK_MAIN` des Hauptmoduls und damit mit dem Register `CLOCK_A`. `CLOCK_B` muss ebenfalls vom Typ `wire` sein, da er indirekt mit dem Register `CLOCK_A` verbunden ist.

Durch solche Zuweisungen können sich also Ketten von Wires über mehrere Module hinweg bilden. Dabei ist jedoch nur der Anfang der Kette ein Register; alle anderen Leitungen sind Wires. Jeder Wire kann natürlich seinen momentanen Wert durch eine normale (vorübergehende) Zuweisung an Register übergeben (V27, V28).

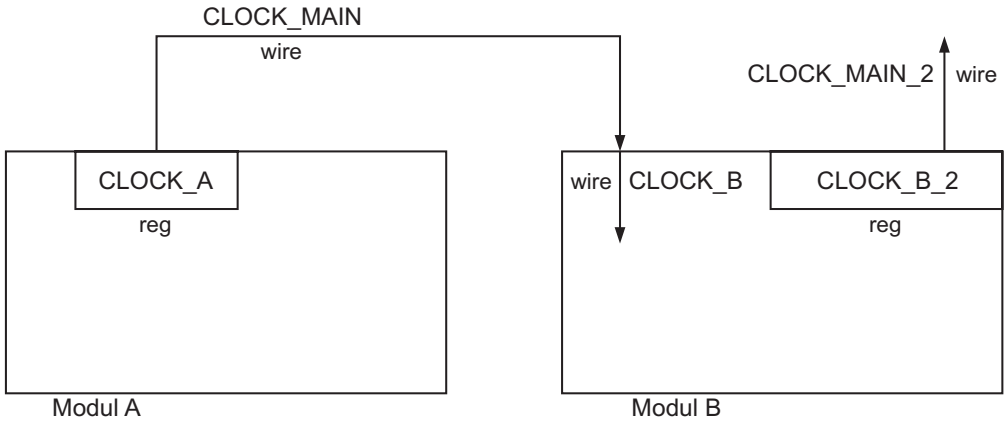


Bild 3.20 Wires verbinden Module

```
module a (
output reg CLOCK_A
);

always begin
  CLOCK_A = 0;                    // Clock auf 0 setzen
  #10;                            // 10 Zeiteinheiten warten
  CLOCK_A = 1;                    // Clock auf 1 setzen
  #10;                            // 10 Zeiteinheiten warten
end
endmodule // a


module b(
input  wire CLOCK_B,              // Haupttakt
output reg  CLOCK_B_2             // halb so schneller Takt
);

always begin
  CLOCK_B_2 = 0;                  // Clock auf 0
  @(CLOCK_B);                     // zwei Haupttakte warten
  @(CLOCK_B);
  CLOCK_B_2 = 1;                  // Clock auf 1
  @(CLOCK_B);                     // zwei Haupttakte warten
  @(CLOCK_B);
end
endmodule // b


module main;
wire CLOCK_MAIN,
     CLOCK_MAIN_2;

a A (CLOCK_MAIN);                  // Instanzen
b B (CLOCK_MAIN, CLOCK_MAIN_2);

always @(CLOCK_MAIN)
  $display ("CLOCK_MAIN");

always @(CLOCK_MAIN_2)
  $display ("CLOCK_MAIN_2");

initial begin
  #100;                           // 100 Zeiteinheiten warten
  $finish;                        // und die Simulation beenden
end
endmodule // main
```

```
CLOCK_MAIN
CLOCK_MAIN_2
CLOCK_MAIN
CLOCK_MAIN_2
CLOCK_MAIN
CLOCK_MAIN
CLOCK_MAIN_2
CLOCK_MAIN
CLOCK_MAIN
CLOCK_MAIN_2
CLOCK_MAIN
CLOCK_MAIN
CLOCK_MAIN_2
CLOCK_MAIN
CLOCK_MAIN
CLOCK_MAIN_2
```

Beispiel 3.21 Wires

V18 reg

Variablen des Typs `reg` können pro Bit die Werte `0`, `1`, `x` (unbestimmt) und `z` (hochohmig) annehmen. Dabei kann die Breite der Variablen in Bit angegeben werden, indem sie der Liste mit den Variablennamen vorangestellt wird. Die Breite ist bei allen Variablen der Liste gleich. Eine Registervariable (kurz: Register) stellt ein Speicherelement dar und behält einen zugewiesenen Wert bis zur nächsten Zuweisung.

Im Beispiel 3.22 gibt ein Zähler die Zahlen von 1 bis 10 aus.

```
module counter;
reg [3:0] R;

initial
  for (R=1; R <= 10; R = R + 1)
    $display ("R= %d", R);
endmodule
```

Beispiel 3.22 Register

V19 Felder von Variablen

Variablen können aus einem oder mehreren Bits bestehen. Zusätzlich gibt es Felder von Variablen. Beispielsweise wird ein eindimensionales Feld `A` mit 1000 Variablen der Breite 1 Bit vom Typ reg definiert durch

```
reg A [1:1000]   oder   reg A [1000:1]
```

Auf die 500. Variable dieses Feldes greift

```
RESULT = A[500];
```

zu. Die Syntax entspricht der Auswahl des 500. Bits einer normalen Variablen. Ein Feld mit 1000 Registern der Breite 16 Bit definiert

```
reg [16:1] A [1:1000];
```

Auf das 8. Bit der 500. Variablen greift man so zu:

```
RESULT = A [500][8];
```

Mehrdimensionale Felder sind ebenfalls möglich. Ein dreidimensionales Feld aus 16-Bit-Registern definiert beispielsweise

```
reg [16:1] B [1:100][1:200][1:300];
```

Dabei kann statt [1:100] auch [100:1] verwendet werden.

V20 Konstante

Konstanten können zu unterschiedlichen Basen gebildet werden. Wird keine Basis angegeben, wird 10 verwendet. Es sind die Basen 2 ('b), 8 ('o), 10 ('d' oder gar nichts) sowie 16 ('h) möglich. Mit der Angabe einer Konstantenbreite wird die Angabe einer Basis zwingend. Die Breite der Konstante wird der Basis als Dezimalzahl vorangestellt.

Es sind lediglich die Ziffern der gewählten Basis zulässig (bei der Basis 2 etwa nur 0 und 1) sowie die Zeichen x, z, ? und _. Den Wert „hochohmig“ stellt z dar. Als „unbestimmt“ wird x interpretiert. Das Zeichen _ dient nur der besseren Lesbarkeit. Die Bedeutung von z und x wird in Beispiel 3.24 näher erläutert.

Im Beispiel 3.23 werden dem Register DATA einige typische Konstanten zugewiesen und ausgegeben. Die Formatierung ist nicht mehr %d, sondern %b zur Ausgabe einer Binärzahl.

```
module constant_example;
reg  [7:0] DATA;

initial begin
  DATA =           0; $display ("DATA = %b", DATA);  // dezimal   0
  DATA =          10; $display ("DATA = %b", DATA);  // dezimal  10
  DATA =        'h10; $display ("DATA = %b", DATA);  // dezimal  16
  DATA =        'b10; $display ("DATA = %b", DATA);  // dezimal   2
  DATA =         255; $display ("DATA = %b", DATA);  // dezimal 255
  DATA =        1'b1; $display ("DATA = %b", DATA);
  DATA =  'bxxxxzzzz; $display ("DATA = %b", DATA);
  DATA = 'b1010_1010; $display ("DATA = %b", DATA);
  DATA =        1'bz; $display ("DATA = %b", DATA);
  DATA =         'bz; $display ("DATA = %b", DATA);
end
endmodule
```

```
DATA = 00000000
DATA = 00001010
DATA = 00010000
DATA = 00000010
DATA = 11111111
DATA = 00000001
DATA = xxxxzzzz
DATA = 10101010
DATA = 0000000z
DATA = zzzzzzzz
```

Beispiel 3.23 Konstanten

Durch

```
DATA = 1'b1;
```

erhält DATA den Wert 1, obwohl die Konstante explizit die Weite 1 Bit hat. Die restlichen Bits werden durch 0 ergänzt. In

```
DATA = 'bz;
```

wird keine Breite angegeben; es wird in diesem Fall mit z statt mit 0 ergänzt.

Beispiel 3.24 erläutert die Bedeutung von x und z genauer. Die zwei Register REG1 und REG2 treiben beide den Wire W. Dies entspricht dem Zusammenschalten zweier möglicherweise hochohmiger Treiber an eine gemeinsame Ausgangsleitung. Sind jedoch beide Treiber nicht hochohmig, so kommt es bei unterschiedlichen Ausgangswerten zu Konflikten.

```
module constant_example_2;
reg  [7:0] REG1,
           REG2;
wire [7:0] W = REG1;
assign     W = REG2;

initial begin
                                  #1; $display ("W = %b", W);
  REG1 =  0; REG2 =            0; #1; $display ("W = %b", W);
  REG1 = 10; REG2 =         8'bz; #1; $display ("W = %b", W);
             REG2 = 8'b11111111; #1; $display ("W = %b", W);
             REG2 =         8'bx; #1; $display ("W = %b", W);
             REG2 =         8'bz; #1; $display ("W = %b", W);
end
endmodule
```

```
W = xxxxxxxx
W = 00000000
W = 00001010
W = xxxx1x1x
W = xxxxxxxx
W = 00001010
```

Beispiel 3.24 x und z

Zum Beginn der Simulation sind REG1 und REG2 noch undefiniert mit x in allen Bit-Positionen. Ihre Verknüpfung liefert ebenfalls überall x. Werden beide Register auf 0 gesetzt, kommt es zu keinem Konflikt. Durch

```
REG1 = 10; REG2 = 8'bz;
```

wird der zweite Treiber hochohmig, sodass das erste Register den Ausgang W alleine treibt. Nach der Zuweisung

```
REG2 = 8'b11111111;
```

kommt es nur an Bit-Positionen mit unterschiedlichen Werten zu Konflikten. Mit

```
REG2 = 8'bx;
```

werden dann alle Bits undefiniert, was sich auf den Ausgang überträgt. Wichtig ist der Unterschied zwischen den beiden Fällen, bei denen alle Bits von REG2 auf z und auf x gesetzt sind. Im ersten Fall beeinflusst dieses Bit weitere angeschlossene Leitungen *nicht*; im zweiten Fall „treibt“ die Variable die angeschlossenen Leitungen mit einem undefinierten Wert.

3.5 Operationen

V21 Arithmetik +, -, *, /, %

+, -, * und / haben die übliche Bedeutung. Der Operator % liefert den Rest einer Division. + und - können auch das Vorzeichen angeben.

V22 Vergleich ==, !=, ===, !==, <, >, <=, >=

Beim Vergleich von Ausdrücken bedeuten ==, !=, <, >, <= sowie >= „gleich“, „ungleich“, „kleiner“ usw.; === und !== berücksichtigen auch die Werte x und z. Während erstere Operatoren auf einer *logischen* Ebene vergleichen, tun === und seine Negation !== dies *wörtlich*. Die Unterschiede erläutern Beispiel 3.25 und Tabelle 3.26.

```
module compare;

initial begin
  $display ("(1'bx  ==  1'b1 ) = %b", 1'bx  ==  1'b1);
  $display ("(1'bx  === 1'b1 ) = %b", 1'bx  === 1'b1);
  $display ("(1'bx  ==  1'bx ) = %b", 1'bx  ==  1'bx);
  $display ("(1'bx  === 1'bx ) = %b", 1'bx  === 1'bx);
  $display ("(1'bz  !=  1'b1 ) = %b", 1'bz  !=  1'b1);
  $display ("(1'bx  <=  1'b1 ) = %b", 1'bx  <=  1'b1);
  $display ("(2'bxx ==  2'b11) = %b", 2'bxx ==  2'b11);
end
endmodule
```

```
(1'bx  ==  1'b1 ) = x
(1'bx  === 1'b1 ) = 0
(1'bx  ==  1'bx ) = x
(1'bx  === 1'bx ) = 1
(1'bz  !=  1'b1 ) = x
(1'bx  <=  1'b1 ) = x
(2'bxx ==  2'b11) = x
```

Beispiel 3.25 Vergleichsoperatoren

„Logische Gleichheit“ bedeutet, dass beide Seiten mit 0 oder 1 definiert und gleich sind. Der erste logische Vergleich zwischen 1'bx und 1'b1 liefert x, da der erste Ausdruck undefiniert ist. Der zweite und „wörtliche Vergleich“ liefert 0 (falsch), da ein x ein anderer Wert als 1 ist.

Der Vergleich zweier Unbestimmter x ist logisch unbestimmt, wörtlich dagegen gleich (Ergebnis x bzw. 1). Zu beachten ist beim letzten Vergleich, dass das Ergebnis 1 Bit breit ist, obwohl die verglichenen Ausdrücke jeweils zwei Bit umfassen.

linke Seite	rechte Seite	Vergleich mit ==	!=	===	!==
1	1	1	0	1	0
0	1	0	1	0	1
1	x	x	x	0	1
0	x	x	x	0	1
x	x	x	x	1	0
z	x	x	x	0	1
z	1	x	x	0	1
z	z	x	x	1	0

Tabelle 3.26 Vergleichen

V23 Logik !, &&, ||

Die logischen Operatoren Negation (`!`), Und (`&&`) und Oder (`||`) verknüpfen mehrere Operanden. Beispielsweise kann eine `if`-Bedingung die Erfüllung mehrerer Bedingungen fordern. Da jeder Operand aber neben den numerischen Werten auch `x` und `z` enthalten kann, muss für die logischen Operatoren auch die Verknüpfung dieser Werte definiert werden. Das Ergebnis einer Operation ist `x`, wenn in irgendeiner Bit-Position eines Operanden ein `x` oder `z` vorkommt. Eine Ausnahme gilt, wenn das Ergebnis durch einen der beiden Operanden bereits eindeutig festgelegt ist. So ist der Wert von

```
1'bx && 1'b0
```

`0`, da eine Und-Verknüpfung mit `0` stets `0` ergibt. Tabelle 3.27 definiert die Und-Verknüpfung.

&&	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

Tabelle 3.27 Und-Verknüpfung mit `x` und `z`

Beispiel 3.28 verdeutlicht Verknüpfungen auf der logischen Ebene.

```
module logical_operators;

initial begin
  $display ("(! 1'b1        ) = %b", ! 1'b1);
  $display ("(! 1'bx        ) = %b", ! 1'bx);
  $display ("(! 1'bz        ) = %b", ! 1'bz);
  $display ("(1'b1  && 1'b0 ) = %b", 1'b1  && 1'b0 );
  $display ("(1'bx  && 1'b0 ) = %b", 1'bx  && 1'b0 );
  $display ("(1'bx  && 1'b1 ) = %b", 1'bx  && 1'b1 );
  $display ("(1'bx  || 1'b0 ) = %b", 1'bx  || 1'b0 );
  $display ("(1'bx  || 1'b1 ) = %b", 1'bx  || 1'b1 );
  $display ("(2'b00 || 2'bxx) = %b", 2'b00 || 2'bxx);
end
endmodule
```

```
(! 1'b1        ) = 0
(! 1'bx        ) = x
(! 1'bz        ) = x
(1'b1  && 1'b0 ) = 0
(1'bx  && 1'b0 ) = 0
(1'bx  && 1'b1 ) = x
(1'bx  || 1'b0 ) = x
(1'bx  || 1'b1 ) = 1
(2'b00 || 2'bxx) = x
```

Beispiel 3.28 Logische Verknüpfungen

Auch der Ausdruck

```
1'bx || 1'b1
```

ergibt `1`, da der zweite Operand das Ergebnis bereits eindeutig festlegt. Die letzte Anweisung verknüpft zwei Ausdrücke mit jeweils zwei Bit, das Ergebnis ist gleichwohl nur ein Bit breit.

V24 Bit-weise Logik `~`, `&`, `|`

Die bit-weisen Operatoren Negation (`~`), Oder (`|`) und Und (`&`) erlauben die bit-weise Verknüpfung von Ausdrücken. Das Ergebnis hat die gleiche Breite wie der breiteste verknüpfte Ausdruck, dabei werden an jeder Bit-Stelle die Operanden entsprechend verknüpft. Da jedes Bit einen der vier Werte `0`, `1`, `x` oder `z` annehmen kann, müssen die bit-weisen Operatoren erweitert werden. Hier gilt die gleiche Regel wie bei den logischen Operatoren. Besitzt nur ein verknüpftes Bit den Wert `x` oder `z`, so ist das Ergebnis `x`, es sei denn, es ist bereits durch einen Operanden alleine bestimmt. Dies ist beispielsweise der Fall bei der Verknüpfung

```
1'bx & 1'bz & 1'b1 & 1'b0
```

mit dem Ergebnis `0`. Beispiel 3.29 wendet bit-weise Operatoren an.

```
module bitwise_operators;

initial begin
  $display ("(~ 4'bzx10        ) = %b", ~ 4'bzx10);
  $display ("(4'b001x & 4'b0x10) = %b", 4'b001x & 4'b0x10);
  $display ("(2'b1x | 2'b00    ) = %b", 2'b1x | 2'b00);
end
endmodule
```

```
(~ 4'bzx10        ) = xx01
(4'b001x & 4'b0x10) = 0010
(2'b1x | 2'b00    ) = 1x
```

Beispiel 3.29 Bit-weise Verknüpfungen

Die erste `$display`-Anweisung invertiert eine 4 Bit breite Variable. Die zweite zeigt eine Und-Verknüpfung, die dritte eine Oder-Operation. Zu beachten ist, dass das Ergebnis mehrere Bits hat.

V25 Konkatenation {}

Die Konkatenation hängt mehrere Ausdrücke zu einem neuen Ausdruck hintereinander. Zwei konkatenierte 3 Bit breite Ausdrücke sind 6 Bit breit. Beispiel 3.30 konkateniert drei Variablen der Breiten 3, 4 und 2.

```
module concatenation;

initial
  $display ("{3'b100, 4'bxxzz, 2'ha} = %b", {3'b100, 4'bxxzz, 2'ha});
endmodule
```

```
{3'b100, 4'bxxzz, 2'ha} = 100xxzz10
```

Beispiel 3.30 Konkatenation

Zu beachten ist die Angabe `2'ha` für den dritten Ausdruck,. Es soll ein Ausdruck mit dem Wert dezimal `10` in zwei Bits dargestellt werden, obwohl 4 Bits benötigt würden. Durch die Angabe `2'` werden jedoch nur die untersten beiden Bits verwendet.

Konstante Wiederholungsfaktoren sind möglich. Beispielsweise wird durch { 4 { 2'b00, 2'b11} } die Konstante 16'b0011001100110011 erzeugt.

V26 Shift <<, >>

Die Shift-Operation << verschiebt die Bits in die Richtung höherwertiger Bit-Positionen, >> in die andere Richtung. Dabei steht auf der rechten Seite die Shift-Distanz. Ist sie negativ, wird entgegengesetzt geschoben.

Beispiel 3.31 schiebt zuerst um 4 Bit nach links, dann 4 Bit nach rechts und schließlich um -4 Bit nach links, folglich nach rechts.

```
module shift;

initial begin
  $display ("%b", 8'b0000_1111 << 4);
  $display ("%b", 8'b0000_1111 >> 4);
  $display ("%b", 8'b0000_1111 << -4);
end
endmodule
```

```
11110000
00000000
00000000
```

Beispiel 3.31 Shiften

3.6 Zuweisungen

V27 Blockende Zuweisung =

Durch die blockende Zuweisung wird einer Variablen oder einem Teil davon auf der linken Seite einmalig die rechte Seite zugewiesen. Die linke Seite muss vom Typ `reg` oder `integer` sein. Folgt beispielsweise in

```
A = #2 B
```

hinter dem = eine Zeitkontrolle (Abschnitt 3.2), so werden der momentane Wert der rechten Seite in einen temporären, transparenten Speicher kopiert und zunächst die angegebene Zeit bzw. das angegebene Ereignis abgewartet. Danach findet die eigentliche Zuweisung statt.

Der Name „blockend“ rührt daher, dass die Zuweisung zuerst vollständig ausgeführt wird, bevor die nächste Anweisung bearbeitet wird, selbst wenn die Zuweisung einen zeitkonsumierenden Anteil enthält. Die Ausführung der nächsten Anweisung wird also „blockiert“. Viele weitere Erläuterungen finden sich bei der nichtblockenden Zuweisung in V28.

V28 Nichtblockende Zuweisung <=

Mit der blockenden Zuweisung = aus V27 wird der linken Seite die rechte Seite sofort zugewiesen, sofern nicht zwischen dem Zeichen = und der rechten Seite eine Zeitkontrolle steht. Sie befindet sich typischerweise in einem sequenziellen oder prozeduralen Ablauf, etwa einer Verhaltensbeschreibung. Der Name „blockend“ rührt daher, dass die Zuweisung zuerst vollständig ausgeführt wird, bevor die nächste Anweisung bearbeitet wird, selbst wenn die Zuweisung einen zeitkonsumierenden Anteil enthält. Die Ausführung der nächsten Anweisung wird also „blockiert“.

Die nichtblockende Zuweisung <= dagegen erlaubt Zuweisungen, die den prozeduralen Ablauf nicht unterbrechen. Nichtblockende Anweisungen sind typischerweise sinnvoll, wenn innerhalb eines Zeitschrittes mehrere Zuweisungen an getaktete Register durchgeführt werden sollen, unabhängig von ihrer Reihenfolge oder ihrer gegenseitigen Abhängigkeit (Abschnitt 4.2).

Die Syntax nichtblockender Zuweisungen entspricht genau der der blockenden. Durch den Kontext wird dieser Operator vom Vergleichsoperator „kleiner-gleich“ unterschieden. Der Simulator bearbeitet eine nichtblockende Zuweisung wie folgt.

1. Die rechte Seite wird berechnet und für den Zeitpunkt geplant, der durch eine Zeitkontrolle vorgegeben ist; fehlt diese, wird die rechte Seite für den jetzigen Zeitpunkt geplant.

2. Am *Ende* des geplanten Zeitschritts wird die Zuweisung ausgeführt, indem die linke Seite ihren Wert erhält.

„Am Ende eines Zeitschritts“ bedeutet, dass die nichtblockenden Zuweisungen die letzten Anweisungen sind, die in diesem Zeitschritt ausgeführt bzw. abgeschlossen werden. (Es gibt jedoch Ausnahmen: eine nichtblockende Zuweisung kann eine weitere blockende Zuweisung im gleichen Zeitpunkt auslösen; der Simulator verarbeitet letztere dann nach der Durchführung der nichtblockenden Zuweisung. Auf solche für die Praxis unerheblichen Spitzfindigkeiten wird an dieser Stelle nicht weiter eingegangen.)

```
module blocking_1;                                  //00
                                                    //01
reg A, B;                                           //02
                                                    //03
initial begin                                       //04
  $monitor ("A=%b B=%b", A, B);                     //05
                                                    //06
  A = 0;                                            //07
  B = 1;                                            //08
                                                    //09
  A = B;                                            //10
  B = A;                                            //11
end                                                 //12
                                                    //13
endmodule                                           //14
```

```
A=1 B=1
```

Beispiel 3.32 Blockende Zuweisungen werden nacheinander ausgeführt

Das Beispiel 3.32 produziert genau das Verhalten „herkömmlicher“ Programmiersprachen.

```
module blocking_2;                              //00
                                                //01
reg A, B;                                       //02
                                                //03
initial begin                                   //04
  $monitor ("A=%b B=%b", A, B);                 //05
                                                //06
  A = 0;                                        //07
  B = 1;                                        //08
                                                //09
  A <= B;                                       //10
  B <= A;                                       //11
end                                             //12
                                                //13
endmodule                                       //14
```

```
A=1 B=0
```

Beispiel 3.33 Austausch von Werten mit nichtblockenden Zuweisungen

```
module blocking_3;                                      //00
                                                        //01
reg A, B, C, D, E, F;                                   //02
                                                        //03
//blockende Zuweisungen                                 //04
initial begin                                           //05
  A = #10 1;                                            //06
  B = #2 0;                                             //07
  C = #4 1;                                             //08
end                                                     //09
                                                        //10
//nichtblockende Zuweisungen                            //11
initial begin                                           //12
  D <= #10 1;                                           //13
  E <= #2 0;                                            //14
  F <= #4 1;                                            //15
end                                                     //16
                                                        //17
initial                                                 //18
  $monitor (                                            //19
    "t=%2.0f   A=%b B=%b C=%b   D=%b E=%b F=%b",        //20
    $time, A, B, C, D, E, F);                           //21
                                                        //22
endmodule                                               //23
```

```
t= 0   A=x B=x C=x   D=x E=x F=x
t= 2   A=x B=x C=x   D=x E=0 F=x
t= 4   A=x B=x C=x   D=x E=0 F=1
t=10   A=1 B=x C=x   D=1 E=0 F=1
t=12   A=1 B=0 C=x   D=1 E=0 F=1
t=16   A=1 B=0 C=1   D=1 E=0 F=1
```

Beispiel 3.34 Nichtblockende Zuweisungen hemmen nicht die Ausführung nachfolgender Anweisungen

Beispiel 3.33 dagegen führt mit nichtblockenden Zuweisungen einen Wertetausch durch. In den Zeilen 10-11 berechnet der Simulator die rechten Seiten der nichtblockenden Zuweisungen, nämlich `1` von `B` und `0` von `A`, und weist sie am Ende des momentanen Zeitschritts den linken Seiten zu, sodass dann `A` den Wert `1` und `B` den Wert `0` hat.

In Beispiel 3.34 hemmt eine nichtblockende Zuweisung nicht die Ausführung nachfolgender Anweisungen. In den Zeilen 06-08 weist der Simulator dem Register `A` den Wert `1` zur Zeit 10 zu und `0` dem Register `B` zur Zeit 12 sowie `1` an `C` zur Zeit 16. In den Zeilen 13-15 dagegen weist der Simulator den Wert `1` an Register `D` wiederum zur Zeit 10 zu, aber den Wert `0` an Register `E` *nicht* zur Simulationszeit 10+2, sondern zur Zeit 2, und schließlich `1` an `F` zur Zeit 4.

Damit hat die nichtblockende Zuweisung auch eine parallele Natur. In der Register-Transfer-Logik und der Logiksynthese ist ihr Einsatz besonders empfehlenswert (Kapitel 4).

V29 `assign` (Ständige Zuweisung, Continuous Assignment)

Mit `assign` wird einem Wire oder einem Teil davon ständig ein Ausdruck auf der rechten Seite zugewiesen (Continuous Assignment). Sie wird stets neu berechnet, wenn sich die rechte Seite ändert. Die ständige Zuweisung kann außer durch `assign` auch implizit bei der Definition eines Wire wie im Beispiel 3.35 erfolgen. `REGISTER` wird implizit `WIRE1` und explizit `WIRE2` zugewiesen. Die beiden `always`-Blöcke überwachen `WIRE1` und `WIRE2` und geben bei einer Änderung die Simulationszeit und den aktuellen Wert aus. Die Änderungen werden von dem `initial`-Block erzeugt.

Zu beachten ist, dass sich zur Zeit 20 `REGISTER` zweimal ändert, dies aber nicht zu einer Änderung von `WIRE1` bzw. `WIRE2` führt, da diese beiden Änderungen innerhalb des `initial`-Blocks ununterbrochen ausgeführt werden, sodass die *parallel* ablaufenden ständigen Zuweisungen dies nicht bemerken.

```
module assign_test;
reg    REGISTER;
wire   WIRE1 = REGISTER;          // implizites Continuous Assignment
wire   WIRE2;
assign WIRE2 = REGISTER;          // explizites Continuous Assignment

always @(WIRE1)
  $display ("Zeit = %2.0f: WIRE1 = %d", $time, WIRE1);

always @(WIRE2)
  $display ("Zeit = %2.0f: WIRE2 = %d", $time, WIRE2);

initial begin
  REGISTER = 0; #10;
  REGISTER = 1; #10;
  REGISTER = 0;
  REGISTER = 1; #10;
  $finish;
end
endmodule
```

```
Zeit =  0: WIRE1 = 0
Zeit =  0: WIRE2 = 0
Zeit = 10: WIRE1 = 1
Zeit = 10: WIRE2 = 1
```

Beispiel 3.35 Ständige Zuweisung (Continuous Assignment)

3.7 Sonstige Befehle

V30 `define

Dies ist keine Anweisung im eigentlichen Sinne, sondern eine Vorgabe an einen Präprozessor, überall im folgenden Programm Text zu ersetzen. Den zu ersetzenden Zeichenketten ist ein ` voranzustellen. Dadurch lassen sich besser lesbare Programme schreiben. Unbedingt zu beachten ist, dass es keine *lokalen* `define-Anweisungen gibt, die nur innerhalb eines Moduls gültig sind. Ein `define gilt vielmehr immer bis zum Programmende. Werden mehrere Dateien hintereinander eingebunden, so gilt es bis zur letzten Datei. Auf diese Weise können bei einem größeren Projekt alle `define-Anweisungen in einer einzigen Datei zusammengefasst werden.

Das Beispiel 3.36 ruft sechsmal `Hallo`. Innerhalb von `module_2` werden `TIMES` und `TEXT` nicht durch ein zusätzliches `define definiert.

```
module module_1;

`define TEXT "Hallo"
`define TIMES 3

reg [2:0] COUNTER;

initial
  for (COUNTER = 1; COUNTER <= `TIMES; COUNTER = COUNTER + 1)
    $display (`TEXT);
endmodule


module module_2;
reg [2:0] COUNTER;

initial
  for (COUNTER = 1; COUNTER <= `TIMES; COUNTER = COUNTER + 1)
    $display (`TEXT);
endmodule
```

Beispiel 3.36 `` `define ``

V31 $display, $write

`$display` gibt seine Argumente auf dem Bildschirm aus und beginnt eine neue Zeile. Die Argumente bestehen entweder aus einem in Anführungszeichen eingeschlossenen Text mit optionalen Formatierungsanweisungen oder aus Ausdrücken, welche gemäß den Formatierungsanweisungen ausgegeben werden (Tabelle 3.37).

`\n`	neue Zeile
`\t`	Tabulator
`\\`	das Zeichen \
`\"`	Anführungszeichen
`%%`	das Zeichen %
`%h, %H`	Hexadezimalzahl
`%d, %D`	Dezimalzahl
`%o, %O`	Oktalzahl
`%b, %B`	Binärzahl
`%f, %F`	reelle Zahl
`%c`	einzelnes Zeichen
`%s`	Zeichenkette
`%t`	Zeit
`%m`	aktueller Modulname

Tabelle 3.37 Formatierungsanweisungen

Bei der Ausgabe von Hexadezimal-, Oktal- oder Binärzahlen werden soviele Stellen ausgegeben, wie der Größe des Ausdrucks entspricht. Daher werden führende Nullen, falls vorhanden, ebenfalls ausgegeben. Sollen diese unterdrückt werden, so enthält die Formatierung zusätzlich eine `0` (z.B. `%0b` statt `%b`). Die `$write`-

Anweisung besitzt die gleiche Funktionalität mit der Ausnahme, dass anschließend keine neue Zeile begonnen wird.

Das Beispiel 3.38 demonstriert einige Ausgaben. Bei der Zuweisung

```
STRING = "Testtext";
```

wird `Testtext` in 8 Bit große Zeichen zerlegt, die der Reihe nach zugewiesen werden.

```
module display_write;
reg [71:0] STRING;
reg  [7:0] VARIABLE;

initial begin
  STRING = "Testtext";
  $display ("Dies ist ein %s.", STRING);
  VARIABLE = 100;
  $write ("100 als Dezimalzahl: %d, als Hexadezimalzahl: %H\n",
    VARIABLE, VARIABLE);
  $display ("als Oktalzahl: %O, als Binaerzahl: %b", VARIABLE,VARIABLE);
  $display ("als Binaerzahl ohne fuehrende Nullen: %0b", VARIABLE);
end
endmodule
```

```
Dies ist ein  Testtext.
100 als Dezimalzahl: 100, als Hexadezimalzahl: 64
als Oktalzahl: 144, als Binaerzahl: 01100100
als Binaerzahl ohne fuehrende Nullen: 1100100
```

Beispiel 3.38 Ausgabe mit `$display` und `$write`

Die `$write`-Anweisung ist äquivalent zu einer `$display`-Anweisung, wenn durch `\n` am Ende der Zeichenkette eine neue Zeile begonnen wird.

V32 $finish

`$finish` beendet eine VERILOG-Simulation. Es kann an jeder Stelle im Programm stehen. Es werden sofort alle noch aktiven Module sowie `initial`- und `always`-Blöcke abgebrochen.

Eine `always`-Schleife ohne Ende kann wie im Beispiel 3.39 durch ein `$finish` in einem separaten `initial`-Block gestoppt werden.

```
module finish;
`define SIMULATIONTIME 4

always begin
  $display ("arbeite");
  #1;
end

initial begin
  #`SIMULATIONTIME;
  $finish;
end
endmodule
```

```
arbeite
arbeite
arbeite
arbeite
```

Beispiel 3.39 Simulationsende mit `$finish`

Ohne die Zeitkontrolle `#1` würde zur Zeit 0 der `always`-Block nie die Kontrolle abgeben.

V33 $readmemh, $readmemb

`$readmemb` sowie `$readmemh` lesen eine Datei in eine Variable ein, die ein Feld sein muss. Damit kann ein großes Feld bequem mit Werten einer Datei gefüllt werden. Die Datei darf lediglich Leerzeichen, Neue-Zeile-Zeichen, Tabulatoren, Kommentare sowie Binärzahlen (im Falle von `$readmemb`) bzw. Hexadezimalzahlen (im Falle von `$readmemh`) enthalten.

Beispiel 3.40 realisiert einen Speicher.

```
module memory (
input  wire [3:0] ADDRESS,                    // Adresse
output wire [7:0] DATA                        // Ausgabedaten
);
reg     [7:0] MEMORY [15:0];                  // 16 Worte je 8 Bit

assign        DATA = MEMORY [ADDRESS];        // staendige Zuweisung

initial
  $readmemb ("z:/Speicherdaten.txt", MEMORY); // MEMORY initialisieren
endmodule
```

Beispiel 3.40 Einlesen eines Feldes

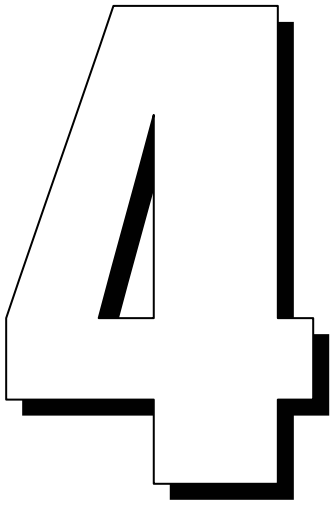
Wird eine Adresse an `ADDRESS` gelegt, so kann das Daten-Byte von `DATA` ausgelesen werden. Dabei wird zum Simulationsstart das Feld `MEMORY` mit dem Inhalt

der Datei `Speicherdaten` geladen. Dieses könnte den Aufbau von Beispiel 3.41 besitzen.

```
// Daten fuer den Speicher
0000_0000 0000_0001 0000_0010 0000_0011
0000_0100 0000_0101 0000_0110 0000_0111
0000_1000 0000_1001 0000_1010 0000_1011
0000_1100 0000_1101 0000_1110 0000_1111
```

Beispiel 3.41 Musterdaten

Dabei wird die erste Binärzahl dem Element des Feldes zugewiesen, welches dem linken Index der Deklaration entspricht, und so fort. Der Speicher wird in dem Beispiel mit den Werten 0 bis 15 initialisiert.



Modellierungskonzepte in VERILOG

Neben den VERILOG-Befehlen selbst ist dieses Kapitel besonders wesentlich und grundlegend.

4.1 Parallelität und Ereignissteuerung des Simulators

In „normalen“ Programmiersprachen wie C werden die Anweisungen der Reihe nach bearbeitet. Ein Programmzähler `PC` zeigt auf die jeweils aktuelle Anweisung. Nach Ausführung dieser Anweisung wird der `PC` um 1 erhöht oder bei Verzweigungen, Schleifen und Ähnlichem geeignet angepasst. Grundsätzlich gibt es nur *einen* Kontrollfluss.

In realen Schaltungen arbeiten alle Komponenten parallel, was sich in der Schaltungssimulation irgendwie niederschlagen muss. Beispielsweise kann eine Taktflanke an vielen Stellen zugleich Aktionen auslösen, oder es gibt parallel arbeitende `always`-Blöcke.

Insgesamt haben wir im vorigen Kapitel folgende Möglichkeiten zur Modellierung der Parallelität in VERILOG kennen gelernt.

1. (Instanzen von) Modulen (V1),
2. `initial`- und `always`-Blöcke (V4, V5),
3. ständige Zuweisungen (Continuous Assignments) (V28),
4. nichtblockende Zuweisungen (V27) und
5. Mischformen aus 1. bis 4.

Wir betrachten die hierbei verwendete ereignisgesteuerte Simulation zunächst etwas abstrakter. Eine globale Variable bestimmt den Simulationszeitpunkt. Zu jedem

Zeitpunkt können ein oder mehrere Ereignisse zur parallelen Ausführung vorgesehen sein. Ein Ereignis-Scheduler eines VERILOG-Simulators übernimmt die Stelle des Programmzählers. Bild 4.1 zeigt die Simulationszeitachse mit mehreren, zu verschiedenen Zeitpunkten vorgesehenen Ereignissen.

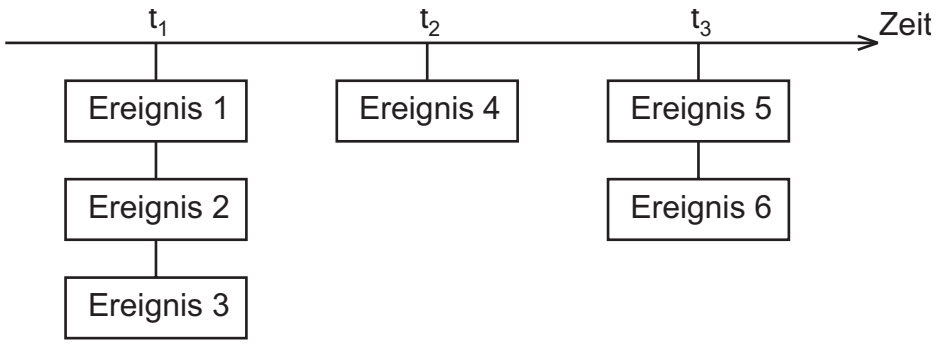


Bild 4.1
Zeit und Parallelität

Der Simulator führt alle Ereignisse aus, die zum gegenwärtigen Simulationszeitpunkt t_1 vorgesehen sind und entfernt sie aus der momentanen Ereignisliste (Ereignisse 1, 2 und 3). Dies geschieht nicht durch echte parallele Bearbeitung, sondern durch sequenzielle Bearbeitung in zufällig ausgewählter Reihenfolge. Wenn keine weiteren Ereignisse zum gegenwärtigen Simulationszeitpunkt t_1 existieren, wird die Simulationszeit weitergeschaltet bis zum Zeitpunkt t_2 eines nächsten vorgesehenen Ereignisses. Durch die Ausführung von Ereignissen werden oft neue Ereignisse für die Zukunft oder sogar für den jetzigen Zeitpunkt erzeugt. Beispielsweise könnte die Ausführung von Ereignis 3 ein Ereignis 7 für t_3 erzeugen, Ereignis 5 ein Ereignis 8 sogar zum gleichen Zeitpunkt t_3 usw.

Die Reihenfolge der Bearbeitung der Ereignisse eines Simulationszeitpunktes ist im allgemeinen Fall unbestimmt.

Man darf sich daher nicht auf eine bestimmte Reihenfolge verlassen. Auch kann sie von Simulator zu Simulator verschieden sein: VERILOG-XL beispielsweise ordnet anders als VeriWell, ohne dass einer von beiden gegen die Spielregeln verstößt.

```
module event_control;

reg[4:0] N;

initial begin
  $display ("AAA");
  $display ("BBB");
end

initial
  for (N=0; N<=3; N=N+1)
    $display (N);

endmodule
```

Beispiel 4.2 Parallelität und Reihenfolge

Es ist jedoch sichergestellt, dass aufeinander folgender Code *ohne* Zeitkontroll-Anweisungen als einziges Ereignis ohne Unterbrechung bearbeitet wird. Beispiel 4.2 illustriert diesen Punkt. Außerdem ist sichergestellt, dass die Reihenfolge der Bearbeitung zwischen zwei gleichen Simulationsläufen gleich ist.
Die Ausführung des Moduls `event_control` produziert die Ergebnisse in Bild 4.3.

```
AAA
BBB
 0
 1
 2
 3
```

Bild 4.3 Simulationsergebnis zum Beispiel 4.2

Beide `initial`-Blöcke sind zum selben Simulationszeitpunkt 0 zur Bearbeitung vorgesehen. Ein anderer Simulator könnte daher

```
 0
 1
 2
 3
AAA
BBB
```

ausgeben. Dagegen mag das Ergebnis

```
AAA
 0
 1
BBB
 2
 3
```

in der Praxis zwar eine sinnvolle Möglichkeit darstellen, es ist hier aber unmöglich, da beide `initial`-Blöcke keine Zeitkontrollen enthalten.

Noch einmal: die Reihenfolge der Bearbeitung der Ereignisse eines Simulationszeitpunktes ist im allgemeinen Fall unbestimmt.

Leider verzweigt sich der Baum der möglichen Simulationszustände des Gesamtsystems in jedem Knoten mit dem Grade `n` zu jedem Zeitpunkt, zu dem `n` Ereignisse zur Bearbeitung anstehen. Der Simulationsbaum eines Systems kann daher im ungünstigen Fall extrem mächtig werden. Leider kann ein Simulator in diesem Baum nur *einen einzigen* Pfad auswählen.

Hierin liegt die große Crux des parallelen Modellierens. Denn *ein* richtiges Simulationsergebnis hat oft noch keine hohe Aussagekraft. Die Simulation des ganzen Baumes ist dagegen aus Komplexitätsgründen schlicht unmöglich. Da der Mensch parallele Ereignisse schlecht überschaut, kann hier nur zu größter Sorgfalt und Vorsicht geraten werden.

Immer dann, wenn parallele Ereignisse nicht völlig unabhängig arbeiten, sondern mit Signalen oder gemeinsamen Daten kommunizieren, sollte im Zweifel lieber eine

konservative Strategie angewendet werden. Dies kann synchron geschehen, indem Ereignisse für *verschiedene* Takt-Zeitpunkte vorgesehen werden. Oder es kann asynchron mit einem Handshake-Mechanismus mit Events oder Synchronisationsregistern die Reihenfolge erzwungen werden.

4.2 Pipelines und Register-Transfer-Logik

Die folgenden Beispiele erscheinen einfach, beinhalten jedoch sehr grundlegende Effekte zur Register-Transfer-Logik und zu parallelen Prozessen.

4.2.1 Eine Flipflop-Kette

Das Beispiel 4.5 (Bild 4.4) demonstriert eine sinnvolle Anwendung einer Zeitverzögerung. Es soll eine Mini-Pipeline simuliert werden, welche aus zwei aufeinander folgenden Speicherelementen besteht. Daten am ersten Speichereingang `INPUT` werden in der Pipeline um zwei Takte verzögert. Der erste Speicherausgang `MIDDLE` ist mit dem zweiten Speichereingang identisch. Die beiden Speichereingänge übernehmen anliegende Werte mit der positiven Flanke von `CP` und setzen ihren Ausgang neu. Dies geschieht in den ersten beiden `always`-Blöcken.

Bild 4.4 Flipflop-Kette

Die Funktionsfähigkeit des Beispiels steht und fällt mit der Zeitverzögerung `#1`, die unten erläutert wird. Der dritte `always`-Block gibt die beiden Eingänge und Ausgänge aus, falls sich einer der Werte ändert. Der vierte generiert einen Takt mit der Periode 20. Der `initial`-Block erzeugt Testwerte, indem er mit der negativen Flanke zu den Zeitpunkten 0, 20 und 40 neue Werte an `INPUT` zuweist. Nach einer positiven Flanke erreicht dieser `MIDDLE` und nach einer weiteren positiven Flanke `OUTPUT`.

Ohne die Zeitverzögerungen `#1` könnte bei einer positiven Flanke der Wert von `INPUT` an `MIDDLE` und zum gleichen Zeitpunkt an `OUTPUT` weitergegeben werden, quasi „durchrauschen“. Mit der Zeitverzögerung wird der Wert von `INPUT` zunächst in einem unsichtbaren Zwischenspeicher gehalten bis der alte Wert von `MIDDLE` nach dem gleichen Schema gerettet wurde. (Auf die zweite Zeitverzögerung könnte sogar verzichtet werden, die zweite Pipeline-Stufe wäre dann aber nicht mehr erweiterbar.)

```
module flipflop_chain;

reg       CP;
reg [7:0] INPUT,
          MIDDLE,
          OUTPUT;

always @(posedge CP)
  MIDDLE = #1 INPUT;

always @(posedge CP)
  OUTPUT = #1 MIDDLE;

always @(INPUT, MIDDLE, OUTPUT)
  $display ("Zeit: %2.0f,  INPUT = %h,  MIDDLE = %h,  OUTPUT = %h",
    $time, INPUT, MIDDLE, OUTPUT);

always begin
  CP = 0; #10;
  CP = 1; #10;
end

initial begin
  INPUT = 0;     #20;
  INPUT = 255;   #20;
  INPUT = 8'haa; #20;
  $finish;
end
endmodule
```

```
Zeit:  0,  INPUT = 00,  MIDDLE = xx,  OUTPUT = xx
Zeit: 11,  INPUT = 00,  MIDDLE = 00,  OUTPUT = xx
Zeit: 20,  INPUT = ff,  MIDDLE = 00,  OUTPUT = xx
Zeit: 31,  INPUT = ff,  MIDDLE = ff,  OUTPUT = 00
Zeit: 40,  INPUT = aa,  MIDDLE = ff,  OUTPUT = 00
Zeit: 51,  INPUT = aa,  MIDDLE = aa,  OUTPUT = ff
```

Beispiel 4.5 Flipflop-Kette mit = und #1

Den gleichen Effekt wie mit #1 können wir in Beispiel 4.6 erreichen, indem wir in Beispiel 4.5 das =#1 durch die nichtblockende Zuweisung <= ersetzen (V28). Denn auch darin steckt ein Warten zwischen der Aufnahme des Wertes der rechten Seite und der Zuweisung an die linke Seite zum Ende des Simulationsschrittes.

```
module flipflop_chain;

reg       CP;
reg [7:0] INPUT,
          MIDDLE,
          OUTPUT;

always @(posedge CP)
  MIDDLE <= INPUT;

always @(posedge CP)
  OUTPUT <= MIDDLE;

always @(INPUT, MIDDLE, OUTPUT)
  $display ("Zeit: %2.0f,  INPUT = %h,  MIDDLE = %h,  OUTPUT = %h",
    $time, INPUT, MIDDLE, OUTPUT);
```

```
always begin
  CP = 0; #10;
  CP = 1; #10;
end

initial begin
  INPUT = 0;     #20;
  INPUT = 255;   #20;
  INPUT = 8'haa; #20;
  $finish;
end
endmodule
```

Beispiel 4.6 Flipflop-Kette mit <=

4.2.2 Eine Pipeline der Register-Transfer-Logik

Die Flipflop-Kette in Beispiel 4.6 war die Vorstufe zu beliebigen Register-Transfer-Logiken. Offenbar war es dabei gar nicht nötig, zwischen den Registern `R1` und `R2` und ihren Ausgängen `MIDDLE` und `OUTPUT` zu unterscheiden, so dass wir jetzt nur noch die Register selbst modellieren.

Es geht um eine vierstufige Pipeline wie in Bild 4.7. Zwischen je zwei Stufen befindet sich jetzt kombinatorische Logik, also eine Funktion ohne Speicher, die berechnet wird, wenn ein Datum in der Pipeline von links nach rechts läuft.

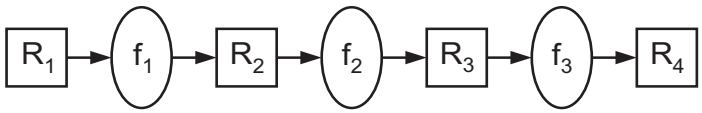


Bild 4.7 Pipeline aus Flipflops und kombinatorischer Logik

Zwischen den vier Registern sind die drei kombinatorische Logiken `f1` bis `f3`. Die Funktion der Pipeline ist

```
R4 = f3(f2(f1(R1))) = ((R1 * 2) + 5)^2.
```

Die Zeilen 21-26 in Beispiel 4.8 sind das Herz der Pipeline. In ihnen wird mit der positiven Flanke von `CP` jeweils für jede Stufe aus dem alten Wert des vorigen Registers ein neuer Wert berechnet.

Die kombinatorischen Logiken sind in den Zeilen 28-41 als Funktionen definiert. Den Takt generieren die Zeilen 43-47. `CP` ist `High` Schritte lang gültig und für `Low` Schritte 0.

In den Zeilen 49-61 werden nach Ausgabe einer Simulationskopfzeile Teststimuli synchron mit jeder fallenden `CP`-Flanke injiziert. Die `for`-Schleife lässt am Ende lediglich die Pipeline leer laufen.

`CP` und alle Register werden in den Zeilen 49-52 überwacht; bei jeder Änderung werden Simulationszeit und alle Zustände ausgegeben. In Zeile 50 verwenden wir

übrigens die Kurzform `always @(*)`, die den always-Block immer dann auslöst, wenn eine der im Block gelesenen Variablen sich ändert. Das wäre hier gleichbedeutend mit `always @(CP or R1 or R2 or R3 or R4)` oder etwas kürzer `always @(CP, R1, R2, R3, R4)`.

Die entscheidende Vereinfachung gegenüber der Mini-Pipeline in Beispiel 4.5 besteht in der nichtblockenden Zuweisung `<=` der Zeilen 23-25 (V28). Durch die Trennung der Berechnung der rechten Seiten und der Zuweisung an die linken Seiten ist ein ungetaktetes „Durchrauschen“ der Berechnung in einem Schritt ausgeschlossen.

```
//-------------------------------------------------------------------   //00
//                                                                     //01
// Pipeline mit nichtblockenden Zuweisungen                           //02
//                                                                     //03
//-------------------------------------------------------------------   //04
                                                                       //05
module pipeline #(                                                     //06
                                                                       //07
parameter     Low   = 10,          // Takt low                       //08
              High  =  5           // Takt high                      //09
);                                                                     //10
reg           CP;                  // Takt                           //11
                                                                       //12
reg     [7:0] R1,                  // Register 1                     //13
              R2,                  // Register 2                     //14
              R3,                  // Register 3                     //15
              R4;                  // Register 4                     //16
                                                                       //17
integer       I;                   // Hilfsvariable                  //18
                                                                       //19
                                                                       //20
// Pipeline steuern und Funktionen berechnen                         //21
 always @(posedge CP) begin                                            //22
   R2 <= f1(R1);                                                       //23
   R3 <= f2(R2);                                                       //24
   R4 <= f3(R3);                                                       //25
 end                                                                   //26
                                                                       //27
// Logik zwischen R1 und R2                                          //28
function [7:0] f1 (input [7:0] IN);                                    //29
  f1 = 2 * IN;                                                         //30
endfunction                                                            //31
                                                                       //32
// Logik zwischen R2 und R3                                          //33
function [7:0] f2 (input [7:0] IN);                                    //34
  f2 = IN + 5;                                                         //35
endfunction                                                            //36
                                                                       //37
// Logik zwischen R3 und R4                                          //38
function [7:0] f3 (input [7:0] IN);                                    //39
  f3 = IN * IN;                                                        //40
endfunction                                                            //41
                                                                       //42
// Ein-Phasen-Takt                                                   //43
always begin                                                           //44
  #Low  CP <= 1;               // Takt low                           //45
  #High CP <= 0;               // Takt high                          //46
end                                                                    //47
                                                                       //48
// Ueberwachung von CP sowie aller Register                          //49
always @(*)                                                            //50
  $display ("%4.0f   %b   %d    %d    %d    %d",                       //51
  $time, CP, R1, R2, R3, R4);                                          //52
                                                                       //53
// Ausgabe, Testmuster                                               //54
initial begin                                                          //55
  $display ("Zeit  CP    R1     R2     R3     R4\n");                  //56
                                                                       //57
  @(negedge CP) R1 <= 1;           // R1 eingeben                    //58
  @(negedge CP) R1 <= 2;           // R1 eingeben                    //59
  @(negedge CP) R1 <= 3;           // R1 eingeben                    //60
  @(negedge CP) R1 <= 4;           // R1 eingeben                    //61
                                                                       //62
  for(I=1;I<=5;I=I+1)              // Pipeline leeren                //63
    @(posedge CP);                                                     //64
  $finish;                                                             //65
end                                                                    //66
                                                                       //67
```

```
endmodule // pipeline                                          //68
```

```
Zeit  CP    R1     R2     R3     R4

  10   1     x      x      x      x
  15   0     x      x      x      x
  15   0     1      x      x      x
  25   1     1      x      x      x
  25   1     1      2      x      x
  30   0     1      2      x      x
  30   0     2      2      x      x
  40   1     2      2      x      x
  40   1     2      4      7      x
  45   0     2      4      7      x
  45   0     3      4      7      x
  55   1     3      4      7      x
  55   1     3      6      9     49
  60   0     3      6      9     49
  60   0     4      6      9     49
  70   1     4      6      9     49
  70   1     4      8     11     81
  75   0     4      8     11     81
  85   1     4      8     11     81
  85   1     4      8     13    121
  90   0     4      8     13    121
 100   1     4      8     13    121
 100   1     4      8     13    169
 105   0     4      8     13    169
 115   1     4      8     13    169
```

Beispiel 4.8 Pipeline mit nichtblockenden Zuweisungen <=

Die Pipeline-Ausgabe von Beispiel 4.8 ist in Bild 4.9 noch etwas professioneller grafisch dargestellt. Hierfür bietet der Simulator ISE/ModelSim eigene Befehle.

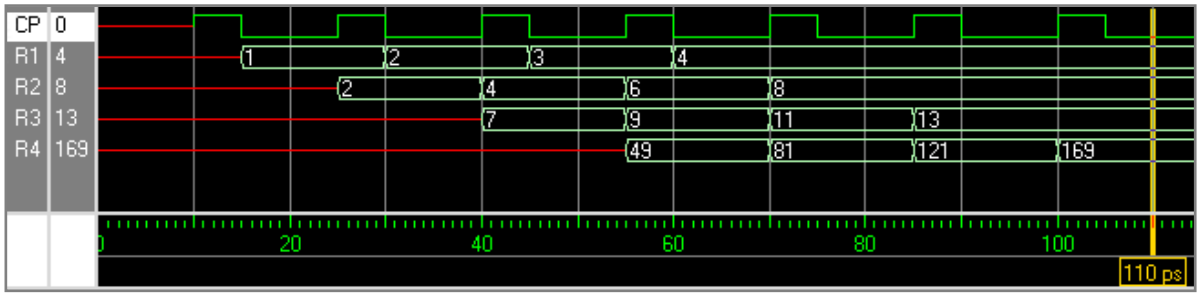


Bild 4.9 Grafische Pipeline-Ausgabe zu Beispiel 4.8

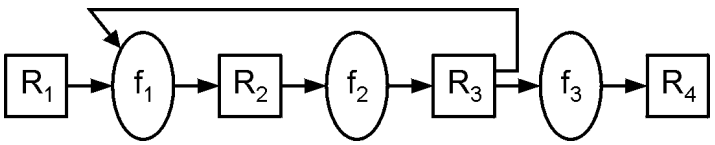


Bild 4.10 RTL-Netz aus Flipflops und kombinatorischer Logik

Die Pipeline kann drei Datensätze gleichzeitig bearbeiten. Pro Takt gibt sie ein fertiges Gesamtergebnis aus und ist daher drei mal so schnell wie die sequenzielle Berechnung aller drei Funktionen. Fügen wir in die Pipeline aus Bild 4.7 noch wie in Bild 4.10 eine Rückkopplung ein, wird aus der Pipeline ein allgemeines Automaten-

netz. Im Beispiel 4.8 ändert sich Zeile 23: aus `R2<=f1(R1)` wird `R2<=f1(R1,R3)`.

4.3 Bidirektionale Kommunikation

Mit bidirektionalen Verbindungen können zwei oder mehr Module durch einen gemeinsamen Datenbus in beiden Richtungen kommunizieren. Diese Konstruktion als Modell für reale bidirektionale Tristate-Busse ist nicht ganz einfach zu verstehen, dieser Abschnitt ist jedoch von grundlegender Bedeutung.

Das Beispiel 4.12 (Bild 4.11) enthält zwei Instanzen `M1` und `M2` eines Untermoduls vom Typ `m`. In den Erläuterungen unterscheiden wir die beiden Instanzen durch den vorangestellten Instanzennamen. `M1.DATA` ist demnach der formale Parameter `DATA` von Instanz `M1`.

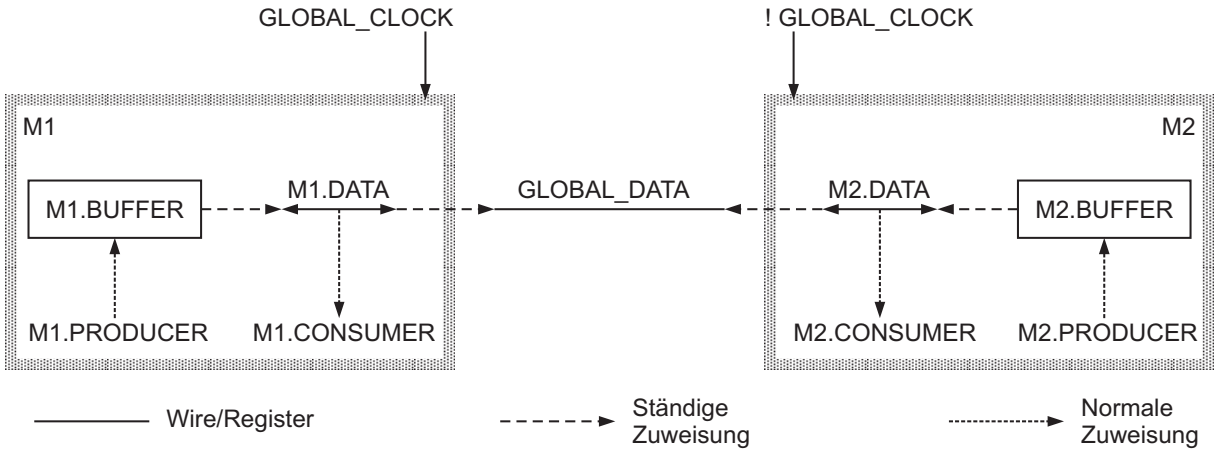


Bild 4.11 Bidirektionale Producer-Consumer

```
module m #(                                                                  //00
parameter       Start = 0                                                    //01
) (                                                                          //02
input wire      CLOCK,          // lokaler Takt                              //03
inout wire [7:0] DATA           // bidirektionaler Daten-Port                //04
);                                                                           //05
                                                                             //06
reg        [7:0] BUFFER;        // Ausgangstreiber                           //07
reg        [7:0] COUNTER;       // Zaehler                                   //08
assign           DATA = BUFFER; // staendige Zuweisung                      //09
                                                                             //10
initial                                                                      //11
  COUNTER = Start;              // von aussen initialisierter Zaehler        //12
                                                                             //13
always @(posedge CLOCK) begin                                                //14
  BUFFER = COUNTER;             // PRODUCER (schreiben)                      //15
  COUNTER = COUNTER + 1;                                                     //16
end                                                                          //17
                                                                             //18
always @(negedge CLOCK)                                                      //19
  BUFFER = 8'bz;                // PRODUCER (Treiber hochohmig)              //20
                                                                             //21
always @(CLOCK, DATA)                                                        //22
  if (Start == 0)                                                            //23
    $display ("%3.0f        %d                  %d", $time, CLOCK, DATA);    //24
  else                                                                       //25
    $display ("%3.0f                %d                %d",                   //26
```

```
      $time, CLOCK, DATA);          // CONSUMER (lesen und display)          //27
endmodule // m                                                              //28
                                                                            //29
                                                                            //30
                                                                            //31
module mainmodule;                                                          //32
wire [7:0] GLOBAL_DATA;             // Datenleitung zwischen Instanzen      //33
reg        GLOBAL_CLOCK;            // globaler Takt                        //34
                                                                            //35
defparam M1.Start =   0;            // Instanzen-Zaehler individuell        //36
defparam M2.Start = 100;            // initialisieren                       //37
                                                                            //38
m M1 ( GLOBAL_CLOCK, GLOBAL_DATA); // Instanz M1                            //39
m M2 (!GLOBAL_CLOCK, GLOBAL_DATA); // Instanz M2 mit invertiertem Takt      //40
                                                                            //41
initial begin                       // globaler Takt                        //42
  $display ("Zeit   M1.CLOCK  M2.CLOCK  M1.DATA  M2.DATA\n");               //43
  #10;                                                                      //44
  GLOBAL_CLOCK = 1; #10; GLOBAL_CLOCK = 0; #10;                             //45
  GLOBAL_CLOCK = 1; #10; GLOBAL_CLOCK = 0; #10;                             //46
  GLOBAL_CLOCK = 1; #10; GLOBAL_CLOCK = 0; #10;                             //47
end                                                                         //48
endmodule // mainmodule                                                     //49
```

Beispiel 4.12 Producer-Consumer-Kommunikation

Die Schnittstelle jeder Instanz besteht aus einem Eingang `CLOCK` und einem bidirektionalen Port `DATA` (Zeile 2). Die beiden Takteingänge werden durch eine externe `GLOBAL_CLOCK` stimuliert, wobei listigerweise `M2` durch den invertierten gegenphasigen Takt gesteuert wird. Die beiden Datenanschlüsse sind durch den externen Wire `GLOBAL_DATA` verbunden (Zeilen 38-39).

Innerhalb einer Instanz wird die Datenleitung in beiden Richtungen benutzt, das heißt es kann von innen nach außen geschrieben und von außen nach innen gelesen werden. Dazu ist zunächst in Zeile 5 eine ständige Verbindung (Continuous Assignment) geschaffen vom Register `BUFFER` zur Leitung `DATA`; demzufolge werden alle Werte von `BUFFER` und deren Änderungen auf den Bus `M1.DATA` bzw. `M2.DATA` und über die ständigen Verbindungen der Instanzenports auch auf die externe Leitung `GLOBAL_DATA` und damit auf `M2.DATA` bzw. `M1.DATA` weitergegeben. Die Gesamtleitung

```
M1.DATA - GLOBAL_DATA - M2.DATA
```

wird durch die *beiden* Quellen `M1.BUFFER` und `M2.BUFFER` getrieben. Sollten diese unterschiedliche 0-1-Werte haben, entsteht auf dem Datenbus der unbestimmte Zustand `x`. Stattdessen wird in Zeile 19 dafür gesorgt, dass während der jeweils zweiten lokalen Taktphase der `BUFFER` hochohmig gesetzt ist und damit auf der Datenleitung keinen Schaden anrichtet; da die beiden lokalen `M1.CLOCK` und `M2.CLOCK` gegenphasig arbeiten, kann hier kein Unglück passieren.

In jedem der beiden Untermodule gibt es einen `CONSUMER`, der ab Zeile 21 bei allen interessierenden Änderungen Werte vom Bus `DATA` abgreift. In diesem Fall ist der `CONSUMER` eine `$display`-Anweisung, es könnte genauso gut eine sich anschließende andere Datenverarbeitung sein. Bei jeder positiven lokalen Taktflanke schickt ein `PRODUCER`, in diesem Fall der `COUNTER` in Zeile 14, einen Wert an den `BUFFER` und damit auf die Datenfernleitung. Durch den Takt ist dafür gesorgt, dass

dies immer dann geschieht, wenn der BUFFER nicht hochohmig ist; das Schreiben von z in den BUFFER ist eine weitere Funktion des PRODUCER.

Die beiden Untermodule M1 und M2 vom Modultyp m sind nun zum einen durch ihren Instanzennamen personalisiert, zum anderen durch den in den Zeilen 35-36 unterschiedlich vereinbarten Startwert 0 bzw. 100 des COUNTER.

Damit spielen die beiden Instanzen Ping-Pong, indem beide ihren hochgezählten Startwert abwechselnd auf die Fernleitung schicken. Diese Nachricht wird vom Partnermodul verstanden und mit `$display` ausgegeben.

```
Zeit   M1.CLOCK  M2.CLOCK  M1.DATA  M2.DATA

 10                 0                   x
 10        1                  x
 10        1                  0
 10                 0                   0
 20                 1                   0
 20        0                  0
 20        0                100
 20                 1                 100
 30                 0                 100
 30        1                100
 30        1                  1
 30                 0                   1
 40                 1                   1
 40        0                  1
 40        0                101
 40                 1                 101
 50                 0                 101
 50        1                101
 50        1                  2
 50                 0                   2
 60                 1                   2
 60        0                  2
 60        0                102
 60                 1                 102
```

Bild 4.13 Simulationsergebnis zum Beispiel 4.12

Das Simulationsergebnis in Bild 4.13 zeigt, dass der Datenbus ständig Werte verschieden von x und z enthält. Außerdem erkennt man, dass beide Untermodule beide Zählerwerte abwechselnd ausgeben. Dieses Beispiel ist eine mögliche Lösung des *Producer-Consumer-Problems*.

5 Einführung in die Logiksynthese

Die Hardware-Beschreibungssprachen verdanken ihren Erfolg beim Entwurf digitaler Schaltungen vor allem auch den Fortschritten in der Logiksynthese. Mit Logiksynthese wird die Entwurfszeit beträchtlich reduziert. Ein Designer kann auf einer hohen Abstraktionsebene entwerfen und so seine Entwurfszeit reduzieren. In diesem Kapitel führen wir die Logiksynthese mit der HDL VERILOG nur oberflächlich ein. Die kleinen Beispiele sind typisch für Synthesesysteme wie Synopsys, Synplicity oder Xilinx XST. Sie stimmen daher natürlich nicht immer mit den Ergebnissen jedes einzelnen Werkzeugs genau überein.

5.1 Logiksynthese im Entwurfsprozess

Die Logiksynthese übersetzt eine Register-Transfer-Beschreibung eines Schaltungsentwurfs in eine optimierte Gatter-Darstellung, wobei eine Standard-Zellbibliothek sowie gewisse *Entwurfsbedingungen* (*Design-Constraints*) gegeben sind. Die Einordnung in den gesamten Entwurfsablauf zeigt Bild 5.1.

Eine Standard-Zellbibliothek kann aus einfachen Zellen bestehen wie grundlegenden Logikgattern (AND, OR, NOR usw.) oder Makrozellen wie Addierern, Multiplexern und speziellen Flipflops. Eine Standard-Zellbibliothek wird auch als *Technologie-Bibliothek* bezeichnet.

Eine Logiksynthese wurde früher vom Designer selbst durchgeführt. Zunächst dachte er sich eine Schaltungsarchitektur aus, zu der er auf dem Papier oder in einer Hardware-Beschreibungssprache ein High-Level-Modell angefertigt hat. Irgendwann hat er auch Bedingungen zum Timing, zur Fläche, zur Testbarkeit und zur Leistung beachtet.

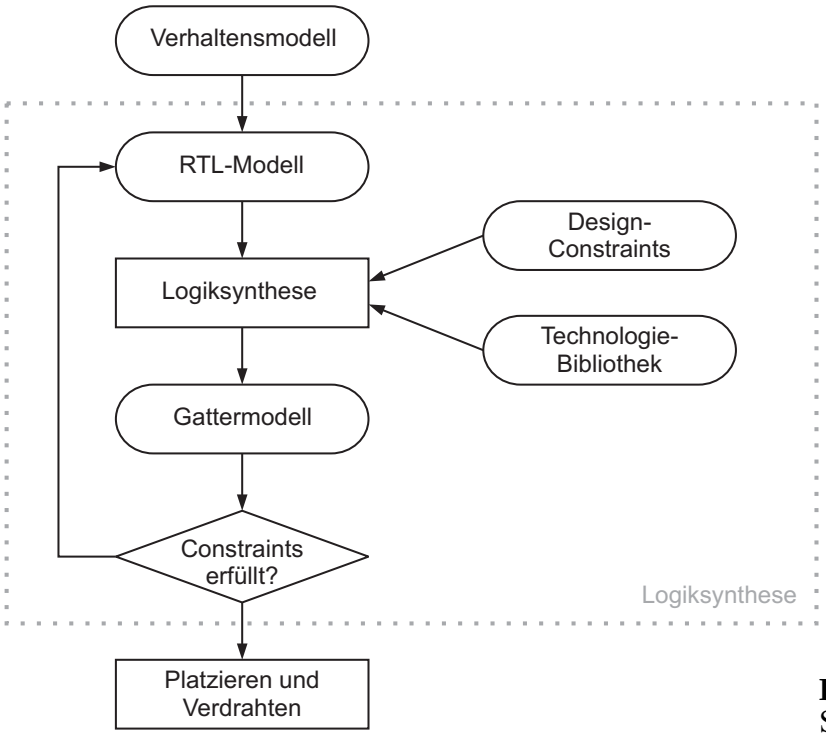


Bild 5.1 Grundlegende Schritte der Logiksynthese

Die eigentliche Logiksynthese bestand dann darin, das High-Level-Modell mithilfe der Zellen einer Standardbibliothek manuell zu einer Gatternetzliste zusammenzusetzen. Dieser letzte Schritt war ein sehr komplexer und zeitaufwändiger Teil des Entwurfs, zumal wenn mehrere Varianten des Gattermodells angefertigt werden mussten, um einen möglichst effizienten Entwurf zu erhalten.

Mit immer leistungsfähigeren Werkzeugen konnte die Synthese von Gattermodellen automatisiert werden. Dadurch kann sich der Designer jetzt auf die Wechselwirkungen auf der Architekturebene konzentrieren, auf die Einhaltung der Design-Constraints und auf die Optimierung einzelner Zellen einer Standard-Zellbibliothek. Mit dieser Standardbibliothek führt das Logiksynthese-Werkzeug mehrere Iterationen durch, um ein optimales Gattermodell zu errechnen.

Als Eingabe für die Logiksynthese haben sich Register-Transfer-Beschreibungen im Rahmen von Hardware-Beschreibungssprachen bewährt. VERILOG HDL wurde hierbei zu einer der zwei populärsten Sprachen.

5.2 Auswirkungen der Logiksynthese

Der kommerzielle digitale Schaltungsentwurf ist durch die Logiksynthese entscheidend produktiver geworden. Folgende Merkmale, Eigenschaften und Auswirkungen sind erwähnenswert.

- Die manuelle Anfertigung großer Gattermodelle war sehr fehleranfällig; jetzt dagegen geht es im Wesentlichen nur noch um High-Level-Fehler, die wesentlich besser zu überschauen sind.

- Über Zeit, Fläche und Leistung wusste man früher erst Bescheid, wenn der gesamte Gatterentwurf fertig und getestet war. Heute brauchen die Bedingungen nur noch vorgegeben zu werden. Sollten sie nicht automatisch erfüllt werden können, sind lediglich Änderungen auf der höheren Ebene der Register-Transfer-Beschreibung nötig.
- Während früher ein wesentlicher Teil der Entwurfszeit, oft viele Monate, mit dem Gatterentwurf vergingen, ist die Logiksynthese relativ schnell und kann in Stunden oder Tagen durchgeführt werden.
- What-if-Fragen waren früher schwer zu beantworten. Beispielsweise mag es zu einem Gattermodell mit einem Zyklus von 2 ns interessant sein, wie groß der Entwurf für einen Taktzyklus von 1,5 ns wäre. Diese Frage erfordert ein komplettes Redesign, das früher selten tatsächlich durchgeführt wurde. Heute lässt sich eine solche Alternative in Stunden erreichen.
- Bei größeren manuellen Entwürfen waren früher Teams von Designern notwendig, die oft recht unterschiedliche Entwurfsstile hatten. Ein großer Entwurf war dadurch schwerer zu verstehen und zu optimieren. Dagegen versuchen Werkzeuge der Logiksynthese, den Entwurf als Ganzes zu optimieren, allerdings unter der Einschränkung der Rechenzeit.
- Zeit, Fläche und Leistung von Bibliothekszellen sind herstellerspezifisch. Wenn daher früher der Halbleiterhersteller gewechselt wurde, musste in aller Regel das Gattermodell neu entwickelt werden. Heute dagegen werden die Eingaben der Logiksynthese auf der Register-Transfer-Ebene technologieunabhängig formuliert und brauchen bei einem Herstellerwechsel daher nur zusammen mit einer anderen Technologie-Bibliothek neu synthetisiert zu werden.
- Eine der wesentlichen Rationalisierungsbestrebungen besteht heute im Design-Reuse, d.h. immer größere Schaltungsteile sollen so standardisiert entwickelt werden, dass sie auch für andere spätere Entwürfe verwendbar sind, möglicherweise nach gewissen Änderungen. Wenn beispielsweise die Funktionalität eines I/O-Blockes eines Mikroprozessors sich nicht ändert, kann dieser bei einem abgeleiteten neuen Mikroprozessor unverändert übernommen werden. Ein solches Design-Reuse wird durch die Logiksynthese ganz wesentlich erleichtert, wenn nicht gar erst ermöglicht.

5.3 Synthese mit der HDL VERILOG

HDL-Beschreibungen für eine Logiksynthese werden gegenwärtig auf der Register-Transfer-Ebene (RTL) formuliert. In diesem Abschnitt diskutieren wir RTL-basierte Logiksynthese mit der Hardware-Beschreibungssprache VERILOG. Daneben gibt es im Rahmen der noch abstrakteren High-Level-Synthese Werkzeuge wie den Behavior-

Compiler, die eine Verhaltensbeschreibung oberhalb der RTL-Ebene synthetisieren. Obwohl seit Jahren Gegenstand intensiver Forschung, wird eine solche High-Level-Synthese bisher nur in Spezialfällen industriell eingesetzt.

5.3.1 Synthetisierbare VERILOG-Konstrukte

Nicht alle VERILOG-Konstrukte sind für eine Logiksynthese geeignet. Außerdem gibt es leichte Unterschiede bei den verschiedenen Synthesesystemen. Wesentliche erlaubte Befehle zeigt Tabelle 5.2.

Typ	Keyword oder Beschreibung	Anmerkungen
Ports	`input`, `inout`, `output`	
Parameter	`parameter`	
Modul-Definition	`module`	
Signale und Variablen	`wire`, `reg`	Vektoren erlaubt
Instanzierung	Modul- oder Gatter-Instanzen	z.B. `mymux M1 (OUT, I0, I1, S)`, `nand (OUT, A, B)`
Funktionen und Tasks	`function`, `task`	Timing-Konstrukte werden ignoriert
prozedural	`always`, `if`, `else`, `case`, `casez`	`initial` nicht möglich oder ignoriert
prozedurale Blöcke	`begin`, `end`, benannte Blöcke	
Datenfluss	`assign`	Zeitkonstrukte werden ignoriert
Schleifen	`for`	

Tabelle 5.2 Wesentliche VERILOG-Konstrukte der Logiksynthese

Eine der wichtigsten Einschränkungen der Logiksynthese besteht darin, dass alle Zeitverzögerungen mit # ignoriert werden. Dies ist nicht etwa eine Schwäche der Synthese, denn wie sollte in Hardware ein `#3` realisiert werden? Dabei ist ja sicher nicht an einen expliziten Verzögerungsbaustein gedacht. Andererseits ist diese Zeitverzögerung nicht ein überflüssiges Element von VERILOG, denn bei der Simulation von VERILOG-Modellen ist die Verzögerung ja gerade ein wichtiger Ersatz für die tatsächliche Laufzeit von Hardware.

Vielmehr hat das synthetisierte Modell seine eigenen Zeitverzögerungen, die letztlich aus der eingesetzten Technologie-Bibliothek resultieren. Eine wesentliche Konsequenz davon ist, dass Prä- und Post-Synthese-Simulationsergebnisse nicht übereinzustimmen brauchen. Der Designer selbst muss hier für eine geeignete Verifikation der beiden Simulationsmodelle sorgen und sollte möglichst einen

Entwurfsstil wählen, der solche Unterschiede reduziert. Dies gelingt im Allgemeinen durch vernünftig getaktete Register-Transfer-Beschreibungen recht gut.

Darüber hinaus wird der `initial`-Block in der Logiksynthese nicht unterstützt oder ignoriert. Stattdessen muss ein geeigneter Reset-Mechanismus zur Initialisierung der Signale im Schaltkreis benutzt werden.

Es wird empfohlen, dass alle Wortbreiten von Signalen und Variablen explizit spezifiziert werden. Andernfalls können bei der Synthese unnötig große Gattermodelle entstehen.

Operator-Typ	Operator-Symbol	Operation
arithmetisch	*	multiplizieren
	/	dividieren durch 2^n
	+	addieren
	-	subtrahieren
	%	Modulus
	+	Vorzeichen
	-	Vorzeichen
logisch	!	logische Negierung
	&&	logisches and
	||	logisches or
Relation	>	größer
	<	kleiner
	>=	größer gleich
	<=	kleiner gleich
Gleichheit	= =	Gleichheit
	! =	Ungleichheit
bit-weise	~	bit-weise Negierung
	&	bit-weises and
	|	bit-weises or
	^	bit-weises xor
	^~ oder ~^	bit-weises xnor
Reduktion	&	Reduktion mit and
	~&	Reduktion mit nand
	|	Reduktion mit or
	~|	Reduktion mit nor
	^	Reduktion mit xor
	^~ oder ~^	Reduktion mit xnor
Shift	>>	Rechts-Shift
	<<	Links-Shift
Konkatenation	{ }	Konkatenation
bedingt	?:	bedingt

Tabelle 5.3 VERILOG-Operatoren der Logiksynthese

5.3.2 VERILOG-Operatoren

Fast alle Operatoren in VERILOG sind in der Logiksynthese erlaubt, die in Tabelle 5.3 zusammengefasst sind; eingeführt wurden diese Operatoren ja bereits

andernorts. Nur die Operationen `===` und `!==` des wörtlichen Vergleichs, die sich auf `x` und `z` beziehen, machen wenig Sinn und sind daher entweder unzulässig oder werden wie == und != behandelt.

5.3.3 Umsetzung einiger VERILOG-Konstrukte

Anhand von ein paar Beispielen wollen wir nun verstehen, wie ein Synthese-Werkzeug Konstrukte interpretiert und sie in Gatterlogik übersetzt.

Anweisung `assign`

Die Anweisung `assign` ist ein grundlegendes Konstrukt, um kombinatorische Logik auf der RTL-Ebene zu beschreiben. Beispielsweise wird die kontinuierliche Zuweisung

```
assign OUT = (A & B) | C;
```

typischerweise in die Gatterkombination aus Bild 5.4 übersetzt.

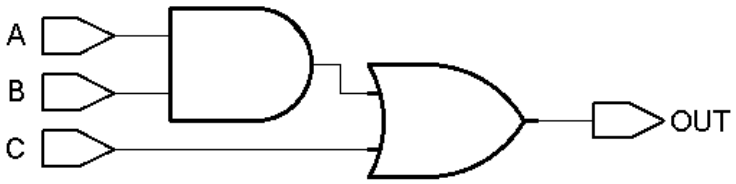


Bild 5.4
Gatter zu `(A&B) | C`

Sind dagegen `A`, `B`, `C` und `OUT` 2-Bit-Vektoren, würde der Schaltkreis aus Bild 5.5 entstehen. Dort ist übrigens der Input `A[1:0]` ein Bus, aus dem z.B. `A[1]` *extrahiert* wird.

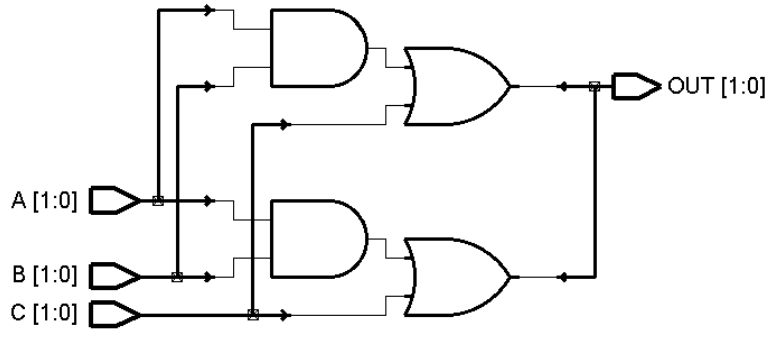


Bild 5.5 `(A&B) | C` mit Vektoren

Bei arithmetischen Operatoren hängt es davon ab, welche Hardware-Blöcke für die Synthese zur Verfügung stehen. Ein 1-Bit-Volladdierer

```
assign {C_OUT, SUM} = A + B + C_IN;
```

kann in der Gatterdarstellung von Bild 5.6 resultieren, während breitere Addierer meist nicht eine simple Vervielfältigung davon sind, sondern durch Optimierung zu anderen Gatterkombinationen führen.

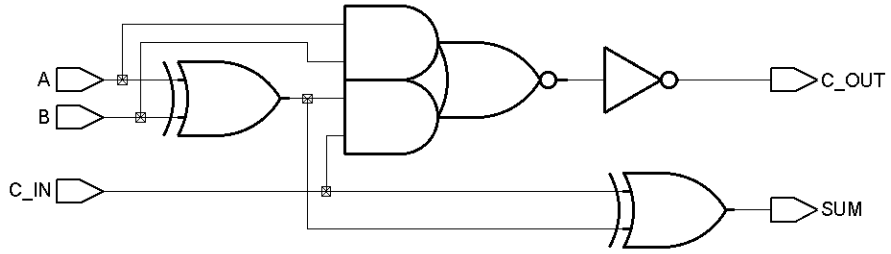


Bild 5.6 1-Bit-Volladdierer

Die VERILOG-Anweisung

```
assign OUT = (S) ? I1 : I0 ;
```

resultiert in einem Multiplexer, etwa wie in Bild 5.7.

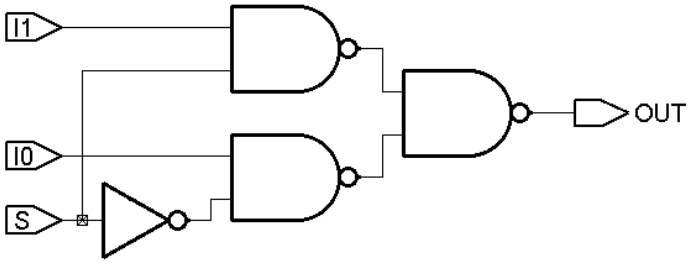


Bild 5.7 Multiplexer

Verzweigungen mit `if-else`

Den gleichen Multiplexer wie in Bild 5.7 erhalten wir mit

```
if (S) OUT = I1;
else   OUT = I0;
```

Im Allgemeinen werden mehrfache `if-else-if`-Anweisungen allerdings nicht in große Multiplexer übersetzt, sondern möglichst in effizientere Konstruktionen.

Die Fallunterscheidung `case`

Schließlich kann der Multiplexer aus Bild 5.7 durch die Fallunterscheidung

```
case (S)
  1´b0: OUT = I0;
  1´b1: OUT = I1;
endcase
```

erzeugt werden. Große `case`-Anweisungen erzeugen große Multiplexer oder verwandte Strukturen.

Schleifen mit `for`

Eine `for`-Schleife kann zur Konstruktion kaskadierter kombinatorischer Logik verwendet werden. So entsteht durch die Schleife in Beispiel 5.8 ein 8-Bit-Volladdierer.

```
C = C_IN;
for (I=0; I<=7; I=I+1)
  {C, SUM[I]} = A[I] + B[I] + C; // 8-Bit-Ripple-Addierer
C_OUT = C;
```

Beispiel 5.8 Kaskadierte kombinatorische Logik

Die Variable `C` wird während des Schleifendurchlaufs dynamisch verändert, indem links von `=` das neue, rechts das alte Carry steht.

Die Anweisung `function`

Funktionen werden in kombinatorische Blöcke mit einem Ausgang synthetisiert, der aus mehreren Bits bestehen kann. Beispiel 5.9 implementiert einen 4-Bit-Volladdierer, wobei das höchstwertige Bit der Funktion für das Carry benutzt wird.

```
function [4:0] fulladd;
input [3:0] A, B;
input C_IN;

begin
  fulladd = A + B + C_IN; // fulladd[3:0]: Summe
end                       // fulladd[4]:   Carry
endfunction
```

Beispiel 5.9 Funktion für einen kombinatorischen Volladdierer

Der `always`-Block

Mit dem fundamental wichtigen `always`-Block können sowohl kombinatorische Logik als auch sequenzielle Register-Transfer-Logik erzeugt werden. Für letztere muss der `always`-Block durch den Wechsel eines Taktsignals kontrolliert sein, etwa `posedge CLK`. Für

```
always @(posedge CLK)
  Q <= D;
```

wird ein positiv flankengesteuertes D-Flipflop wie in Bild 5.10a synthetisiert mit `D` als Eingang, `Q` als Ausgang und `CLK` als Taktsignal. Wir verwenden hierbei stets die nichtblockende Zuweisung `<=`.

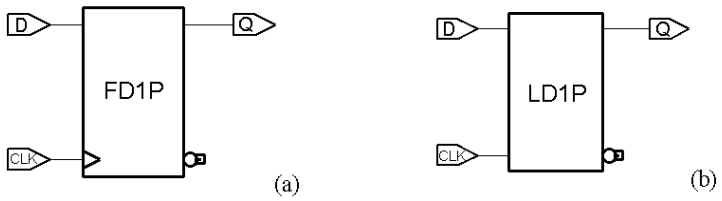


Bild 5.10 Register: flankengesteuertes D-Flipflop (a) und pegelgesteuertes Latch (b)

Ähnlich erzeugt die VERILOG-Beschreibung in Beispiel 5.11 das pegelgesteuerte Latch aus Bild 5.10b, also ein Register, das alle Wertänderungen während der gesamten positiven Taktphase weitergibt.

```
always @ (CLK, D)
  if (CLK)
    Q = D;
```

Beispiel 5.11 Pegelgesteuertes Latch

Dagegen erzeugt Beispiel 5.12 kombinatorische Logik ohne Register, hier erneut einen 1-Bit-Volladdierer wie in Bild 5.6.

```
always @ (A, B, C_IN)
  {C_OUT, SUM} = A + B + C_IN;
```

Beispiel 5.12 Volladdierer als kombinatorischer `always`-Block

Bei größeren kombinatorischen Logiken ist die Modellierung mit `always`-Blöcken oder als Funktion besser strukturierbar und lesbar als mit `assign`. Wir wiederholen an dieser Stelle das Bild 5.13 einer RTL-Logik aus dem vorigen Kapitel, das in der Form von Beispiel 5.14 voll synthetisierbar ist.

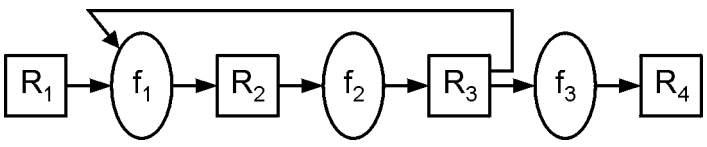


Bild 5.13 RTL-Netz aus Flipflops und kombinatorischer Logik

```
module rtl (                                                          //00
input  wire       CP,              // Takt                          //01
       wire [7:0] R1,              // Eingang R1                    //02
output reg  [7:0] R4               // Ausgang R4                    //03
);                                                                    //04
                                                                      //05
reg     [7:0] R2,                  // Register R2                   //06
              R3;                  // Register R3                   //07
                                                                      //08
integer       I;                   // Hilfsvariable                 //09
                                                                      //10
                                                                      //11
// Register-Transfer-Synthese                                         //12
 always @(posedge CP) begin                                           //13
   R2 <= f1(R1,R3);                                                   //14
   R3 <= f2(R2);                                                      //15
   R4 <= f3(R3);                                                      //16
 end                                                                  //17
                                                                      //18
// kombinatorische Logik zwischen den Registern                       //19
function [7:0] f1 (input [7:0] IN);                                   //20
  f1 = 2 * IN;                                                        //21
endfunction                                                           //22
                                                                      //23
function [7:0] f2 (input [7:0] IN);                                   //24
  f2 = IN + 5;                                                        //25
endfunction                                                           //26
                                                                      //27
function [7:0] f3 (input [7:0] IN);                                   //28
  f3 = IN * IN;                                                       //29
endfunction                                                           //30
                                                                      //31
endmodule // rtl                                                      //32
```

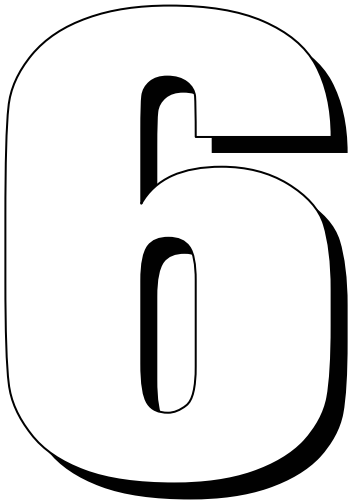
Beispiel 5.14 Synthetisierbares RTL-Modell zu Bild 5.13

5.4 Ausblick

Nur eine allererste Einführung in die Logiksynthese konnte dieses Kapitel geben. So gilt es bei der Register-Synthese zu klären, wann aus einem VERILOG-Register (Datentyp `reg`) ein getaktetes flankengesteuertes Flipflop oder ein (meist unerwünschtes) pegelgesteuertes Latch entsteht oder wann es sich um eine nur der übersichtlicheren Programmierung dienende Hilfsvariable handelt, die von der Logiksynthese anschließend zu Recht wegoptimiert wird.

Überhaupt kann die Optimierung wesentlich beeinflusst werden durch unterschiedliche RTL-Modelle, d.h. der RTL-Designer hat es durch geschickte RTL-Modellierung in der Hand, wie effizient das synthetisierte Gattermodell wird.

Schließlich kann man nicht erwarten, dass das Prä-Synthese-RTL-Modell und das Post-Synthese-Gattermodell exakt das gleiche Verhalten haben, so dass die Prä- und Post-Synthese in Simulationen gegeneinander verifiziert werden müssen. Eine weitere Verifikation, allerdings immer mit dem gleichen Testrahmen, bedingt die Layout-Synthese, wenn nämlich das Gattermodell in ein reales Layout platziert und verdrahtet wird.



Programmierbare Logikbausteine

Bisher haben wir gelernt, in einer Hardware-Beschreibungssprache Probleme zu spezifizieren und bevorzugt auf der Register-Transfer-Ebene (RTL) zu modellieren. In Simulationen können wir uns von der Richtigkeit eines RTL-Modells überzeugen und erste Abschätzungen zum Zeitverhalten einer künftigen Hardware machen.

Die Logiksynthese des vorigen Kapitels transformiert ein RTL-Modell in ein Gattermodell, das bereits nach unsern Wünschen optimiert ist hinsichtlich Schaltzeit oder Fläche oder auch Leistungsaufnahme. Auch wenn diese Parameter jetzt schon ziemlich genau bekannt sind, haben wir doch noch keinen realen Chip in den Händen.

Wenn wir überzeugt sind, dass unsere Schaltung vollständig korrekt ist und sich in hohen Stückzahlen verkaufen lässt, können wir für sehr viel Geld einen Prototypen fertigen lassen. Dies ist direkt nach der Logiksynthese jedoch äußerst selten der Fall.

Stattdessen gibt es programmierbare Logikbausteine, insbesondere *feld-programmierbare Gate-Arrays*, kurz FPGAs, die man geradezu als eierlegende Wollmilchsau ansehen kann. Denn in einen FPGA kann ein Anwender praktisch jede digitale Schaltung von außen „hineinprogrammieren“, indem ein passender Bitstrom geladen wird. Außerdem kann die momentane Schaltung im FPGA jederzeit durch eine völlig andere ersetzt werden.

FPGAs sind technologisch gesehen zwar reine Standardbausteine, die vom Anwender nur noch individuell programmiert werden. Obwohl sie sich daher *technologisch* von individuell gefertigten Chips völlig unterscheiden, haben sie mit letzteren *entwurfsmethodisch* sehr große Gemeinsamkeiten, indem der Entwurf von den obersten Entwurfsebenen bis hinunter zur Logik- und Gatterebene fast identisch ist. Um diese FPGAs und andere programmierbare Logikbausteine geht es im vorliegenden Kapitel.

6.1 Überblick

Zwei wesentliche Faktoren beim kommerziellen Erfolg einer neuen Schaltung sind zum einen die „Time-to-Market“ als die Zeit von der ersten Idee bis zur Marktreife und zum anderen die finanzielle Anfangsinvestition. In dieser Hinsicht sind FPGAs fast ideal, da die Entwicklungszeit nicht die Fertigungszeit von einigen Wochen bis Monaten enthält und da die Anfangsinvestition nur aus den Entwurfskosten besteht, während die Materialkosten je Prototyp zwischen einhundert und einigen tausend Euro liegen im Gegensatz zu 10.000 bis einer Million Euro für die erste Fertigung von Bausteinen vergleichbarer Komplexität.

Den Vorteilen geringer Prototypenkosten und kurzer Produktionszeiten stehen als Nachteile eine um den Faktor 2 bis 50 kleinere Operationsgeschwindigkeit und eine um den Faktor 8 bis 12 geringere Komplexität gegenüber sowie höhere Stückkosten bei der Serienfertigung. Obwohl wertmäßig der FPGA-Markt noch immer klein ist, ist er methodisch äußerst wichtig, da etwa die Hälfte aller Projekte mit FPGAs begonnen werden (Rapid Prototyping), um den korrekten Prototypen später gegebenenfalls zu fertigen.

Auch macht die sofortige und wiederholbare Programmierbarkeit FPGAs zum geradezu idealen Trainingsobjekt in der Lehre. Erfreulicherweise wird dabei nicht nur der spezielle FPGA-Entwurf, sondern der Chip-Entwurf schlechthin geübt.

Neben dem Rapid Prototyping sind FPGAs auch immer häufiger als Endprodukt in einem eingebetteten System zu finden. Beispielhaft genannt seien Controller aller Art für Speicher, FIFOs, Schnittstellen, Grafik, Peripherie, viele Anwendungen in der Kommunikation uvam. Dann ist es von Vorteil, dass sich die FPGA-Schaltungen wie normale Software updaten lassen, etwa wenn der Standard sich geändert hat oder auch nur im nie endenden Marathon von Versionen, Fehlerbeseitigung und neuen Fehlern…

Dies führt im nächsten Kapitel zum besonders faszinierenden Gebiet des *Hardware-Software-Codesigns*, wo rekonfigurierbare FPGAs zusätzlich auf ihrem Chip einen oder gar mehrere Standardprozessoren enthalten. Während der normale Rechner Routineaufgaben erledigt, können besonders rechenintensive oder echtzeitbedürftige Teile eines Programms oder bestimmte Schnittstellen in den FPGA als programmierte Spezial-Hardware ausgelagert werden. Besonders attraktiv ist dabei die Möglichkeit, FPGAs im laufenden Betrieb umzuprogrammieren.

Ein solches Multi-Processor-System-on-Chip (MPSoC) samt zugehöriger Entwicklungsumgebung wird uns im Abschnitt 6.4, im nächsten Kapitel und in den Übungen beschäftigen. Zunächst aber schauen wir uns FPGAs und programmierbare Logik etwas genauer an.

6.1.1 Evolution und Begriffe

Die ersten nennenswerten programmierbaren digitalen Schaltungen waren die *Read-Only-Speicher*: während ein ROM seinen Inhalt bei der Fertigung erhält, kann ein PROM einmal durch den Anwender „feldprogrammiert“ werden.

Gefertigte ROMs sind wesentlich schneller, da die internen Verbindungen während der Herstellung fest verdrahtet werden. Dagegen beinhalten feldprogrammierbare Verbindungen immer eine Art von *programmierbarem Schalter* (z.B. eine „Fuse“), der deutlich langsamer ist. Dem Effizienznachteil stehen als Vorteile gegenüber die geringeren Kosten bei kleinen Stückzahlen und die sofortige Programmierbarkeit. Beim Erasable- bzw. Electrically-Erasable-Programmable-Read-Only-Memory (EPROM bzw. EEPROM) kommt als wesentlicher Vorteil hinzu, dass sie gelöscht und häufig neu programmiert werden können. In vielen Anwendungen und gerade in den frühen Phasen einer Schaltkreisentwicklung ist die Reprogrammierbarkeit sehr nützlich.

Obwohl prinzipiell jede Logikfunktion als ROM-Schaltung realisiert werden kann, liegt die Hauptanwendung von ROMs in der Verwendung als Speicher. Dagegen ist das *Programmable-Logic-Device* oder PLD eine programmierbare Schaltung speziell zur Realisierung logischer Schaltungen. Ein einfaches PLD enthält typischerweise ein Feld von AND-Gattern verbunden mit einem Feld von OR-Gattern und nutzt aus, dass sich jede logische Funktion in einer zweistufigen Summen-Produkt-Form darstellen lässt (beispielsweise $f1 = a \cdot b \cdot c' + a' \cdot c$, $f2 = \ldots$). In der ersten AND-Ebene werden dann zunächst alle Produktterme wie $a \cdot b \cdot c'$ gebildet, die dann in der zweiten OR-Ebene verodert werden.

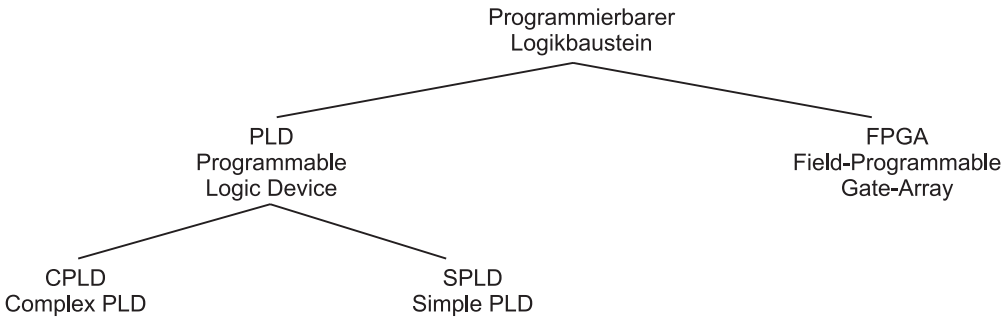


Bild 6.1 Programmierbare Logikbausteine

Diese einfachen PLDs aus genau einer AND- und einer OR-Ebene (Simple-PLD, SPLD) erlauben mit ihrer einfachen zweistufigen Struktur zwar sehr schnelle Schaltkreisrealisierungen, jedoch können nur kleine Logikfunktionen mit einer mäßigen Anzahl von Produkttermen dargestellt werden, weil sonst die Verbindungsstruktur unverhältnismäßig stark wachsen würde. Diesen Nachteil überwinden Complex-PLDs oder CPLDs, die aus einem ganzen Feld von SPLDs bestehen.

Universeller und bedeutender sind heute die FPGAs, auf die wir nun eingehen. Bild 6.1 zeigt die Hierarchie der gerade verwendeten Begriffe und Abkürzungen.

6.1.2 Grundaufbau eines FPGA

Feldprogrammierbare (field-programmable) Gate-Arrays bestehen aus einem Feld (Array) von allgemeinen Elementen, die in allgemeiner Weise verbunden werden können. Seit der Einführung von FPGAs 1985 durch Xilinx wurden zahlreiche FPGA-Familien entwickelt, wobei zu den wichtigsten Herstellern neben Xilinx heute Actel und Altera gehören.

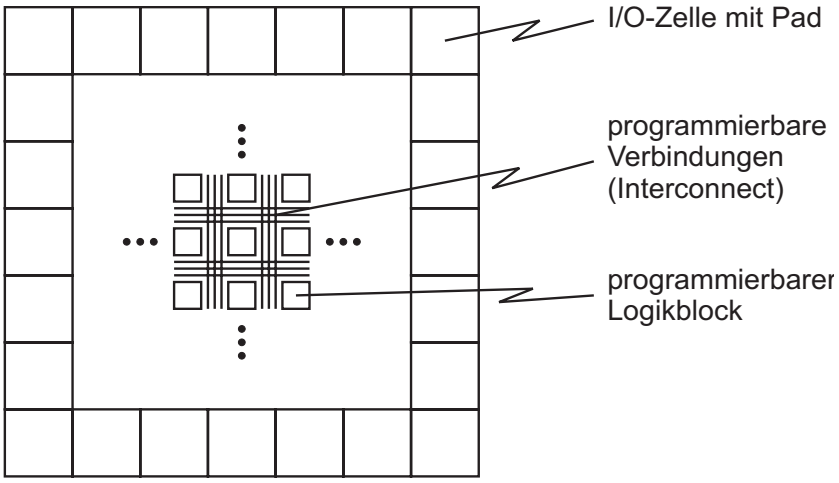


Bild 6.2
Typischer Aufbau eines FPGA

Bild 6.2 skizziert das Konzept eines typischen FPGA. Es besteht aus einem zweidimensionalen Feld von *Logikblöcken*, die durch das *Interconnect* beliebig verbunden werden können. Das Interconnect besteht aus Verbindungsabschnitten, deren Länge unterschiedlich ist. Es enthält programmierbare Schalter, um einerseits die Logikblöcke an die Verbindungsabschnitte anzuschließen und andererseits die Verbindungen untereinander aufzubauen. Ein Schaltkreis wird in einem FPGA implementiert, indem die Logik auf die einzelnen Logikblöcke aufgeteilt wird und dann diese Blöcke mithilfe der Schalter verbunden werden.

Um möglichst viele verschiedene Schaltkreise effizient realisieren zu können, ist die Vielseitigkeit eines FPGA entscheidend. Dies muss durch die Logikblöcke und das Interconnect unterstützt werden. Es gibt vielfältige Wechselbeziehungen zwischen der Fläche, der Komplexität, der Schaltgeschwindigkeit und der Flexibilität der Logikblöcke und dem Interconnect.

Struktur und Inhalt eines Logikblocks stellen die *Architektur* eines FPGA dar, die auf sehr unterschiedliche Weise realisiert sein kann. Bei manchen FPGAs sind die Logikblöcke lediglich einfache Transistorpaare. Etwas komplexer sind mit Multiplexern realisierte Logikblöcke, und die allgemeinste Form der Blöcke ist durch je einen Lookup-Table (LUT) pro Block gegeben. Manche FPGAs haben sogar

Logikblöcke mit je einer vollständigen AND-OR-ähnlichen Struktur und gehören daher auch in die Klasse der CPLDs. Zur Realisierung sequenzieller Schaltungen enthalten die meisten Logikblöcke außerdem speichernde Elemente.

Die Verdrahtungsarchitektur eines FPGA bestimmt sein Interconnect, das wie erwähnt aus Verbindungsabschnitten unterschiedlicher Länge und programmierbaren Schaltern besteht. Diese Schalter können auf unterschiedliche Art realisiert sein, zum Beispiel als SRAM-kontrollierte Pass-Transistoren, als Antifuses sowie als EPROM oder EEPROM-Transistoren (Abschnitt 6.2). Einige FPGAs bieten eine große Anzahl einfacher Verbindungen zwischen Blöcken, andere bieten weniger, aber komplexere Verbindungen.

6.1.3 Entwurfsablauf

Ein besonders interessantes Merkmal des FPGA-Entwurfs besteht darin, dass er sich bis hinab zur Gatterebene praktisch kaum von „echten“ Chip-Entwürfen unterscheidet. Demzufolge werden eine Spezifikation und ein Register-Transfer-Modell in einer Hardware-Beschreibungssprache angefertigt, die die Logiksynthese in ein Gattermodell umsetzt (Kapitel 5). Dabei berücksichtigt die Technologie-Bibliothek die technischen Möglichkeiten der konkreten FPGA-Architektur.

Neben dem RTL-Modell gibt es die Möglichkeit, die Schaltung durch boolesche Gleichungen zu spezifizieren oder als Zustandsdiagramm einzugeben.

Ein *Platzierer* versucht anschließend, die gegebenen Logikblöcke günstig auf dem FPGA zu verteilen. Typischerweise wird versucht, die voraussichtliche Gesamtlänge der Verbindungen bei der Platzierung zu minimieren.

Der letzte Schritt besteht schließlich in der *Verdrahtung*, die geeignete Verbindungsabschnitte aussucht und durch Schalter verbindet. Auch hierbei ist die Verzögerung unter Umständen ein kritischer Faktor, da die programmierbaren Verbindungsschalter relativ viel Verzögerung bedeuten. Die theoretischen Informatiker können beweisen, dass das Platzieren und Verdrahten ein sehr schwieriges Problem ist, nämlich NP-hart, d.h. *praktisch* nicht exakt lösbar. Stattdessen werden gut brauchbare heuristische Näherungslösungen eingesetzt.

Das Ergebnis des Platzierens und Verdrahtens besteht schließlich in einem Bitstrom für eine *Programmiereinheit*, die das fertige FPGA konfiguriert. Dies dauert zwischen einigen Millisekunden und einer Sekunde.

6.2 Programmiertechniken

Ohne auf die elektrischen Grundlagen näher einzugehen, wollen wir einige technologische Möglichkeiten andeuten, Verbindungsschalter in programmierbaren Logikbausteinen zu realisieren. Solche Schalter werden auch *Programmierelemente* genannt, die als statische RAM-Zellen, Antifuses, EPROM- oder EEPROM-Transistoren

realisiert sind. Trotz sehr unterschiedlicher Technik ist ihnen gemeinsam, dass sie als Schalter leitend oder nichtleitend sind.

Diese Programmierelemente realisieren die programmierbaren Verbindungen zwischen den Logikblöcken eines FPGAs, als statische RAMs dienen sie auch dazu, Logikblöcke selbst zu konfigurieren. Ein FPGA kann viele Millionen Programmierelemente enthalten. Diese sollten daher möglichst wenig Chip-Fläche verbrauchen, je nach Zustand einen kleinen oder sehr großen Widerstand besitzen und eine niedrige Kapazität enthalten. Sie sollten zuverlässig in großen Stückzahlen zu fertigen sein. Interessant ist es, ob ein Programmierelement *reprogrammierbar*, d.h. mehrfach einzustellen ist, und ob es *flüchtig* ist, d.h. beim Einschalten des Gerätes neu programmiert werden muss. Schließlich interessiert, ob es im Rahmen eines Standard-CMOS-Prozesses realisiert werden kann.

6.2.1 Programmierung durch SRAMs

Wie in Bild 6.3 lassen sich Verbindungen durch statische RAM-Zellen oder SRAMs[1] aufbauen, die einen Pass-Transistor oder einen Multiplexer ansteuern. SRAMs sind flüchtig, daher müssen die zugehörigen FPGAs bei jedem Einschalten neu konfiguriert werden und einen geeigneten dauerhaften Speicher, etwa einen Flash-Speicher, besitzen.

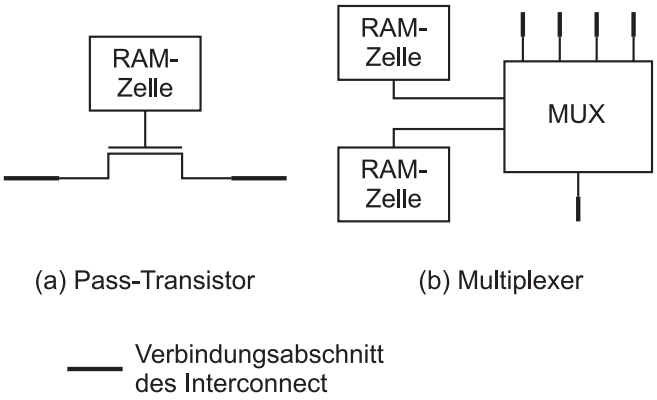


Bild 6.3 Einsatz von statischen RAMs

Im Vergleich zu anderen Programmiertechniken ist der Flächenbedarf von SRAMs relativ groß. Der Hauptvorteil liegt in der häufigen Reprogrammierbarkeit im eingebauten Zustand. Bild 6.4 zeigt als Beispiel eine Verbindung zweier Gatter in verschiedenen Logikblöcken.

[1] SRAM steht für Static RAM: Ein ständig fließender Strom erhält beim SRAM die gespeicherten Informationen, so dass man auf einen Refresh verzichten kann und SRAM daher wesentlich schneller ist als DRAM. Der Nachteil ist eine größere Stromaufnahme und damit stärkere Erwärmung. Dagegen ist der DRAM (Dynamic Random Access Memory) der am häufigsten verwendete Arbeitsspeicher. Er kann Daten nur für eine kurze Zeit halten und muss sie regelmäßig auffrischen.

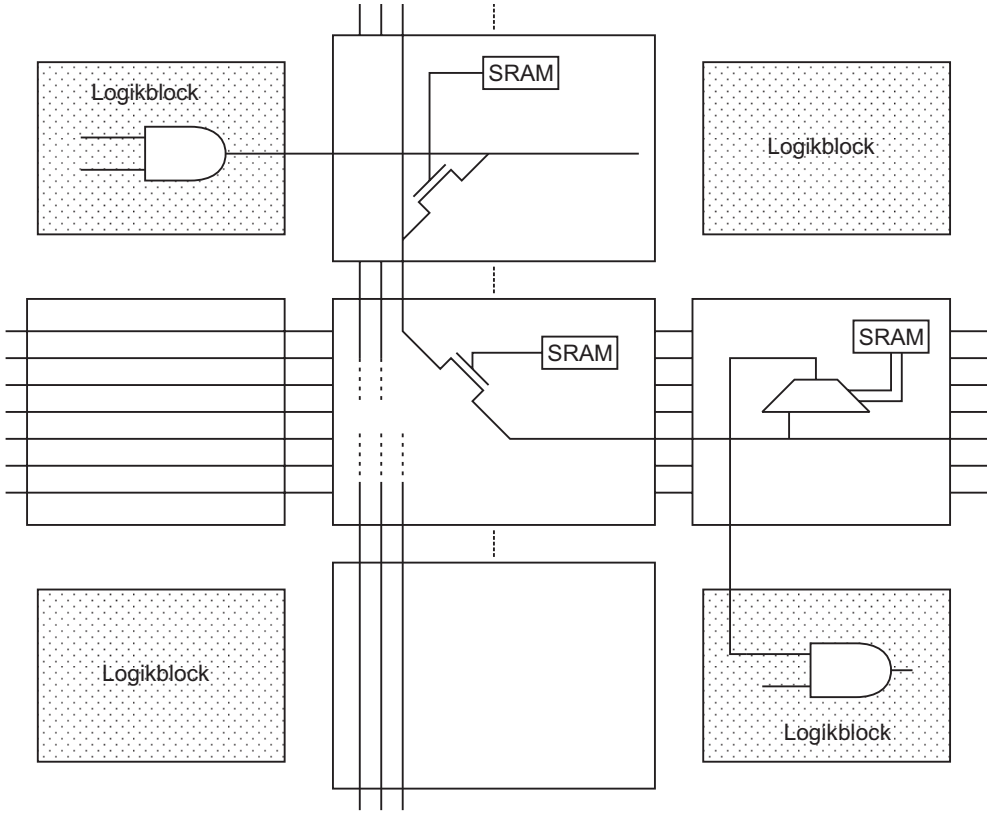


Bild 6.4 SRAM-basierte Verbindungen

6.2.2 Weitere Programmierelemente

Eine Antifuse hat zunächst einen hohen Widerstand und wird durch eine hohe Spannung unwiderruflich in den leitenden Zustand versetzt.

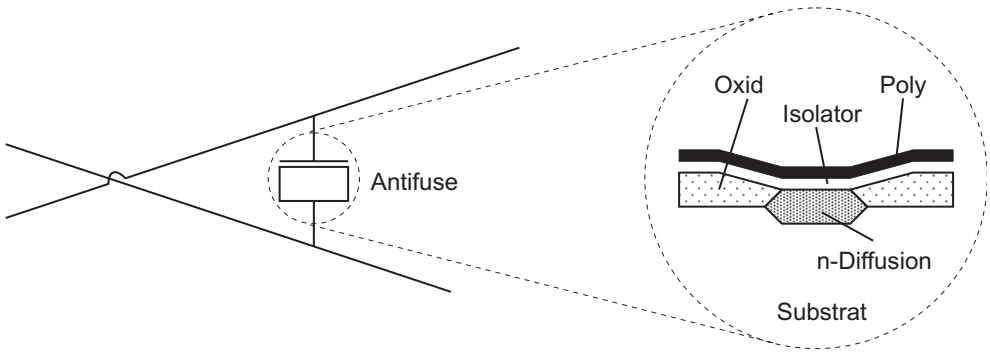


Bild 6.5 PLICE als Beispiel einer Antifuse

Als Beispiel sei die Antifuse PLICE von Actel genannt. Wie in Bild 6.5 besteht sie aus drei Layern: unten n-Diffusion, dazwischen ein Isolator und oben Polysilizium. Programmiert wird die Antifuse durch eine recht hohe Spannung von 18 Volt, wodurch

im Isolator bei einem Strom von etwa 5 mA genug Hitze entsteht, um ihn schmelzen zu lassen. Dadurch entsteht eine leitende Verbindung zwischen Poly und Diffusion von etwa 300 bis 500 Ω (Bild 6.6). Spezielle Hochvolt-Transistoren stellen die großen Spannungen und Ströme zur Verfügung.

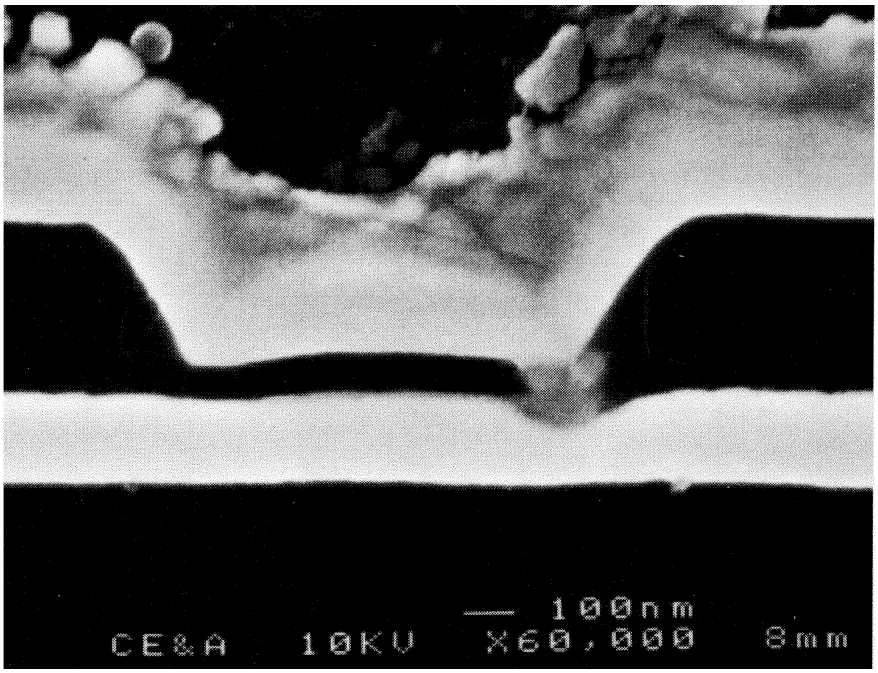


Bild 6.6
Leitende Antifuse

Die Chip-Fläche einer Antifuse ist klein im Vergleich zu anderen Programmiertechniken, jedoch wird viel Fläche für die Hochvolt-Transistoren benötigt. Zwar sind Antifuses nicht flüchtig, aber leider eben auch nicht reprogrammierbar (was manchmal wiederum einen Vorteil darstellt).

Nur am Rande erwähnen wir hier die jetzt nicht mehr so häufig eingesetzte Programmierung durch die bekannten EPROM-Speicher. Sie können durch größere Ströme programmiert und durch ultraviolettes Licht wieder gelöscht werden. Sie können daher nicht durch die Schaltung selbst reprogrammiert werden.

Dies ist dagegen bei EEPROMs begrenzt möglich, die allerdings etwa die doppelte Chip-Fläche benötigen sowie verschiedene Spannungsquellen. Tabelle 6.7 fasst Eigenschaften der verschiedenen Technologien zusammen.

Technik	repro-grammierbar	in Schaltung reprogramm.	flüchtig	Fläche
SRAM	+	+	+	groß
Antifuse	–	–	–	klein
EPROM	+	–	–	klein
EEPROM	+	+	–	2xEPROM

Tabelle 6.7 Merkmale verschiedener Programmiertechniken

6.3 FPGAs von Xilinx – die Wurzeln

Geschichte ist nicht jedermanns Sache, aber ein Blick auf die noch heute gelegentlich anzutreffende FPGA-Familie XC4000 des Marktführers Xilinx vermittelt in übersichtlicher Weise wesentliche Grundideen von FPGAs, die sich auch in modernen „Plattform-FPGAs“ mit Millionen von Gattern nicht geändert haben, auf welche wir im nächsten Abschnitt eingehen werden.

Die allgemeine Architektur von FPGAs wurde bereits in Bild 6.2 skizziert. Sie besteht aus einem zweidimensionalen Feld von Logikblöcken, die bei Xilinx CLBs (Configurable Logic Blocks) heißen. Zwischen den Blöcken gibt es horizontale und vertikale Verdrahtungskanäle. Sowohl die Verbindungen als auch die CLBs werden durch statische RAM-Zellen kontrolliert. Die Komplexität der Familie XC4000 reicht von 2 000 bis 20 000 Gatteräquivalente, es gibt zwischen 64 und 900 CLBs.

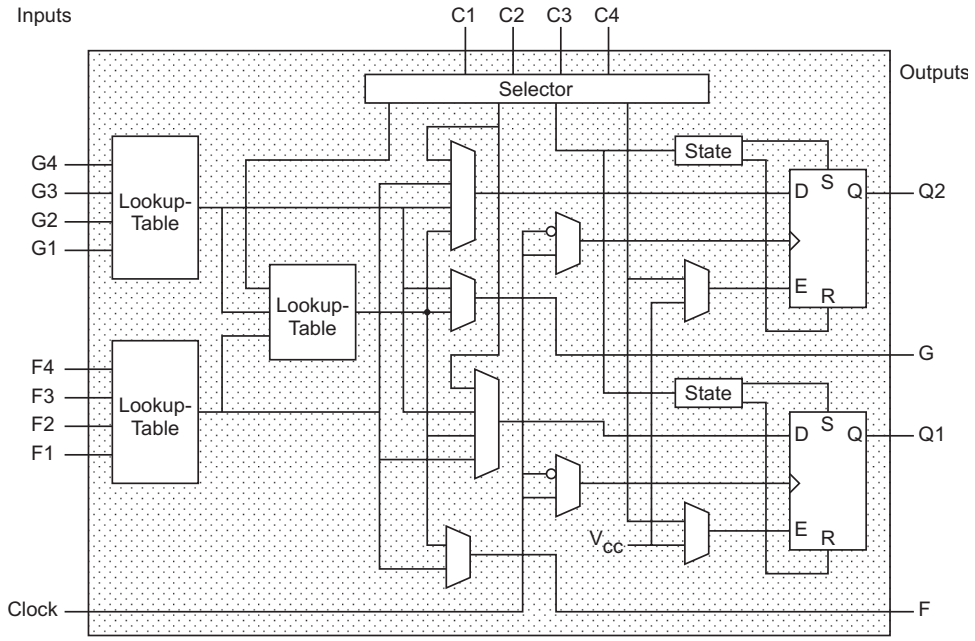


Bild 6.8 CLB der Serie XC4000

Der CLB der Serie XC4000 basiert auf einem Lookup-Table (LUT). Dies ist ein Speicherfeld der Breite 1 Bit, wobei die Adressleitungen die Inputs sind und die 1-Bit-Outputs das Ergebnis des Lookup-Table darstellen. Ein LUT mit k Inputs entspricht einem Speicher der Größe 2^k x 1, und der Anwender kann jede Logikfunktion mit k Inputs realisieren, indem er ihre Wahrheitstafel direkt in den Speicher programmiert. Der CLB in Bild 6.8 enthält zwei Lookup-Tables mit je vier Eingängen verbunden mit den CLB-Inputs C1 bis C4 und einem dritten Lookup-Table, der mit den anderen beiden verbunden ist. Damit kann ein weiter Bereich von Logikfunktionen mit bis zu 9 Inputs realisiert werden, oder zwei verschiedene 4-Input-Funktionen und andere Kombinationen. Die realisierbaren Logiken sind ein- oder zweistufig. Jeder CLB enthält auch zwei Flipflops, deren Initialisierung mit 0 oder 1 durch die SRAM-Zelle

state vorgegeben werden kann. Auch kann bestimmt werden, ob die Flipflops mit der steigenden oder der fallenden Taktflanke gesteuert werden. Jeder CLB hat neben diesen zwei Ausgängen auch zwei ungepufferte kombinatorische Outputs.

Um komplette Systeme realisieren zu können, enthält jeder CLB beispielsweise Schaltungen für die effiziente Ausführung von Arithmetik, insbesondere zur Realisation von schnellen Carry-Operationen. Auch können die Lookup-Tables Read/Write-RAM-Zellen konfigurieren (16x2 oder 32x1 bit). Bei manchen Versionen sind auch Dual-Port-RAMs einstellbar. Jedes XC4000-Chip enthält an der Peripherie breite AND-Ebenen, um Schaltungen wie breite Dekoder zu unterstützen.

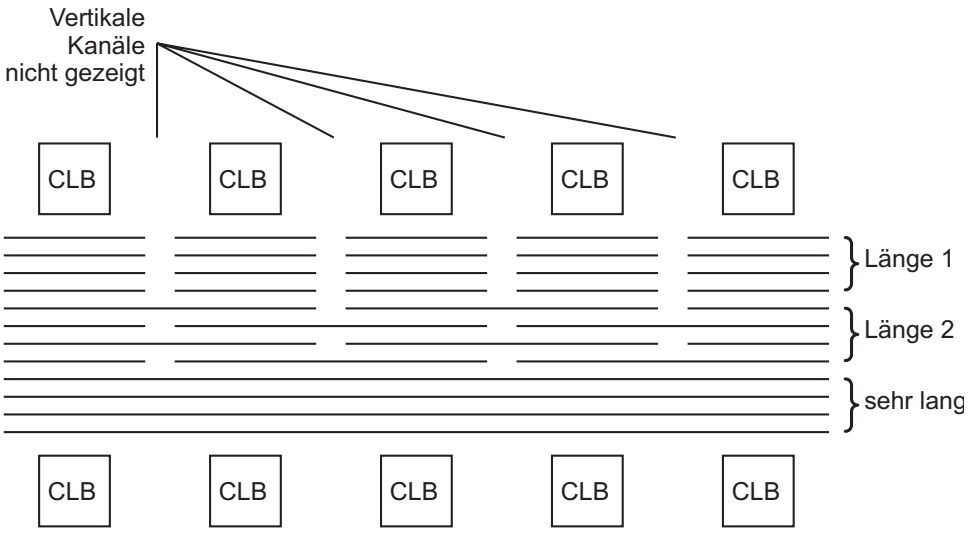


Bild 6.9 Verbindungsabschnitte im XC4000

Die Interconnect-Architektur besteht aus horizontalen und vertikalen Kanälen, die wie in Bild 6.9 jeweils kurze Leitungsabschnitte für benachbarte CLBs enthalten (Länge 1), doppelt so lange (aber immer noch kurze) Abschnitte bis zum übernächsten CLB (Länge 2) und sehr lange Abschnitte über die gesamte Chip-Fläche. Gezeigt sind in Bild 6.9 nur die horizontalen Abschnitte, jedoch nicht die vertikalen, die CLB-Anschlüsse und die Verdrahtungsschalter. Programmierelemente einer Switch-Matrix wie in Bild 6.10 verbinden CLB-Inputs und -Outputs mit den Leitungsabschnitten und diese untereinander. Längere Verbindungen sind wie in Bild 6.11 vorgesehen.

Die Laufzeit einer Verbindung hängt entscheidend, nämlich quadratisch von der Gesamtanzahl der durchlaufenen Schalter ab. Deshalb hängt eine implementierte Schaltkreisgeschwindigkeit wesentlich davon ab, wie CAD-Werkzeuge die Leitungsabschnitte an einzelne Signale vergeben.

Erst nach der Platzierung und Verdrahtung durch spezielle Werkzeuge können exakte Verzögerungszeiten errechnet werden (Zahl der durchlaufenen Schalter, Leitungslängen), und der Entwurf kann erneut durch Simulation verifiziert werden.

Aus den platzierten und verdrahteten Logikblöcken lässt sich schließlich der Bitstrom für die Konfiguration des FPGAs ableiten.

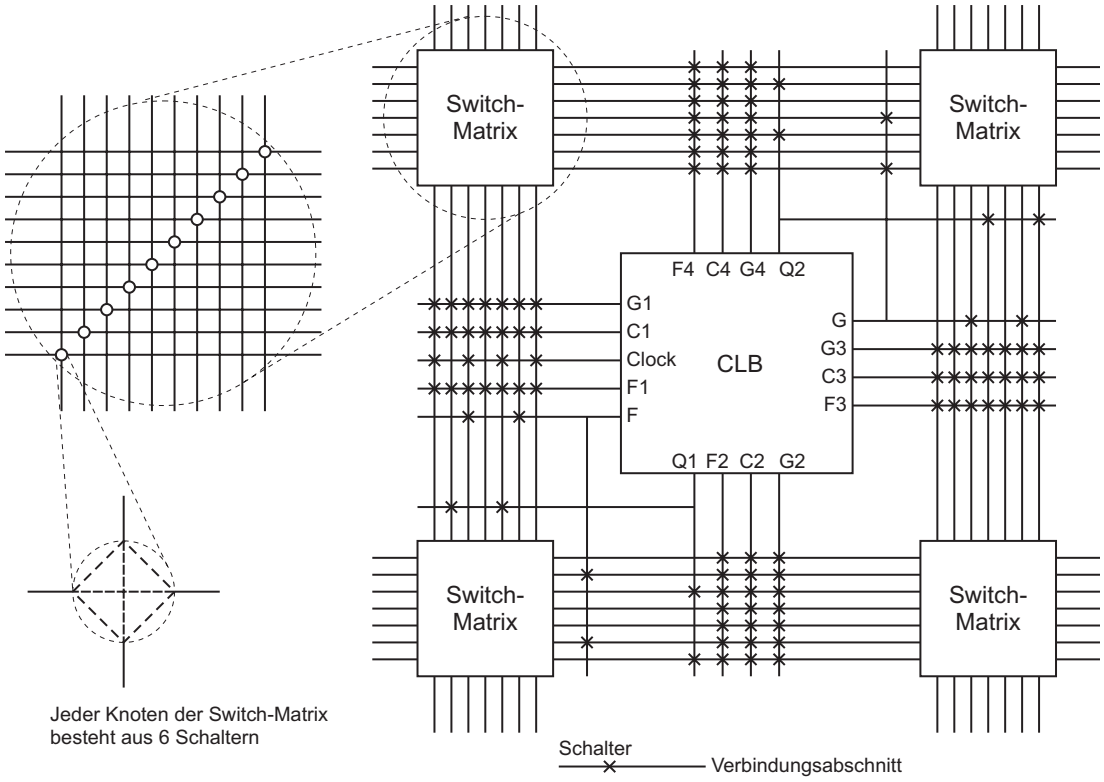


Bild 6.10 Verbindungsabschnitte im XC4000 (Länge 1)

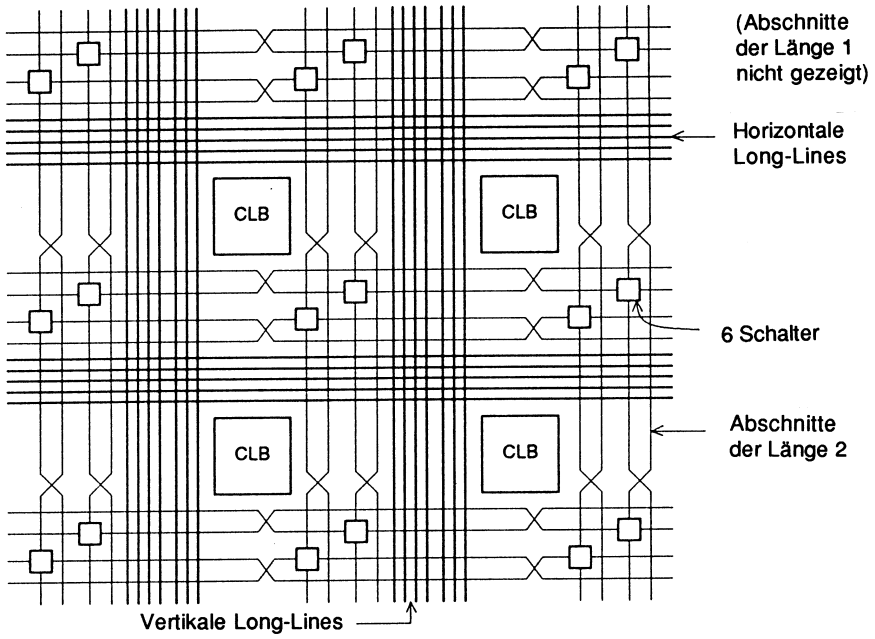


Bild 6.11 Verbindungsabschnitte im XC4000 (Länge 2 und sehr lang)

Der „normale“ FPGA-Designer wendet diese Phasen mehr oder weniger per „Knopfdruck“ an. Sind die Ergebnisse unbefriedigend, können mit Spezialkenntnissen über den FPGA-Aufbau oft wesentliche Verbesserungen erzielt werden.

6.4 Multiprozessor-FPGA-Plattformen

Unter einer *FPGA-Plattform* verstehen wir hier eine Familie von Modellen, die neben einem „reinen“ FPGA wie in den vorigen Abschnitten – also einem regulären Feld aus programmierbaren Logikblöcken mit programmierbaren Verbindungen – zusätzliche Komponenten enthalten: vor allem einen oder mehrere klassische Prozessoren („Multiprozessor“), aber auch Speicherblöcke, Multiplizier-Felder und feste Kommunikations-Architekturen. Die Modelle der Plattform-Familie sind im Wesentlichen gleich aufgebaut und unterscheiden sich vor allem in der Anzahl der Komponenten. Ohne den Entwurf wesentlich ändern zu müssen, kann man also wählen zwischen mehr oder weniger Logikblöcken, mehr oder weniger Speicher und Anderem.

Die derzeit technologisch und kommerziell führenden FPGA-Plattformen sind Virtex-II Pro und ihre Weiterentwicklung Virtex-4. Als Universalbausteine erlauben sie eine unglaubliche Vielfalt von Hardware-Software-Realisierungen für unterschiedlichste eingebettete Systeme, von der Digitalkamera über digitale Fernsehgeräte (DVB, HDTV) bis hin zu Steuergeräten für die Fahrzeugelektronik, Home-Automation oder Raumfahrttechnik.

6.4.1 Architektur des Virtex-II Pro

Mit Einführung der Plattform Virtex-II Pro steht erstmals eine FPGA-Technologie zur Verfügung, die das Prototyping von Multi-Processor-System-on-Chips (MPSoC) mit mehreren Millionen Gattern ermöglicht und dabei Taktfrequenzen bis 400 MHz erreicht. Die Kosten bewegen sich im Bereich einiger tausend Euro, so dass neben dem wichtigen Prototyping auch kurzlebige Technologien wie Mobiltelefon-Chipsätze sehr viel günstiger realisiert werden können als noch vor wenigen Jahren. Übrigens setzen wir die Plattform Virtex-II Pro auch in den Übungen, im Praktikum und in der Forschung zum Hardware-Software-Codesign ein.

Auch früher hat man schon Prozessorkerne in FPGAs integriert und damit auch schon MPSoC-Lösungen realisiert. Damals wurden die Prozessoren jedoch als *Soft-Cores* eingebaut, d.h. als VERILOG- oder Gattermodell, das dann gemeinsam mit der anwendungsspezifischen Hardware auf dem FPGA konfiguriert wurde. Dies hat je nach Komplexität und Anzahl der verwendeten Prozessoren so viele CLBs verbraucht, dass nur noch wenig Platz für die eigentliche Schaltung blieb. Das Platzieren und Verdrahten solcher Soft-Cores konnte mehrere Stunden dauern, und Taktfrequenzen über 50 MHz waren so kaum zu erreichen.

Im Virtex-II Pro integriert Xilinx stattdessen zwei ausgereifte 32-Bit-RISC-Prozessoren vom Typ IBM PowerPC 405 als „echte“ Hardware, so dass sämtliche konfigurierbaren Logikzellen für Anwenderschaltungen zur Verfügung stehen.

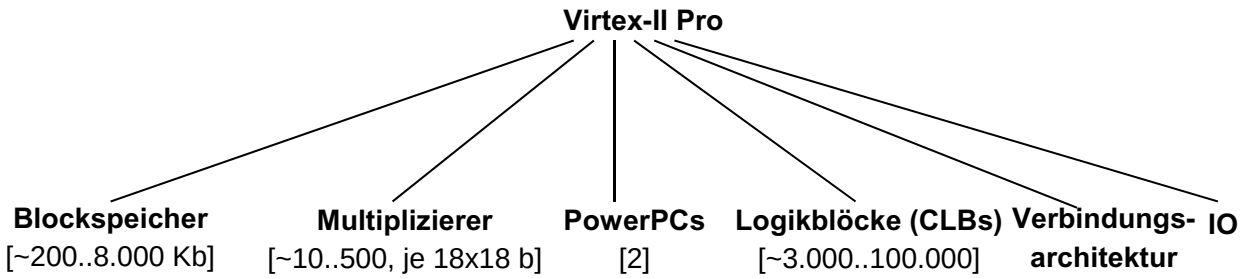


Bild 6.12 Komponenten des Virtex-II Pro

Im Zentrum der Grobarchitektur in Bild 6.12 stehen diese beiden PowerPC und natürlich die frei konfigurierbaren Logikblöcke oder CLBs. Wie in Bild 6.13(a) angedeutet, sind diese CLBs in Spalten angeordnet, neben denen es weitere Spalten von effizientem Blockspeicher und von Multiplizierern gibt (Abschnitt 6.4.3). Die Verbindungsarchitektur umfasst sowohl ein *Network-on-Chip* mit mehreren Bussen als auch ein frei konfigurierbares Interconnect (Abschnitt 6.4.5). Sehr schnelle Transceiver und andere IO-Anschlüsse verbinden die Plattform mit der Außenwelt (Abschnitt 6.4.6).

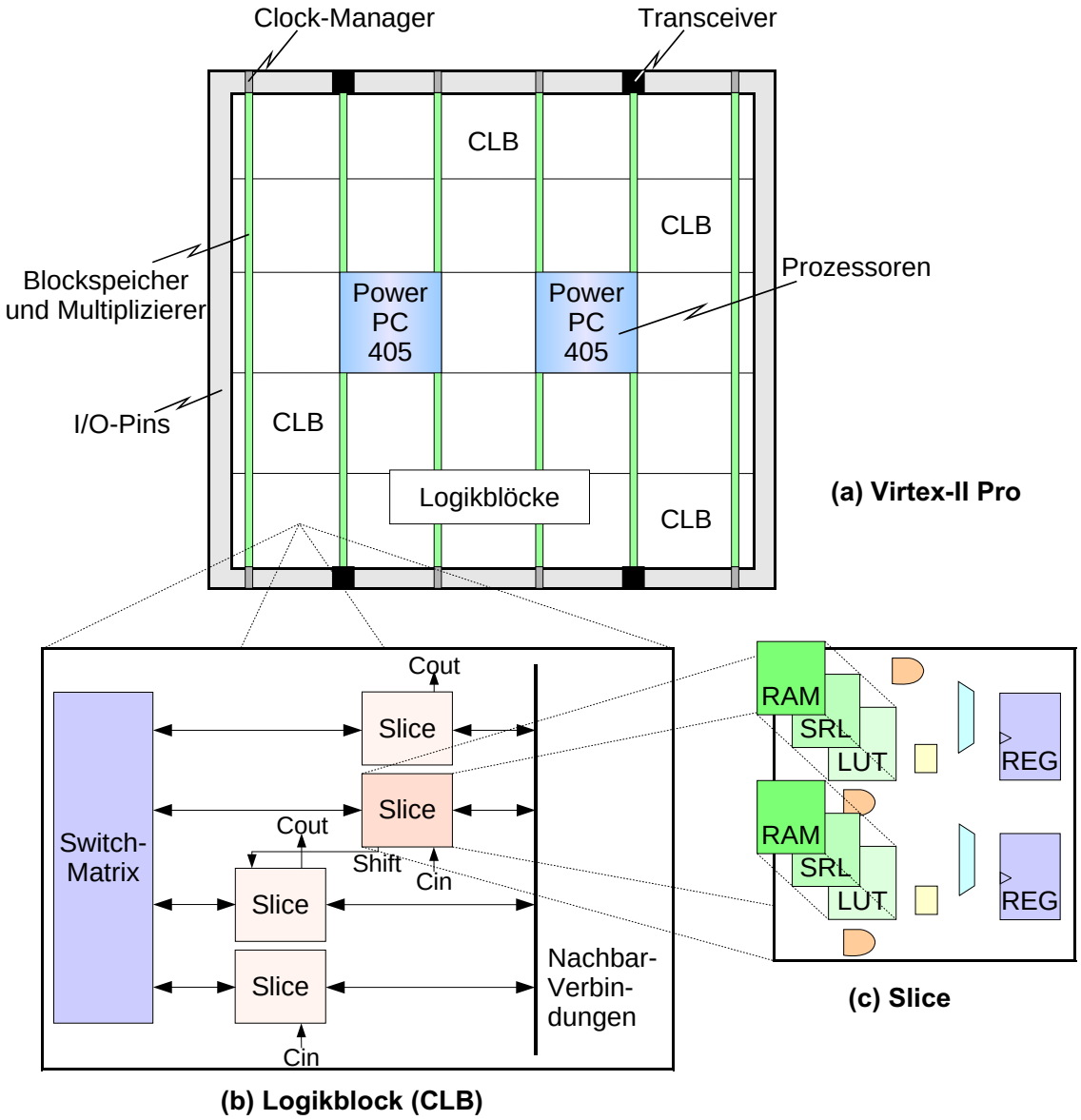


Bild 6.13 Virtex-II Pro: (a) Chip-Layout, (b) Logikblock und (c) Slice

6.4.2 Logikblöcke

Die Plattform-FPGAs übertreffen allein hinsichtlich der Zahl der Logikblöcke die XC4000-Serie im Abschnitt 6.3 bei Weitem und besitzen durch die PowerPCs und die übrigen Komponenten so viele Zusatzfunktionen, dass zum Beispiel schon ein kompletter DVD-Player mit Linux-Betriebssystem darauf realisiert wurde. Allerdings muss die neue Technologie hier erst recht von ausgereiften Entwurfswerkzeugen unterstützt werden, damit sie von allen Entwicklern sinnvoll eingesetzt werden kann. Viele der neuen Eigenschaften können nur mit Expertenwissen voll ausgeschöpft werden.

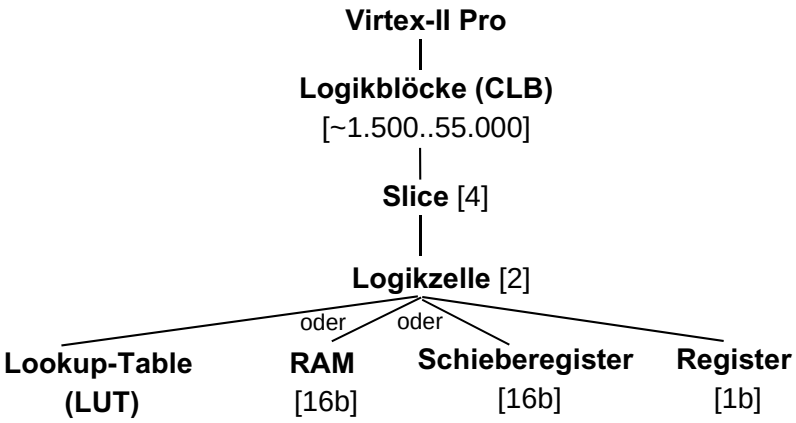


Bild 6.14 CLB-Aufbau

Den hierarchischen Aufbau eines Logikblocks oder CLBs zeigt Bild 6.14. Danach setzt sich jeder CLB wie in Bild 6.13b aus vier Slices zusammen, die untereinander über eine Switch-Matrix und mit benachbarten CLBs über schnelle Verbindungen kommunizieren. Jede Slice hat wiederum zwei Logikzellen, die Bild 6.13c knapp und Bild 6.15 ausführlich zeigen.

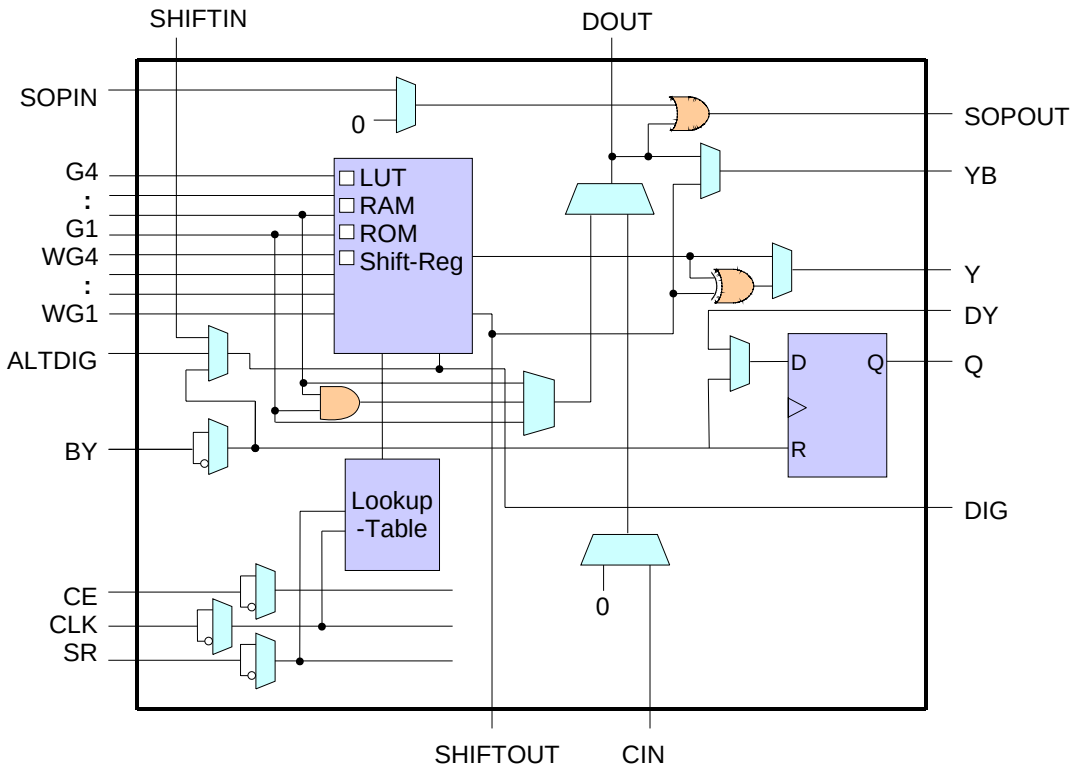


Bild 6.15 Logikzelle des Virtex-II Pro

Das Herz einer Logikzelle ist ein Lookup-Table (LUT) mit vier Eingängen. Dieser kann entweder eine beliebige boolesche Logikfunktion mit vier Variablen implementieren oder als 16-Bit-Speicher fungieren oder auch als 16-Bit-Schieberegister.

Weiter hat eine Logikzelle ein Register, das als flankengesteuertes Flipflop oder als Latch konfiguriert werden kann und welches wahlweise die Ausgaben der LUTs oder Eingangssignale speichern kann.

6.4.3 Blockspeicher und Multiplizierer

Die Lookup-Tables und Register der Logikblöcke lassen sich natürlich zu größeren Speicherbereichen zusammenschalten. Zu bevorzugen ist bei größeren Datenmengen aber der kompakte Blockspeicher (BRAM), der als schneller SRAM-Speicher in den Spalten des FPGA enthalten ist (Bild 6.13a). Jede Zelle des maximal 10 Mbit großen, von Xilinx auch SelectRAM genannten Blockspeichers fasst 18 Kbit und kann über zwei unabhängige *Dual-Ports* angesprochen werden. Die Dual-Ports können auf verschiedene Bitbreiten konfiguriert und mit unterschiedlichen Taktraten verwendet werden. Die Überwindung von Taktgrenzen, wie sie oft zwischen verschiedenen Hardware-Modulen auf einem System-on-Chip vorkommen, wird dadurch erleichtert (*Clock-Domain-Crossing*). Pro Taktzyklus können gleichzeitig 18 Bit geschrieben und gelesen werden.

Speziell für Anwendungen aus der digitalen Signalverarbeitung wurden 18x18 Bit große Multiplizier-Blöcke entwickelt. In Kombination mit einer Blockspeicher-Zelle und einem LUT-basierten Akkumulator bilden sie Multiplizier-Akumulator-Funktionen (MAC), die in digitalen Signalprozessoren (DSPs) zur Realisierung von Filterfunktionen eingesetzt werden.

6.4.4 Die eingebetteten Prozessoren PowerPC 405

Hardware und Software auf einem Chip zu integrieren, ist der Schlüssel für leistungsfähige und zugleich kleine und stromsparende eingebettete Systeme wie MP3-Player oder Marsroboter. In den FPGA-Plattformen Virtex-II Pro und Virtex-4 stehen die beiden eingebetteten Prozessoren PowerPC 405 auf kürzestem Wege mit den rekonfigurierbaren Logikzellen in Verbindung, nämlich mit bis zu 18 GBit/s. Solche HW-SW-Plattformen erleichtern erheblich ein Rapid Prototyping, das heißt die schnelle Realisierung eines ersten Versuchssystems, ebenso wie die Realisierung komplexer Endprodukte.

Der PowerPC 405 ist ein 32-Bit-RISC-Prozessor mit fünfstufiger Pipeline: Holen eines Befehls aus dem Speicher, Dekodieren, Ausführen des Befehls und Zurückschreiben der Ergebnisse in den Speicher (zwei Stufen). Mit der Fixpunkt-Rechenheit können die meisten Operationen in einem Takt ausgeführt werden. Durch minimale Reaktionszeiten auf Hardware-Interrupts und programmierbare Timer mit hoher Zeitauflösung ist der Prozessor gut für Echtzeit-Anwendungen geeignet. Neben dem Standard-Befehlssatz des PowerPC werden zusätzlich spezielle Instruktionen zur digitalen Signalverarbeitung unterstützt.

Wie bei PowerPC-Prozessoren üblich, ist der Zugriff auf den Arbeitsspeicher mit einem Dual-Port-Speicher realisiert, es kann also beispielsweise gleichzeitig ein Befehl geholt und das Ergebnis eines früheren Befehls zurückgeschrieben werden. Die zwei unabhängigen Ports arbeiten jeweils mit einem eigenem Cache. Während klassische *Von-Neumann-Rechner* nur mit einem Speicher-Port auskommen müssen, liegt hier eine *Harvard-Architektur* vor.

Bild 6.16 zeigt zwei *On-Chip-Memory-Controller* (OCM), die einen PowerPC direkt an den Blockspeicher des FPGA anschließen. Der Datenport überträgt auf diese Weise 32 Bit pro Taktzyklus, der Instruktionsport sogar 64 Bit.

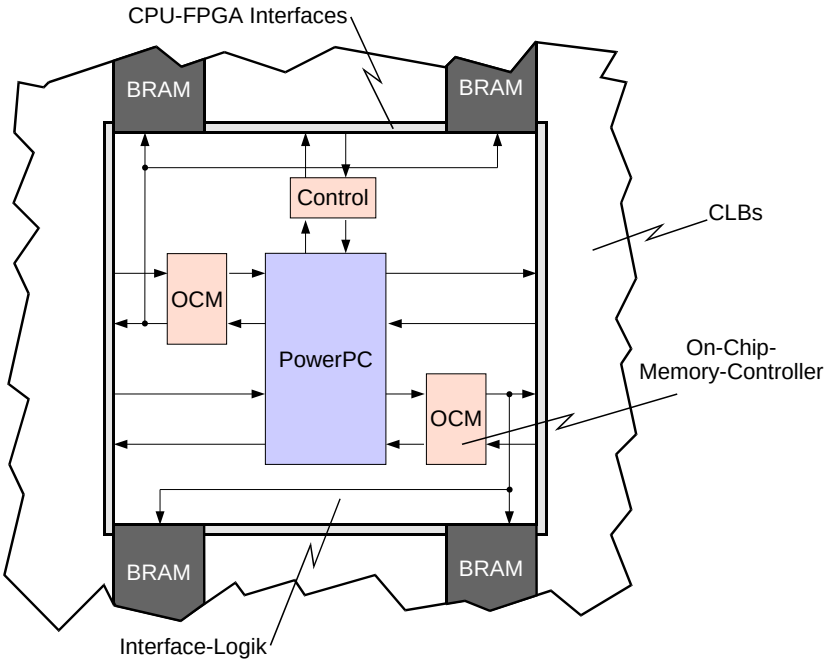


Bild 6.16
Eingebetteter PowerPC

Programme für den PowerPC 405 werden ganz normal in den Sprachen C, C++ oder auch Java entwickelt und können mit einem *Instruction-Set-Simulator* (ISS) getestet werden. Dabei können wir den ISS sogar mit unserem VERILOG-Simulator koppeln und so das gesamte Hardware-Software-System im PC simulieren. Die Simulation weniger Sekunden FPGA-Betriebszeit dauert dann je nach Komplexität des Designs allerdings mehrere Stunden!

Effizienter ist es, die Software mit einem *Cross-Compiler* für den PowerPC 405 zu übersetzen und in das FPGA zu laden (Kapitel 7). Ein *Remote-Debugger*, der über ein USB-Kabel mit dem Virtex-II Pro in Verbindung steht, ermöglicht dann die volle Kontrolle des Programmflusses, und Prozessor-Register, Programm-Variablen sowie der Speicherinhalt können im laufenden Betrieb beobachtet werden. Diese leicht zu bedienende Technik werden wir auch in den praktischen Übungen einsetzen.

Beide Methoden sind bei größeren Projekte jedoch unbefriedigend. Das Simulieren mit ISS ist wegen der langen Wartezeiten oft unpraktisch, das Testen direkt auf dem FPGA ist aber oft auch nicht möglich, da sich die Hardware-Module des

Gesamtsystems ebenfalls erst in der Entwicklung befinden. Bei neuen Ansätzen werden in der Forschung deshalb *High-Level-Modelle* des System-on-Chip für die Software-Entwicklung und -Verifikation verwendet, zum Beispiel mit der System-Beschreibungssprache *SystemC*. Die Simulationsgeschwindigkeit ist dann nur etwa 10 bis 100 mal langsamer als die simulierte Hardware selbst!

6.4.5 Kommunikation im Chip

Typische System-on-Chip-Designs umfassen oft Dutzende von Hardware-Cores, die untereinander Daten austauschen und mit den Prozessoren um den Zugriff auf gemeinsam genutzte Speicherbänke konkurrieren. Eine gut durchdachte Kommunikationsarchitektur hilft, den Bedarf an Verdrahtungs-Ressourcen auf dem FPGA nicht durch unkontrolliert verlegte Punkt-zu-Punkt-Leitungen explodieren zu lassen. Zur Auswahl stehen daher ein *Network-on-Chip* mit eigenen Bussen und wie früher ein ausgeklügeltes Interconnect für die Verdrahtung der Logikblöcke (Bild 6.17).

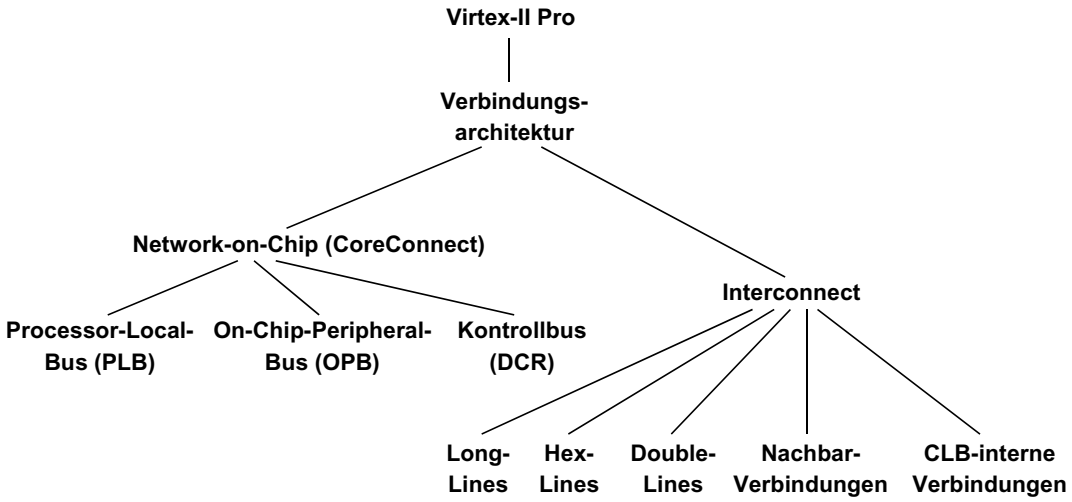


Bild 6.17 Verbindungsarchitektur im Chip

6.4.5.1 Network-on-Chip

Die FPGAs Virtex-II Pro und Virtex-4 verwenden als internes Network-on-Chip die von IBM entwickelte Architektur *CoreConnect* (Bild 6.18).

Der Datenaustausch erfolgt über zwei *On-Chip-Busse*, die von allen Systemkomponenten gemeinsam für die Datenübertragung genutzt werden können. Sie sind jedoch nicht fest auf dem Chip vorhanden, sondern werden je nach Bedarf aus Logikzellen und mit dem Interconnect (Abschnitt 6.4.5.2) rekonfigueriert.

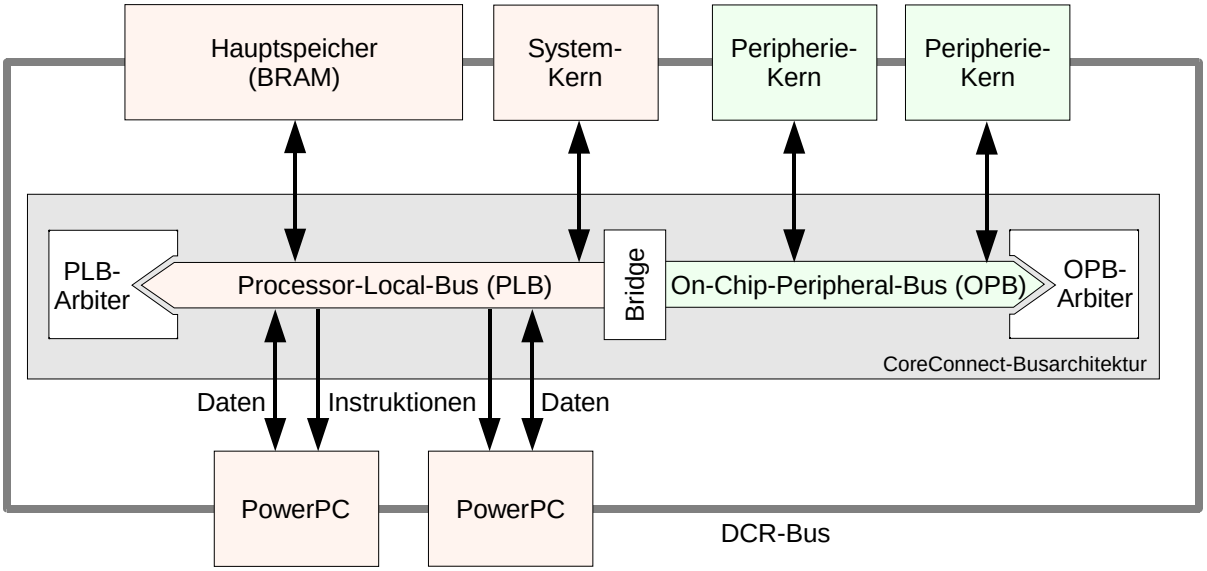


Bild 6.18 Busarchitektur CoreConnect

Der *Processor-Local-Bus* (PLB) ist mit 100 MHz getaktet und verwendet 2x64 Leitungen für die Datenübertragung und 32 zur Adressierung. Er ist für die schnelle Anbindung der PowerPC-Prozessoren an den Hauptspeicher und andere Systemkerne vorgesehen und bietet mehrere Funktionen speziell für die optimierte Datenübertragung von und zu den Caches der Prozessoren, wobei durch getrennte 64 Bit breite Schreib- und Leseleitungen der eine Prozessor Daten in den Speicher schreiben kann, während der andere gleichzeitig Daten aus einem anderen Speicherbereich einliest.

An den *On-Chip-Peripheral-Bus* (OPB) werden alle anderen Hardware-Module des System-on-Chip angeschlossen. Das Zugriffsprotokoll ist einfacher als beim PLB, so dass für den Anschluss eines Moduls an den OPB weniger Logik benötigt wird. Für die Datenübertragung und die Adressierung stehen je 32 Leitungen zur Verfügung, so dass pro Taktzyklus 32 Bit Daten gelesen oder geschrieben werden können.

Durch die Verteilung der Kommunikation auf zwei Busse stören die häufigen Hauptspeicherzugriffe der beiden PowerPCs nicht die Datenübertragungen zwischen anderen Systemkomponenten. Ist eine Verbindung zwischen einer PLB- und einer OPB-Komponente gewünscht, können die beiden Busse über eine *Bridge* verbunden werden.

Bei allen On-Chip-Bussen konkurrieren die angeschlossenen Module um den Zugriff auf das gemeinsam genutzte Medium. Ein streng einzuhaltendes Bus-Zugriffsprotokoll regelt den Kommunikationsablauf, und ein ebenfalls an den Bus angeschlossener *Arbiter* gewährleistet den kollisionsfreien Zugriff. Je nach Bus können hierbei verschiedene Dienste wie prioritäten-basiertes Scheduling und Burst-Transfers für die unterbrechungsfreie Übertragung großer Datenmengen realisiert werden.

Ein großer Vorteil von On-Chip-Bussen ist der geringe Verbrauch an FPGA-Ressourcen durch gemeinsam genutzte Leitungen. Dem gegenüber steht jedoch die zusätzliche Logik, die bei jedem Bus-Teilnehmer für die Zugriffssteuerung benötigt wird, so dass bei nur zwei oder drei Kommunikationspartnern Punkt-zu-Punkt-Verbindungen sinnvoller sind.

Neben CoreConnect gibt es noch weitere On-Chip-Busse wie AMBA von ARM oder WishBone von der Initiative opencores.org. Allen gemeinsam ist eine jeweils standardisierte Kommunikations-Schnittstelle. So können wir jedes mit CoreConnect kompatible Hardware-Modul sofort in unser Virtex-II-Pro-Design integrieren, ohne Kommunikationsprobleme befürchten zu müssen. Ein weiterer Vorteil ist, dass das zeitliche Übertragungsverhalten von Bussen mit Systembeschreibungs-Sprachen wie SystemC schon im High-Level-Modell exakt simuliert werden kann. So können wir den für unser MPSoC-Design am besten geeigneten Bus auswählen, *bevor* wir mit der Entwicklung der Hardware-Module (und ihrer Kommunikation-Schnittstellen) in VERILOG beginnen.

6.4.5.2 Interconnect

Angesichts der Größe von Virtex-II-Pro-FPGAs spielen Laufzeitverzögerungen durch lange Leitungen eine wichtige Rolle. Schaltet man viele kurze Leitungen durch Schalter zusammen, so wächst die Laufzeit quadratisch mit der Zahl der Schalter (Abschnitt 6.3). Längere schalterfreie Leitungsstücke werten ein Interconnect daher auf.

Vor allem die Verteilung der Taktsignale über den gesamten Chip ist äußerst kritisch, denn eine zu große Signalverzögerung würde die maximale Taktfrequenz für eine Schaltung empfindlich dezimieren. Aber auch normale Signale sollten immer auf kürzestem Wege transportiert werden, um den kritischen Pfad großer kombinatorischer Blöcke zu minimieren. Die FPGA-Plattformen Virtex-II Pro und Virtex-4 bieten dazu ein hierarchisch organisiertes Netz von Verdrahtungsressourcen.

Das Interconnect des Virtex-II Pro zeigt Bild 6.19. Innerhalb eines CLBs gibt es schnelle interne Verbindungen der Slices (siehe auch Bild 6.13), nach außen ist jedes CLB über 16 Nachbarverbindungen angeschlossen. Double-, Hex- und Long-Lines stellen in horizontaler und vertikaler Richtung Leitungsabschnitte passender Länge zur Verfügung.

Für die schnelle Taktverteilung ist jeder Quadrant des FPGAs außerdem mit acht Taktverteilungsnetzen ausgestattet, die sich durch eine besonders geringe Signalverzögerung auszeichnen. Sie sind an alle taktgesteuerten Elemente wie CLB-Register und Blockspeicher-Zellen angeschlossen und können entweder mit programmierbaren Taktgeneratoren (Digital Clock Manager, DCM) oder über spezielle Clock-Pins des FPGA angesteuert werden.

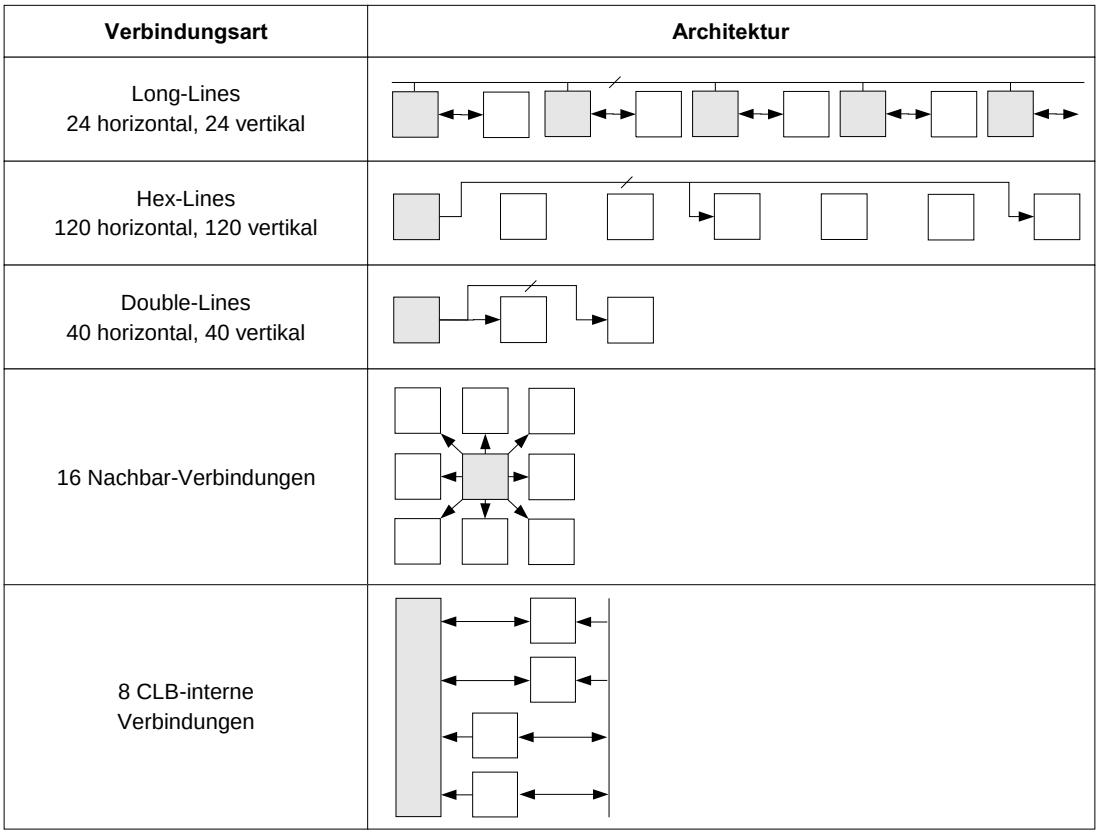


Bild 6.19 Hierarchisches Interconnect

Angesichts dieser komplexen Verdrahtungshierarchie wird deutlich, welche Anforderungen an die Synthesewerkzeuge gestellt werden, hier eine gute Platzierung und Verdrahtung für eine Schaltung mit Millionen von Gattern zu berechnen (vgl. Abschnitt 6.1.3).

6.4.6 Kommunikation mit dem Chip

Für die Verbindung des FPGAs mit seiner Umgebung, etwa mit der Entwicklungsumgebung ML310, gibt es diverse IO-Ports (Input-Output). Interessant sind beispielsweise serielle und sehr schnelle Transceiver mit 3,125 Gbit/s oder IO-Register für den Anschluss externer DDR-Speicher (Double Data Rate, Datenübertragung zur positiven und negativen Taktflanke), auf die wir im nächsten Kapitel genauer eingehen werden.

6.4.7 Rekonfiguration

In Abschnitt 6.4.1 hatten wir die spaltenweise Anordnung der Logikblöcke, Blockspeicher und Multiplizier-Einheiten erörtert. Für die Programmierung oder Rekonfiguration dieser Elemente werden alle Spalten als *Boundary-Scan* hintereinander geschaltet und mit einem seriellen Konfigurations-Bitstrom geladen. Neu ist, dass

dabei nicht nur das gesamte FPGA auf einmal, sondern auch eine Auswahl von Spalten rekonfiguriert werden kann. Bei dieser *partiellen Rekonfiguration* läuft der Rest des Systems ungestört weiter, so dass beispielsweise bei einem Video-Dekoder je nach Bedarf der passende Dekompressions-Algorithmus nachgeladen werden kann.

Während Standard-Bitströme mit der ISE-Entwicklungsumgebung (Kapitel C) aus jeder RTL-Netzliste komfortabel erzeugt werden können, sind partielle Bitströme allerdings noch problembehaftet und Gegenstand aktueller Forschung. Zum Beispiel muss sichergestellt werden, dass Signale an den Rändern des rekonfigurierten Bereichs korrekt mit der Umgebung verbunden werden, was eine exakte Kontrolle des Platzierungs- und Verdrahtungsprozesses erfordert.

Zum Schutz vor Reverse-Engineering (Ausspionieren des teuer entwickelten System-on-Chip) kann der Bitstrom mit dem DES-Verfahren verschlüsselt werden. Die Entschlüsselung wird auf dem FPGA mit einem nicht-flüchtig einprogrammierten DES-Schlüsselsatz vorgenommen.

6.5 Technische Daten und Ausblick

Als Hintergrundinformation wollen wir für die Experten einige (nicht prüfungsrelevante) technische Angaben zusammentragen. Wesentliche Daten der FPGA-Plattform und ihrer Varianten sowie der Nachfolge-Plattform Virtex-4 zeigt Tabelle 6.20. Die rasante Entwicklung des FPGA-Sektors wird an der letzten Tabellenzeile deutlich, in der die Daten der XC4000-Familie vom Anfang des Kapitels gegenübergestellt werden. Die XC4000-Serie wurde bis 1999 regelmäßig erweitert.

Typ	Logikzellen	Blockspeicher (KBit)	PowerPC-CPUs	Multiplizierer 18x18 Bit	IO-Pins
Virtex-II	576 - 104.882	72 - 3.024	-	4 - 168	88 - 1.108
Virtex-II Pro	3.168 - 99.216	216 - 7.992	0 - 2	12 - 444	204 - 1.164
Virtex-4 LX	13.824 - 200.448	864 - 6.048	-	32 - 96	320 - 960
Virtex-4 SX	23.040 - 55.296	2.304 - 5.760	-	128 - 512	320 - 640
Virtex-4 FX	12.312 - 142.128	648 - 9.936	1 - 2	32 - 192	320 - 896
XC4000	1.368 - 20.102	-	-	-	192 - 448

Tabelle 6.20 Varianten der Virtex-FPGA-Plattformen

Seit ihrer Einführung wurde die FPGA-Plattform Virtex-II Pro fortlaufend um leistungsfähigere Bausteine erweitert. Das derzeit größte Familienmitglied V2P100 bietet ca. 8 Mio. Gatteräquivalente in Form von etwa 100.000 Logikzellen, zwei PowerPC-Prozessoren, 8 MBit Blockspeicher, 444 Multiplizier-Blöcken, 12 Taktgeneratoren und über 1.100 frei verwendbaren IO-Pins. Im Praktikum setzen

wir „nur“ ein V2P30 ein, mit etwa 30.000 Logikzellen, 2,4 MBit Blockspeicher und 136 Multiplizier-Blöcken.

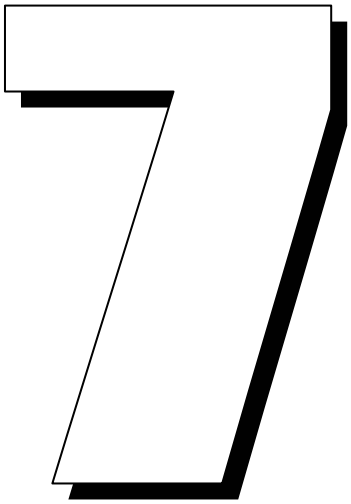
Der Virtex-II Pro wird als 130-nm-Prozess gefertigt, d.h. die effektive Transistor-Gate-Länge ist $1{,}3 * 10^{-7}$ m. Die maximal 430 Millionen Transistoren des V2P100 werden auf 10 Kupferlagen verdrahtet. Die Laufzeit eines kombinatorischen Logikblocks liegt bei weniger als 0,28 ns. Die Herstellung entspricht einem normalen Standardprozess für SRAM-Speicher. Die Chips werden auf 300 mm großen Silizium-Scheiben (Wafers) gefertigt.

Ein Konfigurations-Bitstrom beispielsweise für das V2P30 ist 1,4 MB groß, eine Rekonfiguration dauert hier 29 ms.

Die seit 2005 verfügbaren Virtex-4-Bausteine verwenden eine im Vergleich zu Virtex-II Pro kaum veränderte Technologie, erreichen aber durch den neueren und teureren 90-nm-Prozess sogar 200.448 Logikzellen bei einer maximalen Taktfrequenz von 450 MHz.

Der aktuelle Trend, FPGAs zunehmend auch in Endprodukten einzusetzen, führt dazu, dass die Hersteller ihre Modellpalette stärker diversifizieren. So kann für jede Anwendung ein passendes und gleichzeitig wirtschaftliches FPGA gefunden werden. Xilinx basiert seine Virtex-4-FPGAs hierzu auf der so genannten ASMBL-Architektur (Advanced Silicon Modular BLock), mit der die FPGAs bausatzartig aus einer Bibliothek von Logik-, DSP- und Prozessorblöcken zusammengesetzt werden. Verschiedene Serien bieten entweder mehr Logikblöcke, mehr DSP-Blöcke oder integrierte PowerPC405-Prozessoren und schnelle IO-Ports für eingebettete Anwendungen. Werden ausschließlich Logikblöcke benötigt, bieten sich auch die speziell auf den Low-Cost-Markt ausgerichteten Spartan-3-FPGAs von Xilinx oder Cyclone-II von Altera an, die ebenfalls beide in 90 nm gefertigt schon ab 2 $ pro 100.000 Gatteräquivalente zu bekommen sind.

Eine interessante Alternative zu den immer größeren und damit auch leistungshungrigeren FPGAs ist das Konzept der software-konfigurierbaren Prozessoren. Bei der Stretch-CPU wird dazu ein programmierbares Logik-Array um einen Standard-Prozessorkern herumgelegt, so dass ein vollständig software-konfigurierbarer Datenpfad entsteht. Für häufig wiederkehrende Berechnungen wie bei der DES-Verschlüsselung oder beim Skalieren von Bildern können dem Prozessor Spezial-Instruktionen einprogrammiert werden, mit denen *Hot Spots* aus einigen 10 bis 100 Instruktionen in nur einem Takt ausgeführt werden können. Das hier verwendete Prinzip ist aber keinesfalls neu, denn auch digitalen Signalprozessoren (DSP) werden applikationsspezifische Instruktionen einprogrammiert, um die Geschwindigkeit komplexer mathematischer Operationen zu optimieren. Anstatt auf eine Auswahl festgelegter DSP-Befehle beschränkt zu sein, können auf der Stretch-CPU jedoch beliebige Logikschaltungen feingranular konfiguriert werden. Die Erweiterungs-Instruktionen werden dazu mit Hilfe eines speziellen C/C++-Compilers erzeugt.



Hardware-Software-Codesign

Der Begriff des *Hardware-Software-Codesign* wird häufig verwendet, aber worin besteht das Neue? Schon seit jeher wurden Computer entworfen, die offensichtlich aus Hardware *und* Software bestehen. Auch der Begriff *Design* ist nur eine chice Fassung für *Entwurf*, den wir schon immer betrachtet haben, nämlich den kreativen Teil einer Entwicklung, auf den die weitgehend automatische Implementierung folgt.

Das Neue am Hardware-Software-Codesign besteht in dem Zusatz „Co“, der gleich mehrere Interpretationen erlaubt:

Concurrent: Neben der Interpretation parallel arbeitender Hard- und Software ist vor allem der *gleichzeitige Entwurf* von Hard- und Software gemeint im Gegensatz zum klassischen „hardware first“, bei dem zunächst ein Hardware-Baustein entwickelt wird und dann eine passende Software.

Communicating: Die richtige Kommunikation zwischen Hardware und Software spielt eine wesentliche Rolle.

Coordinated: Es geht um den koordinierten systematischen Entwurf von Hardware und Software.

Complex: Systeme aus Hardware und Software sind gewöhnlich besonders komplex, vor allem wenn sie eng aufeinander abgestimmt sind.

Correct: Selbstverständlich soll ein korrekt zusammenspielendes System entstehen, auch wenn das in der Praxis oft nur aufwändig und unvollständig gelingt.

Cooperation: Hardware und Software arbeiten auf vielfältige Weise zusammen.

Es geht also um meist heterogene Systeme aus diversen Hardware- und Software-Bausteinen, die oft als *eingebettete Systeme* ihre Anwendung finden. Ein typisches Anwendungsbeispiel ist das Kraftfahrzeug, das in der Oberklasse etwa 100 Mikroprozessoren und Spezialchips enthält. Weitere Anwendungsfelder finden sich in Abschnitt 7.3.

7.1 Hardware-Software-Bausteine

Eine Systemlösung durch einen universellen Rechner wird gewöhnlich als reine Software-Lösung bezeichnet, obwohl natürlich die Rechner-Hardware dazu gehört. Umgekehrt wird ein FPGA gewöhnlich als Hardware-Lösung bezeichnet, obwohl auch sie die software-mäßige Programmierung der realisierten Schaltung enthält. In diesem Sinne besteht ein Hardware-Software-System aus Software (auf einem Standard-Baustein) und maßgeschneiderter Hardware (die programmierbar sein kann).

7.1.1 Software

Der Software-Anteil eines Hardware-Software-Codesigns kann auf Universalrechnern realisiert sein, auf Mikrocontrollern, durch digitale Signalprozessoren oder auch durch ASIPs (Application-Specific Instruction Processor).

Die bekannten Universalrechner wie PCs oder Workstations zeichnen sich durch Vielseitigkeit aus. Obwohl die zugrunde liegende Hardware auf höchste Leistung optimiert ist, sind darauf realisierte Programme oft nicht sonderlich effizient. Wegen der komplexen Betriebssysteme sind sie für Echtzeit-Anwendungen schlecht geeignet. Universalrechner erfordern einen sehr hohen Entwurfsaufwand (beispielsweise 300 Mannjahre) und sind daher nur in hohen Stückzahlen rentabel.

Mikrocontroller sind mehr für Steuerungen geeignet, also kontrollflussorientierte Aufgaben mit nur geringen Datendurchsätzen. Typische Anwendungen bestehen aus vielen Tasks und benötigen vor allem Logik- und Bit-Operationen. Oft ist passende Peripherie bereits integriert. Es geht zum einen um Low-Cost-Bausteine, für die häufig immer noch billige 8-Bit-Mikrocontroller eingesetzt werden, etwa Intel 8051. Es werden aber auch High-Performance-Mikrocontroller benötigt, die oft hohe Datenraten verarbeiten, etwa beim elektronischen Stabilitätsprogramm und Anti-Blockier-System im Kraftfahrzeug. Schließlich gibt es Anwendungen mit hohen Datenraten und einer intensiven Datenmanipulation, etwa in der Telekommunikation, oder solche mit hohen Berechnungsanteilen, etwa in der Regelungstechnik.

Digitale Signalprozessoren (DSP) zeichnen sich durch hohe Rechenleistung aus, bei denen oft mit regelmäßigen Operationen sehr große Datenmengen verarbeitet werden, etwa in der Bildverarbeitung. Hier stehen Multiplikationen und Additionen im Vordergrund. Für diese Spezialrechner werden optimierende Compiler benötigt, die die regelmäßige Grundstruktur gut ausnutzen. Oft ist eine geringe Verlustleistung wünschenswert. Angeboten wird eine Vielzahl von Spezialarchitekturen, die tief in das Gebiet der Rechnerarchitektur hineinführen, z.B. Multi-DSP-Systeme oder VLIW-Architekturen (Very Long Instruction Word).

Schließlich seien noch die ASIPs genannt (Application-Specific Instruction Processor). Diese Familie weist oft sehr unterschiedliche Architekturen auf und enthält Hardware für stark spezialisierte Aufgaben. ASIPs arbeiten oft effizienter als

Universalrechner oder Mikrocontroller und möglicherweise auch digitale Signalprozessoren. Sie benötigen in der Regel weniger Pins, weniger Leistung und sind so billiger, erfordern aber einen speziellen Compiler. Oft besteht eine Instruktion aus einer Kette von Operatoren.

7.1.2 Hardware

Neben den im vorigen Abschnitt genannten Hardware-Bausteinen für Software-Lösungen kommen für den eigentlichen Hardware-Teil des Hardware-Software-Codesigns Bausteine zum Tragen wie kundenspezifische Hardware, Co-Prozessoren, digitale Chips, analoge Chips, Speicher und FPGAs.

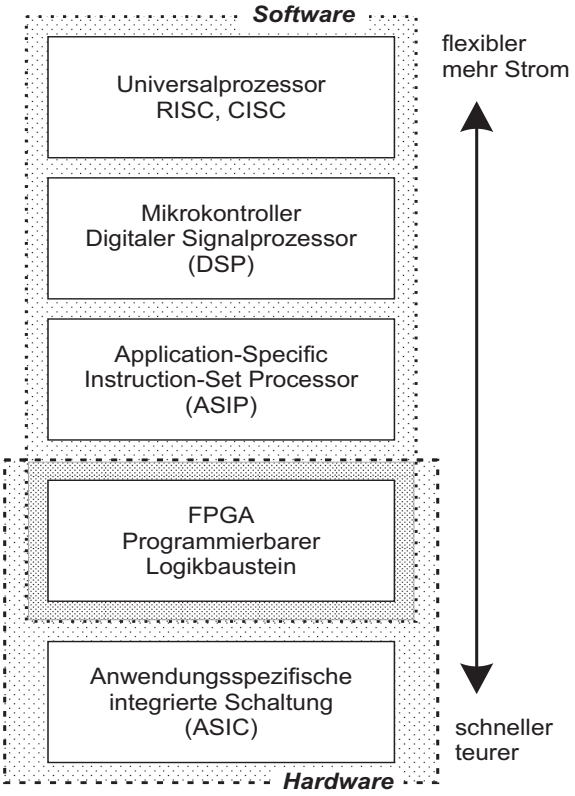


Bild 7.1 Hardware-Software-Bausteine

Bild 7.1 vergleicht die verschiedenen Implementierungsarten. Offensichtlich besteht ein *Trade-off* zwischen der Effizienz einerseits und der Flexibilität andererseits, und die Kunst des Hardware-Software-Codesign kann gerade darin bestehen, die Flexibilität erfordernden Anteile in Software zu realisieren, die rechenintensiven dagegen in spezifischer Hardware.

7.1.3 Hardware-Software-Systeme

Die oben diskutierten Software- und Hardware-Anteile können entweder auf einem Board zusammengefasst werden oder sogar auf einem einzigen Chip. Hardware-Software-Codesign kann darüber hinaus auch verteilte Systeme aus vernetzten Einzelsystemen umfassen.

Bei Board-Systemen werden die Software- und Hardware-Bausteine auf einer Platine montiert, was bei kleinen Stückzahlen preiswerter und schneller zu realisieren ist. Auch sind solche Systeme leichter änderbar und skalierbar durch die Erweiterung um zusätzliche Aufsatzplatinen. Allerdings ist die Kommunikation auf dem Board komplex und ineffizient.

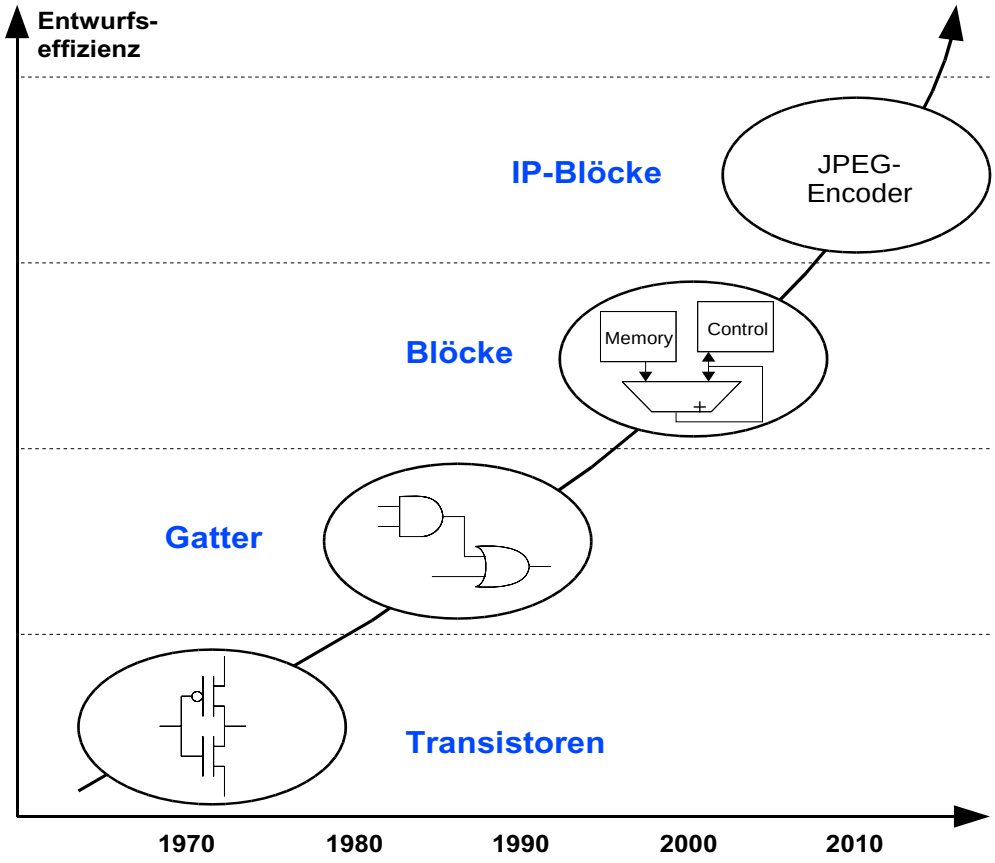


Bild 7.2 Entwurfsgrundlagen

Ein System-on-Chip (SoC) hat große Vorteile, die sich allerdings erst in der Massenproduktion rentieren: geringes Gewicht, geringe Leistungsaufnahme, die besonders bei Mobilsystemen wichtig ist, hohe Verlässlichkeit und die Integrierbarkeit analoger und peripherer Bausteine (z.B. Leistungstreiber, Sensoren). SoC-Lösungen haben einen noch viel größeren Entwurfsraum als klassische Rechner. Um diese gewaltige Vielfalt besser beherrschen zu können, wird intensiv mit der Kombination fertiger Einheiten gearbeitet wie Prozessorkernen, Speichern und Soft-, Firm- und

Hard-Blöcken, die wie in Bild 7.2 die komplexe Fortsetzung der klassischen Bibliothekselemente darstellen. In diesem Zusammenhang spricht man oft von *Intellectual Property* oder kurz IP und meint damit rechtlich geschützte Hardware-Bausteine, die meist als verschlüsselte Gatternetzlisten lizensiert werden.

7.2 Der Entwurfsprozess

Das *Productivity-Gap* – die Schere zwischen technisch möglicher und entwurfsmethodisch gut beherrschbarer Komplexität – klafft hier erst recht: nicht nur die Hardware mit ihren hunderten von Millionen von Transistoren ist exponenziell in ihrer Komplexität gewachsen, sondern erst recht Systeme aus Hardware und Software. Daher sind klassische Entwurfsmethoden zu ineffizient geworden; neben einem Design-Reuse großer fertiger Bausteine (vgl. Intellectual Property im vorigen Abschnitt) ist eine weitere Entwurfsautomatisierung gefragt, etwa die High-Level-Synthese zur Entwicklung von Spezial-Hardware, oder optimierende Compiler, die sich an alternative Hardware-Architekturen anpassen, sowie die Einbeziehung des Entwurfs geeigneter Betriebssysteme.

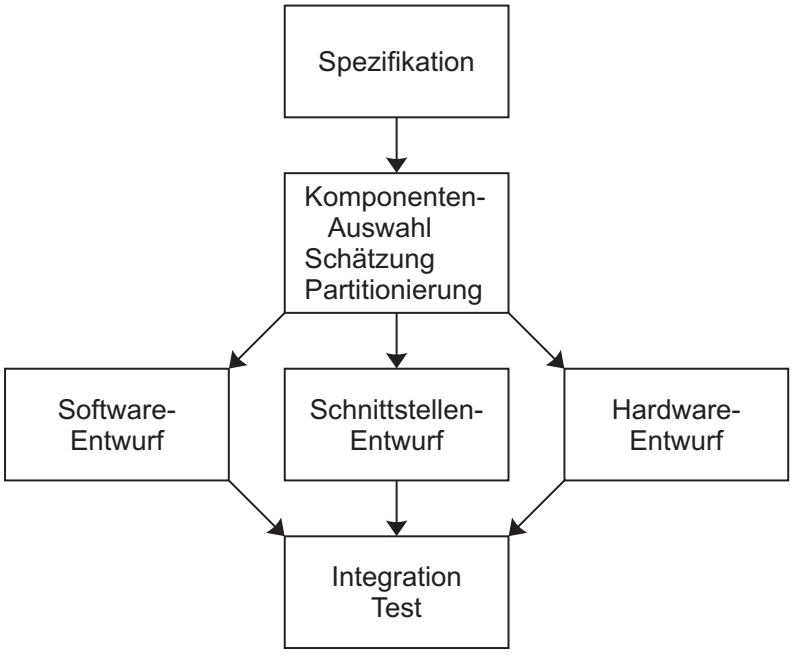


Bild 7.3 Entwurfsablauf im Hardware-Software-Codesign

Bild 7.3 zeigt die wesentlichen Schritte eines Hardware-Software-Codesign. Nach einer Analyse der Anforderungen führt die Spezifikation zu einem möglichst schon ausführbaren Modell in einer geeigneten Hochsprache, etwa UML und SystemC, Statecharts oder auch Mathlab/Simulink oder SDL.

Es folgt ein sehr schwieriger und komplexer Schritt zur Auswahl einer Grobarchitektur mit bestimmten Hardware- und Software-Komponenten, die zu einer Aufteilung oder Partitionierung in Software auf einem Standard-Prozessor (Abschnitt 7.1.1) und Spezial-Hardware (Abschnitt 7.1.2) führt.

Beispielsweise gehört zur Komponenten-Auswahl, ob ein System-on-Chip oder ein Board-System entwickelt werden soll. Von der gewählten Prozessorarchitektur hängt ab, welche Software-Entwicklungswerkzeuge (z.B. Compiler einer bestimmten Sprache) genutzt werden können. Es gehört Fingerspitzengefühl dazu, dass der Prozessor am Ende weder zu klein noch zu teuer ist. Für die Software muss gegebenenfalls auch ein Betriebssystem bestimmt werden.

Oft schon während der Komponenten-Auswahl wird die Partitionierung berücksichtigt, bei der das Gesamtsystem in einen Hardware- und einen Software-Teil zerlegt wird. Neben der Partitionierung sind für die Kommunikation zwischen Software und Hardware geeignete Schnittstellen, Interfaces, Protokolle etc. zu definieren und zu entwerfen. Ohne sorgfältige Schnittstellen-Spezifikation können später teure Redesigns von Software und Hardware notwendig werden.

Nach dem Software-, Hardware- und Schnittstellen-Entwurf wird das System integriert.

Auf allen Ebenen sind eine Co-Simulation und Co-Verifikation von Hard- und Software gefragt bis hin zum Test des integrierten Systems. Ein Problem von Co-Simulation und Co-Verifikation besteht darin, dass im Allgemeinen die Software-Prozessoren abstrakter modelliert werden als die Spezial-Hardware. Das Zeitverhalten des Prozessors wird oft nicht mehr taktgenau dargestellt, sondern nur noch bezüglich seiner Ein- und Ausgaben. Dadurch können in der gleichen Simulationszeit größere Zeitabschnitte der Software simuliert werden als der Hardware.

In der Co-Simulation können dann aus Gründen der Simulationsdauer neben vielen detaillierten Hardware-Schritten nur kurze Software-Abschnitte gleichzeitig simuliert werden. Dies rechtfertigt den Einsatz einer Co-Emulation in Form eines Prototypen-Boards (Rapid-Prototyping).

Die automatische oder zumindest halbautomatische Partitionierung in Hardware und Software ist ein außerordentlich komplexes Problem und Gegenstand jahrelanger intensiver Forschungen. Bisher gibt es nur in Einzelfällen wirklich praxisrelevante Ergebnisse. Bei manchen HW-SW-Codesigns ist die Partitionierung jedoch schon in natürlicher Weise vorgegeben, etwa bei der Auslagerung eines vorgegebenen rechenintensiven Programms zur kryptografischen Verschlüsselung.

Die automatische Partitionierung basiert wesentlich auf Schätzverfahren, die den Nutzen einer konkreten Partitionierung möglichst exakt vorhersagen oder zumindest relativ richtige Vorhersagen treffen und bei denen der Schätzaufwand wesentlich kleiner ist als das Finden der fertigen Lösung. Qualitätskriterien bei der Schätzung sind die Effizienz (Takt, Latenz, Ausführungszeit, Datenrate) sowie für die beteiligte

Hardware Schätzungen zur Fläche, zu den Kosten, zur Leistung und zur Testbarkeit. Auch die Kommunikationskosten müssen berücksichtigt werden.

Ebenfalls große Bedeutung hat neben dem Hardware- und Software-Entwurf das Kommunikations-Design, besonders wenn das komplette System auf einem einzigen System-on-Chip integriert werden soll. Man unterscheidet zwischen *Protokollen* auf relativ abstrakter Ebene und *Interfaces* auf niedriger Ebene zwischen den Hardware- und Software-Bausteinen.

7.3 Eingebettete Systeme

Eingebettete Systeme (*Embedded Systems*) sind im Gegensatz zu klassischen Universalrechnern in einem technischen System oder Produkt verborgen. Eingebettete Systeme können nur aus Software (auf einem Standardprozessor) bestehen, nur aus (anwendungsspezifischer) Hardware, oft aber als Hardware-Software-Codesign aus beidem. Anwendungsbeispiele finden sich in Vermittlungssystemen und Endgeräten der Telekommunikation, vor allem auch der Mobilkommunikation, in der Unterhaltungselektronik, in den Bereichen Messen, Prüfen und Regeln, im Kraftfahrzeug, in der Automatisierung oder bei verteilten Systemen. Tabelle 7.4 zeigt konkrete Beispiele, weitere finden sich in der Sammlung 1.1

Beispiel	Sensoren	Aktoren	Interfaces	Kommunikation
Laserdrucker	Temperatur Füllstand Papierzufuhr	Motoren Heizung Anzeigen	A/D-Wandler D/A-Wandler Pulsformer	Ethernet Parallel Seriell (USB)
Auto	Drehzahl Position Druck Haftreibung Kollision Temperatur	Zündung Einspritzung Elektromotor Bremsen Kupplung Kühlung	A/D-Wandler D/A-Wandler Zähler	CAN-Bus LIN-Bus FlexRay
Handy	Tasten Akkuzustand Mikrofon Empfangsteil	Lautsprecher Display Vibration Funksender	A/D-Wandler D/A-Wandler	Seriell (USB) Infrarot Bluetooth
Videokamera	Tasten Mikrofon Akkuzustand CCD-Sensor	Display Fokus, Blende Bandlaufwerk	A/D-Wandler D/A-Wandler	FireWire, USB Analog-Video Audio

Tabelle 7.4 Beispiele eingebetteter Systeme

Man unterscheidet reaktive und transformierende Systeme. Die *transformierenden* Systeme entsprechen mehr den klassischen Rechnern, während *reaktive* Systeme potenziell ohne Ende auf Stimuli der Umwelt geeignet antworten.

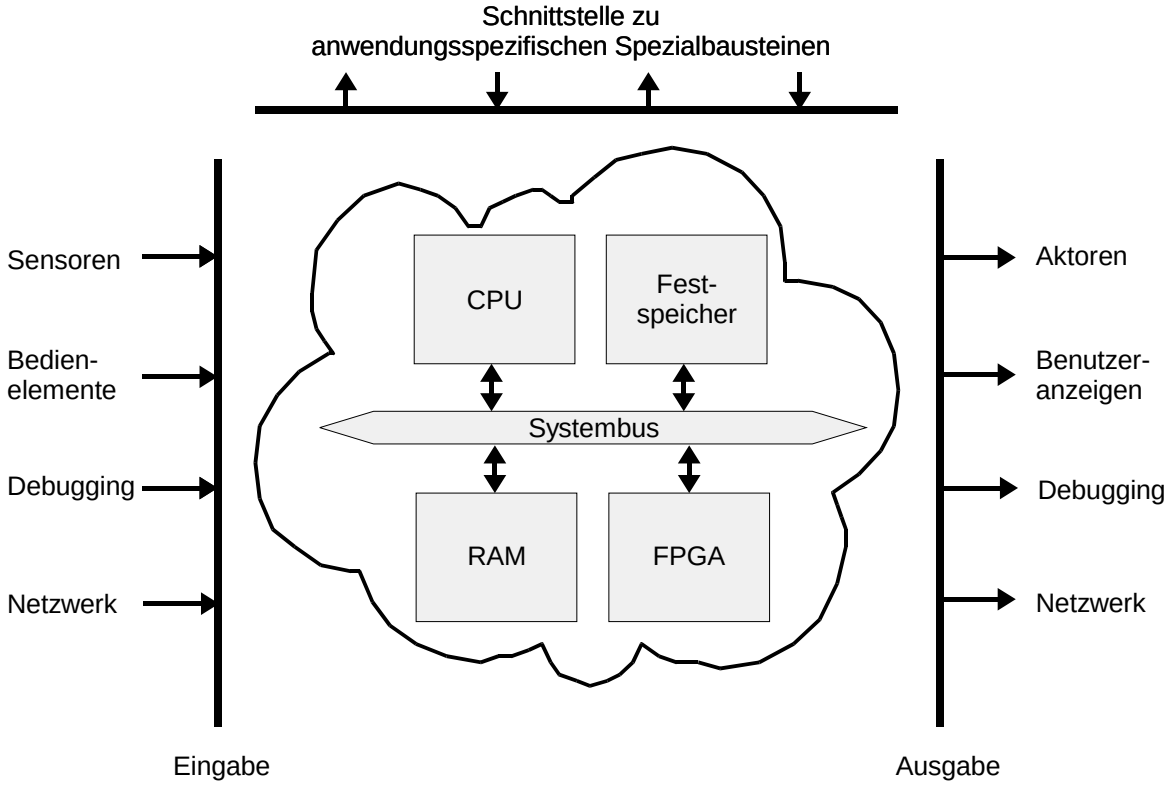


Bild 7.5 Basisarchitektur für eingebettete Systeme

Viele eingebettete Systeme basieren zwar auf einem Mikrocontroller mit CPU, unterscheiden sich aber wie in Bild 7.5 von klassischen Rechnern: oft fehlt ein Massenspeicher wie eine Festplatte, die Bedienelemente und Benutzeranzeigen sind eingeschränkt, dafür gibt es eine Debug-Schnittstelle zur Fehlersuche durch einen externen Host-Rechner, das System steht reaktiv über spezielle Sensoren und Aktoren mit der Umwelt in Verbindung und ist in vielen Fällen an ein Netzwerk angeschlossen.

Je nach Anwendung kommen noch Schnittstellen für zusätzliche Komponenten hinzu wie etwa einen Videoprozessor oder einen Mobilfunk-Übertrager, und immer häufiger werden auch FPGAs für diverse andere IO-Schnittstellen und eine höhere Flexibilität eingesetzt. So kann beispielsweise ein DVD-Player mit FPGA leichter an einen neuen Video-Codec angepasst werden.

Beispielsweise sind bei einem eingebetteten System für einen PKW-Fensterheber in Bild 7.6 Schnittstellen für Schalter, Steuerbusse (hier den CAN-Bus), Verklemmungs-Sensoren sowie zur Motorsteuerung vorhanden. Die notwendige Vermittlung zwischen der digitalen Mikrocontroller-Welt und den meist analog arbeitenden Sensoren und Aktoren wird über Analog-Digital-Wandler (A/D bzw. D/A) realisiert, PWM steht hier für Pulsweiten-Modulation, ein einfaches Verfahren zur digitalen Signalübermittlung.

Das Leistungsspektrum eingebetteter Systeme ist sehr breit gefächert von einfachen *Low-End-Systemen* mit 4-Bit-Architektur und wenigen hundert Byte Speicher, etwa in Armbanduhren, bis zu *High-End-Systemen* mit 128-Bit-Datenbus

und mehreren 100 MB, die als „Rechenboliden“ beispielsweise in einer Spiele-Konsole arbeiten.

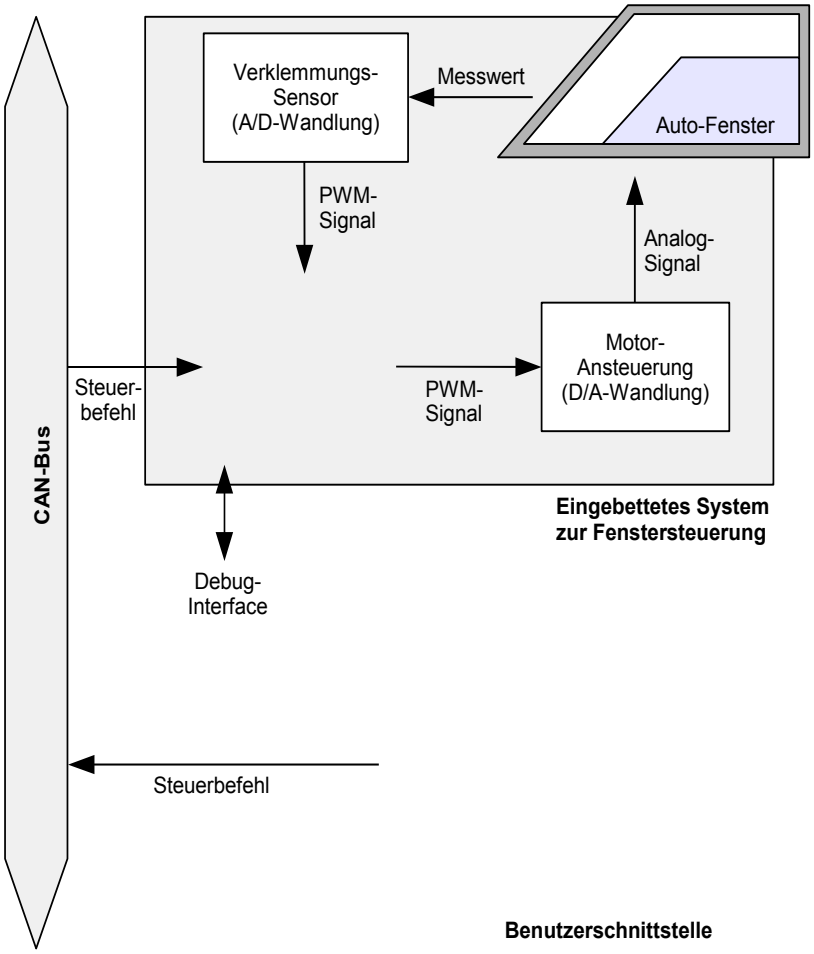


Bild 7.6 Einfaches eingebettetes System „Fensterheber“

Jede Anwendung hat ihre charakteristischen Besonderheiten, ein universelles eingebettetes System gibt es nicht. Beispielsweise werden an eine Waschmaschinensteuerung andere Anforderungen gestellt als an ein Handy-System. Neben der Aufgabenstellung selbst sind daher eine ganze Reihe von Anforderungen (*Constraints*) mit unterschiedlicher Gewichtung zu berücksichtigen (Bild 7.7).

Bei Anwendungen mit hohen Fertigungsstückzahlen stehen generell die Bauteil- und Fertigungskosten an erster Stelle, in der Weltraum- oder Medizintechnik (z.B. Herzschrittmacher) dagegen die Zuverlässigkeit. Zu den Constraints können aber auch scheinbar nebensächliche Dinge gehören, etwa dass eine Platine exakt in ein vorgegebenes Gehäuse passen muss.

Während bei eingebetteten Systemen oft die Anforderungsparameter Kosten, Stromverbrauch, Baugröße und Zuverlässigkeit dominieren, kann auch hier zusätzliche Spezial-Hardware sinnvoll sein, beispielsweise um einen sparsamen Energieverbrauch zu erreichen (low Power).

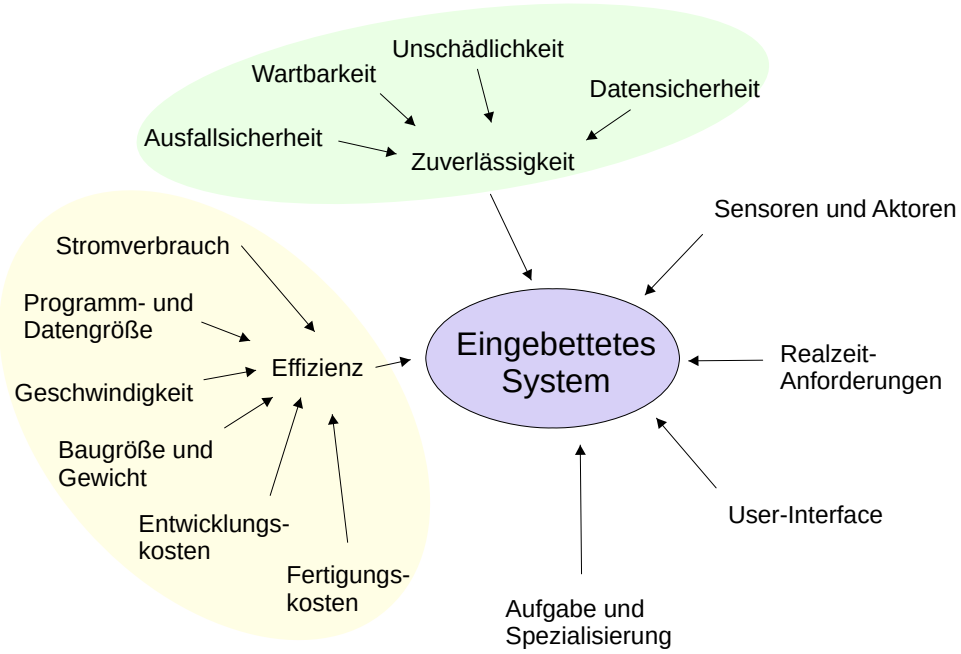


Bild 7.7 Anforderungen an ein eingebettetes System

Einfachere eingebettete Systeme, die nur in Hardware realisiert sind, kommen ohne ein Betriebssystem aus, bei einer komplexeren Software-Lösung dagegen wird ein Betriebssystem benötigt. Während einfache Hardware-Systeme meist auch in Echtzeit arbeiten, gewährleistet bei Software-Systemen ggf. ein Echtzeit-Betriebssystem, dass jeder einzelne Software-Prozess innerhalb einer fest vorgegebenen Zeit ausgeführt und nicht von anderen Prozessen verzögert wird. Man unterscheidet zwischen „harten“ und „weichen“ Echtzeitanforderungen, für die bei der Komponenten-Auswahl das richtige Betriebssystem gewählt werden muss.

7.4 Beispiele für eingebettete Systeme

Wie wir gesehen haben, decken eingebettete Systeme ein immens breites Spektrum von Anwendungen ab, vom einfachen digitalen Fahrradtachometer bis zum High-End-Navigationssystem. Dennoch basieren fast alle eingebetteten Systeme auf der gleichen typischen Basisarchitektur aus Bild 7.5. Abhängig von den Anforderungen an das Produkt werden die Standardbausteine CPU bzw. Mikrocontroller, Speicher und oft FPGA sinnvoll ausgewählt und dimensioniert, so dass als Ergebnis ein integriertes Gesamtsystem entsteht, das – entweder als System-on-Chip oder aus diskreten Bausteinen auf einer Platine aufgebaut – genau für die zu bewältigenden Aufgaben maßgeschneidert ist.

Um etwas Gefühl für den typischen Aufbau eingebetteter Systeme zu bekommen, wollen wir uns nun zwei sehr unterschiedliche Beispiele etwas genauer ansehen: eine Steuereinheit für die Home-Automation zum Fernsteuern von Lampen und Jalousien

auf Basis eines kostengünstigen 8-Bit-Mikrocontrollers und eine digitale Fotokamera, die mit der FPGA-Plattform Virtex-II Pro aus Abschnitt 6.4 realisiert wird.

7.4.1 Eine Steuereinheit für die Home-Automation

In der Home-Automation werden Fenster, Jalousien, Lampen, HiFi-, TV- und Haushaltsgeräte sowie die Warmwasser- und Heizungsanlage mit kleinen Steuergeräten ausgestattet, die eine Fernsteuerung aller wichtigen Funktionen ermöglichen. Durch die Einbindung dieser Steuergeräte in einen Netzwerkverbund (In-Home-Network) können Beleuchtung, Klima und Unterhaltungselektronik von jedem beliebigen Punkt im Haus überwacht und kontrolliert werden, beispielsweise durch Bedien-Terminals an der Wand, einen PC mit Wireless-LAN oder ein Bluetooth-Handy.

Wichtig für ein programmierbares eingebettetes Steuergerät, das in einem *Smart-Home* in jede Lampe, Jalousie etc. integriert wird, sind daher vor allem die folgenden Faktoren:

- Low-Cost (wenige Euro)
- Low-Power (wenige Milliwatt)
- verschiedene Ein- und Ausgänge für Sensoren und Aktoren
- Netzwerkanbindung

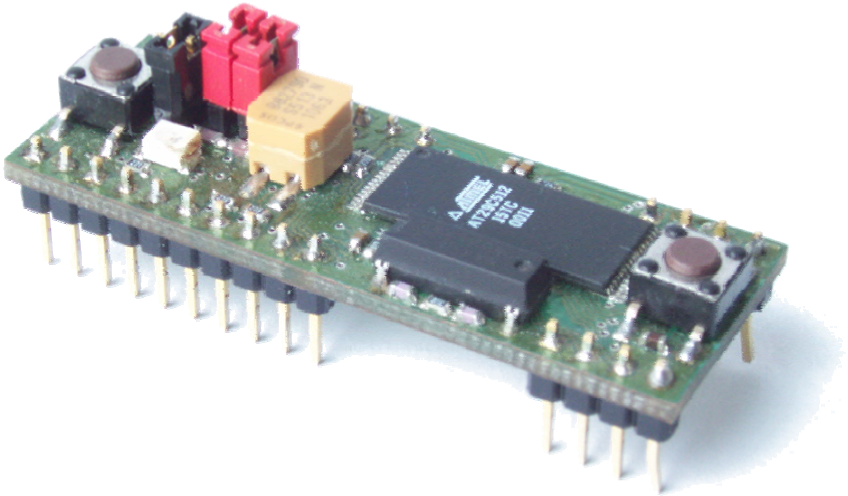


Bild 7.8
Steuereinheit JControl
für die Home-Automation

Die Steuereinheit JControl aus Bild 7.8 erfüllt diese Anforderungen durch Einsatz des sehr kostengünstigen 8-Bit-Mikrocontrollers ST7. Der Mikrocontroller kombiniert einen Prozessorkern mit verschiedenen IO-Schnittstellen, so dass außer einem Speicherchip für die Software keine weiteren Bauteile benötigt werden.

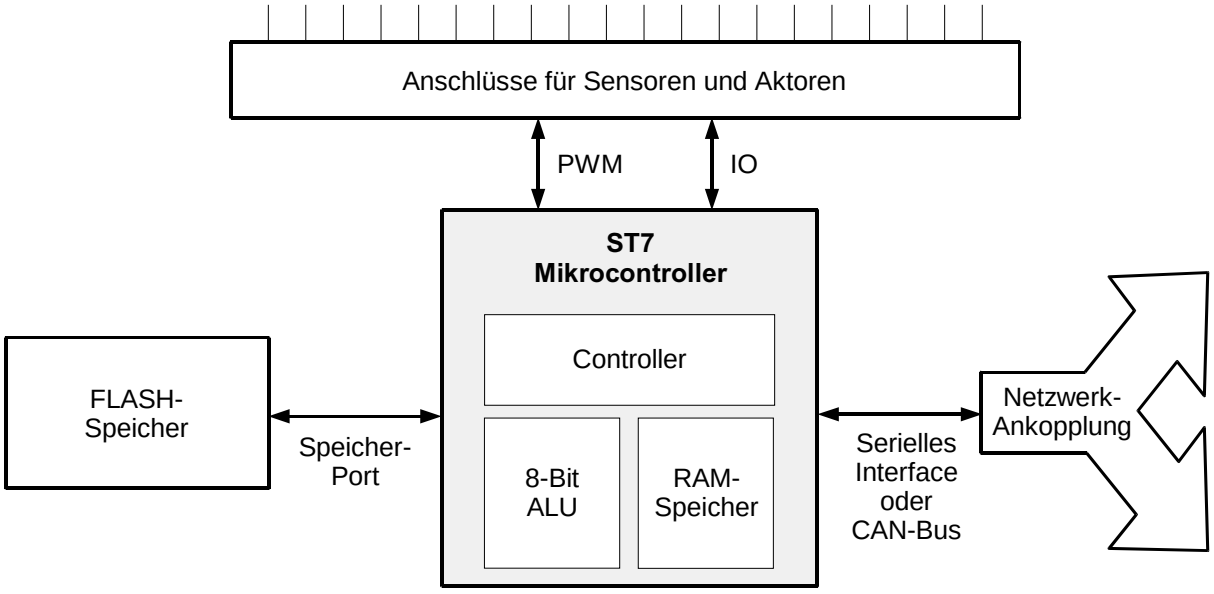


Bild 7.9 Blockschaltbild der JControl-Steuereinheit

Das Blockschaltbild dieses eingebetteten Systems zeigt Bild 7.9. Für die Kommunikation zwischen Mikrocontroller und Speicher kann eine sehr einfache Punkt-zu-Punkt-Schnittstelle verwendet werden, da der Mikrocontroller als einzige Systemkomponente auf den Speicherbaustein zugreift. Auch die Verbindung mit der Außenwelt erfolgt über simple Ein- oder Zweidrahtleitungen. Da die Aufgaben des Systems nicht echtzeit-kritisch sind und auch keine besondere Dienstgüte oder Ausfallsicherheit garantiert werden muss, kann der Systemaufbau mit sehr einfachen Mitteln erfolgen.

7.4.2 Eingebettetes System für eine Digitalkamera

Bild 7.10 zeigt den möglichen Aufbau eines eingebetteten Systems für eine digitale Fotokamera auf Basis der FPGA-Plattform Virtex-II Pro aus dem letzten Kapitel. Die Kamera nimmt kontinuierlich Bilder mit dem digitalen Fotosensor auf und stellt diese auf einem LC-Display dar. Informationen zu Fokus, Blende, Auslösezeit, Aufnahmemodus etc. werden gleichzeitig eingeblendet und kontinuierlich aktualisiert. Drückt der Benutzer auf den Aufnahmeknopf, speichert das eingebettete System das aktuelle Bild als JPEG-Datei auf einer CompactFlash-Karte. Dabei sollte möglichst keine Auslöseverzögerung bemerkbar sein. Schließt der Benutzer die Kamera an einen Computer oder Fernseher an, soll das Live-Kamerabild auch auf diesem zu sehen sein.

Im Gegensatz zur JControl-Steuereinheit aus dem vorherigen Abschnitt haben wir es hier mit Echtzeit-Anforderungen und einer konkurrierenden Nutzung gemeinsamer Ressourcen durch verschiedene Systemkomponenten zu tun.

Gleich mehrere unabhängige Systembausteine (Display, JPEG-Encoder, USB-Schnittstelle, TV-Ausgang) greifen gleichzeitig auf den Speicherbereich zu, der das

aktuelle Kamerabild beinhaltet. Parallel dazu laufen verschiedene Software-Prozesse auf den beiden PowerPCs, die für die Benutzerschnittstelle und die Funktionssteuerung der Digitalkamera zuständig sind.

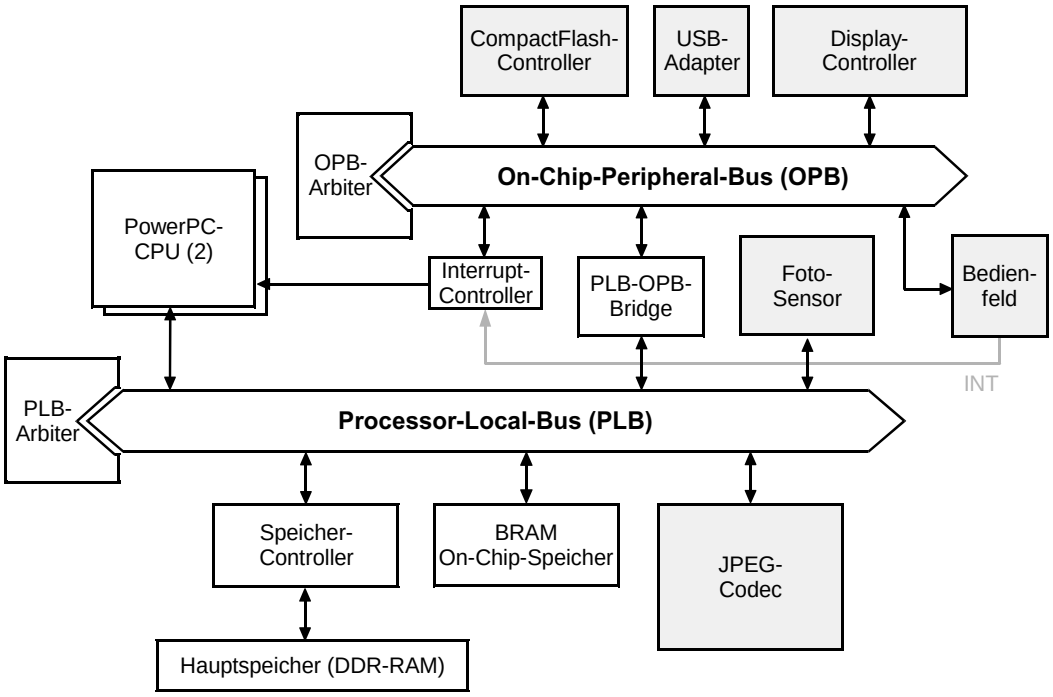


Bild 7.10 Blockschaltbild des Digitalkamera-Systems

An die Stelle der einfachen Punkt-zu-Punkt-Verbindungen des JControl-Systems treten in diesem Beispiel die On-Chip-Busse PLB und OPB (Abschnitt 6.4.5), mit denen ein geregelter und effizienter Zugriff auf die gemeinsam genutzten Systemressourcen möglich wird. Angepasste Busprotokolle garantieren eine schnelle Datenübertragung für Systemfunktionen mit hoher Priorität, wie etwa die Aufnahme und JPEG-Kompression eines Fotos mit anschließendem Speichern auf der CompactFlash-Karte.

Tabelle 7.11 vermittelt einen Überblick zu den verschiedenen Komponenten des Digitalkamera-Systems. Sie lassen sich alle auf der FPGA-Plattform Virtex-II Pro realisieren, die durch ihre CoreConnect-Busarchitektur (Abschnitt 6.4.5) und die beiden PowerPC-Prozessoren sehr gut geeignet ist, einen Prototyp des Systems zu entwickeln. So können umfangreiche Tests lange vor der tatsächlichen Fertigung des endgültigen Produkts durchgeführt werden.

Eine einfach zu handhabende und sehr komfortable Entwicklungsplattform für diese Aufgabe ist das ML310, dem wir uns im folgenden Abschnitt zuwenden und das wir auch im Praktikum einsetzen – allerdings mit nicht ganz so komplizierten Systemen.

Komponente	Bus	Beschreibung
PowerPCs	PLB	Zwei 32-Bit-Prozessorkerne mit je 400 MHz Taktrate
PLB-Arbiter	PLB	Zugriffssteuerung für den Processor-Local-Bus (64 Bit, 100 MHz)
OPB-Arbiter	OPB	Zugriffssteuerung für den On-Chip-Peripheral-Bus (32 Bit, 100 MHz)
Speicher-Controller	PLB	Ermöglicht Lese- und Schreibzugriffe auf den Hauptspeicher mit Double-Data-Rate (DDR). Zugriffe erfolgen im Burst-Modus, d.h. pro Takt werden 64 Bit übertragen, der Verbindungsaufbau (Handshake) benötigt 17 Buszyklen.
BRAM	PLB	Schneller Blockspeicher direkt auf dem FPGA (max. 128 Kbyte pro BRAM-Modul), pro Zugriff werden 32 Bit übertragen, der Verbindungsaufbau (Handshake) benötigt nur 1 Buszyklus.
Interrupt-Controller	OPB	Über den Interrupt-Controller kann der Interrupt-Eingang des PowerPC von mehreren Hardware-Komponenten genutzt werden. Über ein Statusregister kann durch die Software-Interrupt-Behandlungsroutine abgefragt werden, welches Modul den Interrupt ausgelöst hat.
PLB-OPB-Bridge	PLB / OPB	Leitet Daten aus dem OPB an den PLB weiter und umgekehrt. Dabei wird eine konfigurierbare Adresstabelle verwendet, um nur solche Transfers in den jeweils anderen Bus weiterzuleiten, für die dies auch gewünscht ist (vermeidet unnötige „Verschmutzung“ der Busse).
Foto-Sensor	PLB	Holen des aktuellen Bildinhalts über den PLB
JPEG-Codec	PLB	Hardware-beschleunigte Kompression und Dekompression von JPEG-Bildern, arbeitet auf BRAM und Hauptspeicher
Bedienfeld	OPB	Abfrage von Benutzereingaben und Steuerung von Anzeigen (Leuchtdioden, Einblendungen in Sucher)
CompactFlash-Controller	OPB	Lesen und Schreiben von Daten auf eine CompactFlash-Karte
USB-Adapter	OPB	Steuert die Datenübertragung über einen USB-Port (Live-Streaming zu einem angeschlossenen PC sowie Auslesen der Fotos von der CompactFlash-Karte)
Display-Controller	OPB	Bietet einen per OPB beschreibbaren Speicherbereich, dessen Inhalt auf einem LC-Display dargestellt wird.

Tabelle 7.11 Komponenten des Digitalkamera-Systems

7.5 HW-SW-Codesign in der Praxis – die universelle Entwicklungsplattform ML310

In der Theorie macht das Hardware-Software-Codesign einen schlüssigen Eindruck – aber wie sieht es in der Praxis aus? Die Vorstellungen von zeit- und geldsorgenfreien Akademikern und stressgeplagten Systemingenieuren driften hier schnell auseinander. Ist der Zeitdruck („Time-to-Market“) zu hoch, werden kritische Entwurfsentscheidungen dann doch eher „aus dem Bauch“ nach der jeweiligen Erfahrung des Entwurfs-Teams entschieden und nicht auf Basis einer aufwändigen Entwurfsraum-Analyse aus dem Lehrbuch. Das Ergebnis ist in vielen Fällen ein gutes Produkt, aber

mit steigendem Funktionsumfang (HDTV-DVB-Receiver-Digital-Timeshift-Videorekorder) auch immer häufiger ein von unerwünschten Denkpausen und Abstürzen geplagtes Gerät.

Vor diesem Hintergrund sind in den letzten Jahren neue Technologien wie die FPGA-Plattformen aus Abschnitt 6.4 integriert worden in Hardware-Software-Entwicklungsumgebungen, mit denen auch für anspruchsvolle Geräte ein Hardware-Software-Codesign unterstützt wird. Am Beispiel der besonders universellen Entwicklungsplattform ML310 wollen wir uns nun einen Überblick zum aktuellen Stand der Technik verschaffen. Dabei sollten uns die vielen Komponenten und Möglichkeiten nicht abschrecken: es handelt sich um eine möglichst universelle *Entwicklungsumgebung*; praktisch jede konkrete Realisierung benötigt davon nur einen kleinen Teil. Im Endprodukt wird dann natürlich nicht die Entwicklungs-Plattform eingesetzt, sondern eine viel schlankere Hardware, die nur die tatsächlich benötigten Zutaten enthält.

Das Herz der Entwicklungsplattform ist ein FPGA Virtex-II Pro vom Typ V2P30 (Abschnitt 6.4). Das ML310-Board versorgt nicht nur das FPGA mit den nötigen Taktsignalen und Strom, sondern bietet über den Funktionsumfang des FPGA hinaus eine umfangreiche Ausstattung für die Entwicklung moderner eingebetteter Systeme, mit besonderer Unterstützung multimedialer Belange.

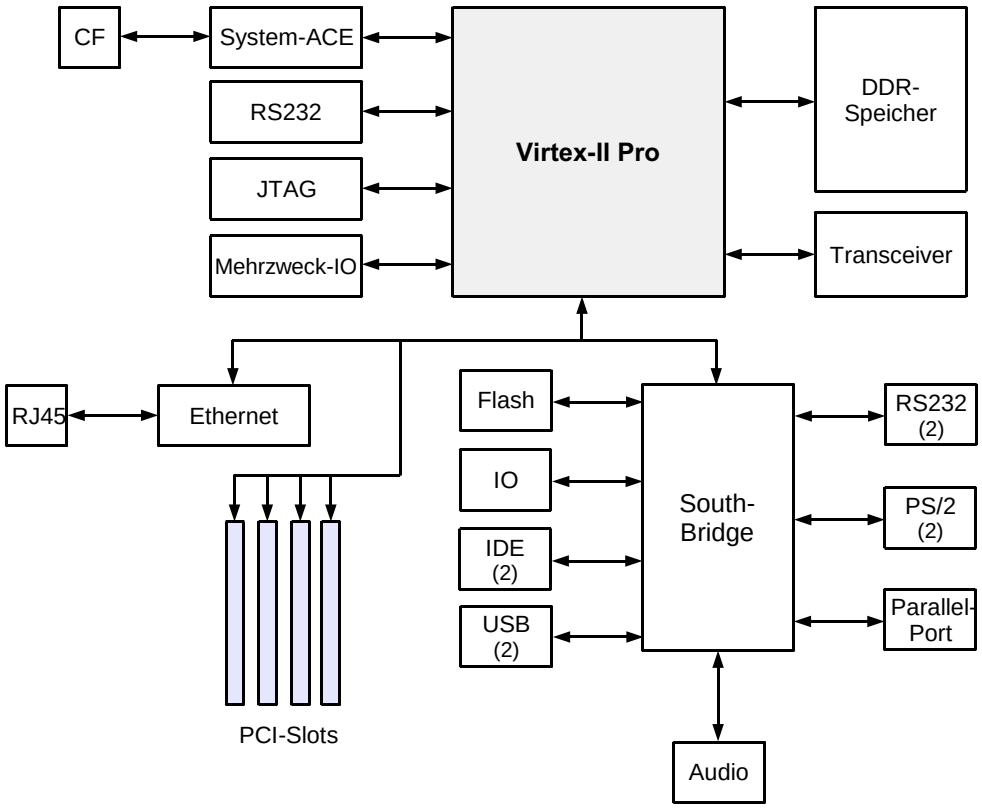


Bild 7.12 Blockdiagramm des ML310

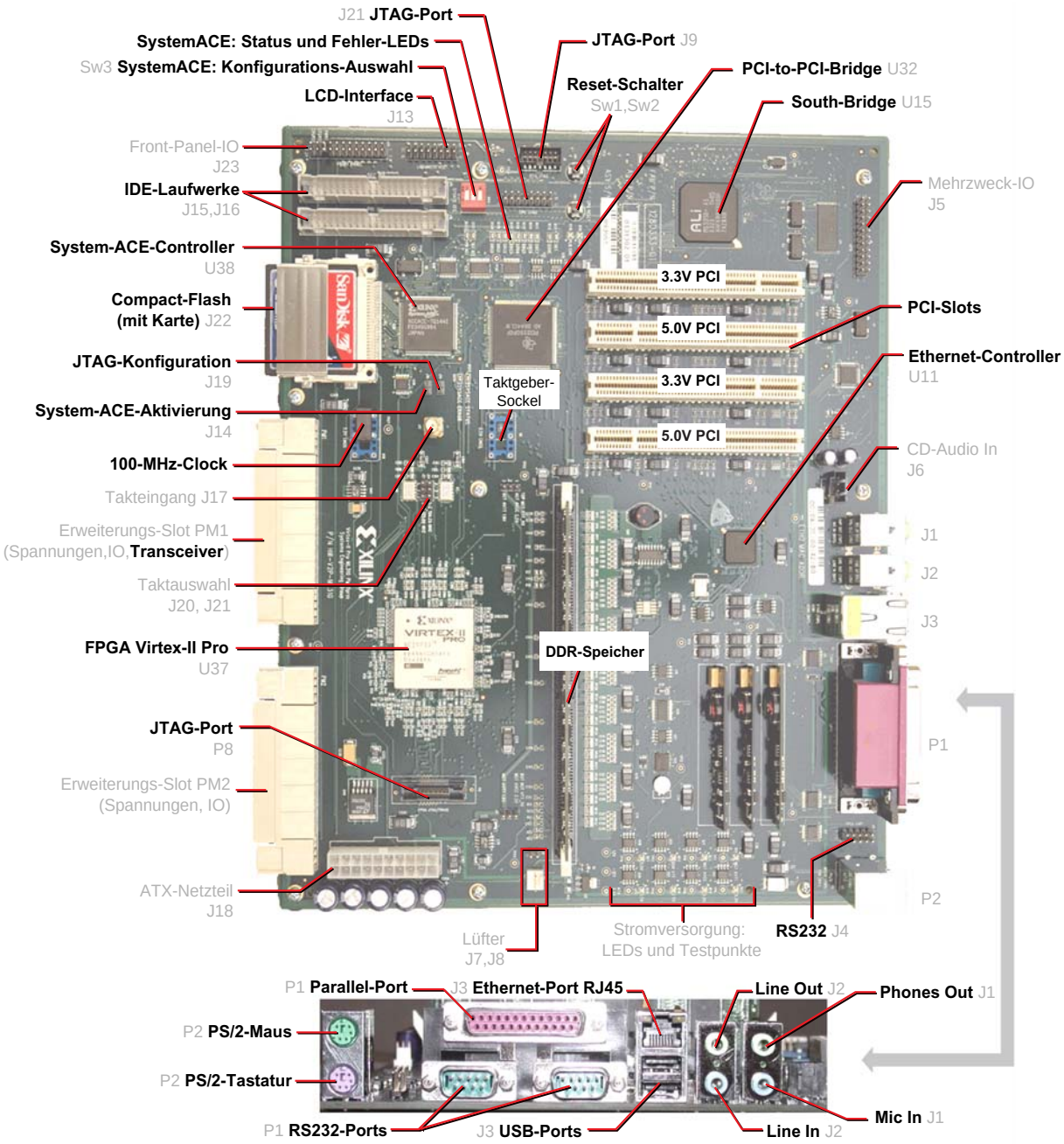


Bild 7.13 Das Prototyping-Board ML310

Bild 7.12 zeigt ein Blockdiagramm mit den wichtigsten Baugruppen, Bild 7.13 gibt einen Überblick zum Aufbau der Platine. Das FPGA ist über seine IO-Pins verbunden mit dem DDR-Speichersockel, der South-Bridge und verschiedenen Peripheriebausteinen. Für die Kommunikation auf der Platine wird der PCI-Bus verwendet, der über vier PCI-Steckplätze herausgeführt ist. Der South-Bridge-Chipsatz bietet zwei serielle RS232-Schnittstellen, PS/2-Anschlüsse, einen Parallelport, zwei USB-Ports, zwei IDE-Adapter für Festplatten, Audio-Eingabe und -Ausgabe sowie weitere IO-Schnittstellen. Ebenfalls über PCI angebunden ist ein

Ethernet-NIC (Network Interface Controller), mit dem 10/100 Mbit-Netzwerke angesteuert werden können.

Mit Hilfe der PCI-Steckplätze können leicht handelsübliche Erweiterungskarten in einen Systementwurf integriert werden, etwa eine Grafikkarte. Für weitere Hardware-Erweiterungen bietet das ML310 acht serielle Hochgeschwindigkeits-Übertrager (Transceiver). Sie erlauben Datenraten bis 3,125 Gbit/s und können für noch höhere Geschwindigkeiten gebündelt werden, womit auch Spezialanwendungen wie Switches für optische Netzwerke realisiert werden können.

Ein interessanter Baustein ist der *System-ACE*-Chip (Advanced Configuration Environment). Da die Konfiguration des Virtex-II Pro auf flüchtigen SRAM-Zellen basiert, liegt direkt nach dem Einschalten keine Konfiguration vor, das FPGA ist also funktionslos. Erst durch einen Bitstrom von außen, etwa über ein USB-Kabel, wird das System mit Leben erfüllt. Beim Einschalten des Systems übernimmt der System-ACE automatisch die Konfiguration mit Daten aus einem nicht-flüchtigen Speicher (Bild 7.14). Dazu ist er mit einem CompactFlash-Slot verbunden, in dem sich eine Speicherkarte mit dem vorgesehenen FPGA-Bitstrom befindet. Der gesamte Konfigurationsvorgang dauert weniger als eine Sekunde, so dass das ML310 kurz nach dem Einschalten betriebsbereit ist. Der Bitstrom kann dabei neben der reinen Hardware-Konfiguration auch Software in den Blockspeicher des FPGA laden.

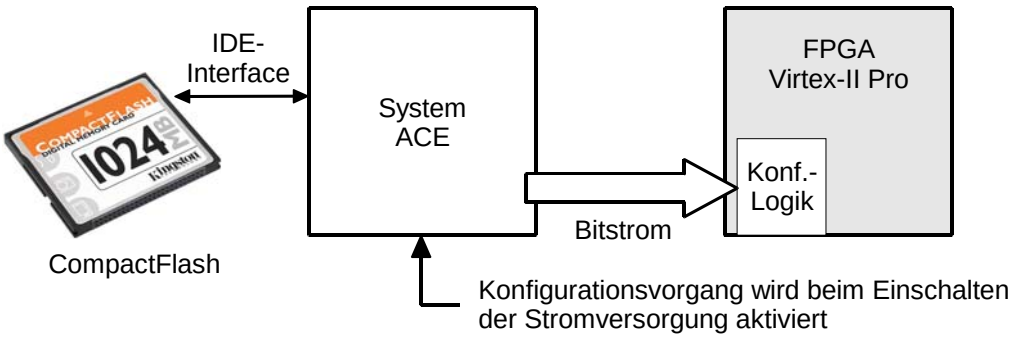


Bild 7.14 FPGA-Konfiguration mit System-ACE

Das ML310 bietet noch eine ganze Reihe weiterer Bausteine, die wir gemeinsam mit den bereits beschriebenen Komponenten in Tabelle 7.15 zusammengefasst haben.

Gruppe	Komponente	Beschreibung
Audio / Video	Audio	Audio-Ein- und Ausgabe (Mic In, CD-Audio In, Line In, Line Out, Phones Out), Stereo, 44,1 KHz; hierzu ist ein AC97-Controller im Southbridge-Chip integriert
	LC-Display	2x16 Zeichen LC-Display (nicht abgebildet)
Speicher	CF	Steckplatz für CompactFlash-Karten zum nicht-flüchtigen Einprogrammieren einer FPGA-Konfiguration
	DDR-Speicher	Steckplatz für DDR-Speichermodule
	Flash	Flash-Speicherbaustein (512 KB) zur freien Verwendung, z.B. nicht-flüchtiges Sichern von Anwendungseinstellungen
Kommunikation und Peripherie	IDE	zwei IDE-Konnektoren zum Anschluss von Festplatten und CD/DVD-Laufwerk
	IIC-Bus	einfacher Kommunikationsbus für den Datenaustausch zwischen verschiedenen ICs auf einer Platine; sowohl das FPGA als auch die Southbridge verfügen über einen IIC-Anschluss (nicht abgebildet)
	Mehrzweck-IO	122 frei verwendbare IO-Leitungen, die direkt vom FPGA an Erweiterungsstecker auf der Platine angeschlossen sind (u.a. LC-Display)
	IO	12 weitere Mehrzweck-IO-Anschlüsse zum Steuern von Status-LEDs auf der Platine, die über den PCI-Bus angesprochen werden können
	Ethernet	Network-Interface-Controller für 10/100 Mbit-Ethernet
	Parallel-Port	Standard-Parallelport zum Anschluss von Drucker etc.
	PCI	Erweiterungs-Slots für den PCI-Bus; zwei Slots mit direkter Verbindung zum FPGA (3,3V) und zwei Slots über PCI-Chip (5V)
	PS/2	PS/2-Ports zum Anschluss von Tastatur und Maus
	RJ45	Standard-Steckverbinder für 10/100 Mbit-Ethernet
	RS232	serieller Anschluss zur Datenübertragung zwischen FPGA und Außenwelt; wird vor allem als Textkonsole in Verbindung mit einem Terminal-Programm verwendet (Grafikkarte/Monitor-Ersatz); es sind ein RS232-Konnektor direkt mit dem FPGA und zwei mit der Southbridge verbunden
	Southbridge	PCI-Chip, der verschiedene IO-Anschlüsse und Peripherie-Controller in einem IC vereint
	Transceiver	acht schnelle serielle Transceiver (3,125 Gbit/s) zur Realisierung von High-Speed-Anwendungen, vor allem aus dem Bereich der digitalen Nachrichtenübertragung (DSL-Provider etc.); durch Kopplung der Transceiver können bis zu 80 Gbit/s erreicht werden
	USB	Universal Serial Bus zum Anschluss verschiedener Ein- und Ausgabegeräte wie Maus, Tastatur, externe Festplatte etc.

Debug / Konfigu-ration	JTAG-Port	Debugschnittstelle für Remote-Debugger und Logik-Analysator; über den JTAG-Port kann auch das FPGA von außen programmiert werden
	Reset	zwei Reset-Schalter für PowerPC-Prozessoren und Gesamtsystem
	System-ACE	Konfigurationsbaustein, erlaubt die FPGA-Programmierung direkt nach dem Einschalten über eine CompactFlash-Karte
	Virtex-II Pro	rekonfigurierbares FPGA mit integrierten PowerPC-Prozessoren als Herz des ML310

Tabelle 7.15 Komponenten des ML310

Bei Prototyping-Boards wie dem ML310 unterscheidet man typischerweise die folgenden zwei Bauformen.

1. *Steckkarten*, etwa für den PCI-Bus eines Entwicklungs-PCs, haben den Vorteil, dass sie vom PC mit Strom versorgt werden und sich Ressourcen mit dem Host-Rechner teilen können, etwa den Hauptspeicher. Die feste Kopplung an den PC ist oft aber auch ein erheblicher Nachteil für eingebettete Anwendungen, da sie nicht so leicht mit weiteren Systemen vernetzt oder auch in ein Produktgehäuse eingebaut werden können, etwa einen Mobiltelefon-Prototyp.
2. Die Alternative eines *Stand-Alone-Boards* ist daher immer häufiger zu finden. Das ML310 stellt hier zwar einen besonders universellen Vertreter dar, ist aber für viele Anwendungen gewissermaßen ein „Overkill“. Das Überangebot an Peripherie und Schnittstellen ist aber gut geeignet, verschiedene Implementierungsvarianten für einen System-Entwurf durchzuspielen, um sich dann für die beste zu entscheiden. Anschließend kann der Prototyp leicht auf ein kleineres Board wie in Bild 7.16 portiert werden, wobei sich das verwendete FPGA selbst nicht ändern muss.

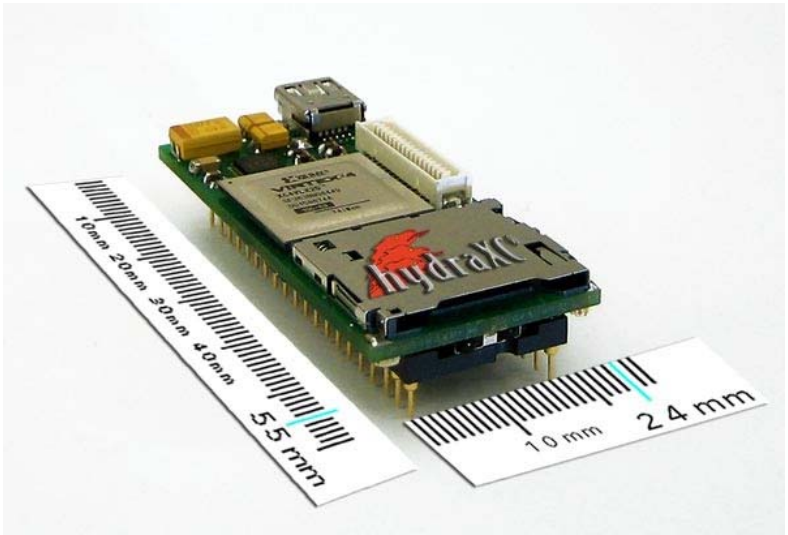


Bild 7.16
Ultra-portables Prototypen-Board hydraXC mit Virtex-4-FPGA

7.5.1 Intellectual Property – Design mit Fertigbausteinen

Beim Hardware-Entwurf mit dem ML310 kann auf eine Vielzahl fertiger Hardware-Bausteine (Intellectual Property, IP, Abschnitt 7.1.3) zurückgegriffen werden. Es handelt sich dabei um synthetisierbare VERILOG- oder Gattermodelle, mit denen durch wenige Mausklicks Standardaufgaben realisiert werden können wie der Zugriff auf die Netzwerkschnittstellen (Ethernet, USB) und den Hauptspeicher (DDR-RAM), die Ansteuerung der PCI-Steckplätze oder die Datenausgabe auf einem LC-Display. Als Ergebnis entsteht eine vollständige FPGA-Konfiguration mit allen notwendigen Bausteinen für den gewünschten Funktionsumfang, die um eigene VERILOG-Module erweitert und dann für das Virtex-II Pro synthetisiert werden kann.

Abhängig von den verwendeten Kommunikations-Schnittstellen und -Protokollen müssen IP-Cores, die nicht direkt für Virtex-II Pro entwickelt wurden, noch an unsere CoreConnect-Kommunikationsarchitektur angepasst werden (Abschnitt 6.4.5). Für viele typische IP-Interfaces wie AMBA oder WishBone stehen hierfür jedoch fertige Adapter, so genannte *Wrapper* zur Verfügung. Einen Eindruck der für das ML310 verfügbaren IP vermittelt Tabelle 7.17. Viele weitere IP-Cores, etwa TFT-Display-Controller oder digitale Signalfilter, sind unabhängig vom Typ des eingesetzten FPGAs und des Prototyping-Boards verfügbar. Sie können vom jeweiligen Hersteller gekauft und für den Einsatz passend konfiguriert werden.

IP-Core	Bus	Parameter (Auswahl)	Slices	LUTs	FFs
BRAM-Controller	PLB	Speichergröße (Anzahl Blockspeicher-Zellen), Burst-Mode	171	234	133
DDR-Synchronous-DRAM-Controller	PLB	32/64-Bit-DDR-RAM, Auto-Refresh, Burst-Mode, Error-Correction-Code (ECC), Anzahl Speicherbänke	1.970	2.261	1.841
Interrupt-Controller	OPB	8/16/32 IRQ-Eingänge, kaskadierbar, flanken-/pegelgesteuert	99	395	342
Serielle Schnittstelle (UART)	OPB	5/6/7/8 Bit pro Zeichen, gerade/ungerade Parität, Stop-Bits, verschiedene BAUD-Raten etc.	442	534	401
Ethernet-Controller	OPB / PLB	10/100/1.000-Mbit-Ethernet, Memory-Mapped oder Direct-Memory-Access (DMA), für Gigabit-Ethernet müssen PLB und DMA verwendet werden	2.570	5.140	2.592
OPB-to-PLB-Bridge	OPB / PLB	32/64-Bit-Busbreiten	449	898	720
PLB-to-OPB-Bridge	OPB / PLB	32/64-Bit-Busbreiten	662	787	620

Tabelle 7.17 Beispiele von Intellectual Property für das ML310

7.5.2 Das ML310 als Entwicklungsumgebung für eingebettete Systeme

Das besondere an FPGA-Prototyping-Boards wie dem ML310 ist, dass wir selbst entwickelte Hardware und Software für eingebettete Systeme „live“ auf dem Board erproben und verbessern können, bis wir schließlich mit unserem Produkt zufrieden sind. Anschließend können wir die entstandene Hardware-Netzliste direkt in die Fertigung geben. Beim Hardware-Software-Codesign mit dem ML310 können sehr unterschiedliche Programmiersprachen und Methoden zum Einsatz kommen, nämlich VERILOG und VHDL für die Beschreibung von Hardware auf RTL-Ebene, C/C++, Java und auch UML dagegen für die Software-Entwicklung.

Zwar kann jede Komponente zunächst separat in der jeweils geeigneten Entwicklungsumgebung implementiert werden (etwa Eclipse für Software und ISE für Hardware), aber um Probleme frühzeitig zu erkennen, die erst beim Zusammenstecken aller Bausteine in der Integrationsphase entstehen, erstellen wir so früh wie möglich ein vollständiges High-Level-Modell des Gesamtsystems. In diesem Modell ist bereits jeder Baustein als Verhaltensmodell („Blackbox“) mit Kommunikationsschnittstelle vorhanden, muss aber noch nicht in seiner endgültigen Form vorliegen. Hierfür eignet sich besonders eine System-Beschreibungssprache wie SystemC, wir können aber auch VERILOG und C++ in friedlicher Co-Existenz verwenden.

Gleichzeitig entwickeln wir einen Testrahmen (Kapitel 2), mit dem die Funktionalität des Gesamtsystems auf allen Entwurfsebenen überprüft werden kann. Anschließend können wir Hardware und Software Baustein für Baustein zu synthetisierbaren Modellen verfeinern und anschließend mit dem Testrahmen verifizieren. Da sich die einzelnen Systemkomponenten dabei auf unterschiedlichen Abstraktionsebenen befinden und darüber hinaus auch in verschiedenen Programmiersprachen vorliegen können, benötigen wir hierzu einen Mixed-Mode-Simulator (Kapitel 2) wie ModelSim. Eine integrierte Hardware-Software-Entwicklungsumgebung wie das *Xilinx-Platform-Studio* (XPS) aus Bild 7.18, das wir auch in den praktischen Übungen einsetzen, unterstützt uns bei diesem Prozess.

Alle Hardware- und Software-Bausteine des Systems werden zusammen mit den eingesetzten On-Chip-Bussen und anderen Kommunikationsverbindungen in einem Projekt verwaltet. Ein hierarchisch gegliederter Modulbaum bietet schnellen Zugriff auf alle Quelltexte; Moduleinstellungen wie Adressbereich, Interface und Kommunikationsprotokoll können in übersichtlichen Dialogen vorgenommen werden. Mögliche Probleme, etwa eine Überlappung von Adressbereichen oder eine nicht verbundene Modulschnittstelle, werden angezeigt und können teilweise automatisch behoben werden.

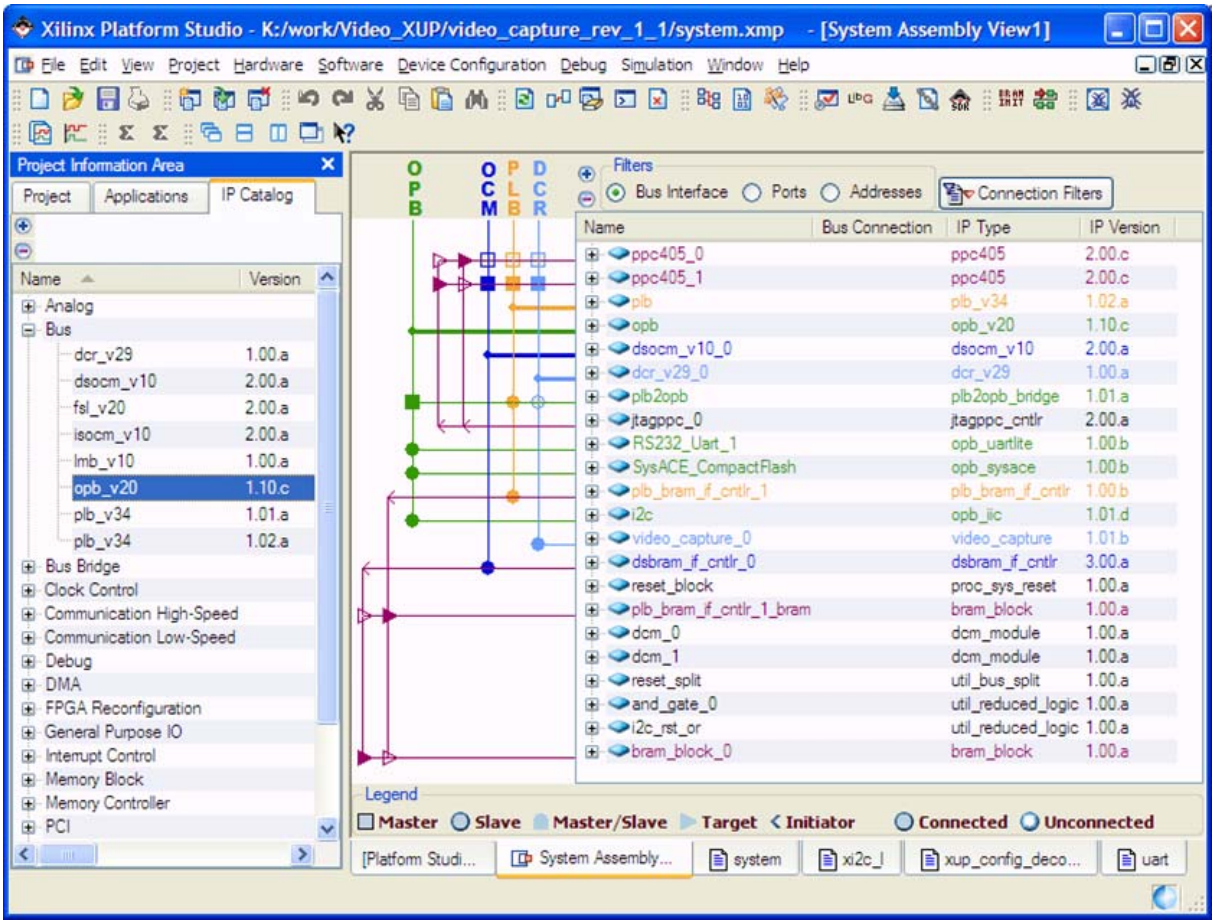


Bild 7.18 Übersichtliche Darstellung eines System-on-Chip im Xilinx-Platform-Studio (XPS)[1]

Da die verwendeten Technologien beim Hardware-Software-Codesign so vielfältig sind – quasi die gesamte Bandbreite der Informatik und Elektrotechnik wird hier berührt – müssen wir trotzdem mit einer ganzen Reihe weiterer Entwurfswerkzeuge jonglieren. Angefangen bei den verschiedenen Programmiersprachen und Compilern für Hardware und Software über Simulatoren, Remote-Debugger und Logik-Analyzer bis hin zur Konfiguration von Prozessorkernen und Echtzeit-Betriebssystem haben wir es mit den unterschiedlichsten Tools zu tun. Die meisten davon können aber direkt aus der integrierten Entwicklungsumgebung heraus aufgerufen werden und sind dann gleich passend zum aktuellen System-Design konfiguriert. Darüber hinaus wurden moderne Entwicklungen aus der Software-Welt wie die Programmierumgebung *Eclipse* ebenfalls bereits für das Hardware-Software-Codesign angepasst und steigern so den Entwurfskomfort.

[1] Dieser und andere Screenshots sind im schwarz-weißen A5-Ausdruck weniger aussagekräftig als in einer vergrößerten Farbansicht unter Acrobat. (Am besten sind sie natürlich live zu erforschen.)

7.5.2.1 Software-Entwicklung mit dem ML310

Während die Hardware-Synthese einen VERILOG-Quelltext auf eine bestimmte Zieltechnologie abbildet, etwa ein Virtex-II-Pro-FPGA, erzeugen Software-Compiler normalerweise Code, der für *dieselbe* Hardware-Plattform bestimmt ist, auf der auch der Quelltext entwickelt wurde.

Beim Übersetzen von Software für eingebettete Systeme ist dies nicht der Fall. Hier kommen eingebettete Prozessoren wie der PowerPC aus Kapitel 6.4.4 und Mikrocontroller wie der ST7 aus Abschnitt 7.4.1 oder Spezial-CPUs mit angepasstem Befehlssatz (ASIP, Abschnitt 7.1.1) zum Einsatz, so dass wir beim Hardware-Software-Codesign einen speziellen *Cross-Compiler* benötigen (Bild 7.19).

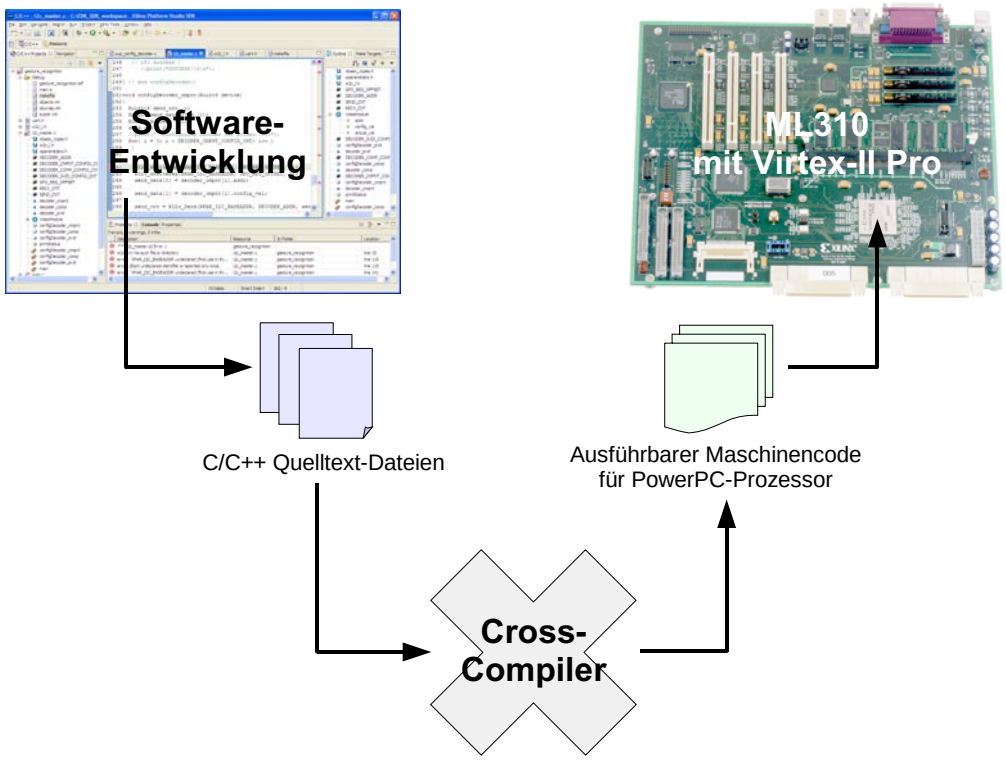


Bild 7.19 Cross-Compiler erzeugt Maschinencode für die Zieltechnologie

Der durch Cross-Compiler erzeugte Maschinencode kann auf dem Entwicklungsrechner selbst nicht ausgeführt werden. Ein direkter Test, wie wir dies bei der Software-Programmierung gewohnt sind, ist dadurch unmöglich, was die Entwicklung fehlerfreier Software nicht gerade erleichtert. Nahe liegend ist natürlich die Verlagerung des Tests vom Entwicklungsrechner auf das Zielsystem, denn dafür haben wir die Software schließlich auch entwickelt. Aber im Gegensatz zu unserem Entwicklungs-PC ist die Zielplattform normalerweise nicht mit Tastatur, Maus und Monitor ausgestattet. Schlimmer noch ist aber, dass wir auf dem eingebetteten System in den meisten Fällen auch kein Standard-Betriebssystem wie Linux oder Windows einsetzen.

Aus diesem Grund sind Prozessoren und Mikrocontroller für eingebettete Systeme mit einem *JTAG-Interface* ausgestattet. Diese von der „Joint Test Action Group“ standardisierte Debug-Schnittstelle wird beim ML310 über einen USB-Anschluss herausgeführt und erlaubt die Fernsteuerung beider PowerPCs mit Hilfe eines *Remote-Debuggers*.

Bild 7.20 zeigt die Oberfläche des von uns verwendeten Remote-Debuggers GDB, der die gerade vom Virtex-II-Pro-FPGA auf dem ML310 abgearbeitete Programmzeile anzeigt. So können wir zum Beispiel die Programmausführung bei Entdecken eines Fehlers anhalten, den Quelltext korrigieren, die überarbeitete Stelle neu übersetzen und die Ausführung anschließend mit dem reparierten Code fortsetzen.

Die Arbeit mit dem Remote-Debugger werden wir auch in den praktischen Übungen trainieren. Das Ausführen von Programmcode auf einem ferngesteuerten ML310-Board ist zwar zunächst etwas gewöhnungsbedürftig, hat aber besonders bei der Entwicklung von hardware-nahen Gerätetreibern den großen Vorteil, dass sich ein Programmabsturz nicht auch auf die Stabilität des Entwicklungs-PCs auswirkt.

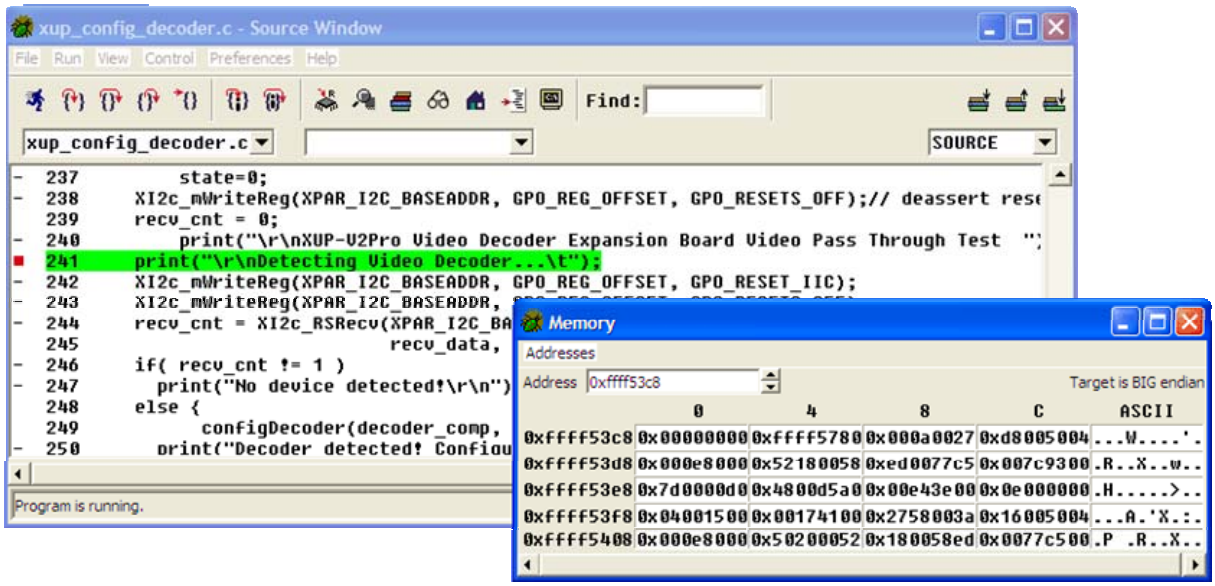


Bild 7.20 Remote-Debugger zur Fehlersuche und Fernsteuerung des PowerPC405

7.5.2.2 Hardware-Entwicklung mit dem ML310

Die Erprobung und Fehlersuche bei Hardware-Blöcken erfolgt üblicherweise mit einem Simulator, in den wir zunächst unseren VERILOG-Quelltext auf Register-Transfer-Ebene eingeben. Die Waveform-Darstellung einer VERILOG-Simulation zeigt Bild 7.21, in diesem Fall handelt es sich um eine Addierer-Schaltung aus den praktischen Übungen. Erscheint uns das Design fehlerfrei, können wir anschließend eine Hardware-Synthese durchführen und das Ergebnis – eine auf dem FPGA

platzierte und verdrahtete Schaltung – auf Übereinstimmung mit dem VERILOG-Modell überprüfen.

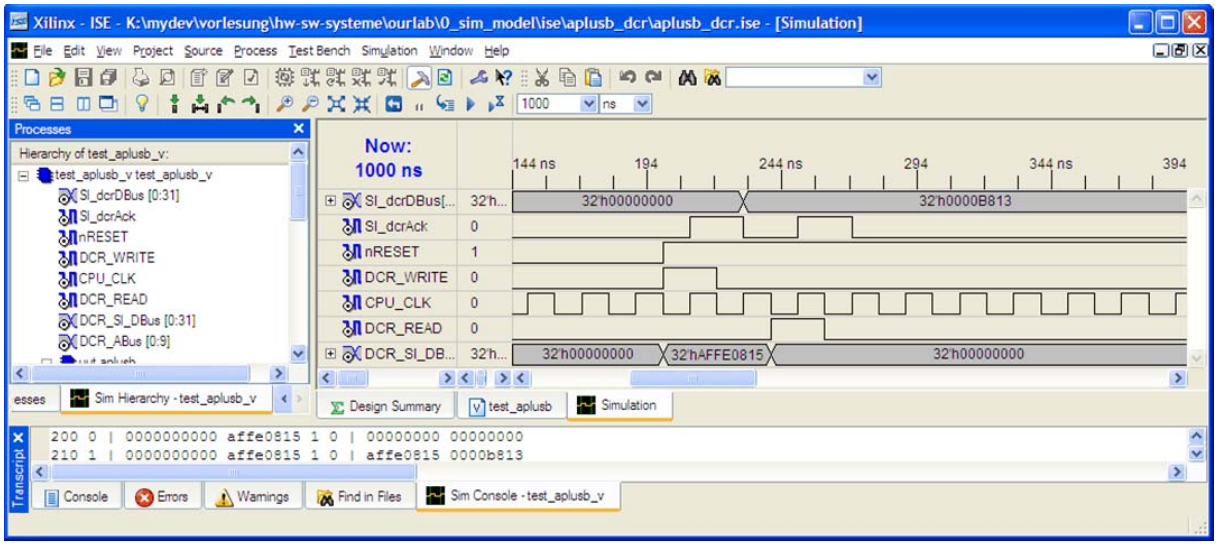


Bild 7.21 Simulation eines VERILOG-Modells

Problematisch bei dieser Entwurfsmethode ist die Simulationsdauer. Für einzelne Systembausteine mit einigen Tausend Gattern noch im Minutenbereich, steigt sie für das vollständig zusammengesetzte System-on-Chip mit mehreren Millionen Gattern schnell auf mehrere Stunden. Kommt noch die Verifikation der eingebetteten Software mit Hilfe eines Prozessor-Simulators (Instruction-Set-Simulator, ISS) hinzu, kann die Simulation sogar einige Tage dauern.

Eine einfachere und kostengünstigere Lösung ist die Co-Verifikation des Hardware-Software-Gesamtsystems direkt auf dem ML310. Komfortable Debug-Werkzeuge für die Software-Verifikation haben wir bereits kennen gelernt (Bild 7.19), aber wie können wir unsere Hardware effizient auf Fehlverhalten überprüfen? Wir möchten beispielsweise überprüfen, ob bestimmte Signalverläufe innerhalb des Chips unseren Vorstellungen entsprechen, etwa ob sich eine State-Machine im gewünschten Zustand befindet oder ob beim Zugriff auf den On-Chip-Bus das Kommunikations-protokoll korrekt eingehalten wird. Vor allem aber möchten wir Fehler erkennen, die durch das komplexe Zusammenspiel der vielen vernetzten System-on-Chip-Komponenten entstehen und die wir in den Einzelsimulationen der Systembausteine durch unsere Testmuster nicht abdecken können.

Für diese Aufgabe können wir auf FPGAs einen *integrierten Logik-Analysator* einsetzen (ILA, Bild 7.22). Der ILA bietet eine konfigurierbare Anzahl Ports und wird als zusätzliches Hardware-Modul in unser System-on-Chip-Design eingebaut. Bei Xilinx-FPGAs kann jeder ILA-Port mit 256 Signalen verbunden werden, so dass bei maximal 16 Ports bis zu 4.096 Signale überwacht werden können. Der ILA verwendet BRAM-Blöcke auf dem FPGA, um die Signalverläufe an seinen Ports aufzuzeichnen.

Die Größe des verwendeten Speicherbereichs bestimmt die maximale Aufzeichnungsdauer.

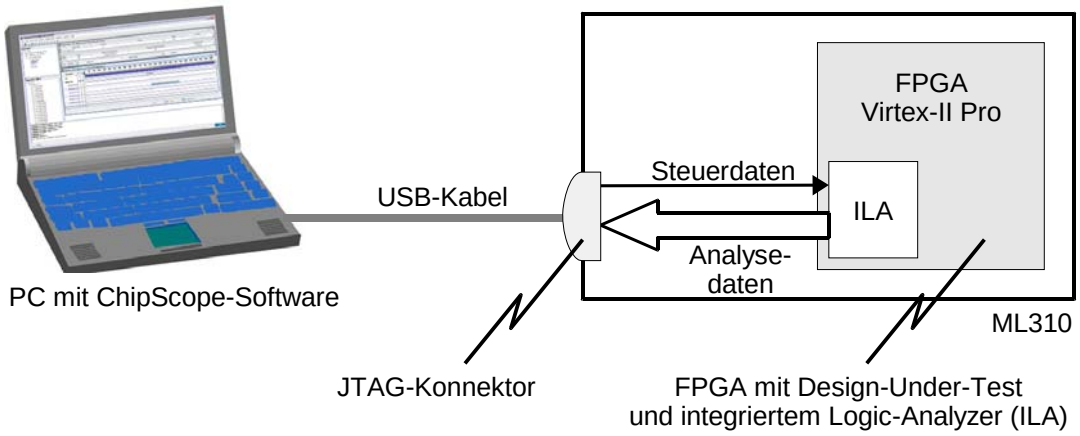


Bild 7.22 Hardware-Verifikation mit integriertem Logik-Analysator (ILA)

Sich den gesamten Datenverkehr auf den überwachten Signalen anzusehen ist wenig sinnvoll, da die wirklich interessanten Stellen dann nur schwer im Datenwust zu finden sind. Typischerweise *triggern* wir deshalb auf eine Signalflanke, wobei meist mehrere Signale auf einmal betrachtet werden, um z.B. einen bestimmten Zustand einer State-Machine zu erkennen. So wäre es etwa im Addierer-Beispiel aus Bild 7.21 sinnvoll, auf die positive Flanke des Signals DCR_WRITE zu triggern, um jeden Schreibzugriff auf den Addierer und den nachfolgenden Signalverlauf während der Addition zu beobachten. Weniger hilfreich dagegen wäre das Triggern auf das Signal CPU_CLK, denn dieses ändert sich in jedem Taktzyklus, unabhängig vom Zustand des Addierers.

Durch Verketten von Triggerbedingungen können schließlich auch ganze Datensequenzen beschrieben werden, so dass der Logik-Analysator nur bei Auftreten einer ganz bestimmten Zustands*reihenfolge* auslöst.

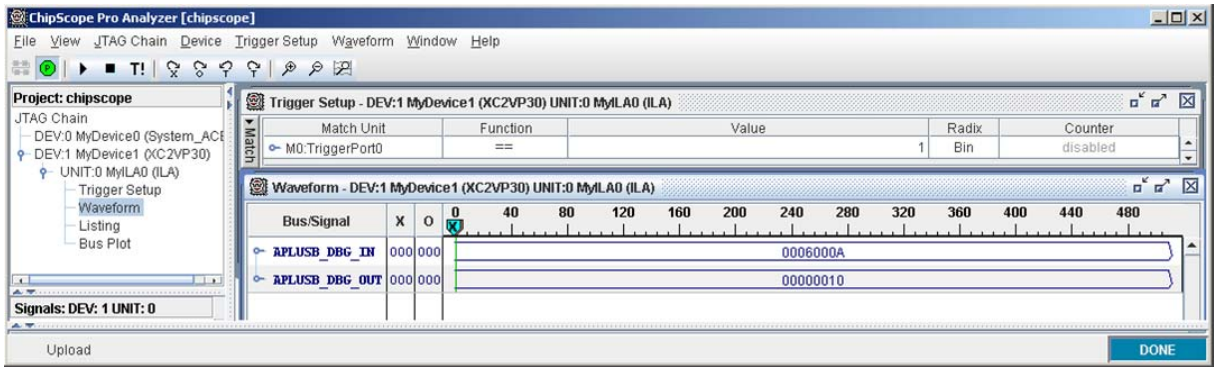


Bild 7.23 Verifikation eines Hardware-Addieres mit integriertem Logik-Analysator (ILA)

Die vom ILA aufgezeichneten Signalverläufe werden über die JTAG-Schnittstelle des ML310 ausgegeben und können auf dem Entwicklungs-PC grafisch dargestellt werden. Bild 7.23 zeigt die Logikanalyse des Hardware-Addierers, den wir in Bild 7.21 simuliert hatten. Hier arbeitet der Addierer nun auf dem Virtex-II-Pro-FPGA unserer ML310-Entwicklungsplattform. Es ist erkennbar, dass die Eingabewerte 0006 und 000A korrekt zum Ergebnis 00000010 summiert werden.

Das Einfügen eines ILA in unser Design können wir leicht in VERILOG vornehmen. Dazu genügt es, das in Bild 7.24 gezeigte VERILOG-Modul zu instanzieren und mit den zu analysierenden Signalen zu verbinden.

```
module ila_example (
input  wire         CLK,     // Taktsignal
input  wire [31:0] DATA,     // Eingänge für zu überwachende Signale (hier 32)
input  wire [ 7:0] TRIGGER, // Trigger-Eingänge (hier 8)
output wire [35:0] CONTROL  // Steuerleitung zum Speichern und Weiterleiten
                            // der aufgezeichneten Daten an den PC
);

  // ILA-Core instanzieren und verbinden
  ila Ila (CONTROL, CLK, DATA, TRIGGER[0]);

endmodule
```

Bild 7.24 Einfügen eines integrierten Logik-Analysators mit VERILOG

7.5.3 Schnittstelle zwischen Hardware und Software

Die Schnittstelle zwischen Hardware und Software spielt beim HW-SW-Codesign eine zentrale Rolle (Bild 7.3). Im Vergleich zu Standard-PCs sind eingebettete Systeme typischerweise mit vielfältigeren und oft exotischen Peripherie-Schnittstellen ausgestattet, da sie als Steuergeräte und Sensorsysteme in ganz anderen Umgebungen als der PC unter dem Schreibtisch eingesetzt werden. Bussysteme wie CAN, LIN und FlexRay für die Kommunikation von Steuergeräten im Automobil sind nur ein Beispiel. Weitere Beispiele sind die Ankopplung von Sensoren, Aktoren und Bedien-Panels über Spezialschnittstellen (etwa in der Medizintechnik oder der Robotersteuerung) oder die Anbindung eines bildgebenden CCD-Chips in Digitalkameras. FPGAs wie der Virtex-II Pro sind sehr gut geeignet, solche Spezialschnittstellen mit weniger (Kosten-)Aufwand anzusteuern als ein Spezialbaustein. Die notwendigen Controller entwickeln wir dazu in VERILOG und SystemC.

Die Hardware-Anbindung ist damit realisiert, aber wie können die verschiedenen Netzwerke, Sensoren und Aktoren nun mit unserer Software verwendet werden? Der Hardware-Software-Schnittstellen-Entwurf beschäftigt sich mit dieser Frage. Die Schwierigkeit liegt vor allem darin, die komplexen Hardware-Eigenschaften („hell of physics“) elegant in die abstrakte Welt der Software („heaven of software“) abzubilden. Hierfür kommen typischerweise zwei Verfahren in Frage: Memory-Mapping und Interrupts. Weiter unterscheidet man zwischen direkten und Shared-Memory-Datentransfers sowie dem Direct-Memory-Access (DMA). Solche Fragen

wollen wir im Folgenden genauer untersuchen. Generell spielen bei der Kopplung die Datenrichtung eine Rolle (von der Software zur Hardware oder umgekehrt) und der Initiator des Datentransfers (Software oder Hardware).

7.5.3.1 Memory-Mapping

Memory-Mapping (oder auch *Address-Mapping*) ist eine einfache und gebräuchliche Schnittstelle zwischen Hardware und Software. Sie beruht auf der Erkenntnis, dass jedes Software-Programm (oder zumindest das Betriebssystem) auf beliebige Bereiche im Hauptspeicher zugreifen kann, indem es einfach Variablen schreibt oder liest, die auf den gewünschten Speicherbereich verweisen. Um nun den Zugriff auf FPGA-Register zu ermöglichen, müssen diese Register lediglich in den Speicherbereich des Prozessors eingeblendet werden – es wird quasi eine Speichererweiterung vorgegaukelt. Die Software kann anschließend auf einen erweiterten Adressbereich zugreifen, wobei sie entweder tatsächlich mit dem Hauptspeicher kommuniziert, oder aber mit einer Hardware-Komponente auf dem FPGA.

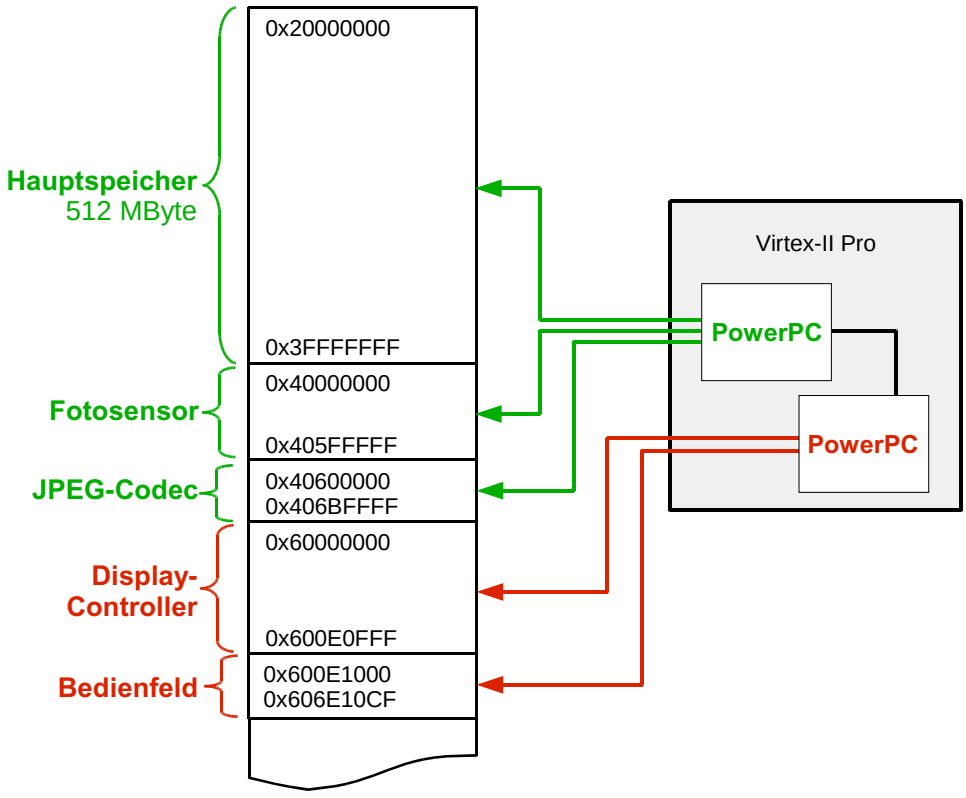


Bild 7.25 Memory-Mapping

Während einfache Prozessoren wie der ST7-Mikrocontroller aus Abschnitt 7.4.1 lediglich über ein simples Speicher-Interface verfügen, besitzen leistungsfähigere Bausteine mehrere Kommunikationsschnittstellen. Der PowerPC des Virtex-II Pro zum Beispiel verfügt sowohl über ein Bus-Interface für den Processor-Local-Bus

(PLB) als auch über eine schnelle BRAM-Schnittstelle. Eine Adresstabelle, mit der der Prozessor beim Booten konfiguriert wird, bestimmt, welches Interface für welchen Adressbereich verwendet werden soll. Anschließend werden Speicherzugriffe anhand ihrer Zieladresse ausgewertet und über die richtige Schnittstelle abgewickelt.

Bild 7.25 zeigt das Memory-Mapping für ein typisches System-on-Chip, es handelt sich in diesem Fall um die Digitalkamera aus Abschnitt 7.4.2. Beide Prozessoren des Virtex-II Pro sind im Einsatz. Während die Software auf dem ersten Prozessor Bilder vom Fotosensor entgegen nimmt und mit Hilfe des JPEG-Codecs komprimiert im Hauptspeicher ablegt, steuert die Software auf dem zweiten Prozessor den Display-Controller und das Bedienfeld der Digitalkamera an. Das Einschalten einer Leuchtdiode des Kamera-Bedienfelds beispielsweise kann durch das Memory-Mapping mit dem einfachen Quelltext aus Bild 7.26 erfolgen.

```
void setLED(bool on) {
  volatile unsigned int *led = 0x600E1000;
  *led = on;
}
```

Bild 7.26 Einfacher Hardware-Zugriff durch Memory-Mapping

Hier wird lediglich ein Pointer auf den Speicherbereich der Bedienfeld-Hardware angelegt, auf den wie auf eine normale Variable geschrieben werden kann. Der Schreibzugriff führt dazu, dass tatsächlich Daten vom Prozessor an das Bedienfeld-Hardware-Modul gesendet werden. Wichtig ist hierbei, dass der Wert der Variablen `led` nicht im Hauptspeicher abgelegt wird!

Das Vorgehen beim Memory-Mapping ist nun deutlich geworden. Wer initiiert aber den Datentransfer? Bei obigem Beispiel ist dies eindeutig der Prozessor. Da sich das angesprochene Hardware-Modul hier wie ein normaler Speicher verhält, sprechen wir vom Memory-Mapped-Zugriff.

Ein Nachteil kann bei größeren Datenmengen sein, dass der Prozessor die gesamte Zeit mit der Datenübertragung beschäftigt ist. Um dies zu vermeiden, können wir stattdessen der Hardware signalisieren, dass für sie ein größeres Datenpaket im Hauptspeicher bereit steht, etwa Bilddaten für die Kompression in das JPEG-Format. Das JPEG-Hardware-Modul kann nun *selbständig* den entsprechenden Speicherbereich auslesen, die JPEG-Komprimierung vornehmen und das Ergebnis wieder in den Speicher zurück schreiben. Der gesamte Vorgang geschieht ohne weiteres Zutun des Prozessors, der nebenbei andere Dinge erledigen kann. Da die Hardware hier als Initiator oder *Master* der Kommunikation auftritt, nennen wir diese Kommunikationsvariante *Master-Mode*.

Schließlich wollen wir noch eine gebräuchliche Variante des Master-Mode zumindest erwähnen, den so genannten *Direct-Memory-Access* oder DMA. Ein spezielles Hardware-Modul, der DMA-Controller, steuert hier die Kommunikation

zwischen Hardware-Komponente (z.B. JPEG-Encoder) und Hauptspeicher. Dieses Verfahren bietet sich für Architekturen an, bei denen keine On-Chip-Busse wie CoreConnect verwendet werden, sondern lediglich einfache Speicherschnittstellen. Bei System-on-Chips wird es eher selten eingesetzt.

7.5.3.2 Interrupts

Während wir im vorigen Abschnitt untersucht haben, wie Daten zwischen Hardware und Software übertragen werden können, haben wir noch nicht geklärt, wie wir die Software auf interessante Ereignisse aufmerksam machen können, etwa die Betätigung des Auslösers an unserer Digitalkamera.

Hardware-Module auf dem FPGA können dazu einen *Interrupt* auslösen. Dieser wird an einem speziellen Interrupt-Eingang des Prozessors signalisiert und sorgt für eine sofortige Unterbrechung der laufenden Programmbearbeitung. Der Prozessor speichert den aktuellen Zustand seiner Pipeline für die spätere Fortsetzung der Programmbearbeitung in so genannten *Schattenregistern* und führt anschließend eine spezielle Interrupt-Behandlungsroutine aus. Diese können wir mit unserer eigenen Software-Routine überschreiben, so dass wir beispielsweise per Memory-Mapped-Zugriff abfragen können, was denn die Unterbrechung ausgelöst hat. Anschließend werden die Werte aus den Schattenregistern wieder in die Prozessor-Pipeline zurück kopiert und die normale Programmbearbeitung fortgesetzt.

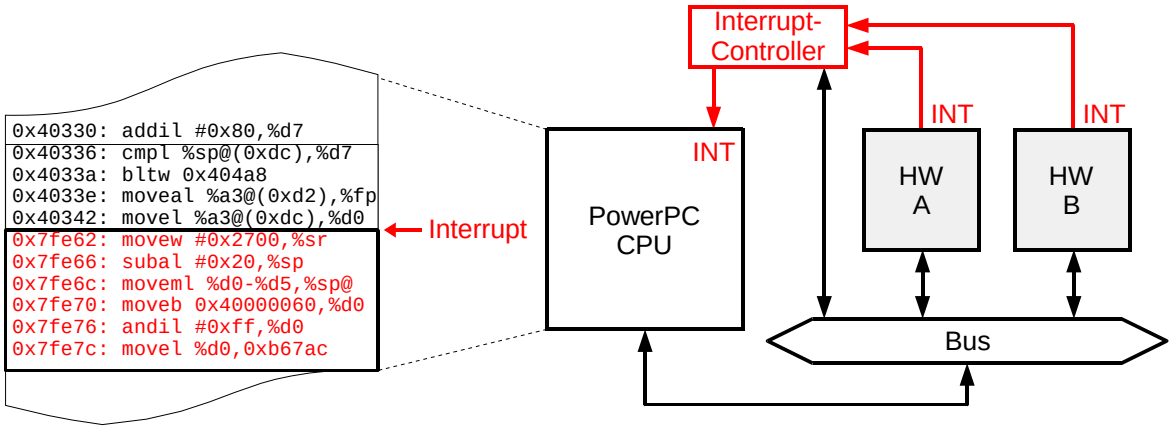


Bild 7.27 Ein Hardware-Modul löst einen Interrupt aus

Bild 7.27 deutet einen Interrupt an, den ein am On-Chip-Peripheral-Bus angeschlossenes Hardware-Modul unserer Digitalkamera signalisiert. Der Prozessor wechselt daraufhin vom aktuellen Programmkontext in die Interrupt-Behandlungsroutine, führt dort die notwendigen Aktionen aus (z.B. Digitalfoto aufnehmen) und setzt anschließend das unterbrochene Programm fort.

Bei den PowerPC-Prozessoren des Virtex-II Pro stehen ein kritischer und ein unkritischer Interrupt-Eingang zur Verfügung, die mit unterschiedlichen Prioritäten

behandelt werden. Durch einen Interrupt-Controller kann die Anzahl verfügbarer Interrupts vergrößert werden.

7.6 Ausblick

„Den eingebetteten Systemen gehört die Zukunft!“ ist die typische Antwort, wenn man Marktanalysten nach dem *Quo-Vadis* der Computerindustrie fragt. Hardware-Software-Codesign ist sicher eine Schlüsseltechnologie auf diesem Gebiet. Viele Schwierigkeiten beim Entwurf eingebetteter Systeme können mit Hilfe der vorgestellten Entwurfswerkzeuge bereits leichter bewältigt werden als noch vor wenigen Jahren. Aber das grundsätzliche Problem ist die entwurfsmethodisch bedingte Kluft zwischen Hardware und Software, die durch verschiedene Programmiersprachen, andere Vorgehensweisen und unterschiedliche Tools entsteht, vor allem aber auch durch die große Vielfalt von realer Hardware.

Zwar gibt es mehr und mehr Konzepte, mit denen Hardware und Software besser verschmolzen werden, die jährlichen Erfahrungsberichte der Industrie zeigen aber, dass die meisten Probleme immer noch bei der Integration von Hard- und Software für ein neues Produkt entstehen. Viele Entwickler sehnen sich daher nach einer Möglichkeit, moderne Software-Techniken wie UML und objektorientierte Programmierung für Hardware und Software gemeinsam verwenden zu können.

Intensiv erforscht werden deshalb *System-Beschreibungssprachen*, besonders das aus C++ hervorgegangene *SystemC*, mit dem das gesamte eingebettete System „aus einem Guss“ erstellt werden kann. Dabei werden zunächst alle Systemkomponenten in SystemC modelliert, unabhängig davon, ob es sich später um Hardware oder Software handeln wird. Bereits vorhandene Bausteine in UML, C++, Java, VERILOG, VHDL oder als Netzliste können direkt in das SystemC-Projekt integriert werden, dabei werden abstrakte und hardware-nahe Beschreibungen beliebig gemischt. Aus dem SystemC-Modell wird anschließend ein Simulationsmodell erstellt, mit dem das gesamte Hardware-Software-System ausgiebig getestet werden kann.

Bisher ist es noch nicht möglich, solche SystemC-Modelle direkt in Hardware und Software für die Zielarchitektur zu übersetzen. Aber Teillösungen existieren bereits, z.B. ermöglicht unsere FPGA-Middleware TRAIN die automatische Generierung der Hardware-Software-Schnittstelle aus High-Level-Beschreibungen in SystemC, so dass wir das Ergebnis sofort auf dem ML310 testen können. Andere Forschungsprojekte beschäftigen sich mit der Generierung von Software für eingebettete Systeme aus SystemC, wobei Berechnungen in Echtzeit eine wichtige Rolle spielen. Erste Werkzeuge zur Hardware-Synthese sind ebenfalls verfügbar.

Ein weniger bekanntes Beispiel für Hardware-Software-Codesign sind die *adaptiven Rechner*. Bei adaptiven Rechnern werden FPGAs verwendet, um häufig wiederkehrende Programmsequenzen in Hardware auszulagern. Bei rechenintensiven Aufgaben, etwa in der Biotechnologie (DNA-Analyse), kann die Gesamtlaufzeit so

von Tagen auf Stunden zu reduziert werden. Im Gegensatz zum eingebetteten System wird beim adaptiven Rechner die FGPA-Konfiguration häufig geändert, spätestens beim Ausführen eines neuen Programms. Hierzu werden spezielle Compiler entwickelt (z.B. COMRADE an der Abteilung E.I.S. in Zusammenarbeit mit der TU Darmstadt), die rechenintensive Programmsequenzen automatisch identifizieren und in VERILOG-Code umwandeln.

Wie wir gesehen haben, ist das ML310 ein äußerst vielseitiges Board, das beim Entwurf moderner eingebetteter Multimedia-Systeme und in der Forschung zu adaptiven Rechnern eingesetzt wird. In den Praktika der folgenden Semester sehen wir uns diese Entwicklungsplattform genauer an und nutzen die hier vorgestellten Entwurfswerkzeuge, um selbst ein Hardware-Software-Codesign von der Idee bis zum fertigen System-on-Chip durchzuführen. Dabei werden Sie sicher auch mit dem einen oder anderen „Bug“ zu kämpfen haben, der mit Hilfe der komfortablen Debug-Möglichkeiten aber meist schnell behoben werden kann.

Angesichts der rasant steigenden Komplexität eingebetteter Systeme – man denke nur an Visionen wie selbststeuernde Automobile oder Brillen mit eingeblendeten Umgebungsinformationen – werden zukünftige Prototyping-Boards aber vermutlich noch wesentlich ausgereiftere Analyse-Möglichkeiten mitbringen – der Fortschritt kennt schließlich kein Ende…

Hardware-Software-Systeme

Teil 2: Übungen

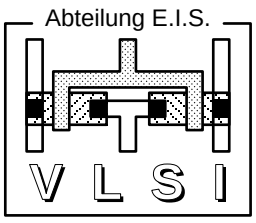
Ulrich Golze

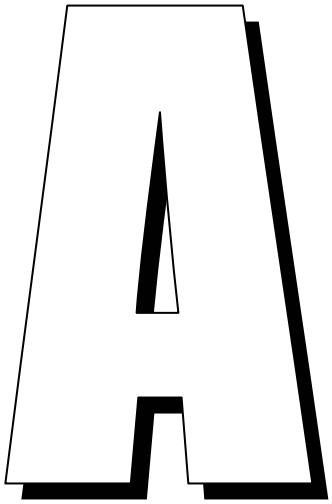
Technische Universität Braunschweig

Abteilung Entwurf integrierter

Schaltungen (E.I.S.)

Oktober 2010





Die drei Labs zu den HW-SW-Systemen

Herzlich willkommen zu unseren drei Labs[1] zur Veranstaltung Hardware-Software-Systeme!

In diesen drei Labs üben Sie live am Rechner die Hardware-Beschreibungssprache VERILOG (Kapitel B) und staunen, wie einfach Sie VERILOG-Modelle mit einer automatischen Logiksynthese in Gattermodelle übersetzen und damit fast schon eigene Chips „bauen“ können (Kapitel C). Das dritte Lab verdient seinen Namen wirklich, denn jetzt tauchen Sie ein in eine industrielle Profi-Umgebung und bauen dort reale Hardware, die mit Ihrer eigenen Software kommuniziert (Kapitel D). Mit Kapitel E steht Ihnen zusätzlich ein Nachschlagewerk zur Verfügung, wo die wichtigsten Schritte kurz zusammengefasst sind.

Im Einzelnen beginnen Sie mit leichten Aufwärmübungen wie dem Anlegen von VERILOG-Projekten und der Durchführung einfacher Hardware-Simulationen. Dabei machen Sie sich spielend mit den Grundlagen der Hardware-Programmierung bekannt und verwenden aktuelle CAD-Tools wie Xilinx ISE und ModelSim, um aus Ihrem VERILOG-Quelltext „echte“ Hardware zu synthetisieren. Beide Programme sind im kommerziellen Chip- und System-Entwurf weit verbreitet.

Anschließend trainieren Sie weitere Fertigkeiten wie den Umgang mit Testrahmen und grafischen Waveforms und gewinnen ganz nebenbei Vertrauen zu den wichtigsten VERILOG-Begriffen in Form praktischer Beispiele. Am Ende des zweiten Labs, also nach etwa vier Wochen, synthetisieren Sie bereits eigene Logikschaltungen für das FPGA Xilinx Virtex-II Pro (Kapitel 6). Auch das nötige FPGA-Wissen erwerben Sie gleichzeitig in der Vorlesung.

Im dritten Lab schließlich kombinieren Sie Ihre Hardware mit passender Software und erwecken sie auf dem Prototypen-Board ML310 als komplett selbst entwickeltes Hardware-Software-System zum Leben! Das ML310 ist eine moderne Entwicklungs-

[1] Sprich: Lääbs, im deutschen Sprachraum auch Labore genannt; noch konservativer: praktische Übungen.

umgebung, die in der Industrie beim Design typischer Hardware-Software-Systeme eingesetzt wird, z.B. beim Entwurf digitaler Videokameras. Es steht Ihnen für Ihre Experimente zur Verfügung und wird in Kapitel 7 der Vorlesung ausführlich diskutiert.

In der zweiten Semesterhälfte nehmen Sie sich dann Zeit, fortgeschrittene Tools wie Remote-Debugger und ChipScope-Pro zu erforschen, mit denen Sie die Zusammenarbeit von Hardware und Software beobachten und optimieren werden.

Schließlich befindet sich am Ende jedes Labs ein Quiz aus mehreren Verständnisfragen. Hiermit können Sie sich selbst testen und überprüfen, ob sie alles verstanden haben und wichtige Feinheiten selbst formulieren können.

Wir hoffen, Ihnen mit den folgenden Übungen einen spannenden und abwechslungsreichen Einstieg in das praktische Hardware-Software-Codesign zu bieten, das bereits heute und in Zukunft mehr denn je eine Hauptrolle in allen möglichen Industriezweigen spielen wird. (Nebenbei gefragt, wie viel Geld haben Sie bereits für Handies, MP3-Player und Digitalkameras ausgegeben?) Und natürlich finden Sie mit solch praxisnahen Erfahrungen anschließend Ihren Traumjob.

Soweit die optimistische Motivation. Aber ganz ehrlich, die Labs haben auch Ihre Schattenseiten. Gerade weil wir mit dem ML310 eine große Industrie-Plattform einsetzen, sind die zugehörigen Design-Tools auch *sehr* mächtig und komplex. Möchten Sie lieber mit 1.000 Seiten Handbüchern allein gelassen werden? Natürlich nicht. Deswegen haben wir liebevoll jeden einzelnen Schritt für Sie aufgeschrieben mit der Folge, dass Sie das Gefühl haben könnten, am Hundehalsband geführt zu werden. (Eine erstmalige Einführung in Word oder Photoshop wäre übrigens ähnlich nervig.) Wir hoffen, dass Sie bei allen Detail-Anweisungen nicht den roten Faden aus den Augen verlieren.

Und ebenso paradox ist, wenn Sie im dritten Lab den ganzen Aufwand nur betreiben, um zwei Zahlen zu addieren. Zum Glück ist Ihnen aber klar, dass ein solches Mini-Beispiel nötig ist, weil man in einer einstündigen Übung nicht mehr erreichen kann. Und zum Glück haben Sie genug Phantasie zu verstehen, dass die HW-Realisierung und die HW-SW-Kommunikation des lächerlichen kleinen Addierers viel mit großen realen Entwürfen gemeinsam haben könnten.

Wir bemühen uns um guten Service, weil wir möchten, dass etliche von Ihnen bei uns weitermachen. Zu gutem Service gehört übrigens auch, dass wir offen sind für Ihr Feedback, positives, vor allem aber auch kritisches Feedback.

B

Lab 1: VERILOG und seine Tools

Das erste der drei Labs führt in die praktische Arbeit mit der Hardware-Beschreibungssprache VERILOG ein. Sie lernen erst mal die Oberflächen unserer CAD-Tools ISE und ModelSim kennen und dann Schritt für Schritt neue Features wie automatische Testrahmen-Generierung und grafische Simulationsausgabe. Vor allem werden natürlich auch die Konzepte aus der Vorlesung vertieft und praktisch angewandt.

Längere Quelltexte müssen Sie nicht abtippen, sondern können sie von unserer Übungs-Webseite herunterladen:
http://www.eis.cs.tu-bs.de/schroeder/hw-sw-uebungen/ .

B.1 Hello World!

Dass aller Anfang gar nicht so schwer ist, werden Sie jetzt feststellen. Sie legen ein Projekt an, geben ein winziges VERILOG-Programm ein und simulieren es.

B.1.1 Projekt-Navigator starten

Starten Sie den Projekt-Navigatorvon Xilinx über das Icon in Bild B.1**,** und Sie sehen seine grafische Oberfläche in Bild B.2.

Xilinx ISE 10.1

Bild B.1
Projekt-Navigator

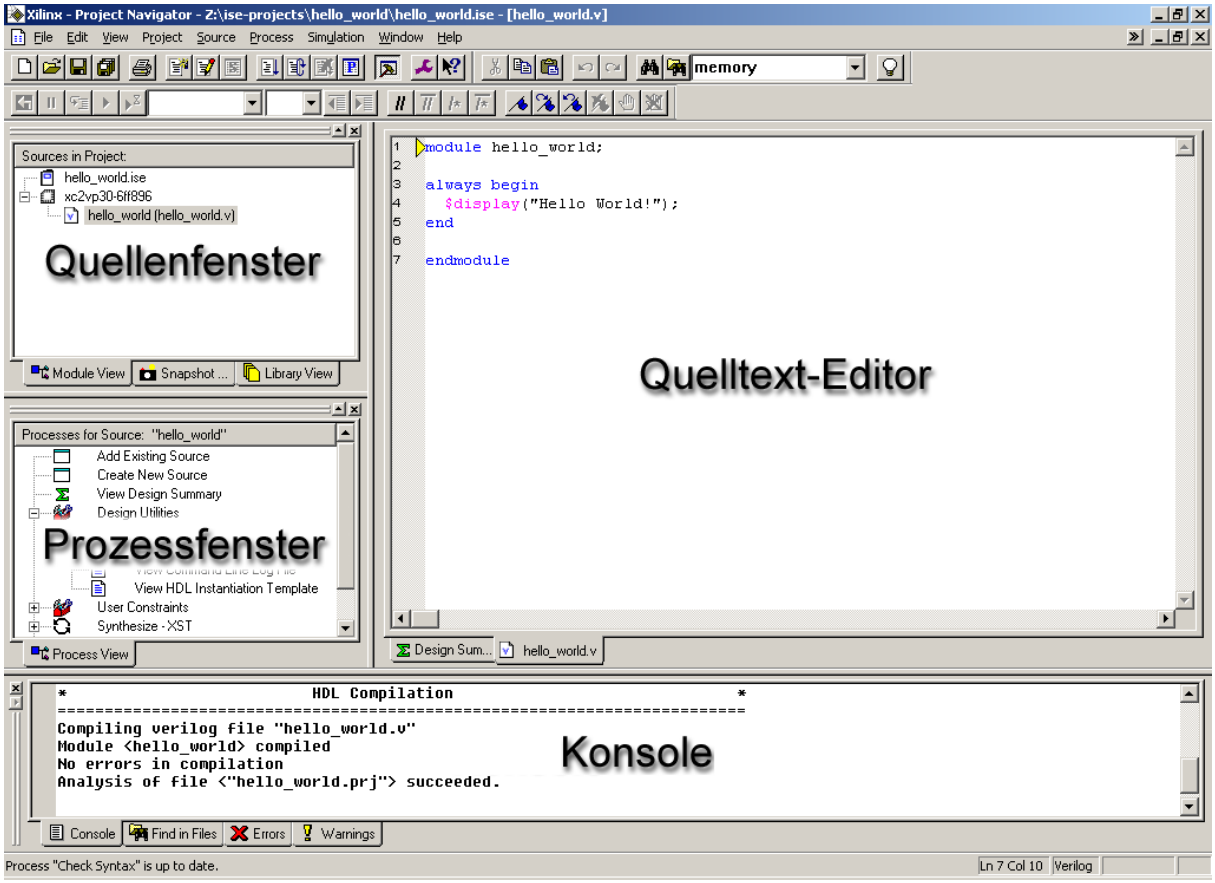


Bild B.2 Überblick zum Projekt-Navigator

B.1.2 Ein neues Projekt

Zuerst legen Sie ein neues Projekt mit File → New Project… an. Wählen Sie als Projektnamen hello_world und als Projectverzeichnis den Pfad D:\gruppeXX\hello_world (wobei XX Ihrer jeweiligen Gruppennummer entspricht). Als Top-Level Module Type sollte HDL ausgewählt sein (Bild B.3).

Achtung: Laufwerk D ist ein lokales Laufwerk, auf dem Ihre Daten nicht gesichert sind. Ihr persönliches Laufwerk ist U. Wenn Sie neue Projekte anlegen, tun Sie dies bitte immer lokal auf D:\. Am Ende einer Übung verschieben Sie Ihre Daten bitte immer in Ihr persönliches Verzeichnis unter U:\. Zu Beginn der nächsten Übung können Sie es sich zum Arbeiten dann von dort wieder in ein lokales Verzeichnis kopieren.

Klicken Sie Next.

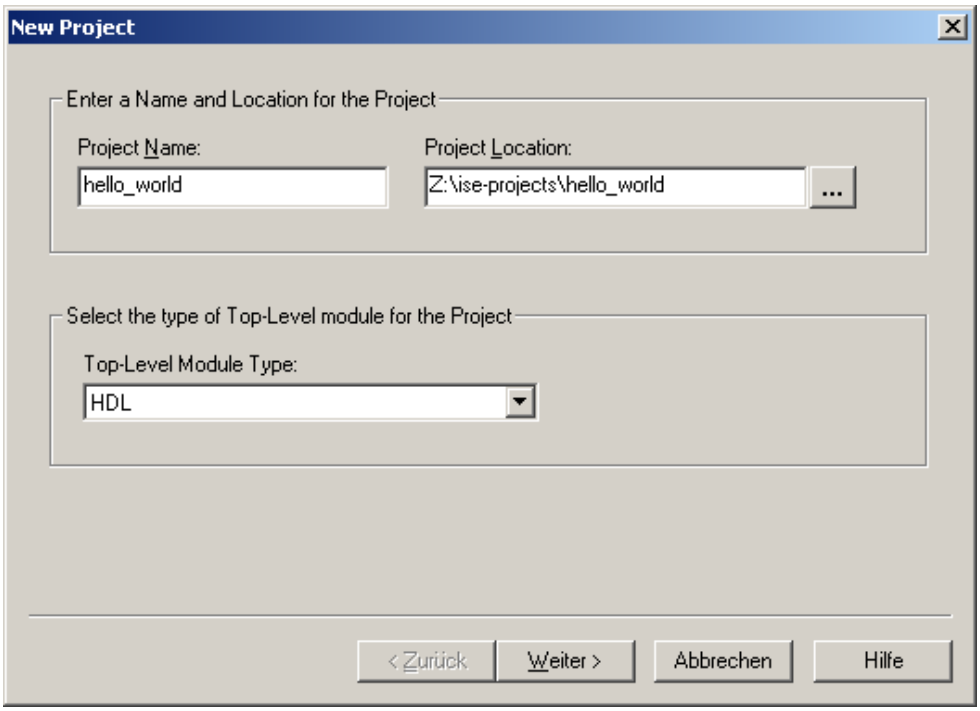


Bild B.3
Neues Projekt: Projektname

Stellen Sie im nächsten Dialog in Bild B.4 sicher, dass die Device Family Virtex2P, das Device XC2VP30 und das Package FF896, sowie Speed Grade -6 ausgewählt sind. Wählen Sie als Simulator den Modelsim-SE Verilog. Klicken Sie wie bei den drei folgenden Dialogfeldern Next bzw. Finish.

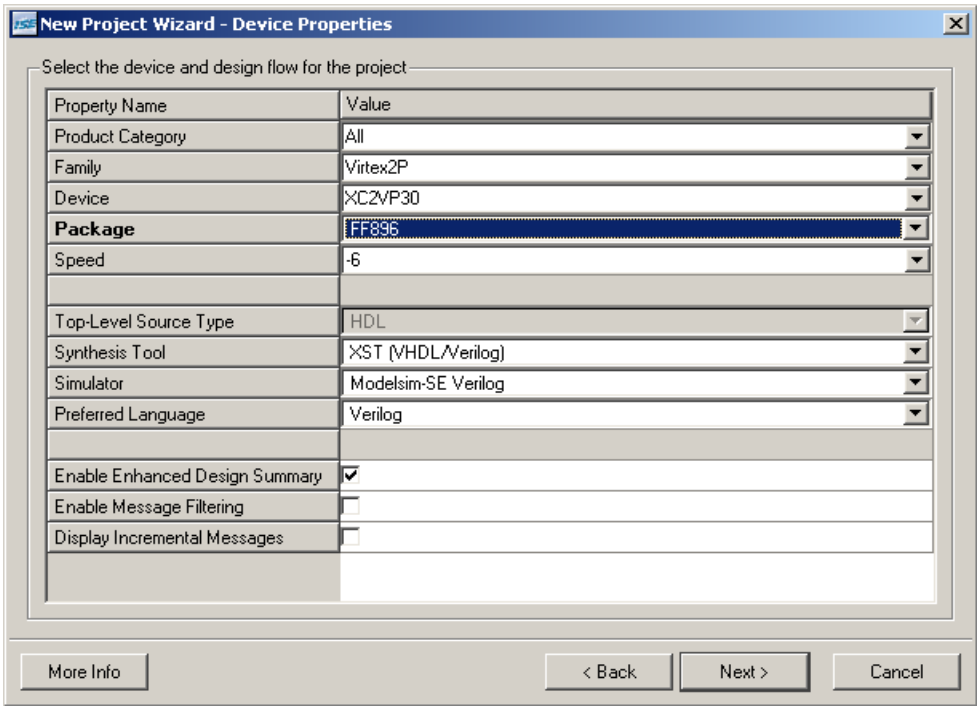


Bild B.4
Neues Projekt: Einstellungen

B.1.3 Benutzeroberfläche von ISE

Sie haben nun ein neues Projekt angelegt. Die Benutzeroberfläche des Projekt-Navigators hat wie in Bild B.2 vier Bereiche. Das Quellenfenster oben links zeigt die Module Ihres Designs. Mit einem Rechtsklick auf xc2vp30-6ff896 können Sie bei Bedarf unter Properties Einstellungen ändern. Mit einem einfachen linken Mausklick wählen Sie ein Objekt aus, für das darunter das Prozessfenster vordefinierte Abläufe anbietet wie einen Syntax-Check oder eine Simulation. Die Konsole unten meldet Hinweise und Fehler. Was der Quelltext-Editor tut, versteht sich von selbst.

B.1.4 Quelltext erzeugen und ins Projekt einfügen

Erzeugen Sie nun eine neue VERILOG-Moduldatei, indem Sie im Quellenfenster rechtsklicken und New Source… wählen. Wählen Sie im Dialog in Bild B.5 Verilog Module an und tragen Sie unter File Name den Dateinamen `hello_world` ein. Bestätigen Sie diesen und die weiteren Dialoge mit Next bzw. Finish.

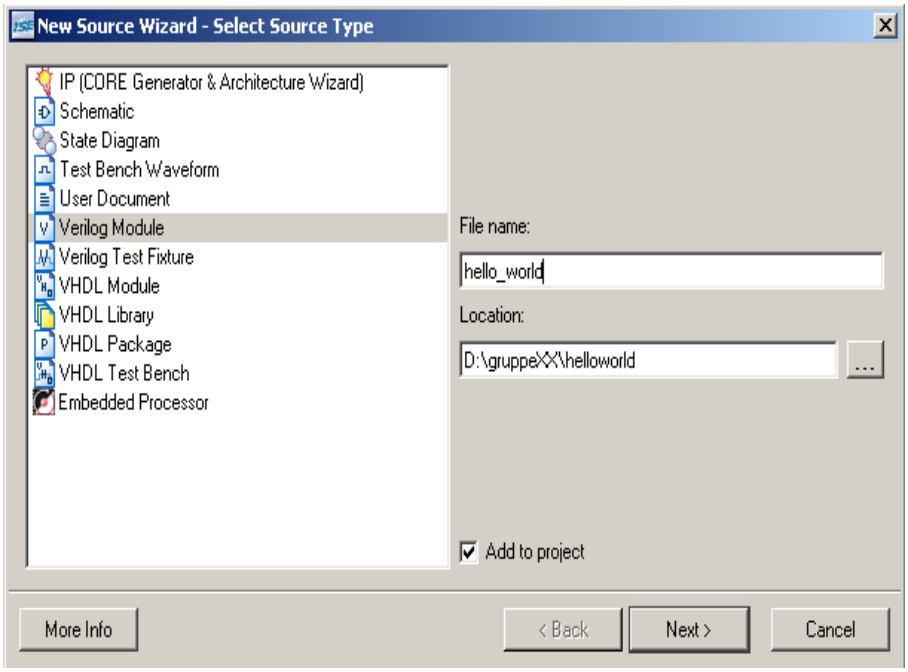


Bild B.5 Neue VERILOG-Moduldatei anlegen

Das Objekt hello_world (hello_world.v) im Quellenfenster bedeutet, dass nun das Modul hello_world in Datei hello_world.v Teil Ihres Projektes ist. Durch einen Doppelklick können Sie den entsprechenden Quelltext im Hauptfenster betrachten und verändern.

Fügen Sie in das Modul den Quelltext aus Beispiel B.6 ein und speichern Sie mit Strg-s.

```
module hello_world(
    );
initial begin
  $display("Hello World!");
end

endmodule
```

Beispiel B.6 Hello-World

B.1.5 Simulator aufrufen

Um ModelSim aus dem Project Navigator zu starten, müssen Sie diesen zunächst einmal einbinden. Dies tun Sie über den Menüpunkt Edit | Preferences…, klicken dort unter ISE General auf Integrated Tools und geben für den Model Tech Simulator den Pfad C:\Programme\modeltech\win32\modelsim.exe ein. Bestätigen Sie mit OK.
Klicken Sie einmal das Modul hello_world im Quellenfenster und wählen Sie im Ausklappmenü Sources for Behavioral Simulation aus. Betrachten Sie die zugehörigen Prozesse im Prozessfenster. Unter ModelSim Simulator verbirgt sich unter anderem der für uns interessante Prozess Simulate Behavioral Model. Doppelklicken Sie diesen Prozess (Bild B.7).

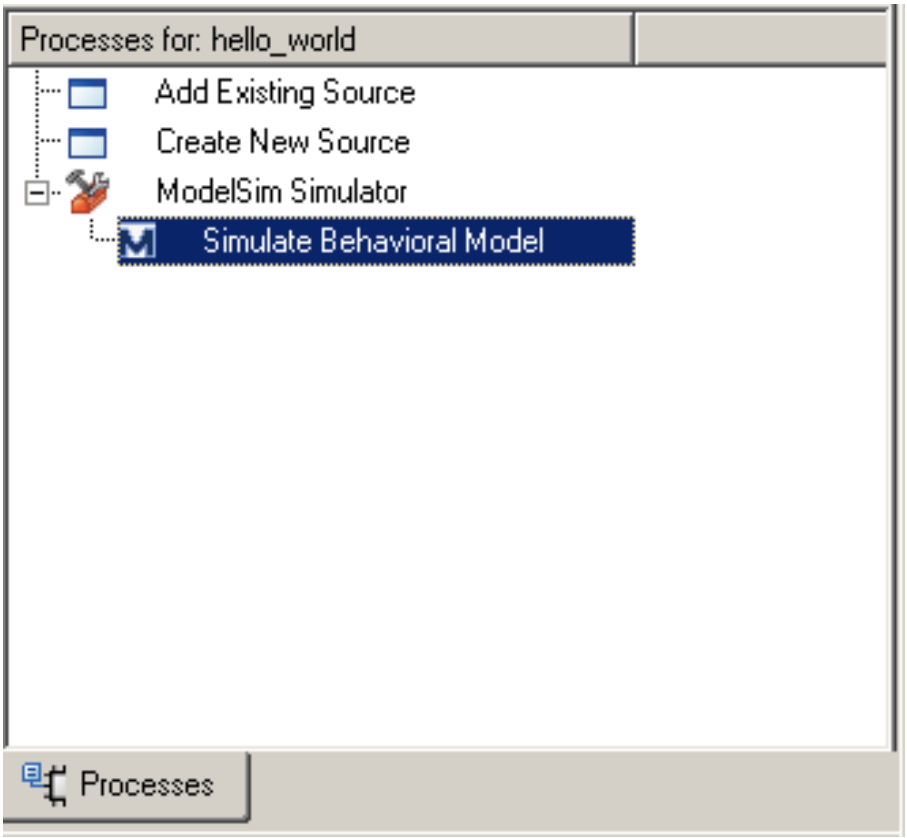


Bild B.7
Aufruf des Simulators ModelSim

B.1.6 Benutzeroberfläche von ModelSim

Der VERILOG-Simulator ModelSim wird nun gestartet. Das Workspace-Fenster oben links in Bild B.8 zeigt alle Module, Modulinstanzen, `initial`- und `always`-Blöcke sowie Continuous Assignments. Die beiden Fenster rechts davon geben Waveforms aus. Die Konsole unten dient der Kommunikation mit ModelSim.

Im Workspace-Fenster sehen wir momentan die zwei Module hello_world und glbl. Letzteres ist automatisch hinzugefügt und steuert den globalen Reset und Taktverzögerungen. Wir akzeptieren lediglich seine Existenz und wenden uns stattdessen unserem hello_world-Modul zu. Die Angabe #INITIAL#23 direkt unter hello_world verrät den Beginn eines `initial`-Blocks in der 23. Zeile. Klicken Sie mit rechts auf hello_world und wählen View Declaration. Im Waveform-Fenster können Sie nun den Quelltext von hello_world sehen.

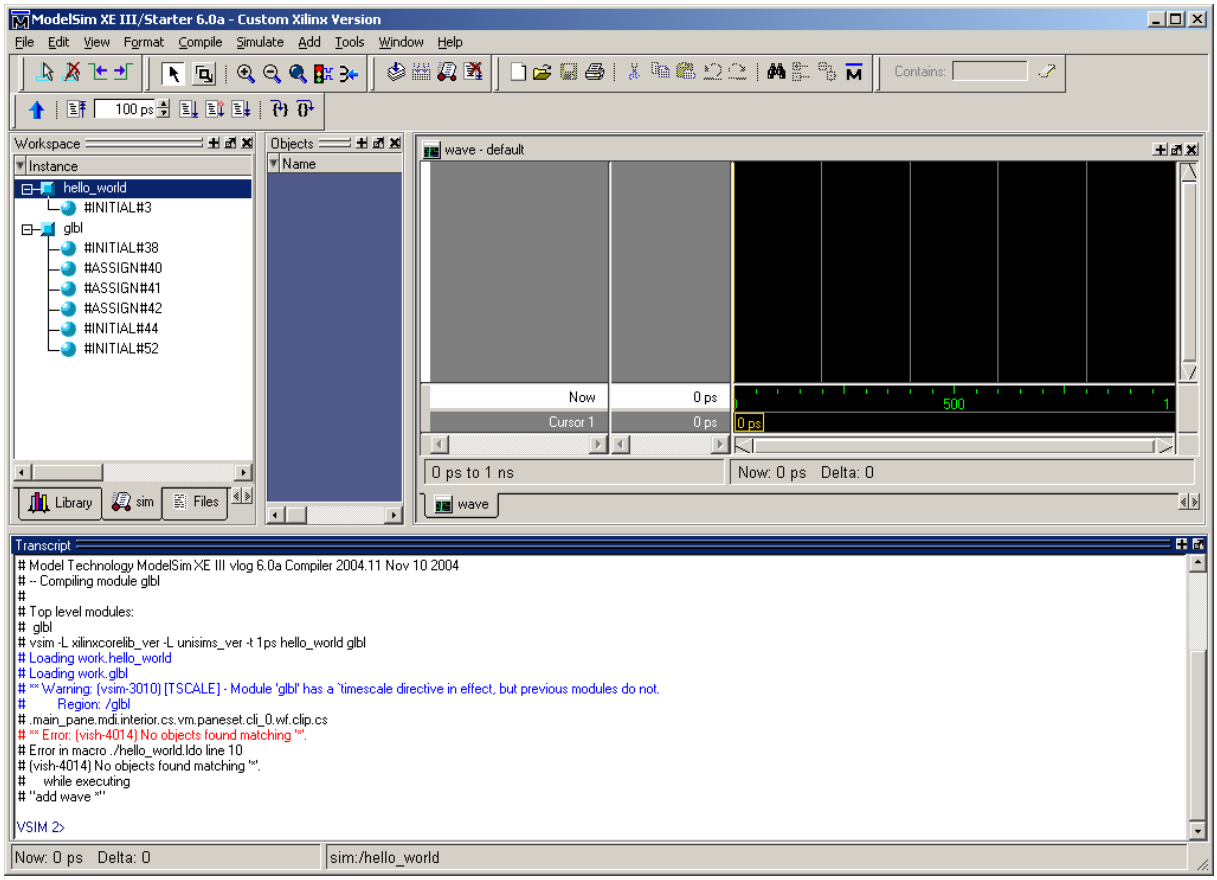


Bild B.8 Benutzeroberfläche des Simulators ModelSim

B.1.7 Simulation starten

Klicken Sie im Konsolenfenster einmal mit der linken Maustaste rechts neben `VSIM 2>`. Dort blinkt nun der Cursor, und Sie können die Simulation mit `run` starten

(Beispiel B.9). Das Modul hello_world gibt nun `Hello World!` aus, dem wie allen Simulationsausgaben noch ein # vorangestellt ist.

```
VSIM 2> run
# Hello World!

VSIM 3>
```

Beispiel B.9 Simulationsausgabe

Herzlichen Glückwunsch – Sie haben soeben Ihre erste VERILOG-Simulation erfolgreich durchgeführt! Schließen Sie ModelSim und kehren Sie zur ISE-Oberfläche zurück.

B.2 Simulierte Zeit

Als nächstes wollen wir uns mit der simulierten Zeit beschäftigen. Erstellen Sie ein neues ISE-Projekt, in das Sie ein neues Modul fortime einfügen (Beispiel B.10).

```
`timescale 1ns / 1ps

module fortime;

integer INDEX;    // Deklaration der Schleifenvariablen

initial begin
  for (INDEX = 1; INDEX <= 5; INDEX = INDEX + 1) begin
    $display("Simulationszeit ist %d", $time);
  end
end

endmodule
```

Beispiel B.10 Modul `fortime`

Das Modul `fortime` besteht aus einem `initial`-Block, in dem sich eine Schleife befindet, welche fünf mal die interne Simulationszeit des Simulators ausgibt. Als Datentyp für die Schleifenvariable wird `integer` verwendet.

Führen Sie – wieder per Doppelklick auf Simulate Behavioral Model – eine Simulation durch und beobachten Sie die einzelnen Ausgaben der jeweiligen simulierten Zeit. Diese beträgt bei jedem Schleifendurchlauf 0, da keine simulierte Zeit vergangen ist.

Lassen Sie ModelSim geöffnet und verändern Sie nun das Modul, indem Sie direkt vor der `$display`-Anweisung die Zeitkontrolle `#INDEX` einfügen. Sichern Sie die Datei mit File | Save und wechseln Sie zur noch geöffneten ModelSim-Applikation.

Sie haben zuvor vielleicht schon festgestellt, dass direkt nach dem Aufruf von ModelSim automatisch ein Simulationslauf durchgeführt wird, und zwar mit einer ausreichend langen simulierten Zeit. Dies wird durch ein Skript gesteuert, das ISE automatisch generiert. Durch Drücken der Pfeil-Hoch-Taste auf der ModelSim-Konsole können Sie den Aufrufbefehl do {fortime.fdo} für das Skript aus der Befehlshistorie hervorholen und das Skript erneut starten. Dann werden alle VERILOG-Module neu compiliert und die Simulation neu gestartet. Aber Vorsicht: dieser Trick ist nur für kleine Designs praktikabel, bei denen das Recompilieren aller Module nicht zu langwierig ist. Die Länge der simulierten Zeit können Sie übrigens in ISE über die Properties von ModelSim steuern.

Dies ist die schnellste Möglichkeit, die Änderungen für die Simulation zu übernehmen, ohne den Simulator neu starten zu müssen. Alternativ können Sie diesen Vorgang auch manuell durchführen: Klicken Sie dazu im Workspace-Fenster auf den Tab Library, und betrachten Sie den Inhalt der Library work (Bild B.11).

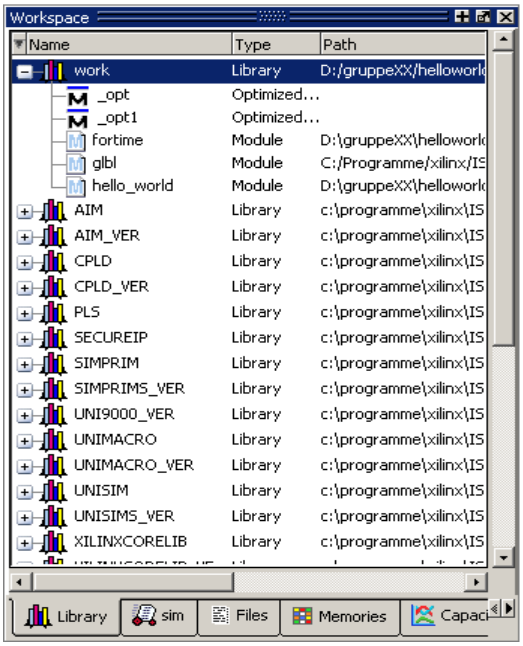


Bild B.11
Aktualisierung der Module
in der Simulationsbibliothek

Durch einen Rechtsklick auf fortime und die Aktion Recompile werden die Änderungen, die Sie in ISE vorgenommen haben, von ModelSim übernommen. Führen Sie nun noch einen Simulations-Restart durch, indem Sie in der Konsole den Befehl `restart` eingeben und den sich öffnenden Dialog mit Restart bestätigen. Starten Sie nun die Simulation mit `run`.

Sie werden bei dieser Variante allerdings feststellen, dass Sie als Simulationsausgabe nur eine leere Zeile erhalten. Der Grund liegt in der zu kurz eingestellten zu simulierenden Zeit. Oben unter der ModelSim-Menüleiste sehen sie ein Textfeld, in dem `100 ps` steht, so dass die Simulation nur für den Zeitraum t = 0 bis t = 100 ps (Picosekunden) durchgeführt wird. Ändern Sie diesen Wert auf 100 ns (Nanosekunden), führen Sie wieder einen `restart` durch und starten Sie erneut mit `run`.

Nun sehen Sie, dass die simulierte Zeit bei jeder Ausgabe um `INDEX` erhöht wird. Beachten Sie, dass der Simulator eine Umrechnung von *logischer Zeit* (#1) in *reale Zeit* (1 ns) vornimmt. Der Schlüssel hierzu ist die Anweisung `` `timescale 1ns / 1ps ``, die ISE beim Erzeugen Ihrer VERILOG-Moduldatei automatisch eingefügt hat. Der erste Wert bezieht sich auf die Zeiteinheit, d.h. #1 entspricht 1 ns. Der zweite Wert gibt die interne Auflösung des Simulators an. Beide Werte haben für uns hier keine weitere Bedeutung und werden nur aus technischen Gründen aufgeführt.

B.3 Konkurrierende Ereignisse

Betrachten Sie nun Beispiel B.12. Das Modul hat zwei Dateneingänge und zwei `always`-Blöcke. Um das Modul zu testen, d.h. Daten einzugeben und seine Reaktion zu beobachten, benötigen wir einen Testrahmen (Beispiel B.13). Im Testrahmen werden benötigte Register deklariert, das zu testende Modul instanziert sowie Stimuli erzeugt.

```
`timescale 1ns / 1ps

module activation (
  input wire IVAR1,                                    // Modulschnittstelle
  input wire IVAR2
);

always @(IVAR1)
  $display("Block 1 ausgeloest bei Zeit %d", $time);

always @(IVAR1, IVAR2)
  $display("Block 2 ausgeloest bei Zeit %d", $time);

endmodule
```

Beispiel B.12 Aktivierungslisten

Sie sollen nun das Modul `activation` mit dem Testrahmen `test_activation` simulieren. Erzeugen Sie dazu ein neues ISE-Projekt und nehmen Sie das Modul `activation` wie gewohnt ins Projekt auf. Für den Testrahmen erzeugen Sie mit File | New | Text File eine neue Datei, in die Sie den Quellcode aus Beispiel B.13 eintragen. Speichern Sie die Datei als `test_activation.v`, und fügen Sie sie per Rechtsklick im Quellenfenster | Add Source ins Projekt ein. Nun erscheint `test_activation` im Quellenfenster. Klicken Sie einmal darauf und starten Sie die Verhaltenssimulation über Simulate Behavioral Model im Prozessfenster. Simulieren Sie das Modell für eine Zeit von 10 ns.

```
`timescale 1ns / 1ps

module test_activation;

reg IVAR1, IVAR2;

// Instanzierung des Moduls activation;
// die Instanz heißt Activation
activation Activation(IVAR1, IVAR2);

// Stimuli
initial begin
  #1;           // eine Simulationszeiteinheit warten
  IVAR1 = 0;    // IVAR1 auf 0 initialisieren
  #1;           // eine Simulationszeiteinheit warten
  IVAR2 = 1;    // IVAR2 auf 1 initialisieren
end

endmodule
```

Beispiel B.13 Testrahmen für das Modul `activation`

Die `always`-Blöcke werden immer dann ausgeführt, wenn ein bestimmtes Ereignis vorliegt (`always @`). Die Auslöser der Ereignisse haben in diesem Fall ihren Ursprung in dem `initial`-Block des Testrahmens. Der erste `always`-Block reagiert auf Änderungen von `IVAR1`. Der zweite `always`-Block wird ausgeführt, sobald sich `IVAR1` oder `IVAR2` ändern. Aufgrund der Initialisierung aller Variablen eines VERILOG-Modells auf den unbestimmten Wert `x` zum Simulationszeitpunkt 0 stellen die Zuweisungen `IVAR1=0` sowie `IVAR2=0` eine solche Änderung dar.

Simulieren Sie nun verschiedene Variationen, bei denen Sie die Reihenfolge der `always`-Blöcke vertauschen sowie die Zeitpunkte und die Abfolge der Zuweisungen im Testrahmen verändern. Betrachten Sie dabei auch Fälle, bei denen keine Wartezeiten vor oder zwischen den einzelnen Zuweisungen vorhanden sind, also Ereignisse konkurrieren. Versuchen Sie jeweils vor der Simulation Aussagen über das Verhalten zu machen.

Sie werden dabei feststellen, dass der Simulator konkurrierende Ereignisse nicht unbedingt intuitiv vorhersagbar abarbeitet. Der Simulator optimiert die Menge der Ereignisse und die Bearbeitungsreihenfolge, um die Rechenzeit für die Simulation zu minimieren. Innerhalb eines Simulationszeitpunktes kann der Simulator Ereignisse für einen Block zusammenfassen und die Reihenfolge der Bearbeitung von Blöcken variieren.

Besonders problematisch ist dies zur Zeit 0, wenn `initial`- und `always`-Blöcke zum ersten Mal gestartet werden. Wird ein `always`-Block aufgrund der beliebigen Bearbeitungsreihenfolge erst für ein Ereignis empfangsbereit, nachdem ein anderer Block bereits ein Ereignis für ihn generiert hat, so kann er dieses Ereignis nicht wahrnehmen.

Die Ausnutzung einer auf diese Weise erreichten Bearbeitungsreihenfolge von konkurrierenden Ereignissen und Blöcken in einem Simulationszeitpunkt ist ein

Modellierungsfehler. Er ist eine häufige Ursache für fehlerhaftes Verhalten eines VERILOG-Modells nach einem Wechsel oder Update des Simulators, nach kleinen Änderungen im Modell selbst, aber auch nach der Integration in ein größeres Modell.

B.4 Flip-Flop

Machen Sie sich nun mit dem VERILOG-Modell in Beispiel B.14 vertraut und fügen Sie es in ein neues ISE-Projekt ein. Es handelt sich hierbei um ein taktflankengesteuertes Register mit Reset, welches Daten bei steigenden (positiven) Taktflanken übernimmt.

```
`timescale 1ns / 1ps

module flipflop(
  output reg  [7:0] DATA,      // Ausgang, Register
  input  wire [7:0] IN,        // Eingaenge, Wires
  input             CLOCK,
                    RESET
  );

always @(posedge CLOCK) begin
  if (RESET == 1) DATA <= 0;
  else DATA <= IN;
end

endmodule
```

Beispiel B.14 Flankengesteuertes 8-Bit-Register

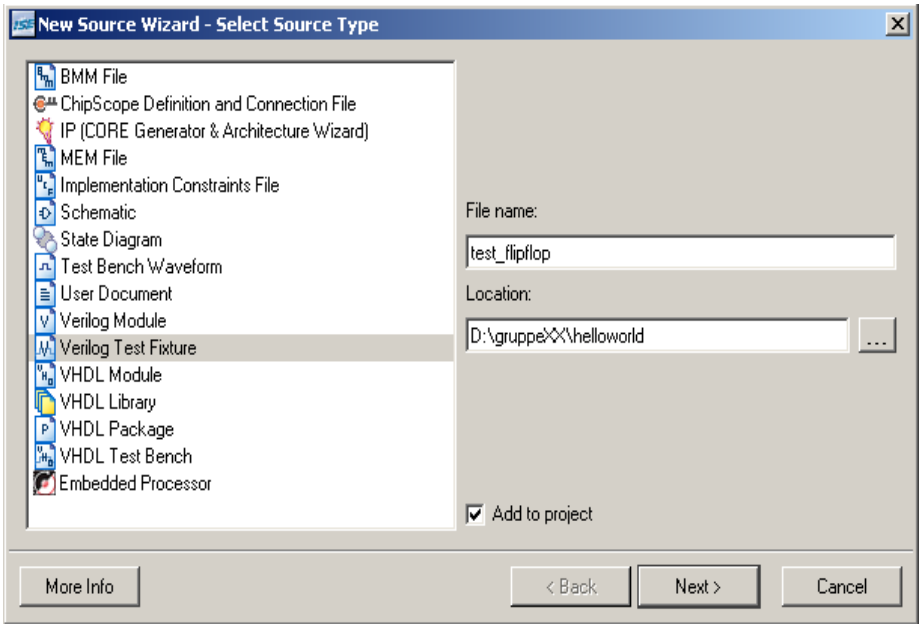


Bild B.15 Testrahmen erzeugen

Wegen der Bearbeitungsbedingung `posedge CLOCK` wird der `always`-Block nur dann bearbeitet, wenn sich `CLOCK` auf 1 ändert, also bei einer positiven Taktflanke. Eine Reaktion auf negative Flanken wäre durch `negedge` realisierbar.

Desweiteren ist `<=` neben `=` die zweite Möglichkeit einer Wertzuweisung (Abschnitt 3.6). Die Unterschiede werden erst ab einer späteren Aufgabe bedeutsam.

Um das Modul zu testen, benötigen wir wieder einen Testrahmen. ISE bietet die Möglichkeit, Teile des Testrahmens automatisch anzulegen, was wir im Folgenden versuchen wollen. Klicken Sie rechts ins Quellenfenster | New Source…, wählen Verilog Test Fixture und geben den Dateinamen `test_flipflop` ein (Bild B.15). Bestätigen Sie mit Next / Finish.

Der automatisch generierte Testrahmen test_flipflop erscheint nun im Quellenfenster, wenn bei Sources for der Eintrag Behavioral Simulation gewählt wurde. Betrachten Sie seinen Quelltext (Beispiel B.16).

```
`timescale 1ns / 1ps

module test_flipflop;

// Inputs
reg [7:0] IN;
reg CLOCK;
reg RESET;

// Outputs
wire [7:0] DATA;

// Instantiate the Unit Under Test (UUT)
flipflop uut (
   .DATA(DATA),
   .IN(IN),
   .CLOCK(CLOCK),
   .RESET(RESET)
);

initial begin
  // Initialize Inputs
  IN = 0;
  CLOCK = 0;
  RESET = 0;

  // Wait 100 ns for global reset to finish
  #100;

  // Add stimulus here

end
endmodule
```

Beispiel B.16 Automatisch generierter Testrahmen

Nach der Deklaration von Input- und Output-Signalen folgt die Instanzierung des Moduls `flipflop`. Die I/O-Ports werden hier durch die Port-Namen zugeordnet.

Bevor Sie die Simulation durchführen können, muss der Testrahmen noch um eine Takterzeugung, Stimuli und ein Monitoring erweitert werden. Komplettieren Sie

den Testrahmen wie in Beispiel B.17. Achten Sie darauf, die Zeile `CLOCK = 0` unter `// Initialize Inputs` zu löschen.

```
`timescale 1ns / 1ps

module test_flipflop;

// Inputs
reg [7:0] IN;
reg CLOCK;
reg RESET;

// Outputs
wire [7:0] DATA;

// Instantiate the Unit Under Test (UUT)
flipflop uut (
  .DATA(DATA),
  .IN(IN),
  .CLOCK(CLOCK),
  .RESET(RESET)
);

// Takterzeugung
always begin
  CLOCK = 1; #10;
  CLOCK = 0; #10;
end

initial begin
  // Monitoring
  $monitor("Zeit %d: CLOCK = %b, IN = %b, RESET = %b, DATA = %b",
           $time, CLOCK, IN, RESET, DATA);

  // Initialize Inputs
  IN = 0;
  // CLOCK = 0; ← geloescht wegen Takterzeugung!
  RESET = 0;

  // Wait 100 ns for global reset to finish
  #100;

  // Add stimulus here
  // Stimuli
  IN = 100; RESET = 1; #10;
  IN = 0;   RESET = 0; #10;
  IN = 201; RESET = 0; #10;
  // Ende der Simulation
  $stop;
end
endmodule
```

Beispiel B.17 Komplettierter Testrahmen

Führen Sie nun eine Simulation durch, indem Sie im Quellenfenster test_flipflop einmal anklicken und den zugehörigen Prozess Simulate Behavioral Model starten.

Tipp: Die Simulationsausgaben werden im Konsolenfenster in einer proportionalen Schrift dargestellt. Über den Menüpunkt Tools | Edit Preferences | Main Window können Sie eine besser lesbare fixe Schrift, z.B. Courier New auswählen.

B.5 VERILOG-Quiz

In Beispiel B.19 und Beispiel B.20 hat Karl Klammer unerwartet alle Variablen- und Modulnamen durch unverständliche Bezeichner ausgetauscht. Versuchen Sie, per Simulation herauszufinden, welche Berechnung hier durchgeführt wird! Sie können sich den Quelltext von unserer Übungs-Webseite herunterladen.

Bild B.18
Karl Klammer
hat zugeschlagen

```
`timescale 1ns / 1ps

module y(
  input  wire       A,
                    B,
  output reg  [7:0] C,
                    D
);

always @(posedge A) begin
  if (B) begin
    C <= 1;
    D <= 1;
  end
  else begin
    D <= C;
    C <= C + D;
  end
end

endmodule
```

Beispiel B.19 Unverständliches Codestück 1

Haben Sie das Berechnungsergebnis erkannt? Prima! Ändern Sie nun die nicht-blockenden Zuweisungen `<=` im `else`-Zweig des `always`-Blocks in blockende Zuweisungen `=` und simulieren Sie erneut. Welche Auswirkungen haben die Änderungen auf das Berechnungsergebnis? Versuchen Sie, die Ursache zu erklären.

```
`timescale 1ns / 1ps

module x;

reg        A,
           B;
wire [7:0] C,
           D;

always begin
  A = 1; #10;
  A = 0; #10;
end

y Y(A, B, C, D);

initial begin
  $display("F   A B | C    D   ");
  $display("====================");
  $monitor("%.3d %.1b %.1b | %.4d %.4d", $time, A, B, C, D);
  B = 1; #100; B = 0; #200; $stop;
end

endmodule
```

Beispiel B.20 Unverständliches Codestück 2

B.6 Parametrisiertes Schieberegister

Legen Sie ein neues ISE-Projekt an, und fügen Sie den Code aus Beispiel B.21 ein (downloadbar von der Übungs-Webseite).

```
`timescale 1ns / 1ps

module shiftreg #(parameter SIZE = 4) // Default-Laenge
(
  output reg [SIZE-1:0] DATA,         // Ausgang
  input  wire           DIN,          // Eingaenge
                        CLOCK
);

always @(posedge CLOCK) begin
  DATA <= {DIN, DATA[SIZE-1:1]};
end

endmodule
```

Beispiel B.21 Parametrisiertes Schieberegister

Dabei handelt es sich um ein Schieberegister von parametrisierter Länge mit einem seriellen Eingang und einem parallelen Ausgang. (Schieberegister sind Grundbausteine, die z. B. in den Bereichen Fehlerkorrektur und Netzwerkanbindung eine wichtige Rolle spielen.)

Fügen Sie den Testrahmen aus Beispiel B.22 hinzu und simulieren Sie. Was passiert, wenn Sie im Testrahmen bei der Instanzierung des Schieberegister-Moduls auf die Übergabe des Parameters TSIZE verzichten?

```
`timescale 1ns / 1ps

module test_shiftreg;

parameter TSIZE = 8;              // Testlaenge

// Inputs
reg DIN;
reg CLOCK;

// Outputs
wire [7:0] DATA;

// Instantiate the Unit Under Test (UUT)
shiftreg #TSIZE uut (
  .DATA(DATA),
  .DIN(DIN),
  .CLOCK(CLOCK)
);

always begin
  CLOCK = 0; #10;
  CLOCK = 1; #10;                 // => 1 Takt = #20
end

initial begin
  $display("TIME CLOCK | DIN DATA    ");
  $display("=========================");
  $monitor("%.4d %.5b | %.3b %.8b",
           $time, CLOCK, DIN, DATA);

  // Initialize Inputs
  DIN = 0;

  // Wait 100 ns for global reset to finish
  #100;

  // Add stimulus here
  DIN = 0; #20;                   // 1 Takt warten
  DIN = 1; #20;
  DIN = 1; #20;
  DIN = 0; #20;
  DIN = 1; #20;
  DIN = 1; #20;
  DIN = 1; #20;
  DIN = 0; #20;
  $stop;
end
endmodule
```

Beispiel B.22 Testrahmen für das Schieberegister

Neben der textuellen Ausgabe zeigt ModelSim ausgewählte Signale auch grafisch als *Waveform* an (Bild B.23).

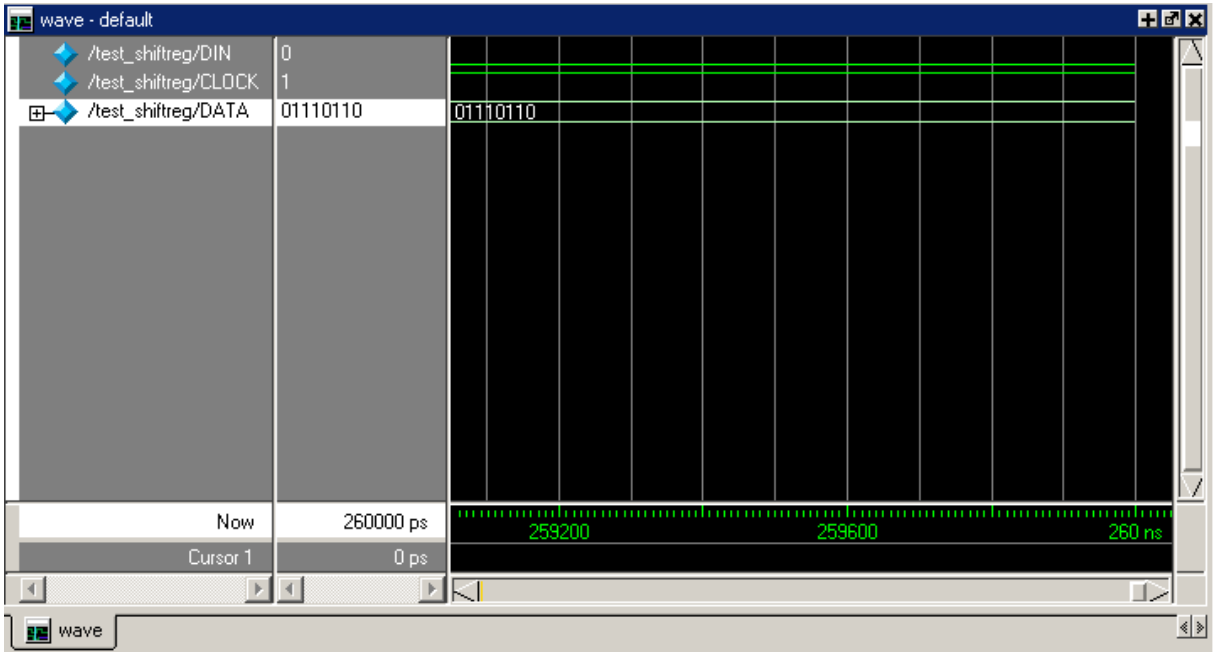


Bild B.23 Waveform-Ansicht in ModelSim

Klicken Sie in das Waveform-Fenster und betätigen dann solange Zoom-Out (Bild B.24), bis Sie die Variation im CLOCK-Signal deutlich sehen können (Bild B.25).

Bild B.24
Zoom-Out-Button

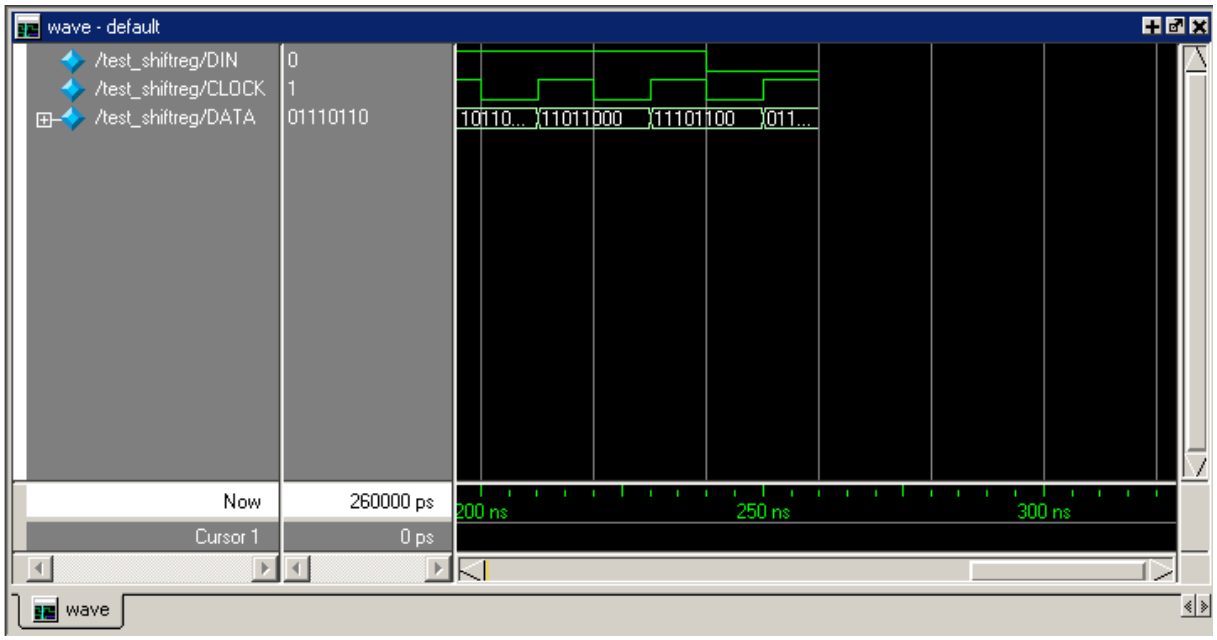


Bild B.25 Waveform nach Zoom-Out

Scrollen Sie dann nach links bis zum Zeitpunkt 0 der Simulation. Sie sehen, wie sich das Register DATA zu jeder steigenden Taktflanke mit einem weiteren Datenbit füllt. Sobald keins der Bits mehr den unbestimmten Wert x enthält, wird der Signalverlauf in grün dargestellt.

Wählen Sie ein oder zwei Stellen in den Stimuli des Testrahmens aus und versuchen Sie, diese Stellen im grafischen Signalverlauf wiederzufinden.

In anderen Simulationen kann es sinnvoll sein, sich die Waveforms statt binär in einem anderen Zahlenformat anzusehen. Probieren Sie eine hexadezimale Darstellung des Signals DATA aus durch Rechtsklick auf DATA | Radix | Hexadecimal. Ändern Sie die Darstellung danach wieder auf binär.

Sie können weitere Signale als Waveform betrachten, indem Sie im Workspace-Fenster eine Modulinstanz anklicken und dann rechts im Objects-Fenster aus den zugehörigen Signalen die gewünschten per Drag & Drop ins Waveform-Fenster ziehen. Um den zeitlichen Verlauf dieser neuen Signale sehen zu können, muss jedoch ein Restart mit anschließendem Run durchgeführt werden.

Erweiterung Schieberegister

Erweitern Sie nun das Schieberegister um einen high-aktiven Reset auf 0 (d.h. der Ausgang wird auf 0 gesetzt, wenn der Reset-Eingang auf 1 liegt) sowie um einen parallelen Lade-Eingang (d.h. ein zweiter Eingang um mehrere Bits parallel zu laden) und verifizieren Sie alle Funktionen durch Simulation mit Testrahmen.

B.7 Abschlussquiz

Willkommen zum ersten Abschlussquiz! Die folgenden Fragen zielen auf ein Grundverständnis ab, auf dass Sie sich nun selber testen können. *Wichtig:* Bearbeiten Sie die Fragen bitte für sich allein, denn nur so erzielen Sie (und auch wir) den gewünschten Lerneffekt. Sie können sich ihre Antworten in Stichworten notieren oder auch nur grob im Kopf überlegen.

Wenn jeder Ihrer Übungspartner mit dem Quiz fertig ist, sprechen Sie Ihren HiWi an, er wird Ihnen Ihre Antworten abnehmen. Doch keine Sorge: wir werden keinerlei Konsequenzen ziehen. Die Fragen sind nur für Sie gemacht, damit Sie Ihren eigenen Wissensstand kontrollieren können. Deshalb empfehlen wir Ihnen auch, erst einmal nur aus dem Gedächtnis zu antworten, und erst danach im Skript nachzuschauen.

Frage 1

Was bedeutet Parallelität in Verilog? Was für Voraussagen kann man über die Reihenfolge der Blöcke machen? Wovon hängt es ab, welcher Block zuerst ausgeführt wird?

Frage 2

Wie wird der Inhalt eines `always`-Blockes ausgeführt? Wie kann man den `always`-Block unterbrechen und so die Kontrolle an einen anderen Block abgeben?

Frage 3

Was ist der signifikante Unterschied zwischen der blockenden und der nicht-blockenden Zuweisung?

Frage 4

Wie können zwei Variablen ihre Werte in einer Programmiersprache wie C oder Java tauschen? Wie kann man dies viel einfacher in Verilog realisieren?

Frage 5

Was ist serielle, was ist parallele Datenübertragung?

Herzlichen Glückwunsch! Sie haben den regulären Teil von Lab 1 erfolgreich absolviert und die Grundlagen von Verilog erarbeitet. Die nun folgenden Aufgaben sind freiwillig und anspruchsvoller. Je nach dem, wie weit Sie im Zeitplan sind, können Sie diese Aufgaben zusätzlich bearbeiten. Allerdings sollen Sie mit B.8 und B.9 beginnen. Falls Sie sich dann immer noch nicht ausgelastet fühlen, können Sie frei aus den übrigen Aufgaben wählen. Sprechen Sie deswegen Ihr weiteres Vorgehen mit Ihrem Hiwi ab.

B.8 Debouncer des Spiels Trigger Happy

Sie sollen als Vorbereitung für das Spiel Trigger Happy aus Lab 3 ein Modul `debouncer` entwerfen. Es erhält ein Eingangssignal, welches im späteren Spiel von einem mechanischen Taster stammt und daher störende Schwingungen enthält, d.h. ein einziger Tastendruck führt typischerweise zu mehreren aufeinander folgenden 1-0-Übergängen. Der Modul `debouncer` filtert diese heraus, damit ein Tastendruck nicht als mehrere Tastendrücke interpretiert wird.

In Lab 3 werden von dem Modul zwei Instanzen erzeugt, die sich dann getrennt um die Verarbeitung der Signale von Spieler 1 und Spieler 2 kümmern.

Sehen Sie sich das Framework debouncer.v in Beispiel B.26 an. Es gibt die Struktur des Moduls bereits vor. Gehen Sie beim Vervollständigen des Frameworks wie folgt vor:

Schnittstellen

- Als Input bekommt das Modul die „Rohdaten“ der Taster `PLAYER_IN_RAW`.
- Das Taktsignal ist natürlich ebenfalls ein Input.
- Ebenso ist das low-aktive Reset-Signal ein Input.[1]
- Als Output soll das Modul das entprellte Signal `PLAYER_IN` liefern, welches bei einem gültigen Tastendruck für genau einen Takt lang auf 1 liegen soll.

Modulinterne Variablen

- Eine boolesche Variable soll Auskunft darüber geben, ob der Spieler schon im vorigen Takt den Taster betätigt hat: `PLAYER_IN_REG`.
- Ein 32-Bit-Register soll zählen, wie viele Zyklen seit dem letzten korrekten Klick vergangen sind: `LAST_CLICK`.

Always-Block

- Beginnen Sie mit dem Reset-Fall: Welche Register müssen zurückgesetzt werden?
- Die `if`-Abfrage für den Player: Hier soll das Input-Signal entprellt und so ein korrektes Output-Signal zurückgegeben werden. Überlegen Sie sich dazu die korrekten Bedingungen. Welchen Registern müssen noch neue Werte zugewiesen werden?

Testrahmen

- Schreiben Sie für das Modul einen Testrahmen für zwei Fälle: schnell hintereinander gesendete Signale und Signale mit einer größeren Zwischenpause.

[1] Unser lokales Reset-Signal wird in der späteren Aufgabe an ein globales Reset angeschlossen, das low-aktiv ist. Deswegen sehen wir hier für unser lokales Reset ebenfalls Low-Aktivität vor (d.h. ein Reset erfolgt bei einem Wechsel von 1 auf 0).

```
module debouncer(
// Eingang des Tasters (Rohdaten)
...
// Takteingang (ML310: 100 MHz)
...
// Reset-Eingang (low-aktiv)
...
// Entprellte Daten (Ausgang zum gamecontroller)
...
);

// Wurde bereits geklickt?
...
// Takte nach einem regulärem Klick
...

always @(posedge CLK) begin
  if (~nRESET) begin
    ...
  end
  else begin
    // ML310: Takt hat Frequenz von 100MHz.
    // Nur einmal alle 0,25 Sek. soll ein Input-Signal gültig sein.
    if (... (LAST_CLICK > (100000000 >> 2))) begin
      ...
    end
    // zähle weiter die Takte und vermeide einen Pufferüberlauf
    else if (LAST_CLICK < 100000000) begin
      ...
    end

    // übrige Register
    ...
  end
end

endmodule
```

Beispiel B.26 Vorgabe für den Debouncer

B.9 Universal Asynchronous Receiver

In dieser Aufgabe können Sie einen universellen asynchronen Empfänger implementieren.

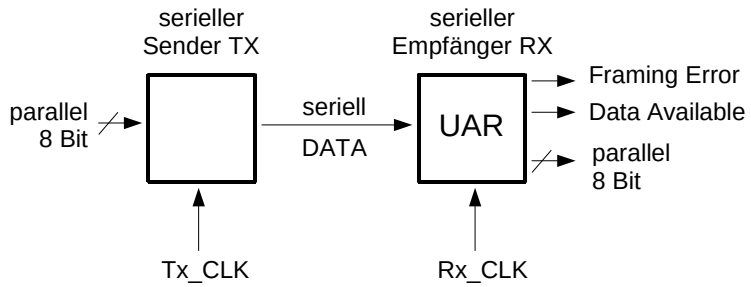


Bild B.27
Schema eines UAR

Bild B.27 zeigt schematisch die Arbeitsweise eines asynchronen seriellen Interfaces, wie es häufig in Computern vorkommt. Ein Sender TX ist mit einem Empfänger RX über einen Übertragungskanal DATA verbunden. Der Sender übermittelt Zeichen

zu je 8 Bit seriell an den Empfänger, also jedes Bit einzeln. Außerdem werden direkt vor einem Zeichen noch ein Start-Bit (DATA=0) und direkt nach einem Zeichen ein Stop-Bit (DATA=1) übertragen.

Die Zeit zwischen dem Ende einer Übertragung und dem Beginn einer neuen Übertragung (also zwischen Stop-Bit des einen Zeichens und Start-Bit des nächsten Zeichens) ist variabel, weswegen wir von einer asynchronen Übertragung sprechen. In dieser Zeit hält der Sender die Übertragungsleitung DATA konstant auf 1.

Sender und Empfänger arbeiten mit unterschiedlichen Taktraten. Der Sender legt zu Beginn jeder positiven Flanke von Tx_CLK ein Bit (Start-, Daten-, oder Stop-Bit) an DATA an. Der Takt Rx_CLK des Empfängers läuft in unserm Fall acht mal so schnell wie Tx_CLK, weswegen jedes gesendete Bit acht mal empfangen wird.

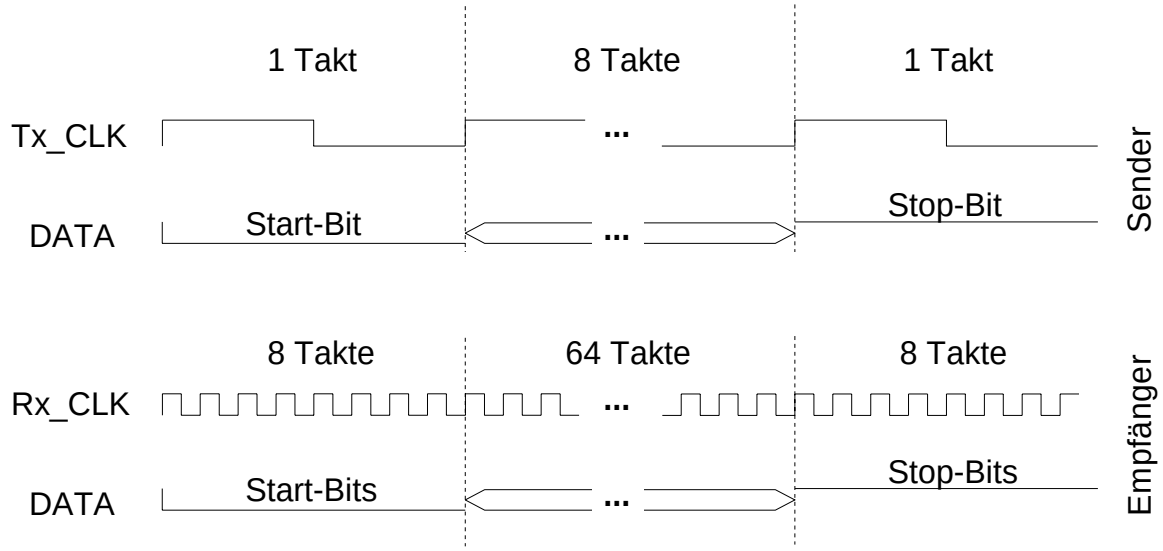


Bild B.28 Signalverlauf der Datenübertragung

Bild B.28 zeigt eine korrekte Datenübertragung. Der Sender überträgt zunächst das Start-Bit, also eine 0. Danach werden die 8 Daten-Bits übertragen und schließlich folgt noch eine 1 als Stop-Bit.

Der Empfänger erkennt das Start-Bit an dem 1-0-Übergang. Um kurzzeitige Spannungssenkungen (*Glitches*) auf dem Datenkanal nicht fälschlicherweise als Start-Bit zu interpretieren, soll ein Start-Bit erst dann als solches erkannt werden, wenn DATA mindestens drei Takte lang auf 0 liegt. Zwölf Takte nach dem 1-0-Übergang des Start-Bits wird das erste Datenbit gelesen (8 Takte Start-Bit + 4 weitere Takte, um zeitlich in der Mitte des ersten gesendeten Daten-Bits zu liegen). Die weiteren Daten-Bits werden jeweils 8 Takte später gelesen. 8 Takte nach dem letzten Daten-Bit bzw. 76 Takte nach dem 1-0 Übergang muss DATA auf 1 liegen (Stop-Bit), damit die Datenübertragung erfolgreich war.

Verilog-Code aus 5 Modulen:

- `start_detect`: erkennt die Startsequenz mit einem Schieberegister.
- `control`: setzt den Receiver in den Running-Modus, nachdem die Startsequenz erkannt wurde.
- `counter`: zählt während des Running-Modus die Takte.
- `ser_par_conv`: konvertiert das serielle Eingangs- in ein paralleles Ausgansgssignal.
- `flags`: setzt die Flags nach dem Stop-Signal.

Vorgehensweise

- Laden Sie sich den vorgegebenen Code von der Übungswebseite herunter und setzen Sie sich mit dem Zusammenhang der Module auseinander.
- Zeichnen Sie ein Blockschaltbild des Modells.
- Vervollständigen Sie den Code, so dass der Receiver korrekt funktioniert.

B.10 RAM mit asynchronem Lese-/Schreibzugriff

In Beispiel B.29 sehen Sie das Grundgerüst für einen RAM mit asynchronem Lese- und Schreibzugriff. Ihre Aufgabe ist es, diesen zu vervollständigen und zu testen.

```
`timescale 1ns / 1ps

module ram_sp_ar_aw(
  inout wire [7:0] DATA,
  input wire [7:0] ADDRESS,
  input wire       WRITE_ENABLE,
  input wire       OUTPUT_ENABLE
);

// Interne Variablen

// Tri-State Pufferkontrolle

// Speicher - schreiben

// Speicher - lesen

endmodule
```

Beispiel B.29 Vorgabe für den RAM

Ergänzen Sie dazu die fehlenden internen Variablen und schreiben Sie die beiden `always`-Blöcke, die den RAM beschreiben und auslesen, sowie den Tri-State Buffer, der den `inout`-Port `DATA` beschreibt.

`DATA` dient als Schnittstelle, die Daten zum Schreiben in den Speicher liefert bzw. beim Lesen aus dem Speicher erhält. Geschrieben und gelesen wird im Speicher an der Stelle `ADDRESS`, allerdings nur bei bestimmten Belegungen der Input-Wires. Überlegen Sie sich, welche Belegungen diese sind.

B.11 Teilbarkeit durch 3

Entwerfen Sie ein Modul `divide_by_3`, das prüft, ob die Anzahl der Takte, die in Ihrer Simulation vergehen, durch 3 teilbar ist. Der Testrahmen ist bereits vorgegeben. Ein Reset ist ebenfalls vorgesehen. Es sollen alle vollständigen Takte nach dem Reset gezählt werden.

Vorgehensweise

- Inputs: `wire CLK`, `RESET`, Output: `wire DIVABLE_BY_3`
- Zählen Sie zur Kontrolle in einem weiteren Register `COMPLETE_CLOCK` komplette Takte mit.

```
module divide_by_3_test();

reg RESET, CLK;
wire DIVABLE_BY_3;

divide_by_3 uut(
  .CLK(CLK),
  .RESET(RESET),
  .DIVABLE_BY_3(DIVABLE_BY_3)
);

initial begin
  CLK = 0;
  RESET = 0;
  #2; RESET = 1;
  #2; RESET = 0;
  //wählen Sie eine dieser Wartezeiten aus:
  #20;
  //#22;
  //#25;
  $display("DIVABLE_BY_3: %b", DIVABLE_BY_3);
  $stop;
end

always #1 CLK = ~CLK;

endmodule
```

Beispiel B.30 Testrahmen des Moduls `divide_by_3`

B.12 Kontrollflussgraph

In dieser Aufgabe sollen Sie den Kontrollflussgraphen aus Bild B.31 realisieren:

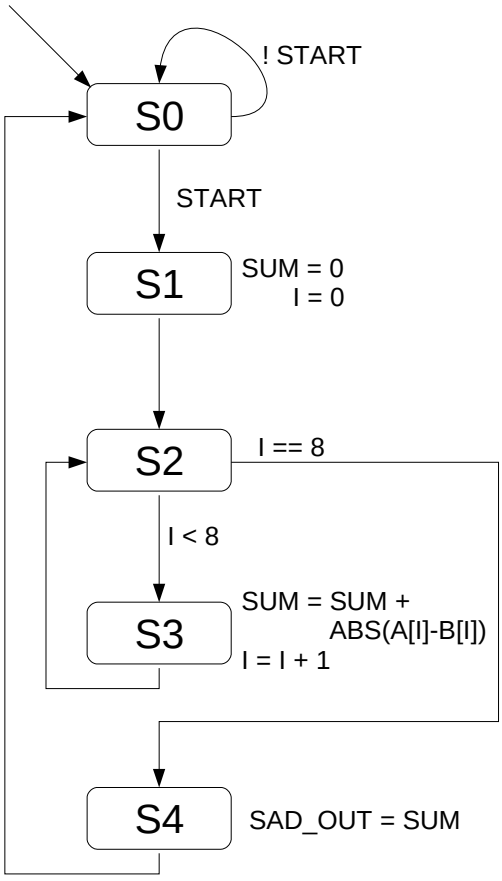


Bild B.31
Kontrollflussgraph

Im Kernzustand S3 summiert der Graph die Summe absoluter Differenzen (SAD) aus zwei Arrays A und B (jeweils 8-elementig, jeweils 8 Bit pro Element) auf. Hierbei sollen die beiden Arrays jeweils aus einem Text-File eingelesen werden:

```
MemA.txt:
00 FF A1 04 B3 58 07 CA
MemB.txt:
00 FE A1 02 B3 12 07 CA
```

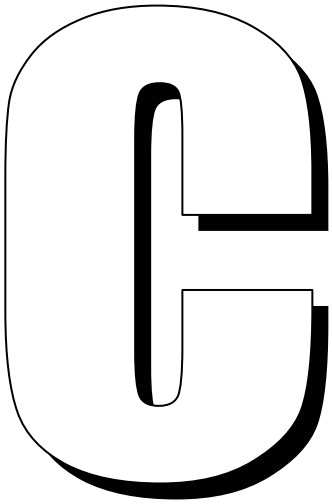
Vorgehensweise

- Berechnen Sie das Ergebnis zuerst manuell.
- Die Funktion ABS in S3 soll als Verilog-Funktion ausgelagert werden.
- Ein Reset soll alle relevanten Variablen wieder auf Null zurücksetzen und den Startzustand ansteuern.
- Schreiben Sie einen Testrahmen mit der Taktperiode 20 ns.

- Lassen Sie sich abschließend im Waveform-Fenster die Integer-Variable SUM anzeigen.

Erweiterung um ein einfaches ROM

- Erweitern Sie nun Ihr Modul um einen einfachen lesbaren Speicher. Fügen Sie also ein drittes Modul SADMem hinzu, das als Input die Adresse, an der zu lesen ist, bekommt, und als Output das Datum, das an dieser Stelle im Speicher-Array liegt.



Lab 2: Logiksynthese mit VERILOG

In diesem zweiten Lab vollziehen Sie den Übergang vom VERILOG-Modell zum Gattermodell. Dabei wird das VERILOG-Modell durch einen Synthese-Compiler automatisch in eine Liste aus Logikgattern (AND, OR usw.) und Registern (FlipFlop, Latch) übersetzt, die miteinander so verdrahtet sind, dass sie die in VERILOG beschriebenen Berechnungen durchführen. Dies ist bekanntlich die *Logiksynthese* (Kapitel 5). Ein erstes Teilergebnis ist eine so genannte *Zwischendarstellung*, die Sie grafisch betrachten und im nächsten Lab auf eine Zieltechnologie abbilden können, etwa ein FPGA Virtex-II Pro auf dem ML310 (Kapitel 6, 7). Bereits an der Zwischendarstellung können Sie abschätzen, mit welcher Taktfrequenz die spätere Hardware voraussichtlich betrieben werden kann.

C.1 Synthese eines Addierers

Zunächst legen Sie wie zuvor ein ISE-Projekt an und fügen ein VERILOG-Modul hinzu. Anschließend begnügen Sie sich nun aber nicht mit der Simulation allein, sondern führen eine Logiksynthese durch und untersuchen das Ergebnis.

C.1.1 ISE-Projekt, VERILOG-Modul und Verhaltenstest

Starten Sie den Projekt-Navigator wie gewohnt und legen Sie ein neues Projekt an. Beachten Sie dabei, dass in der Maske Select the Device and Design Flow for the Project (Bild C.1) als Synthese-Tool XST (VHDL/Verilog) ausgewählt ist.

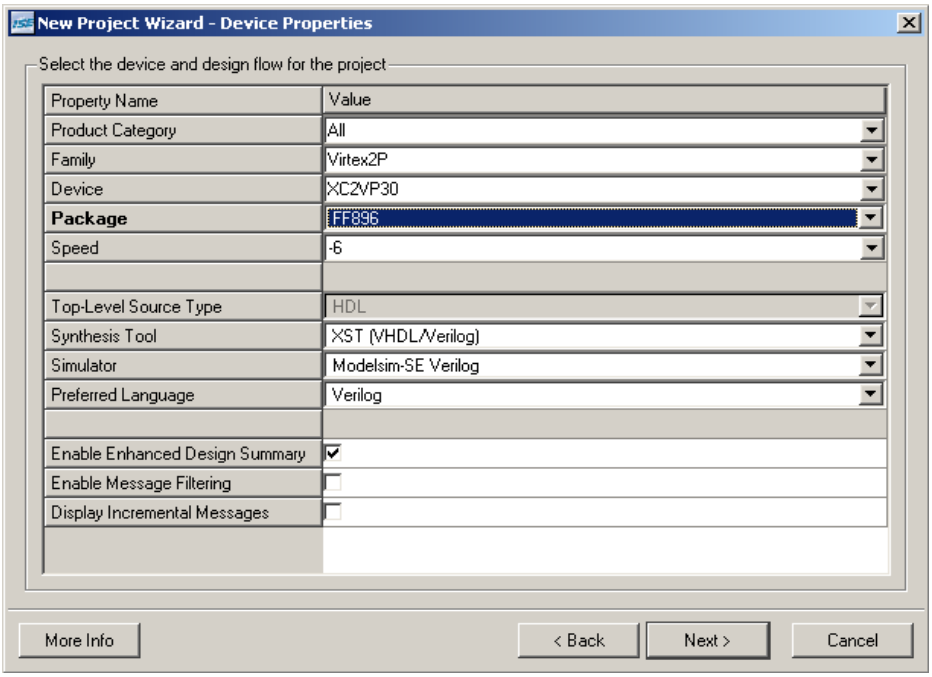


Bild C.1 Anlegen eines ISE-Projekts; Wahl des Synthesewerkzeugs XST

```
`timescale 1ns / 1ps

// getakteter Addierer für 4 positive Zahlen je 8 Bit
module adder (
  input  wire       CLK,
  input  wire [7:0] A,
                    B,
                    C,
                    D,
  output reg  [9:0] OUT
);

always @(posedge CLK)
  OUT <= A + B + C + D;

endmodule
```

Beispiel C.2 Getakteter Addierer für vier Zahlen

Erzeugen Sie sich eine VERILOG-Datei `adder.v` mit dem Quelltext aus Beispiel C.2 sowie einen zugehörigen Testrahmen als Datei `test_adder.v` mit dem Quelltext aus Beispiel C.3 (Download auf der Übungs-Webseite). Das Modul `adder` summiert `A`, `B`, `C` und `D` und weist das Ergebnis zu jeder steigenden Taktflanke dem Ausgangsregister `OUT` zu.

```
`timescale 1ns / 1ps
`define CLK_HALF  5
`define CLK_FULL 10

// Testrahmen fuer den adder
module test_adder;
  reg CLK;                                  // Inputs
  reg [7:0] A, B, C, D;
  wire [9:0] OUT;                           // Outputs

  // Instanz des Prueflings
  adder Adder (CLK, A, B, C, D, OUT);

  // Clock erzeugen
  always begin
    CLK = 1; #`CLK_HALF;
    CLK = 0; #`CLK_HALF;
  end

  // Ueberwachung und Stimuli
  initial begin                             // Ueberwachung
    $display("TIME CLK |  A    B    C    D   OUT");
    $display("=================================");
    $monitor("%.4d %.3b | %.4d %.4d %.4d %.4d %.3d",
             $time, CLK, A, B, C, D, OUT);

    #`CLK_HALF;                             // Stimuli
    A = 1; B = 1; C = 1; D = 1; #`CLK_FULL;
    A = 2; B = 2; C = 2; D = 2; #`CLK_FULL;
    A = 3; B = 4; C = 5; D = 6; #`CLK_FULL;
                                #`CLK_FULL;
    $stop;
  end
endmodule
```

Beispiel C.3 Testrahmen für den Addierer aus Beispiel C.2

Führen Sie zunächst wie gewohnt eine Verhaltenssimulation mit ModelSim durch und überzeugen Sie sich von der korrekten Funktionalität des Addierers.

C.1.2 Logiksynthese

Nachdem der Addierer zufriedenstellend funktioniert, führen Sie eine Logiksynthese durch. Wechseln Sie dazu unter Sources for in den Modus Implementation. Aktivieren Sie dann im Quellenfenster Ihr Modul `adder` durch einmaliges Anklicken und betrachten Sie die im Prozessfenster unter Synthesize – XST angezeigten Prozesse (Bild C.4).

Um die Logiksynthese durchzuführen, klicken Sie lediglich den Prozess Synthesize – XST doppelt an. Nach kurzer Wartezeit sollten zwei grüne Häkchen im Prozessfenster erscheinen, die die Fertigstellung der Synthese und ihres Berichts signalisieren. Ein drittes grünes Häkchen erscheint beim Doppelklick auf Check Syntax (Bild C.5). Falls Fehler auftreten (rote Häkchen), können die Fehlermeldungen im ISE-Konsolenfenster betrachtet und nötigenfalls der HiWi hinzugezogen werden.

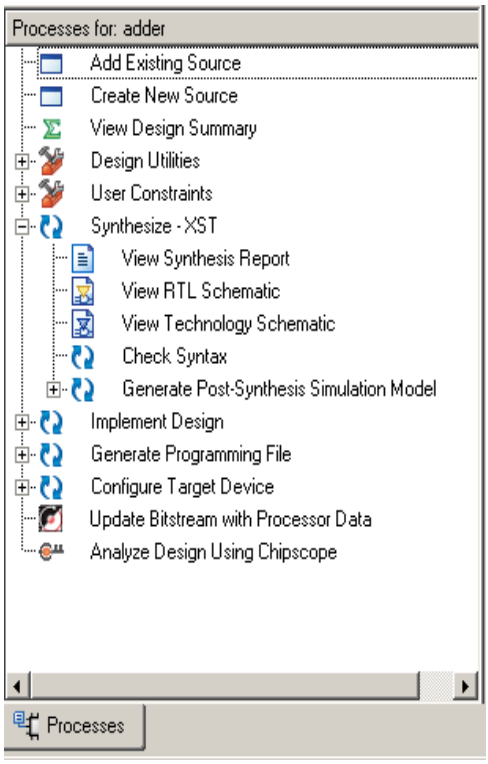


Bild C.4
Synthese-relevante ISE-Prozesse

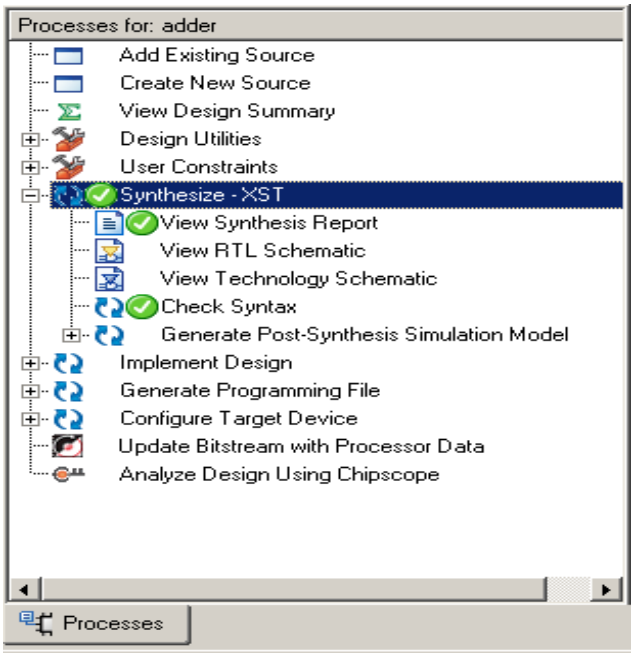


Bild C.5
Die Synthese war erfolgreich

C.1.3 Syntheseergebnis untersuchen

Wirklich spannend wird die Logiksynthese natürlich erst, wenn wir auch das Ergebnis betrachten. Auch hierzu gibt es einen entsprechenden Punkt in der Prozessliste, nämlich View RTL Schematic. Nach Doppelklick auf diesen Prozess erscheint ein Ansichtsfenster, in dem Sie zunächst nur eine fensterfüllende Blackbox für Ihren

Addierer sehen. Um die Zwischendarstellung der synthetisierten Schaltung zu betrachten, klicken Sie doppelt auf diese Blackbox (**Bild C.6**).

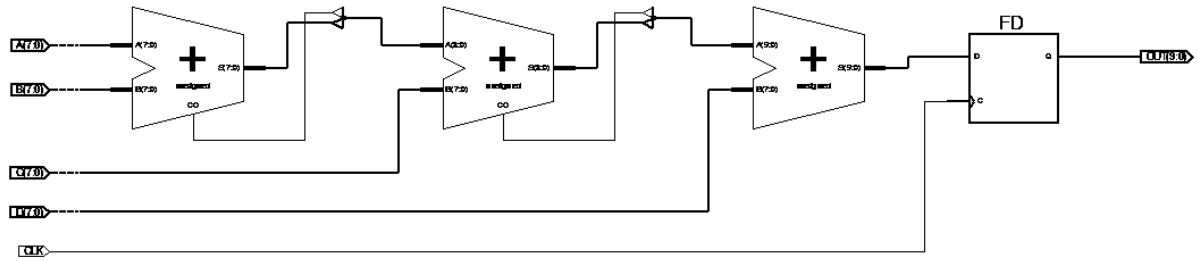


Bild C.6 RTL-Ansicht des Designs

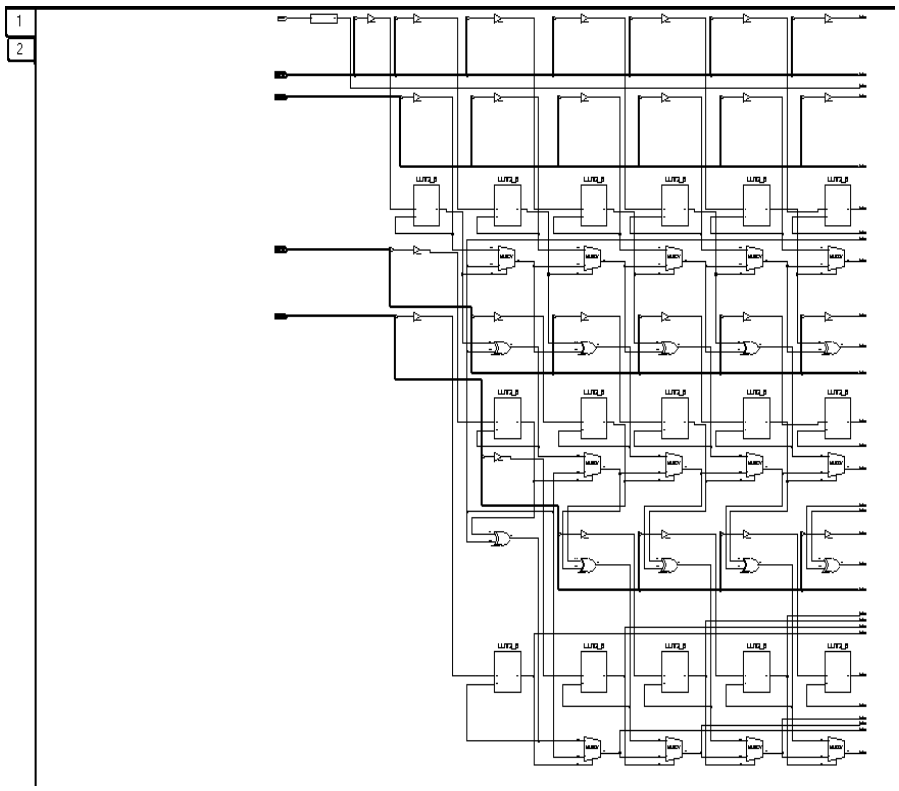


Bild C.7 Technologie-Ansicht

Man erkennt deutlich den Addierer mit seinen vier Eingängen und das D-Flipflop, das die Summe flankengetriggert speichert und an den Ausgang `OUT` anlegt. Was Sie hier sehen, entspricht der Zwischendarstellung vor dem Abbilden auf die Zieltechnologie (Kapitel 5). Wie die VERILOG-Modelle ist auch die Zwischendarstellung hierarchisch geschachtelt. Durch Doppelklick auf die dargestellten Elemente kann man häufig noch tiefer in die verborgenen Details eintauchen und so beispielsweise den genauen Aufbau der einzelnen Addiererbausteine untersuchen. Das geht nicht für

Elemente (z.B. Addierer) der Technologiebibliothek des verwendeten Bausteins, da dies die kleinsten darstellbaren Einheiten sind.

Nachdem Sie sich mit der Zwischendarstellung vertraut gemacht haben, möchten wir Ihnen noch einen Vorgeschmack auf die endgültige Hardware-Schaltung geben, wie sie nach dem (noch nicht durchgeführten) Technology-Mapping für ein FPGA aussehen könnte (FPGAs werden in Kapitel 6 behandelt). Schließen Sie dazu die RTL-Ansicht und starten Sie stattdessen den Prozess View Technology Schematic. Dadurch öffnet sich das Technologie-Ansichtfenster wie in Bild C.7.

Die Technologie-Ansicht zeigt eine Abbildung Ihres Designs auf die FPGA-Technologie. Das Netz aus Logikgattern wurde optimiert und in Lookup-Tabellen und Multiplexer für die Programmierung des FPGA umgewandelt. Außerdem wurden an den Ein- und Ausgängen Puffer eingefügt. Da die Technologie-Ansicht nicht auf eine Bildschirmseite passt, ist sie auf mehrere Karten aufgeteilt, die Sie über die Reiter am linken oberen Rand erreichen können.

Normalerweise kommen Sie mit der Technologie-Ansicht nicht in Berührung, denn die enthaltenen Details sind nur in Spezialfällen interessant, bei denen man eine Schaltung von Hand im Gattermodell optimieren möchte.

C.1.4 Maximale Taktrate

Zum Abschluss unseres Synthese-Lab wollen wir bestimmen, wie hoch die maximale Taktrate ist, mit der wir den Addierer voraussichtlich betreiben dürfen, wenn wir ihn in einem FPGA als Hardware realisieren (was wir im nächsten Lab auch tatsächlich tun werden). Öffnen Sie hierzu in der Prozessansicht View Synthesis Report. Dieser Bericht fasst verschiedene Informationen über Ihre Schaltung zusammen, die während der Logiksynthese notiert wurden.

Scrollen Sie im Textfenster nach unten bis zur Sektion Timing Summary, um die Timing-Analyse zu betrachten. Der Synthese-Compiler listet hier die längsten Datenpfade Ihrer Schaltung auf und gibt die nötige Rechenzeit für diese *kritischen Pfade* an. Genau genommen werden die kritischen Pfade vier verschiedener Messbereiche angegeben: (1) von Registern zu anderen Registern, (2) von FPGA-Input-Pins zu Registern, (3) von Registern zu FPGA-Output-Pins und (4) von FPGA-Inputs zu FPGA-Outputs. Der gesamte kritische Pfad ist das Maximum dieser vier Werte. Bei unserem Addierer erhalten wir einen Wert von 7,675 ns. Dies gibt uns einen ersten Hinweis[1] auf die zu erwartende maximale Taktrate unseres Addierers auf dem FPGA, nämlich 1/7,675 ns = ca. 130 MHz.

[1] Die vom Synthese-Tool ermittelten Zeiten setzen sich aus tatsächlichen Gatterverzögerungen und geschätzten Verdrahtungsverzögerungen zusammen. Letztere können zu diesem Zeitpunkt im Design-Flow noch nicht exakt bestimmt werden, da noch gar keine Platzierung und Verdrahtung stattgefunden hat. Exakte Werte lassen sich mit dem TimingAnalyzer (nach Platzierung und Verdrahtung) ermitteln.

Übrigens gelten die ermittelten Werte für das FPGA Xilinx Virtex-II Pro V2P30, das Sie in dem nun folgenden dritten Lab einsetzen werden. Wenn Sie die Logiksynthese für eine andere Zieltechnologie durchführen möchten, müssen Sie lediglich die Projekt-Optionen für Ihr Addierer-Projekt entsprechend umstellen.

C.2 Logiksynthese-Quiz

Ob Sie sich schon für den Job eines Reverse-Engineer für Logiksynthese bewerben können, sollen Sie nun in unserem spannenden Logiksynthese-Quiz unter Beweis stellen!

In Bild C.8 bis Bild C.11 sehen Sie Zwischendarstellungen für verschiedene einfache Register-Transfer-Logiken. Schreiben Sie für jede dargestellte Schaltung ein VERILOG-Modell, dessen Logiksynthese mit ISE genau die gleiche Zwischendarstellung liefert (ISE-Prozess View RTL Schematic). Zeigen Sie Ihre Ergebnisse dem HiWi!

Hinweise: Die fünfeckigen Kästen mit Spitze sind die Ein- und Ausgänge („Bus-Taps“) der Schaltung.

Sie können alle vier Verilog-Module in einem ISE-Projekt speichern. Um das RTL-Schematic eines Moduls zu betrachten, muss dieses dann zunächst zum Top-Level-Modul bestimmt werden. Dies geschieht im Quellenfenster über Rechtsklick auf das Modul und Set as Top Module.

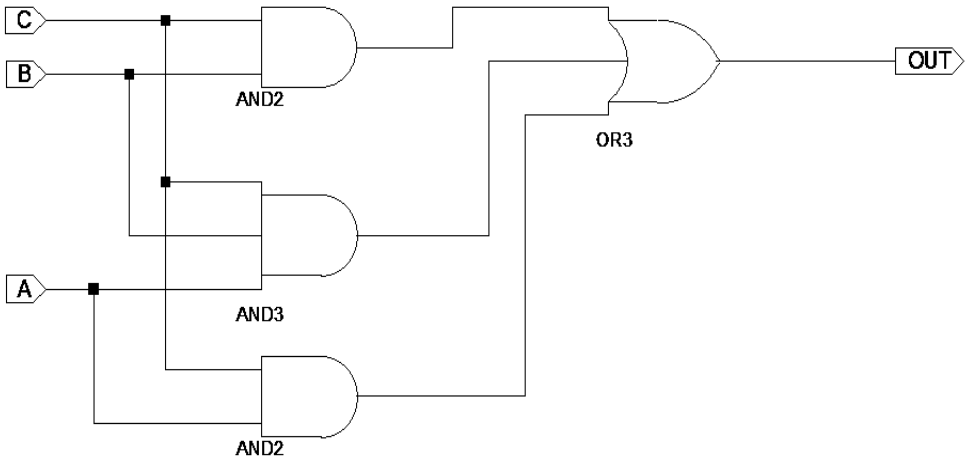


Bild C.8
Logiksynthese-Quiz: Aufgabe 1

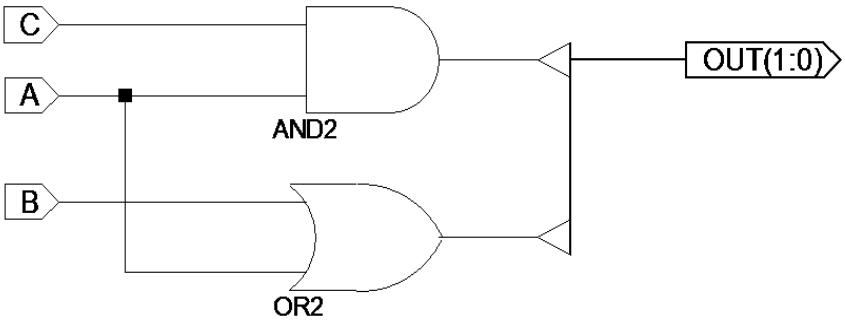


Bild C.9 Logiksynthese-Quiz: Aufgabe 2

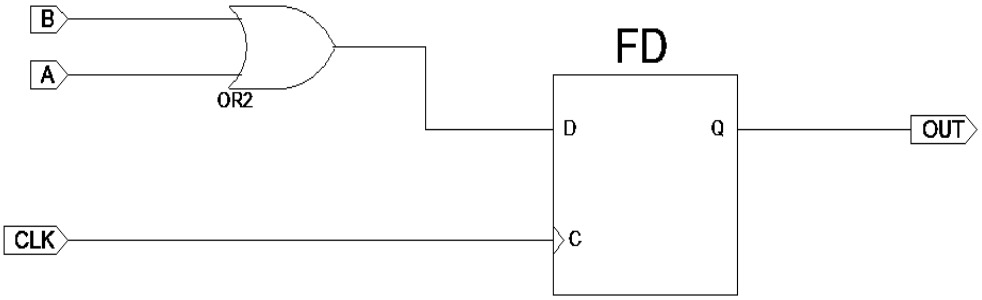


Bild C.10 Logiksynthese-Quiz: Aufgabe 3

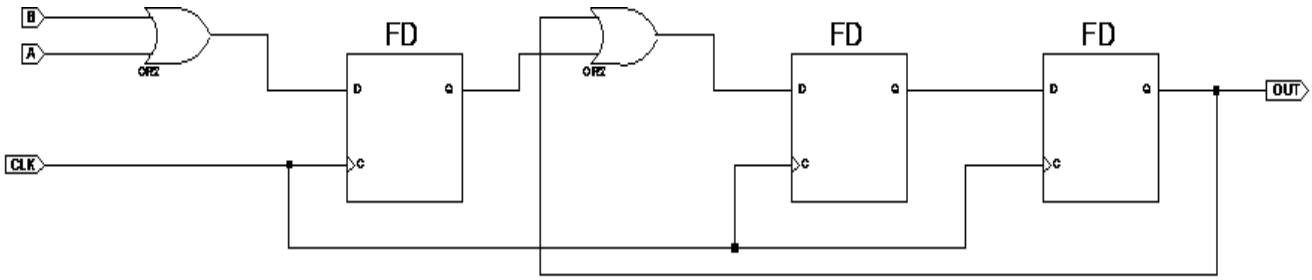


Bild C.11 Logiksynthese-Quiz: Aufgabe 4

C.3 Abschlussquiz

Auch in diesem Quiz gilt wieder: Bearbeiten Sie die Fragen bitte in Ruhe alleine. Sie können sich ihre Antworten in Stichworten notieren oder auch nur grob im Kopf überlegen. Wenn jeder Ihrer Übungspartner fertig ist, sprechen Sie Ihren HiWi an, und er wird Ihnen Ihre Antworten abnehmen.

Frage 1

Was ist Logiksynthese?

Frage 2

Wovon ist die Logiksynthese abhängig?

Frage 3

Was kann nicht synthetisiert werden? Nennen Sie Beispiele.

Frage 4

Wie berechnet man die maximale Taktfrequenz einer Schaltung bei einem kritischen Pfad der Signallänge x?

D

Lab 3: Hardware-Software-Codesign auf dem ML310

In diesem etwas größeren Lab verschaffen Sie sich einen Einblick in den praktischen Entwurf von komplexen eingebetteten Systemen. Das aus der Vorlesung bekannte Entwicklungs-Board Xilinx ML310, das Sie jetzt praktisch ausprobieren, ist so vielseitig, dass Sie selbst mit der Entwicklung eines DVB-T-Empfängers nicht alle Möglichkeiten ausschöpfen würden. Stattdessen begnügen wir uns mit einem sehr einfachen Beispiel, um wichtige Konzepte eines modernen Hardware-Software-Codesign exemplarisch kennenzulernen.

In Abschnitt D.1 betten Sie eigene in VERILOG geschriebene Hardware-Module in ein Gesamtsystem ein und verifizieren ihre Funktion auf dem ML310 im laufenden Betrieb. Hier machen Sie sich Schritt für Schritt mit der Entwicklungsumgebung XPS (Xilinx-Platform-Studio) und dem zugehörigen Logikanalysator Chipscope[1] vertraut.

In Abschnitt D.2 geht es um die wichtige Schnittstelle von Software nach Hardware. Die Argumente für eine Berechnung werden vom Software-Prozessor über einen DCR-Bus an einen Hardware-Addierer auf dem FPGA übergeben; das Ergebnis wird dann vom Logikanalysator angezeigt.

In Abschnitt D.3 wird zusätzlich die entgegengesetzte Schnittstelle von Hardware nach Software eingeführt. Ein von Ihnen entwickeltes Programm liest das Berechnungsergebnis der Hardware über den DCR-Bus zurück und gibt es auf einem LCD sowie über die serielle Schnittstelle aus.

In Abschnitt D.4 werden Sie mit Zeitmessungen untersuchen, ob und wie stark Software-Programme durch Auslagerung rechenintensiver Funktionen in Hardware tatsächlich beschleunigt werden können.

In Abschnitt D.5 schließlich werden Sie das nervenaufreibende Spiel *Trigger Happy* auf dem ML310 realisieren.

---

[1] Genauer: ChipscopePro-Analyzer

D.1 Hardware „bauen“ und testen

In diesem ersten Teil machen Sie sich vorrangig mit der Entwicklungsumgebung für das ML310 vertraut. Auf dem Entwicklungsrechner in Bild D.1 entwickeln Sie die Software und Hardware für die Zielplattform ML310, ohne letztere tatsächlich zu benutzen. In den Abschnitten D.1.1 bis D.1.4 benutzen Sie zunächst das VERILOG-Modell eines Addierers stellvertretend für eine beliebige Schaltung, die auf dem ML310-FPGA als Hardware-Schaltung realisiert werden soll.

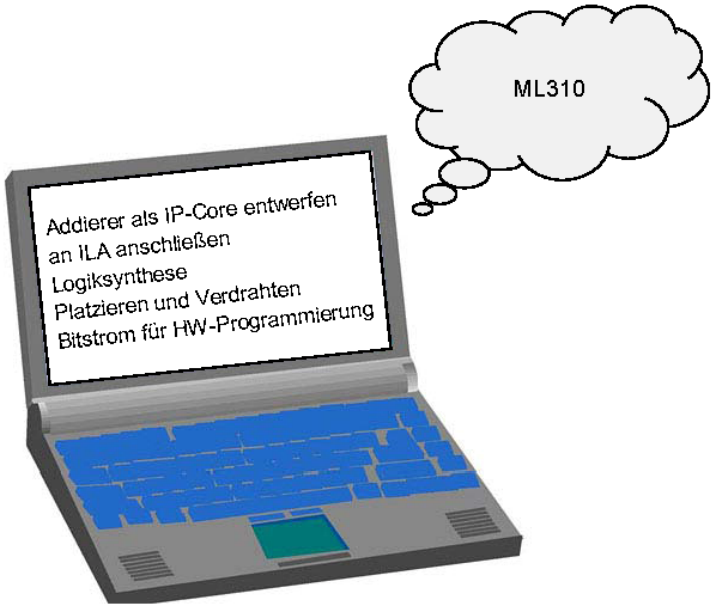


Bild D.1 Hardware-Entwicklung unter XPS

Außerdem setzen Sie einen Logikanalysator ein, der ähnlich wie ein Videorekorder Signalverläufe zu Testzwecken eine Zeit lang aufzeichnen kann. Dabei fasziniert, dass ein solcher Logikanalysator nicht als separates Messgerät in einem eigenen Gehäuse an irgendwelche Testausgänge der Zielplattform angeschlossen wird, sondern dass Sie diesen *integrierten* Logikanalysator (ILA) als bereits fertige Bibliotheksschaltung mit Ihrer eigenen Schaltung zu einer Gesamtschaltung verbinden, die später dann als Ganzes in das FPGA hineingeladen wird.

Im Einzelnen finden Sie auf der Übungs-Webseite die VERILOG-Quellen eines Addierers und ein vorbereitetes Xilinx-Platform-Studio-Projekt. Sie fügen den Addierer und den Logikanalysator in das XPS-Projekt ein und nehmen die nötigen Verdrahtungen vor. Vollautomatisch wird Ihr Modell synthetisiert, platziert und verdrahtet sowie in einen *Bitstrom* für die Programmierung bzw. Konfiguration des FPGAs umgewandelt.

Erst in Abschnitt D.1.5 wird dieser Bitstrom dann tatsächlich in die Zielplattform geladen und in D.1.6 dort getestet.

D.1.1 VERILOG-Modell eines Addierers

Wir wollen zunächst den Quellcode des Addierers betrachten. Entpacken Sie das Archiv `d.1_hardware_bauen_und_testen.zip` und öffnen Sie die Datei `verilog/adder.v` (Beispiel D.2) in einem Texteditor.

```
module adder (
  input  wire [31:0] IN,        // Summanden (A: Bits 15:0; B: Bits 31:16)
  output wire [31:0] OUT        // Summe
);

wire [15:0] A = IN [15: 0];     // Aufschluesselung der Summanden in A
wire [15:0] B = IN [31:16];     // und B

assign OUT = {15'b0, A + B};    // A und B kombinatorisch addieren;
                                // nichtbenoetigte Bits mit Nullen fuellen
endmodule
```

Beispiel D.2 VERILOG-Code des Addierers

Die beiden Summanden A und B werden dem Addierer durch das Signal IN übergeben. OUT wird per ständiger Zuweisung die 17-Bit-Summe zugeordnet, wobei die oberen, nicht benötigten 15 Bits zu 0 gesetzt werden.

Bild D.3 zeigt die Waveform einer Verhaltenssimulation des Addierers. Die beiden hexadezimalen Summanden `0x6` und `0xA` werden korrekt zu `0x10` summiert.

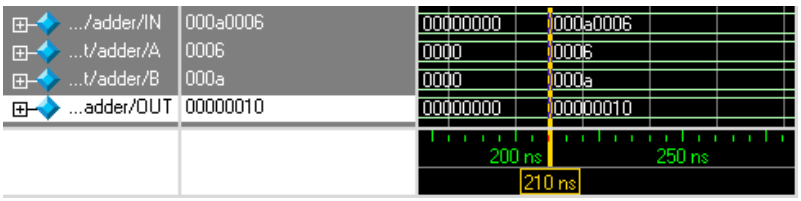


Bild D.3 Verhaltens-simulation des Addierers

Die zweite VERILOG-Datei `aplusb_hw.v` bildet einen „Wrapper“ um den Addierer, der die (in der ersten Version noch fest verdrahteten) Summanden `0x6` und `0xA` liefert.

D.1.2 In XPS einen IP-Core erzeugen und einbinden

Öffnen Sie das vorbereitete XPS-Projekt durch Doppelklicken der Datei `xps/system.xmp`. Sollte das Programm nicht starten, können Sie dieses auch über Startmenü | Alle Programme | Xilinx ISE Design Suite | EDK | Xilinx Platform Studio

starten und über Open a recent Project Ihr Projekt öffnen. Sie sehen nun die XPS-Oberfläche wie in Bild D.4.

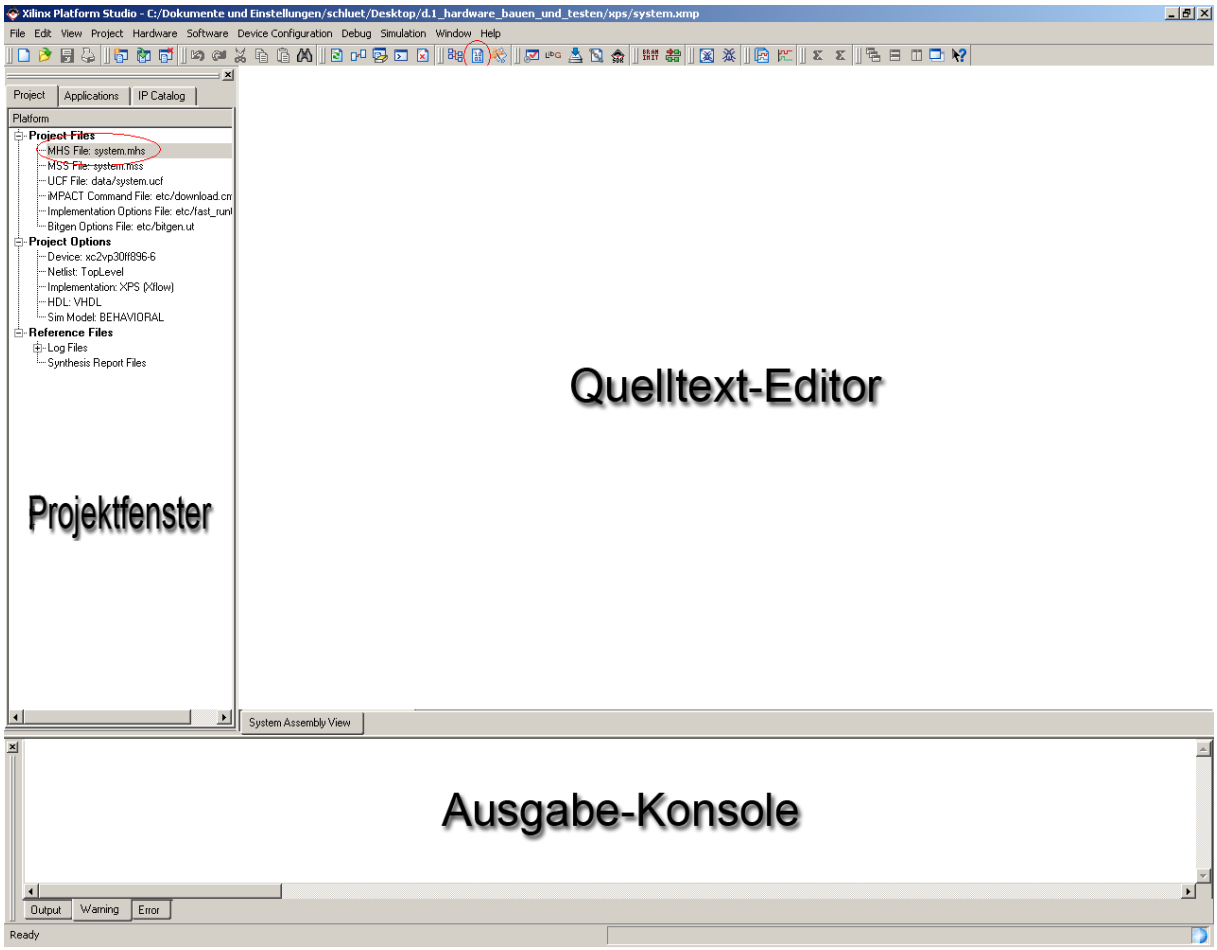


Bild D.4 Die XPS-Benutzeroberfläche

Im linken Projektfenster haben Sie u.a. Zugriff auf die Hardware-Komponenten Ihres Systems, können Module miteinander verdrahten und Projektoptionen einstellen. Den Quelltext-Editor auf der rechten Seite können Sie zum Betrachten und Ändern von projektspezifischen Dateien verwenden, während die Ausgabe-Konsole Statusmeldungen, Warnungen und Fehler ausgibt (z.B. während einer Synthese). Die beiden roten Markierungen heben zwei sehr wichtige Funktionen hervor: die Bitstrom-Generierung (oberer Button) und die Anzeige der Hardware-Modulinstanzen und ihrer Verdrahtungen (untere Ellipse).

Um den Addierer in das Projekt einzubauen, muss zunächst ein Intellectual-Property-Core (IP-Core) aus dem Addierer erstellt werden. IP-Cores dienen dazu, benutzerspezifische Schaltungen (HDL-Code wie auch Netzlisten) und deren Schnittstellen in einem einheitlichen Format zu beschreiben. Klicken Sie auf den Button Create or Import Peripheral (Bild D.5). Der sich öffnende Dialog in Bild D.6 fragt Schritt für Schritt die Einstellungen für Ihren IP-Core ab.

Bild D.5 Create or Import Peripheral

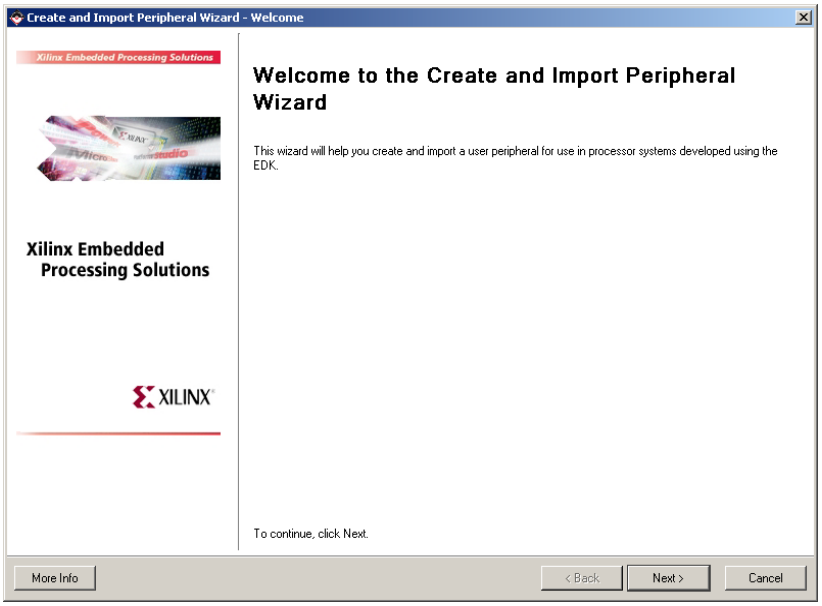


Bild D.6
Dialog zum Erstellen eines IP-Cores

Klicken Sie auf Next. Im nächsten Dialog wählen Sie Import existing peripheral und klicken Next. Betätigen Sie im dritten Dialog den Next-Button, ohne irgendwelche Einstellungen zu verändern. Im vierten Dialog (Import Peripheral – Name and Version, Bild D.7) geben Sie den Namen des Top-Level-Moduls Ihres einzubindenden Codes an, in diesem Fall aplusb_hw. Durch Anklicken von Use Version können Sie eine Versionsnummer für diesen IP-Core festlegen.

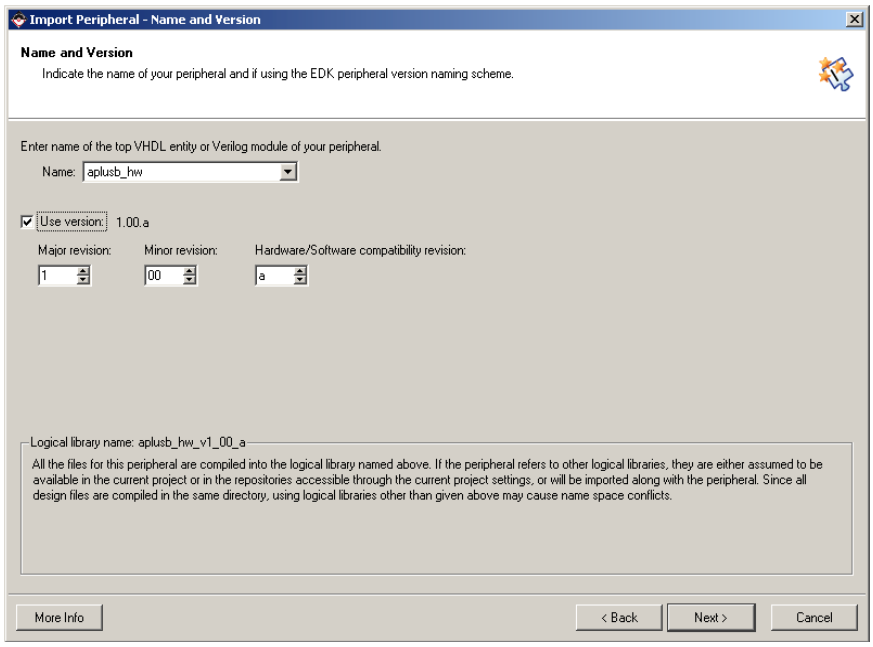


Bild D.7
Namen des Top-Level-Moduls sowie Versionsnummer festlegen

Wählen Sie im nächsten Dialog (Source File Types) den Punkt HDL Source Files (*.vhd, *.vhdl, *.v, *.vh). In Schritt HDL Source Files in Bild D.8 stellen Sie bitte VERILOG als Sprache ein und wählen den Punkt Browse to your HDL source and dependent library files (*.vhd, *.vhdl, *.v, *.vh) in next step.

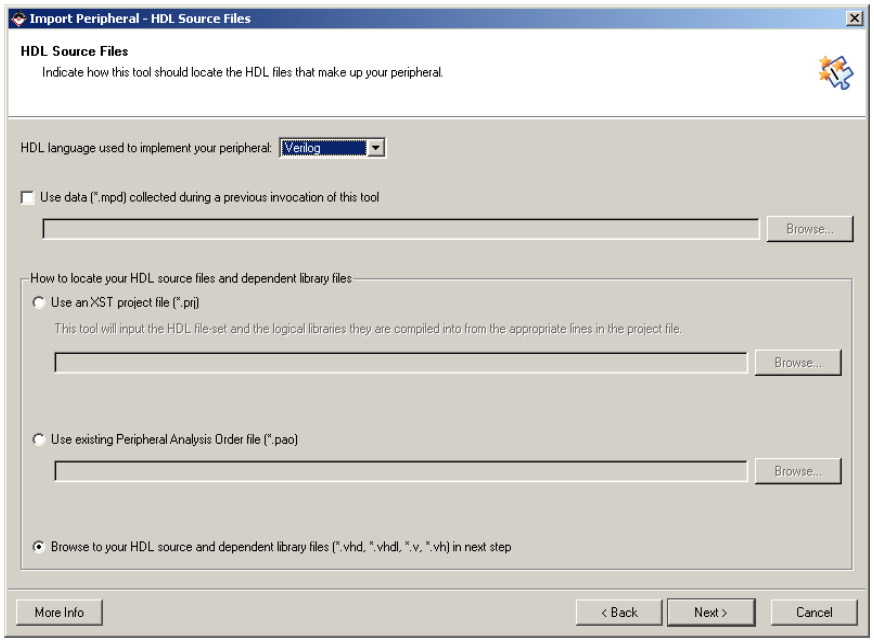


Bild D.8
HDL und Art der Quelldateien festlegen

Im Schritt HDL Analysis Information fügen Sie die beiden VERILOG-Dateien `adder.v` und `aplusb_hw.v` zum Core hinzu. Im Schritt Bus Interfaces müssen Sie den Haken bei Select bus interface(s) entfernen. Entfernen Sie im nächsten Schritt Identify Interrupt Signals lediglich den Haken bei Select and configure interrupt(s), klicken im Schritt Port Attributes einfach auf Next und im letzten Fenster endlich auf Finish. Alle erstellten IP-Cores werden im Unterverzeichnis pcores Ihres Projektverzeichnisses gespeichert; dort befindet sich für den Addierer nun das neue Verzeichnis `aplusb_hw_v1_00_a`, zusammengesetzt aus IP-Name und Versionsnummer.

Herzlichen Glückwunsch – Sie haben Ihren ersten IP-Core selbst erstellt (und können ihn nun theoretisch an Interessenten in aller Welt verteilen)! Fügen Sie als nächstes eine Instanz Ihres IP-Cores zu Ihrem Projekt hinzu, damit Sie den Hardware-Addierer in einem Hardware-Software-Gesamtsystem nutzen können.

Klicken Sie im Projektfenster auf den Reiter IP-Catalog, dann auf USER und nach einem Rechtsklick auf den IP-Core aplusb_hw auf AddIP.

Abgabe 1

Zeigen Sie Ihrem Hiwi den in Ihr XPS-Projekt eingebundenen IP-Core aplusb_hw.

D.1.3 Verdrahtung des Addierers

Der Addierer ist nun im FPGA-Modell vorhanden, jedoch noch nicht mit der Außenwelt verbunden. Die Außenwelt besteht in diesem Fall aus einer weiteren Teilschaltung, dem oben erläuterten integrierten Logikanalysator, mit dem Sie die Arbeit des Addierers im laufenden Betrieb überwachen können.

Klicken Sie im Projektfenster im Reiter Project doppelt auf den Eintrag MHS File: system.mhs. Wenn Sie nun im Quelltext-Editor ganz nach unten scrollen, sehen Sie die Instanzierung des Addierers wie in Beispiel D.9.

```
BEGIN aplusb_hw
 PARAMETER INSTANCE = aplusb_hw_0
 PARAMETER HW_VER = 1.00.a
END
```

Beispiel D.9 Instanzierung des Addierer-Cores in der Datei `system.mhs`

Das Top-Level-Modul `aplusb_hw.v` des Addierers stellt über die beiden Ausgänge `DBG_IN` und `DBG_OUT` die Summanden bzw. die Summe der Addition zur Verfügung. Fügen Sie wie in Beispiel D.10 zwei Zeilen Code ein und speichern Sie mit Strg-s.

```
BEGIN aplusb_hw
 PARAMETER INSTANCE = aplusb_hw_0
 PARAMETER HW_VER = 1.00.a
 PORT DBG_IN = APLUSB_DBG_IN
 PORT DBG_OUT = APLUSB_DBG_OUT
END
```

Beispiel D.10 Verdrahteter Addierer-Core

Der Anschluss `DBG_IN` des Addierers steht nun für die anderen Instanzen, unter anderem den Logikanalysator, in der Datei `system.mhs` unter dem Namen `APLUSB_DBG_IN` zur Verfügung, der Anschluss `DBG_OUT` entsprechend unter `APLUSB_DBG_OUT`.

D.1.4 Bitstrom für die Hardware-Programmierung

Erzeugen Sie einen Bitstrom mit dem in Bild D.4 hervorgehobenen Button (untere der drei Symbolleisten in XPS). Ihr Design wird nun vollautomatisch synthetisiert, auf die Zielplattform ML310 abgebildet, platziert, verdrahtet und in einen Bitstrom zur

FPGA-Konfiguration verpackt. Während des etwa 5-minütigen Vorgangs sehen Sie im Konsolenfenster die Kontrollausgaben der jeweils aktuellen Bearbeitungsphase. Die Bearbeitung endet mit einer Ausgabe wie in Beispiel D.11.

```
DRC detected 0 errors and 6 warnings.
Saving bit stream in "system.bit".
Creating bit mask...
Saving mask bit stream in "system.msk".
Bitstream generation is complete.
Done!
```

Beispiel D.11 XPS-Konsole: erfolgreiche Bitstrom-Erzeugung

Der erzeugte Bitstrom `system.bit` befindet sich im Unterverzeichnis `implementation` Ihres Projektes. Er kann direkt auf das FPGA programmiert werden.

Hinweis: bitte achten Sie in der Konsolenausgabe auf Fehler! Falls anstatt `DRC detected 0 errors` Fehler auftraten, dürfen Sie den Bitstrom auf keinen Fall in das FPGA laden – es könnte beschädigt werden. Eine häufige Fehlerursache sind falsche Verdrahtungen in der Datei `system.mhs`. Überprüfen Sie diese im Fehlerfall und wenden Sie sich an Ihren Hiwi, falls Sie keinen Fehler entdecken können.

D.1.5 Download auf das ML310

Bild D.12 könnte Ihnen bekannt vorkommen, da es Bild 7.12 aus der Vorlesung sehr ähnlich sieht. Hier haben wir zusätzlich angedeutet, dass der zentrale Chip Virtex-II Pro aus einem Software-Teil besteht (zwei konventionell programmierbaren PowerPC-Prozessoren) und einem Hardware-Teil, in dessen Logikblöcken durch den erwähnten Bitstrom individuelle Schaltungen programmiert oder konfiguriert werden können.

Besonders hervorgehoben ist in Bild D.12 der JTAG-Port, über den sowohl der *Download* des Bitstroms erfolgt (Bild D.13) als auch die Ergebnisse des Logikanalysators zurückgegeben werden (Bild D.16). Außerdem sind die Mehrzweck-IOs hervorgehoben, an die später das board-eigene LCD angeschlossen wird, sowie die serielle Schnittstelle RS232, über die später die PowerPC-Software Ergebnisse ausgeben kann. Die übrigen Komponenten dürfen Sie für den Rest der Übung ignorieren.

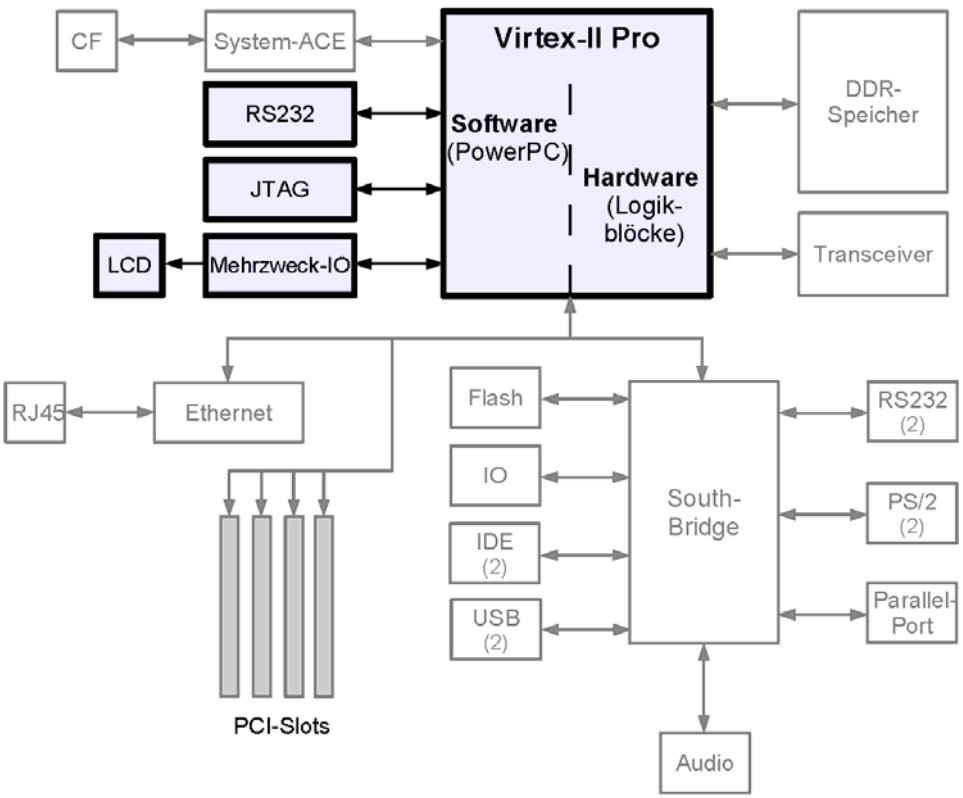


Bild D.12 Blockschaltbild des ML310

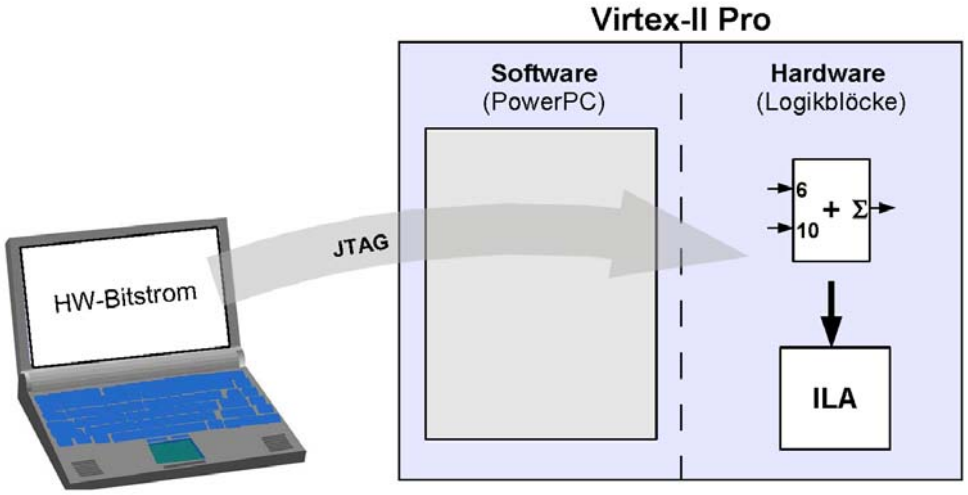


Bild D.13 Download der eigenen Schaltung

Starten Sie nun Chipscope durch Doppelklicken des Symbols in Bild D.14 auf Ihrem Desktop.

Bild D.14
Desktop-Icon zum
Chipscope-Analyzer

Sie sehen nun die Oberfläche des Chipscope-Analyzers vor sich. Überzeugen Sie sich davon, dass das ML310 eingeschaltet ist, und betätigen Sie dann in Chipscope den Button Open Cable/Search JTAG Chain (Bild D.15), der sich oben links unter der Menüleiste befindet.

Bild D.15 Button
Open Cable/Search JTAG Chain

Das daraufhin erscheinende Bestätigungsfenster schließen Sie durch OK. Im oberen linken Chipscope-Fenster wird Ihnen das FPGA des ML310 als DEV:1 MyDevice1 (XC2VP30) angezeigt. Klicken Sie mit der rechten Maustaste darauf und wählen Sie Configure... Öffnen Sie mit Select New File den von XPS erzeugten Bitstrom `system.bit`, und bestätigen Sie mit OK. Unten rechts in Chipscope wird Ihnen der Fortschritt des Downloads angezeigt, der bereits nach wenigen Sekunden abgeschlossen ist.

D.1.6 On-Chip-Test des Addierers

Wenn nichts schiefgegangen ist, wurde Ihr Addierer im vorigen Abschnitt synthetisiert, platziert, verdrahtet, als Bitstrom verpackt und befindet sich nun tatsächlich in Hardware auf dem FPGA des ML310. Daher liegt nichts näher, als diese Hardware nun auch tatsächlich laufen zu lassen und zu überprüfen. Da trifft es sich gut, dass die Entwicklungs-Software Chipscope nicht nur Bitströme downloaden kann, sondern auch den in die FPGA-Hardware integrierten Logikanalysator ILA steuern und seine Hardware-Messergebnisse bequem am Entwicklungsrechner anzeigen kann (Bild D.16).

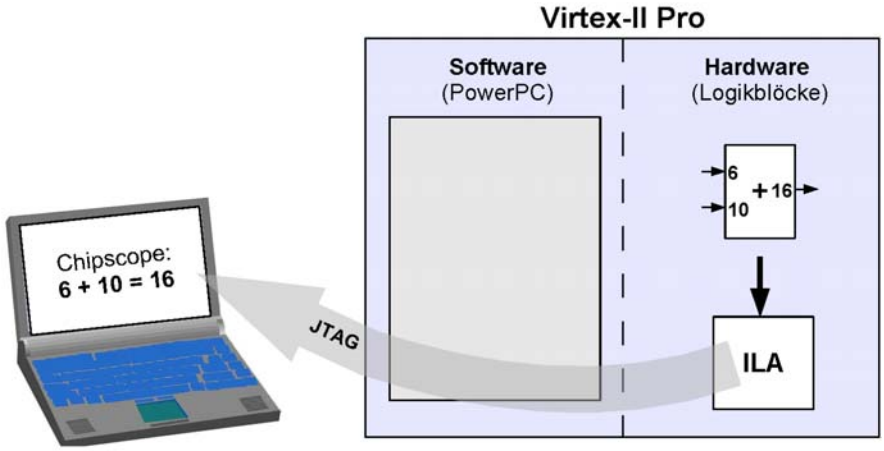


Bild D.16 Ergebnis-Anzeige mit Chipscope

Klicken Sie mit rechts auf Unit:0 und dann auf Open Waveform und zusätzlich auf Open Trigger Setup. Im Waveform-Fenster in Bild D.17 listet Chipscope nach dem Download die 64 Datenbits `DataPort[0..63]` auf.

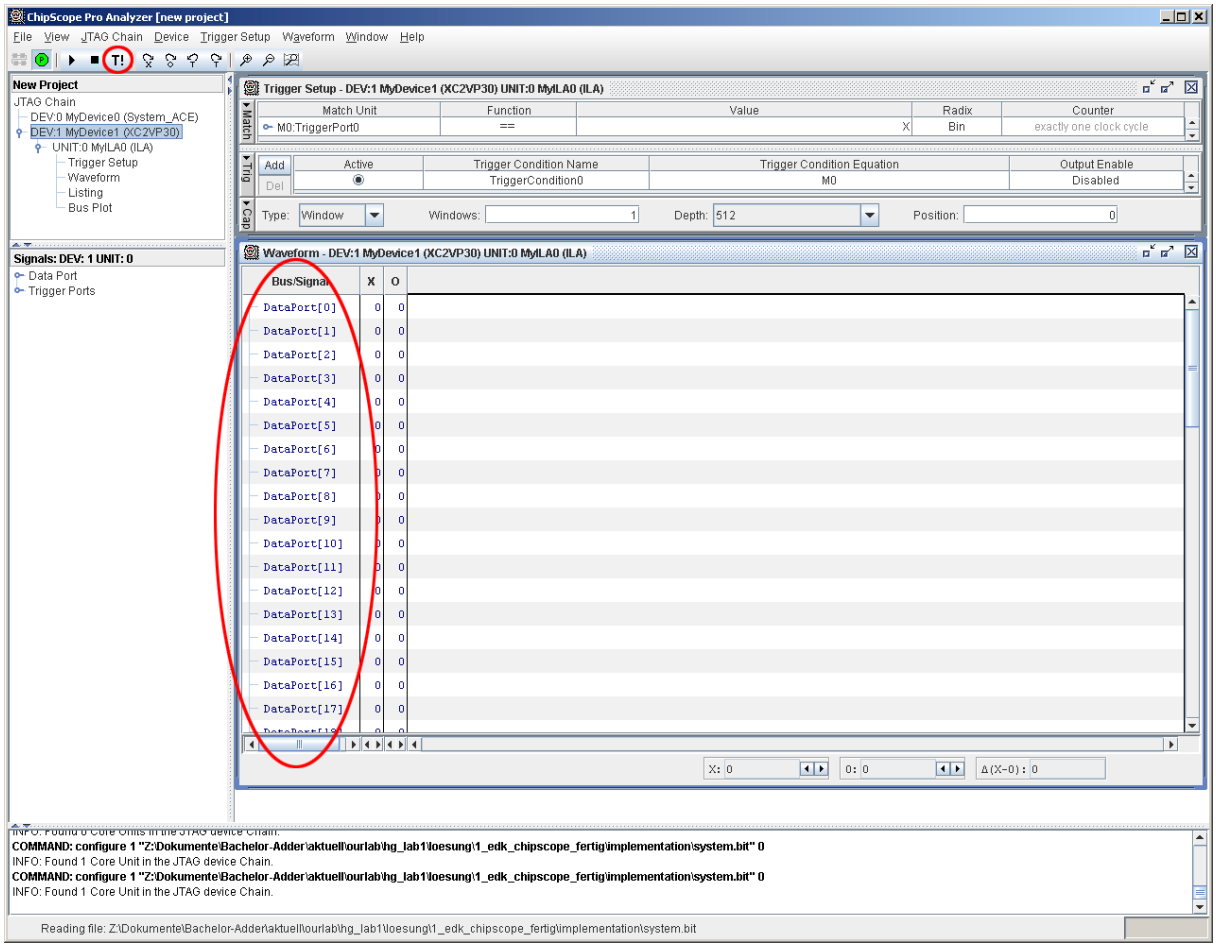


Bild D.17 Oberfläche des Chipscope-Analyzer (nach dem Download)

Die Definition dieser Leitungen befindet sich in der Datei `system.mhs` Ihres XPS-Projektes unter der Sektion `BEGIN chipscope_ila`. Sie finden dort die Zeile

```
PORT DATA = APLUSB_DBG_IN & APLUSB_DBG_OUT
```

Die an `DATA` angeschlossenen Signale werden vom IP-Core `chipscope_ila des` Logikanalysators aufgezeichnet, der ebenfalls Teil Ihres XPS-Projekts ist. Das Zeichen `&` bedeutet hier eine Konkatenation. Demnach wird die Summe `APLUSB_DBG_OUT` der Addition den Bits `31..0` des `DATA`-Signals zugeordnet, während die Summanden `APLUSB_DBG_IN` der Addition auf die Bits `63..32` abgebildet werden.

Betätigen Sie in Chipscope nun den Knopf Trigger Now (Bild D.17, Markierung oben links). Nun werden 512 aufeinanderfolgende Werte pro Datenbit des aktuellen Signalverlaufs im FPGA aufgezeichnet. Im Waveform-Fenster wird ein Signalverlauf für jedes Bit angezeigt.

Das Ergebnis bitweise zu betrachten, ist jedoch sehr umständlich. Fassen Sie daher `DataPort[0..31]` zu einem Bus zusammen. Hierzu klicken Sie auf `DataPort[0]`, halten die Shift-Taste gedrückt, scrollen etwas herunter und klicken auf `DataPort[31]`. Die Bits des Signals `APLUSB_DBG_OUT` wurden dadurch ausgewählt. Klicken Sie auf den ausgewählten Bereich mit der rechten Maustaste und wählen Sie Add to Bus | New Bus. Benennen Sie den neuen Bus `BUS_0` um in `APLUSB_DBG_OUT` (rechte Maustaste | Rename). Gehen Sie analog für die Bits `63..32` von `APLUSB_DBG_IN` vor. Die Werte für `APLUSB_DBG_IN` und `APLUSB_DBG_OUT` lassen sich nun bequem hexadezimal ablesen. Der Addierer-Input lautet `0x0006000A` (d.h. A = `0x000A`; B = `0x0006`), die Summe sollte `0x00000010` betragen.

Um die Verbindung zum ML310 wieder sauber zu trennen, wählen Sie im Menü JTAG Chain den Befehl Close Cable.

Abgabe 2

Führen Sie die Chipscope-Ausgaben Ihrem Hiwi vor!

Sie haben nun Ihr erstes VERILOG-Hardware-Modell in ein Gesamtsystem eingebettet, dieses synthetisiert, platziert und verdrahtet und im laufenden Betrieb auf dem FPGA Virtex-II Pro getestet!

D.2 Software steuert Hardware

In diesem Abschnitt sollen die Summanden A und B für die Addition nicht mehr fest in Hardware vorgegeben sein, sondern von einem Software-Programm übergeben werden. Da Hardware häufig eingesetzt wird, um rechenintensive Aufgaben zu beschleunigen, wird dafür natürlich eine Schnittstelle für den Datenaustausch zwischen Software und Hardware benötigt.

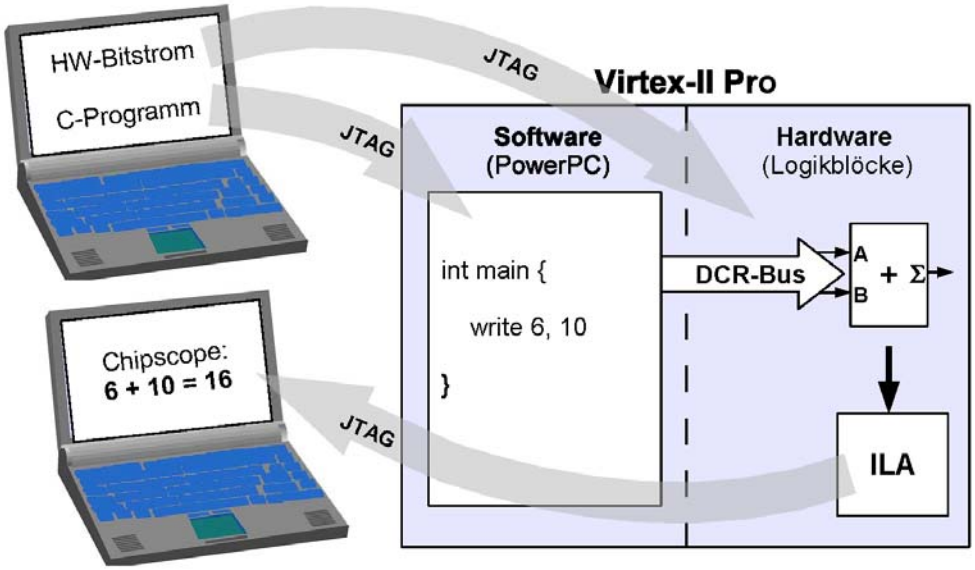


Bild D.18 Addierer mit Software-Hardware-Schnittstelle

Bild D.18 erweitert das bekannte Design um eine Software-Komponente auf dem PowerPC sowie um einen DCR-Bus als Software-Hardware-Schnittstelle. Der JTAG-Port wird jetzt zum vielgefragten Botschafter: Einerseits werden darüber das C-Programm für den PowerPC und die Hardware-Konfiguration (Addierer, DCR-Bus und Logikanalysator) auf das ML310 importiert, andererseits werden die Messergebnisse des Logikanalysators an Chipscope exportiert.

D.2.1 DCR-Bus von der Software zum Addierer

Der Addierer bekommt ein Bus-Interface für den Device-Control-Register-Bus (DCR) des Virtex-II Pro und wird über einen DCR-Controller an den PowerPC angeschlossen. Neben PLB und OPB ist der DCR der dritte On-Chip-Bus auf dem FPGA Virtex-II Pro (Bild 6.18). Es handelt sich dabei um einen sehr einfachen Bus, der vor allem für den sporadischen Austausch von kleinen Datenpaketen mit wenigen Bits zwischen Hardware und Software gedacht ist. Auf diese Weise können wir die Summanden für die Addition aus einem Software-

Programm auf dem PowerPC heraus an unseren Hardware-Addierer übertragen.

Mit dem DCR können pro Zugriff 32 Bit übertragen werden. Ein Zugriff dauert dabei mehrere Takte, der Bus gehört daher zu den langsameren Vertretern seiner Art. Jedes an den DCR angeschlossene Modul verfügt über einen DCR-Eingang und einen DCR-Ausgang, beide sind mit je einer 32-Bit-Datenleitung und einer 10-Bit-Adressleitung versehen. Der DCR-Ausgang des PowerPC ist DCR-Eingang unseres Addierers, dessen DCR-Ausgang wiederum als DCR-Eingang eines anderen IP-Cores dient.

Kommt ein Datum am DCR-Eingang an, wird geprüft, ob die mitgeschickte Adresse im eigenen Adressbereich liegt. Ist dies der Fall, werden die Daten entgegen genommen. Andernfalls werden sie einfach über den DCR-Ausgang weitergeleitet. Auf diese Weise sind alle an den DCR-Bus angeschlossenen Module wie auf einer Perlenkette aufgereiht. Der Zugriff auf den Bus ist ausschließlich dem PowerPC erlaubt, alle IP-Cores dürfen nur passiv auf Schreib- oder Leseanfragen des PowerPC antworten.

Entpacken Sie `d.2_software_steuert_hardware.zip`. Im Verzeichnis `verilog` ist die Datei `adder.v` identisch zu der Version aus Abschnitt D.1. Neu sind `dcr_write_if.v` und `aplusb_writeonly.v`. Die Datei `dcr_write_if.v` beinhaltet VERILOG-Code zur Anbindung des Addierers an den DCR-Bus, `aplusb_writeonly.v` instanziert als Top-Level-Modul das DCR-Interface sowie den Addierer und beinhaltet notwendige Verdrahtungen.

D.2.2 IP-Core erzeugen und einbinden

Im Verzeichnis `xps` befindet sich ein vorbereitetes XPS-Projekt, in das Sie im Folgenden einen mit DCR-Anschluss ausgerüsteten Addierer-Core eingefügen sollen.

Öffnen Sie das XPS-Projekt. Erstellen Sie nun einen IP-Core mittels Create or Import Peripheral, diesmal jedoch aus den drei Dateien `adder.v`, `dcr_write_if.v` und `aplusb_writeonly.v`. Geben Sie als Namen für den Core aplusb_writeonly an. Da es sich diesmal um einen IP-Core mit DCR-Lesezugriff handelt, müssen Sie im Schritt Bus Interfaces das Feld DCR Slave markieren (Bild D.19). Im Schritt SDCR Port wählen Sie für den Bus Connector DCR_Write den Port DCR_WRITE, im Schritt SDCR Parameter wählen Sie für Register Space die Einträge C_DCR_BASEADDR und C_CDR_HIGHADDR, bei Schritt Identify Interrupt Signals muss der Haken aus dem Feld Select and configure interrupt(s) entfernt werden. Die übrigen Schritte bestätigen Sie mit Next bzw Finish. Da wir die Ports der DCR-Schnittstelle in `dcr_write_if.v` genau so benannt haben wie Xilinx dies in seinen eigenen IP-Cores zu tun pflegt, werden unsere DCR-Leitungen von XPS automatisch als solche identifiziert.

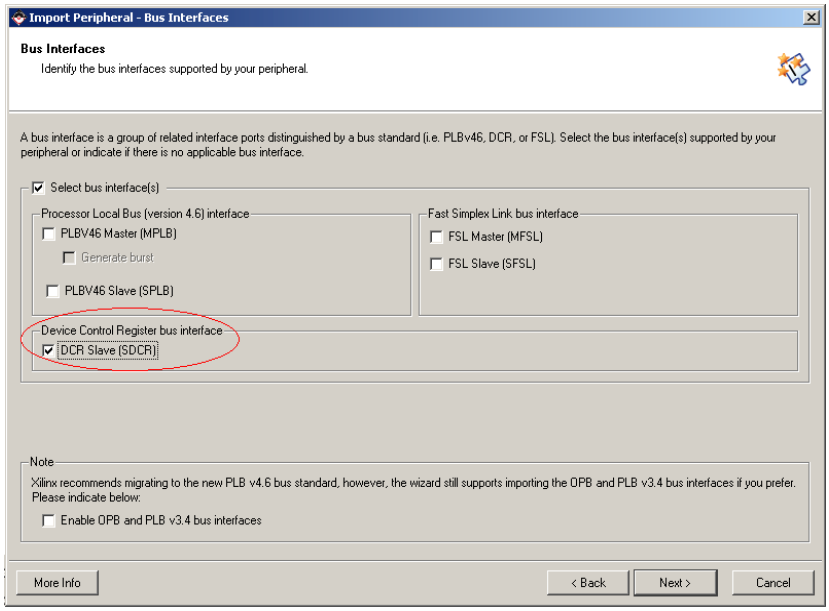


Bild D.19
Anschluss des IP-Cores an den DCR-Bus

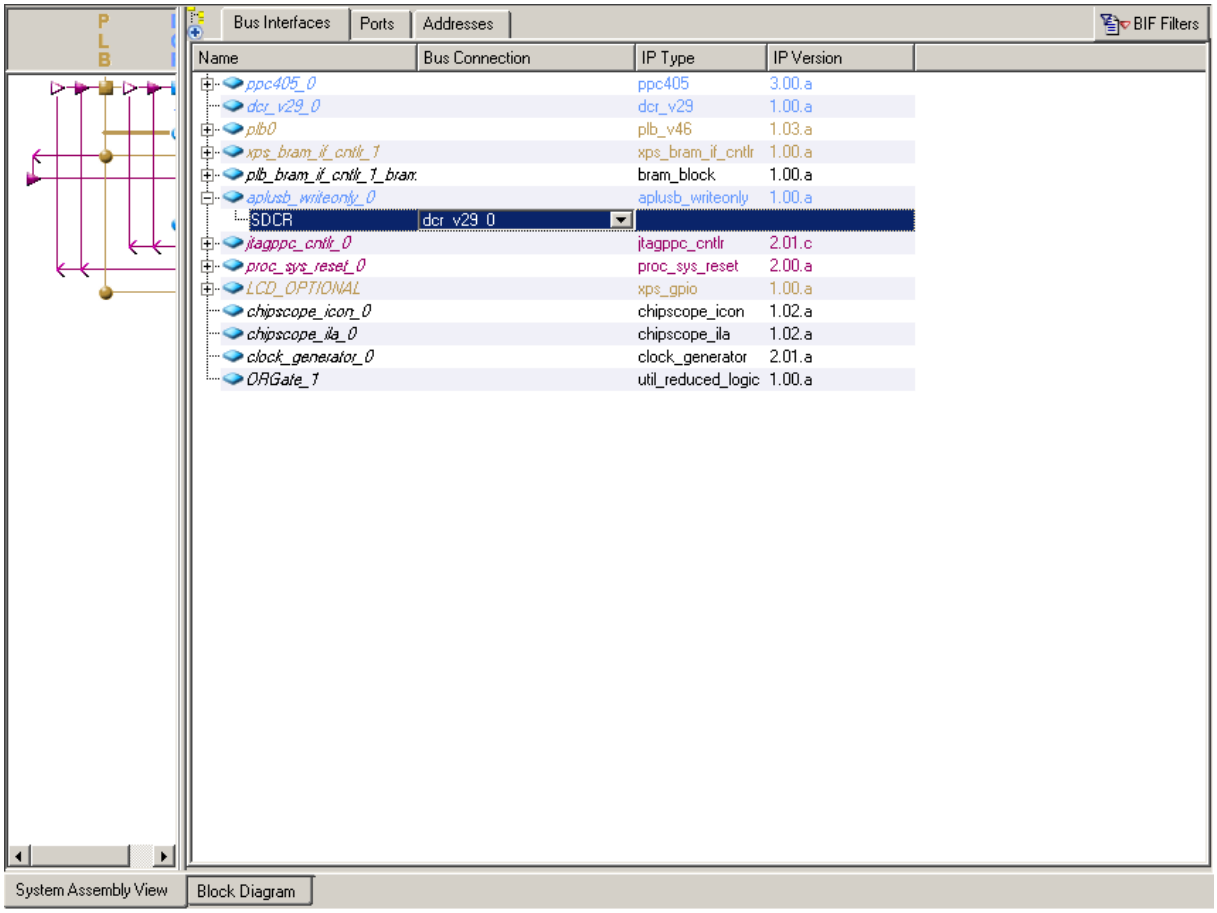


Bild D.20 Anschluss des Addierer-Cores an den DCR-Bus (Ausschnitt der System-Assembly-Ansicht)

Fügen Sie über den Reiter IP Catalog eine aplusb_writeonly-Instanz zu Ihrem Projekt hinzu. Nun muss der Core noch an den bereits vorhandenen DCR-Controller angeschlossen werden. Wechseln Sie dazu in den Tab Bus Interfaces und wählen Sie für das SDCR Interface Ihrer aplusb_writeonly-Instanz den Eintrag dcr_v29_0 (Bild D.20).

Um den Addierer mit den anderen Modulen Ihres Hardware-Software-Systems zu verbinden, muss er in der Datei `system.mhs` noch verdrahtet werden. Zu Ihrem Glück brauchen Sie sich um den Anschluss an den DCR-Bus nicht selbst zu kümmern – das übernimmt XPS für Sie. Aber die Verbindung mit der Clock- und Reset-Leitung sowie mit dem Logikanalysator ChipScope liegt in Ihrer Hand. Fügen Sie also in der Sektion aplusb_writeonly zwischen den beiden Zeilen

```
BUS_INTERFACE SDCR = dcr_v29_0
END
```

folgendes ein:

```
PORT nRESET = sys_rst_s
PORT CPU_CLK = sys_clk_s
PORT DBG_IN = APLUSB_DBG_IN
PORT DBG_OUT = APLUSB_DBG_OUT
PORT DBG_DCR_WRITE = DBG_DCR_WRITE
```

Speichern Sie dann mit Strg-S und klicken Sie auf den Button zur Bitstrom-Erzeugung. Achten Sie auch hier wieder darauf, ob zum Schluss 0 errors angezeigt werden.

D.2.3 Software vervollständigen

Die Hardware ist fertig – jetzt müssen Sie nur noch die Software fertig stellen. Klicken Sie im XPS-Projektfenster auf den Tab Applications. Unter Sources verbergen sich zwei Verweise auf C-Dateien. Sie brauchen sich hier nur um das Hauptprogramm in der Datei `testApp_adder.c` zu kümmern. Öffnen Sie die Datei durch einen Doppelklick und betrachten Sie die `main`-Methode, die das auszuführende Programm enthält (Beispiel D.21).

Nach der Definition von `writeData` wird das LC-Display des ML310 initialisiert, und zur Kontrolle werden zwei Strings ausgegeben. Übrigens wird das LCD auch über den IP-Core `opb_gpio` in Ihrem XPS-Projekt angesprochen. Im Gegensatz zum Addierer dient hier der OPB-Bus aus Bild 6.18 als Hardware-Software-Schnittstelle.

Im unteren Teil von `main` werden schließlich mit `writeData = 0x000a0006` die beiden Summanden `A` und `B` für die Addition festgelegt: `A` erhält den Wert `0x0006`, `B` den Wert `0x000a`. Dort, wo die drei Punkte stehen, müssen Sie nun den Inhalt von `writeData` über den DCR-Bus an die Hardware senden. Dazu nutzen Sie bitte den Befehl

```
int main(int argc, char* argv[]) {

  unsigned int writeData;

  // LCD: Initialisierung
  lcd_init();

  // LCD: String-Ausgabe
  lcd_write("E.I.S. ML310", "A + B example");

  // Argumente fuer den Addierer:
  // A = 0x0006, B = 000a
  writeData = 0x000a0006;

  // EDIT: sende writeData ueber den DCR-Bus an den Addierer
  ...


  return 0;
}
```

Beispiel D.21 Methode `main` der Software für den Addierer

`mtdcr(dcr_address, dcr_data);`

. Der Parameter `dcr_address` bestimmt die Zieladresse des DCR-Schreibzugriffs, `dcr_data` die zu übertragenden Daten. Sie können den DCR-Adressbereich Ihres IP-Cores im Quelltextfenster über den Reiter Addresses einsehen. Setzen Sie die passenden Daten ein und compilieren Sie Ihr Programm. Klicken Sie dazu im Projektfenster mit der rechten Maustaste auf Project: testApp_adder und wählen Sie Build Project. In der Ausgabe-Konsole sehen Sie nach fehlerfreiem Ablauf die Ausgabe aus Beispiel D.22.

```
Text     data      bss      dec      hex   filename
2486      312     8228    11026     2b12   testApp_adder/executable.elf
Done!
```

Beispiel D.22 Erfolgreiche Software-Compilierung

D.2.4 Test auf dem ML310

Hardware und Software sind vorbereitet – Sie können die Anwendung nun auf dem PowerPC des Virtex-II Pro auf dem ML310 testen! Das Ergebnis können Sie sich auch hier wieder vom Logikanalysator anzeigen lassen.

Überprüfen Sie, ob das ML310 eingeschaltet ist und laden Sie den Bitstrom mit Chipscope auf das FPGA. Anschließend können Sie die Software in den RAM-Speicher des ML310 programmieren, und zwar unter Verwendung des Xilinx-Microprocessor-Debuggers (XMD). Dieser lässt sich aus Ihrem XPS-Projekt heraus starten: klicken Sie auf das Icon Start XMD oder über den Menüpunkt Debug | Launch XMD (Bild D.23) und wählen Sie die Prozessorinstanz ppc405_0.

Bild D.23
XMD-Button

Beim ersten Starten werden Sie eventuell aufgefordert, die XMD-Optionen einzustellen. Orientieren Sie sich hierzu an Bild D.24. Dieses Einstellungsfenster erscheint nicht zwangsläufig.

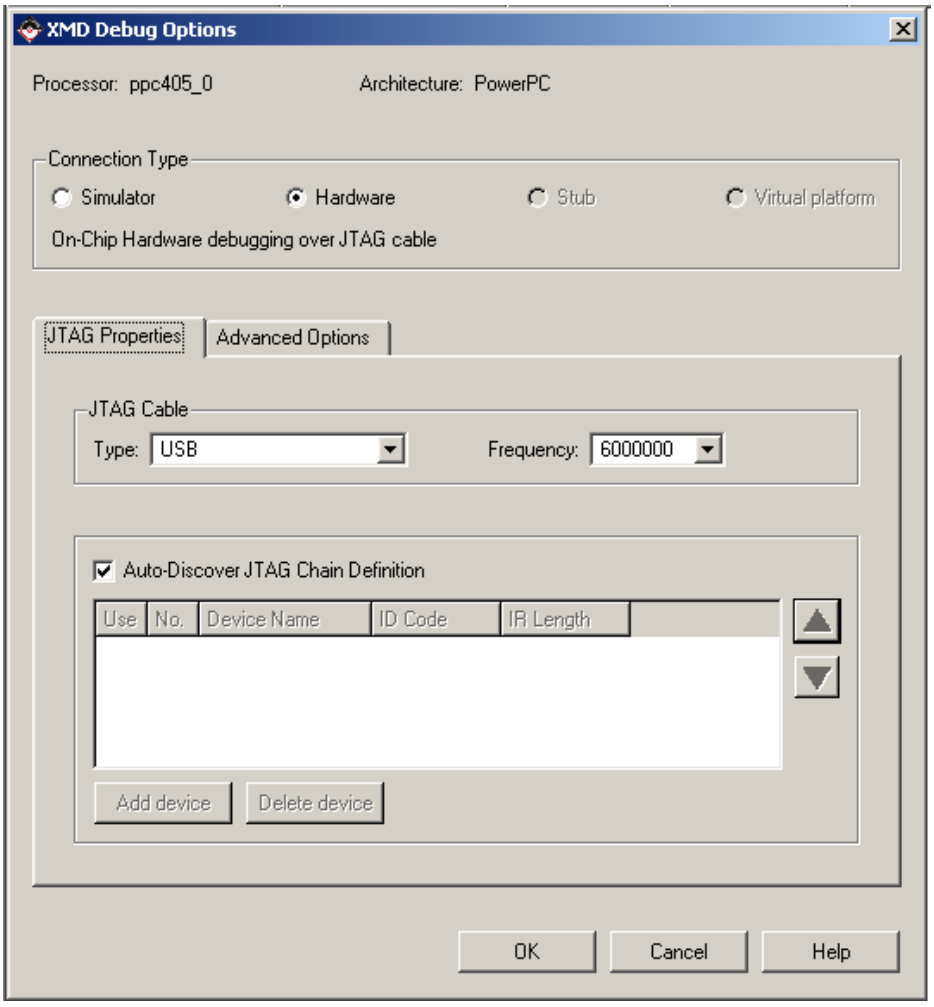


Bild D.24
Einstellen der XMD-Optionen

Die XMD-Konsole öffnet sich. Hinter XMD% kann die Eingabe von Befehlen erfolgen. Als erstes muss sichergestellt werden, dass sich der PowerPC in einem stabilen Ausgangszustand befindet. Dies erreichen Sie durch einen Reset (Kommando `rst`, Beispiel D.25).

```
XMD% rst
WARNING: Cannot stop Processor after Reset. Use “stop” Command to STOP
Power PC
System reset successfully
XMD%
```

Beispiel D.25 Reset zur Stabilisierung der CPU

Wechseln Sie nun mit

```
cd testApp_adder
```

in das Unterverzeichnis, in dem sich die compilierte Software befindet. Danach können Sie mit

```
dow executable.elf
```

die Software auf das Board laden (Beispiel D.26).

```
XMD% cd testApp_adder
XMD% dow executable.elf
        section, .text: 0xffff0000-0xffff0830
        section, .boot0: 0xffff0830-0xffff0840
        section, .boot: 0xfffffffc-0x00000000
        section, .rodata: 0x20800000-0x20800022
        section, .fixup: 0x20800024-0x20800038
        section, .got2: 0x20800038-0x20800058
        section, .sdata2: 0x20800058-0x20800058
        section, .data: 0x20800058-0x20800348
        section, .sdata: 0x20800348-0x2080034c
        section, .bss: 0x2080034c-0x2080034c
        section, bss_stack: 0x2080034c-0x20801350
        section, bss_heap: 0x20801350-0x20802350
Setting PC with program start addr = 0xfffffffc
XMD%
```

Beispiel D.26 Laden der Software in das FPGA

Überprüfen Sie mit dem Befehl `srrd`, ob sich der Programmzähler, nämlich das Register `pc`, an der richtigen Stelle `fffffffc` befindet wie in Beispiel D.27. Andernfalls müssen Sie einen erneuten Reset machen und die Software nochmals hochladen.

Vor dem Start des geladenen Programms stellen wir den Logikanalysator darauf ein, beim Starten der Datenübertragung von Software nach Hardware eine kurze Zeit lang einmal pro Takt die Summanden und die Summe der Addition aufzuzeichnen. So kann man sich davon überzeugen, dass die Datenübertragung fehlerfrei funktioniert.

```
XMD% srrd
    pc: fffffffc     msr: 00000000      cr: 00000000      lr: ffffffff
   ctr: ffffffff     xer: e0000077     pvr: 200108a0   sprg0: ffffffff
 sprg1: ffffffff   sprg2: ffffffff   sprg3: ef7efdff    srr0: ffff0700
  srr1: 00000000     tbl: 4cf1608e     tbu: 0000005e  icdbdr: 55000000
   esr: 00000000    dear: 00000000    evpr: ffff0000     tsr: dc000000
   tcr: 00000000     pit: 00000000    srr2: ffff7ffc    srr3: 00000000
  dbsr: 08100300   dbcr0: 81000000    iac1: fffffffc    iac2: bfffffff
  dac1: fffefe7e    dac2: ffffffff    dccr: 00000000    iccr: 00000000
   zpr: 00000000     pid: 00000000     sgr: ffffffff    dcwr: 00000000
  ccr0: 00700000   dbcr1: 00000000    dvc1: fffffffe    dvc2: 7fbf0fb6
  iac3: fffffffc    iac4: ff9efff0    sler: 00000000   sprg4: 54f7afef
 sprg5: bfffffff   sprg6: 7a8fe2af   sprg7: fffeff5b    su0r: 00000000
usprg0: ffffffff
XMD%
```

Beispiel D.27 Special-Register-Check

Fassen Sie in Chipscope wie in Abschnitt D.1.6 die DataPorts `0..31` zum Bus `APLUSB_DBG_OUT` und die Ports `32..63` zu `APLUSB_DBG_IN` zusammen. Benennen Sie `DataPort[64]` um in `DCR_Write`. Das Signal `DCR_Write` wechselt genau dann von `0` auf `1`, wenn die CPU Daten über den DCR-Bus schreiben möchte.

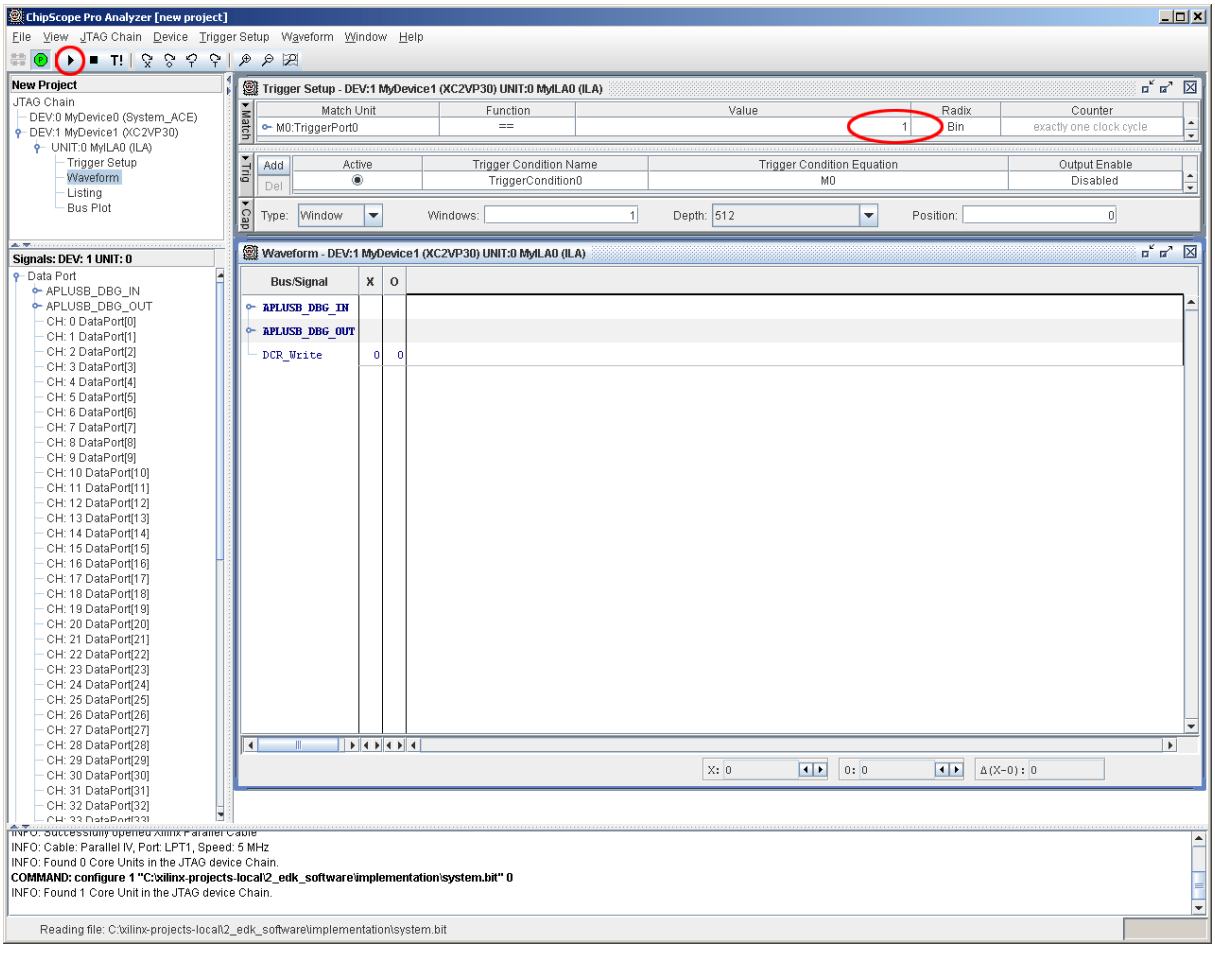


Bild D.28 Triggern in Chipscope

Konfigurieren Sie den Logikanalysator so, dass er, sobald `DCR_Write` auf `1` wechselt, mit dem Aufzeichnen von Werten beginnt. (`DCR_Write` ist für diesen Zweck bereits in der Datei `system.mhs` als sogenanntes *Trigger*-Signal festgelegt worden.) Dazu setzen Sie im Trigger Setup den Value von `X` auf `1` und betätigen den Button Apply Settings and Arm Trigger (Bild D.28).
Starten Sie jetzt das Programm über die XMD-Konsole mit dem Befehl

```
con 0xfffffffc
```

Wenn auf dem LCD die Meldung

```
E.I.S: ML310 – A + B example
```

angezeigt wird, ist Ihr Programm korrekt gestartet worden. Verifizieren Sie die aufgezeichneten Werte in Chipscope.

Abgabe 3
Präsentieren Sie die mit Chipscope gemessenen Signalverläufe sowie die LCD-Ausgabe Ihrem Hiwi.

Stoppen Sie anschließend die CPU mit dem XMD-Befehl `stop` und verlassen Sie die Konsole mit `exit` .

Sie haben hiermit Ihre erste Software-Hardware-Schnittstelle zwischen dem Steuerprogramm auf dem PowerPC und Ihrem Addierer-IP-Core auf dem FPGA realisiert und erprobt. Als nächstes soll auch die umgekehrte Datenübertragung von Hardware nach Software eingebaut werden, indem die vom Addierer berechnete Summe zurück in die Software gemeldet wird.

D.3 Hardware antwortet Software

Um Hardware auf dem FPGA für rechenintensive Aufgaben zu verwenden, müssen die berechneten Ergebnisse natürlich wieder zurück in die Software gelangen. Wir schauen uns die Realisierung am Beispiel des Addierers an. Nebenbei erstellen Sie Ihr XPS-Projekt komplett selbst. Außerdem sollen die Summanden und das an den PowerPC gemeldete Ergebnis auf dem LC-Display des ML310 dargestellt und auch über die serielle Schnittstelle an den Entwicklungsrechner wie in Bild D.29 übertragen werden.

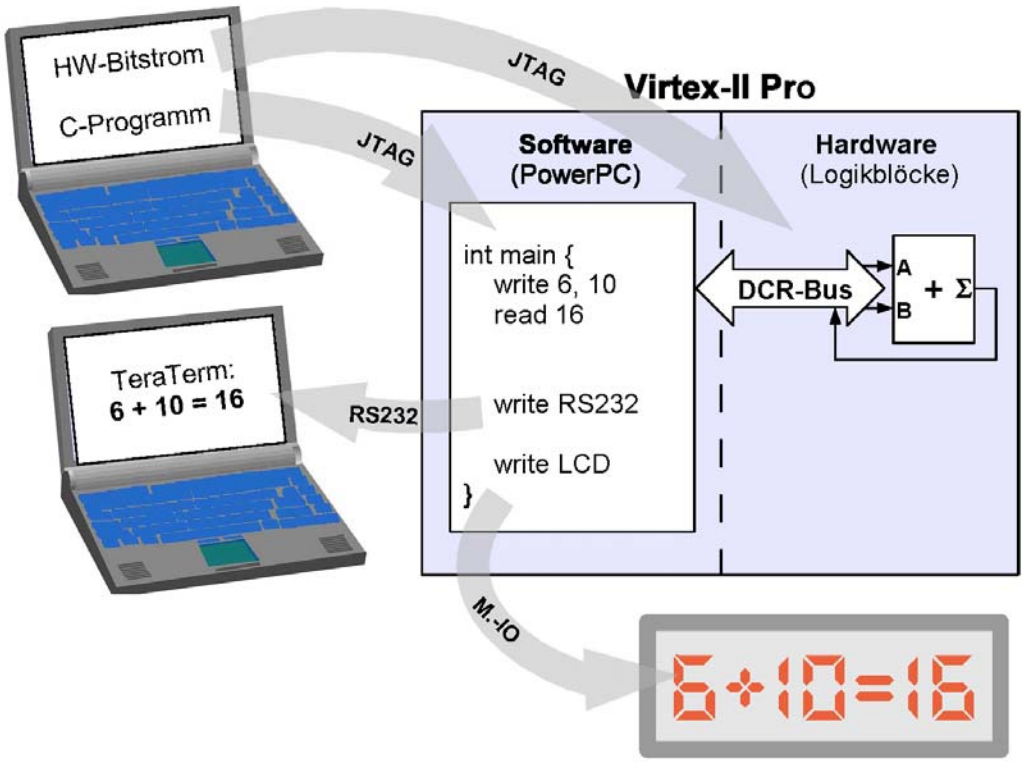


Bild D.29 Addierer mit SW-HW- und HW-SW-Schnittstelle

D.3.1 XPS-Basisprojekt

Starten Sie XPS diesmal über den Startmenüneintrag in Bild D.30.

Bild D.30
Desktop-Icon
von XPS

XPS fragt Sie, an welchem Projekt Sie arbeiten möchten. OK startet den Base System Builder Wizard. Im nächsten Dialog wählen Sie ein Verzeichnis für Ihr Projekt. Klicken Sie auf Browse..., erzeugen Sie sich in Ihrem Arbeitsverzeichnis einen Ordner `d3_self`, wechseln dort hinein und klicken dort Öffnen und speichern Sie das Projekt unter system.xmp. Bestätigen Sie mit OK.

Nun öffnet sich der Base System Builder. Wählen Sie im Welcome-Dialog I would like to create a new design. Im Dialog Select Board wählen Sie als Board Vendor *Xilinx*, als Board Name *Virtex-II Pro ML310 Evaluation Platform*, und als Board Revision *D* aus. Im Dialog Select Processor wählen Sie *PowerPC*.

Im nächsten Dialog werden die Prozessor-Parameter eingestellt. Nehmen Sie keine Änderungen an den Default-Einstellungen vor.

In den nächsten drei Dialogen werden die IO-Interfaces festgelegt. Lassen Sie im ersten Dialog lediglich das Häkchen vor RS232_Uart stehen und entfernen Sie die übrigen Häkchen. Im zweiten Dialog soll nur LCD-OPTIONAL ausgewählt bleiben und im dritten Dialog können Sie alle Häkchen entfernen.

Wählen Sie im Dialog Add Internal Peripherals für den XPS_BRAM_IF_CNTLR eine Größe von 64 KB. Im BRAM-Speicher innerhalb des Virtex-II Pro wird später die Software abgelegt.

Im Dialog Software Setup entfernen Sie bitte die Häkchen vor Memory Test und Peripheral Self Test. Dies sind Testprogramme, die wir für die vorliegende Aufgabe nicht benötigen.

Nun sind Sie nur noch 3 Klicks vom Ziel entfernt: Next - Generate – Finish, und schon ist das Basisprojekt gebaut.

D.3.2 Neuer Core `aplusb`

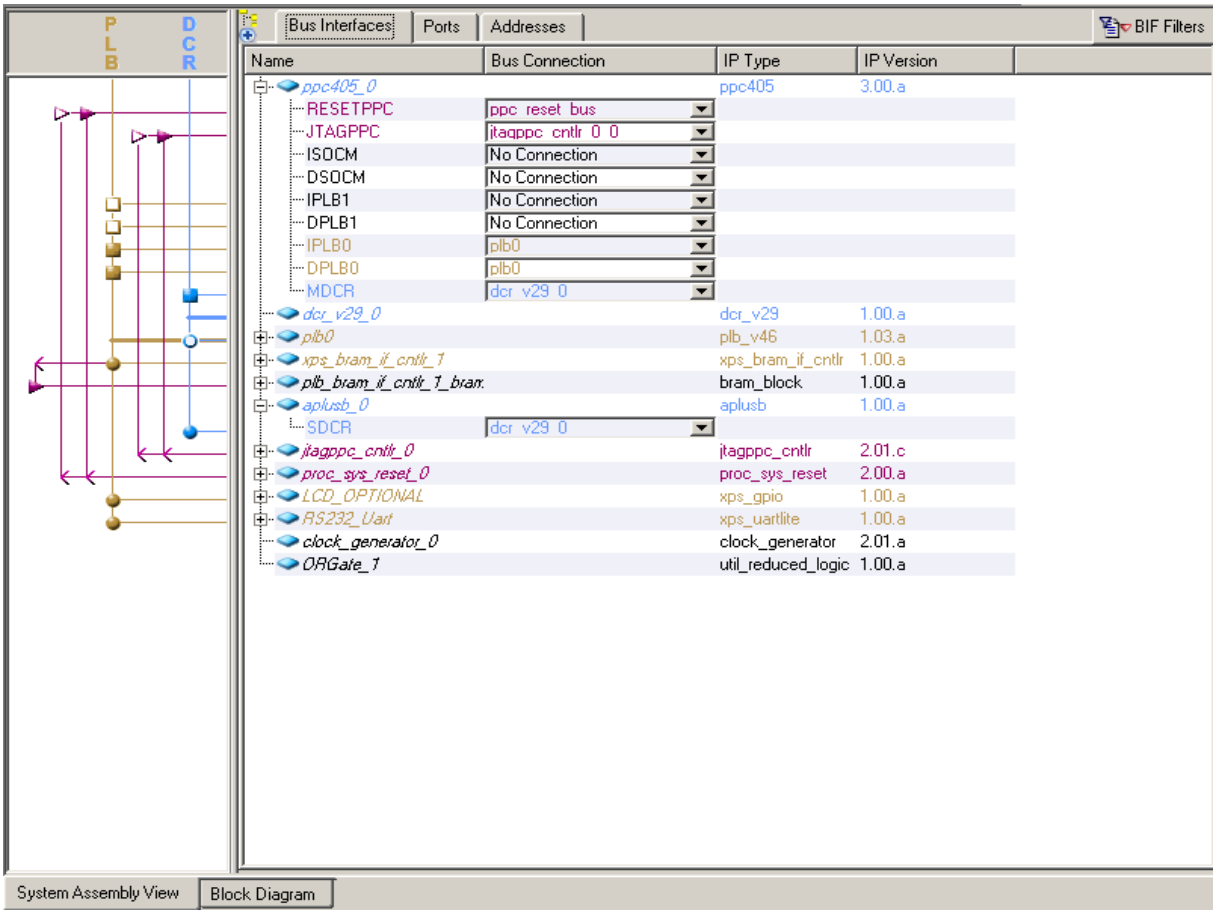


Bild D.31 DCR-Bus hinzufügen und Instanzen anschließen

Entpacken Sie `d.3_hardware_antwortet_software.zip` in Ihr Arbeitsverzeichnis. Der IP-Core, den Sie im vorigen Abschnitt erzeugt haben, kann zwar

Daten vom DCR-Bus lesen, aber keine Daten schreiben. Erstellen Sie sich daher nun einen IP-Core, der auch auf den DCR-Bus schreiben kann. Dafür brauchen Sie die drei Dateien `adder.v`, `dcr_if.v` und `aplusb.v` aus dem Verzeichnis `verilog`. Geben Sie als Namen für den Core aplusb an.

Fügen Sie den neuen IP-Core Ihrem Projekt hinzu. Markieren Sie die Option DCR-Slave, erzeugen Sie, wie in den vorangegangenen Beispielen den IP-Core und fügen Sie ihn ebenso hinzu. Im Tab Bus Interfaces, wählen Sie unter aplusb_0 für SDCR -> New connection das Interface dcr_v29_0. Wählen Sie unter ppc405_0 im Feld MDCR die Connection dcr_v29_0. (Bild D.31)

Verdrahten Sie den Reset- und Clock-Port Ihres IP-Cores durch Hinzufügen der folgenden zwei Zeilen an entsprechender Stelle in die Datei system.mhs:

```
PORT nRESET = sys_rst_s
PORT CPU_CLK = sys_clk_s
```

Erzeugen Sie dann den Bitstrom für das FPGA.

D.3.3 Fertigstellen der Software

Auch das Software-Projekt in XPS sollen Sie diesmal selbst erstellen. Erzeugen Sie in Ihrem XPS-Projektverzeichnis auf der Festplatte einen Ordner `testApp_adder` und kopieren Sie die fünf C-Quelltextdateien dort hinein, die Sie im Ordner `c` des entpackten Zip-Archivs finden. Die Dateien `lcd_functions.h` und `lcd_functions.c` enthalten Funktionen zur Ansteuerung des LCD-Displays; `uart_functions.h` und `uart_functions.c` stellen entsprechende Funktionen für die Ausgabe über die serielle Schnittstelle bereit. Das Hauptprogramm befindet sich in `testApp_adder.c`.

Wechseln Sie im XPS-Projektfenster in den Applications Tab und doppelklicken Sie Add SW Application Project... Geben Sie testApp_adder als Projektnamen ein und wählen Sie die Prozessor-Instanz ppc405_0 aus. Das neue Projekt wird angelegt. Mit einem Rechtsklick auf Sources | Add existing File… fügen Sie die drei Quelltext-dateien zum Projekt hinzu. Außerdem mit einem Rechtsklich auf Headers | Add existing File... die zwei Headerdateien. Die im Projektfenster erscheinenden C-Dateien können per Doppelklick im Quelltext-Editor angezeigt werden.

Sehen Sie sich das Hauptprogramm in `testApp_adder.c` an (Beispiel D.32). Die Zeilen 29 bis 40 dienen lediglich zur Variablendefinition und Initialisierung. In den Zeilen 43 und 44 werden `a` und `b`, die beiden Summanden der Addition, festgelegt, bevor sie in Zeile 46 auf die unteren (`a`) bzw. oberen (`b`) 16 Bit des Wertes `writeData` abgebildet werden.

```
int main(int argc, char* argv[]) {                                    // 28
  // UART-Instanz zur seriellen Kommunikation                         // 29
  XUartLite uartLite;                                                 // 30
  // Strings zur Ausgabe auf dem LCD                                  // 31
  char line1[17], line2[17];                                          // 32
  // CPU sendet Daten an den Addierer                                 // 33
  unsigned int writeData, a, b;                                       // 34
  // CPU liest Daten vom Addierer                                     // 35
  unsigned int readData;                                              // 36
                                                                      // 37
  // LCD + UART: Initialisierung                                      // 38
  lcd_init();                                                         // 39
  uart_init(&uartLite);                                               // 40
                                                                      // 41
  // Argumente fuer den Addierer                                      // 42
  a = 0x000a;                                                         // 43
  b = 0x0008;                                                         // 44
                                                                      // 45
  writeData = (b << 16) + a;                                          // 46
                                                                      // 47
  // ======== EDIT ========                                           // 48
  // ==== Ausgabe der Argumente auf dem LCD ====                      // 49
  strcpy(line1, "Writing to DCR:");                                   // 50
  print_hex(line2, writeData);                                        // 51
  ... // gebe line1 und line2 auf dem LCD aus                        // 52
                                                                      // 53
  // Ausgabe der Argumente ueber die serielle Schnittstelle           // 54
  xil_printf("A = 0x%x;  B = 0x%x  \r\n", a, b);                      // 55
                                                                      // 56
  // warte 2 Sekunden                                                 // 57
  usleep(2000000);                                                    // 58
                                                                      // 59
  // ==== Sende writeData ueber den DCR-Bus an den Addierer ====      // 60
  ...                                                                 // 61
                                                                      // 62
  // ==== Lese Berechnungsergebnis ueber DCR vom Addierer ====        // 63
  readData = ...                                                      // 64
                                                                      // 65
  // ==== Gebe Ergebnis auf dem LDC aus ====                          // 66
  sprintf(line1, "A + B =         ");                                 // 67
  print_hex(line2, ...);                                              // 68
  ... // gebe line1 und line2 auf dem LCD aus                        // 69
                                                                      // 70
  // ==== Gebe Ergebnis ueber serielle Schnittstelle aus ====         // 71
  ...                                                                 // 72
                                                                      // 73
  return 0;                                                           // 74
}                                                                     // 75
```

Beispiel D.32 DCR-Bus hinzufügen und Instanzen anschließen

In Zeile 52 sollen die Strings `line1` und `line2` zur Kontrolle des Wertes `writeData` auf dem LCD-Display des ML310 ausgegeben werden. Dies kann mit Hilfe des Befehls `lcd_write` aus der Datei `lcd_functions.h` geschehen. Sehen Sie sich die Syntax dieses Befehls an und fügen Sie eine entsprechende Anweisung in Zeile 52 ein.

Die Anweisung in Zeile 55 gibt `a` und `b` über die serielle Schnittstelle aus. Nach zwei Sekunden Pause in Zeile 58 soll in Zeile 61 der Wert `writeData` an Ihren Hardware-Addierer übermittelt werden, analog zum vorigen Abschnitt D.2. Fügen Sie den hierzu notwendigen DCR-Schreibbefehl ein.

Da der neu erzeugte Addierer-IP-Core den DCR-Bus nicht nur lesen, sondern auch schreiben kann, lässt sich eine Datenübertragung von Hardware nach Software nun ebenso einfach realisieren wie umgekehrt. Hierzu können Sie das folgende Kommando verwenden:

```
readData = mfdcr(dcr_address);
```

Verwenden Sie den Befehl `mfdcr`, um das Ergebnis der Addition von Ihrem Addierer-IP-Core über den DCR-Bus auszulesen.

Fügen Sie den gelesenen Wert `readData` in Zeile 68 ein, um die Summe in den String `line2` umzuwandeln. In Zeile 69 soll Code eingefügt werden, der `line1` und `line2` auf dem LCD-Display ausgibt (so wie in Zeile 52). Anschließend soll die Summe in Zeile 72 auch noch über die serielle Schnittstelle ausgegeben werden. Kopieren Sie dazu den Code aus Zeile 55 und ändern Sie ihn entsprechend ab.

Compilieren Sie das Programm (im Projektfenster mit der rechten Maustaste auf Project: testApp_adder | Build Project klicken) und beseitigen Sie eventuelle Fehler, die dabei in der Ausgabe-Konsole angezeigt werden.

Nach erfolgreicher Compilierung fehlen nur noch zwei kurze Schritte zur Fertigstellung der Software. Als erstes müssen Sie ein Linker-Skript erzeugen, damit das Programm auf dem ML310 im korrekten Speicherbereich des FPGA abgelegt wird. Wechseln Sie im Projektfenster wieder in das Tab Applications, klicken mit der rechten Maustaste auf Ihr Software-Projekt und wählen Sie Generate Linker Script... XPS hat die korrekten Speicherbereiche Ihres Programms automatisch erkannt, Sie können also einfach mit OK bestätigen. Klicken Sie erneut mit der rechten Maustaste auf Ihr Softwareprojekt und aktivieren Sie die Option Mark To Initialize BRAMs. Compilieren Sie die Software nun erneut. (Das generierte Linker-Skript wird dabei automatisch verwendet.)

Das war's! Hardware und Software sind nun bereit, auf dem ML310 getestet zu werden.

D.3.4 Test auf dem ML310

Laden Sie den Bitstrom `system.bit` Ihres Projektes mit Chipscope hoch. Wundern Sie sich dabei nicht, dass diesmal keine Dataports angezeigt werden – in Ihr jetziges Design ist schließlich kein Logikanalysator eingebaut.

Starten Sie auf Ihrem Arbeitsplatzrechner das Programm TeraTerm (Bild D.33). Es stellt die Ausgaben dar, die Ihr Programm über die serielle Schnittstelle des ML310 sendet.

Bild D.33
Desktop-Icon
zum TeraTerm

TeraTerm fragt beim Starten, über welche Schnittstelle Sie eine Verbindung aufbauen möchten. Wählen Sie als Schnittstelle Serial und als Port COM1. Kontrollieren Sie im Menü Setup | Serial Port..., ob die Einstellungen mit denen in Bild D.34 übereinstimmen.

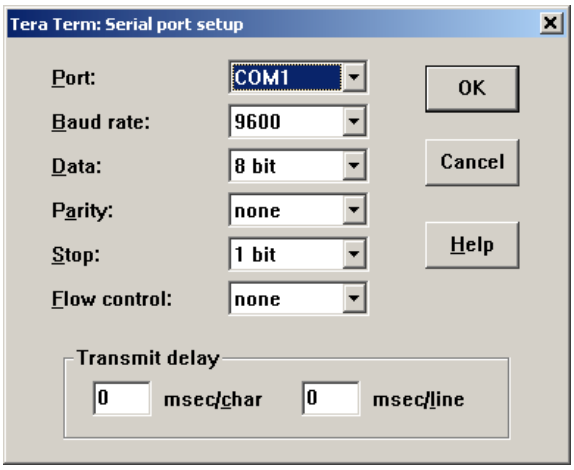


Bild D.34
TeraTerm:
Serial port setup

Laden Sie nun die Software Ihrer Applikation wie in Abschnitt D.2.4 über XMD hoch und starten Sie das Programm. Beobachten Sie dabei die Kontrollausgaben über das LCD und TeraTerm.

Abgabe 4

Präsentieren Sie die LCD- und TeraTerm-Ausgaben Ihrem Hiwi.

Herzlichen Glückwunsch – Sie sind nun in der Lage, Hardware-Software-Projekte komplett selbst zu entwerfen und Daten zwischen Software und Hardware auszutauschen!

D.4 Wettlauf zwischen Hardware und Software

Die Addition aus den vorherigen Abschnitten ist natürlich keine Rechtfertigung für den Einsatz eines FPGAs – dieses Beispiel haben wir nur der Einfachheit halber gewählt. Additionen lassen sich in Software so effizient erledigen, dass eine Auslagerung in Hardware keinen großen Sinn macht. *Aber:* sobald eine Berechnung komplexer wird, kann sich das Auslagern in

Hardware bereits lohnen. Dies wollen wir jetzt an einem Beispiel untersuchen.

D.4.1 Ein Multiplizierer

Der Addierer aus Abschnitt D.3 wird gegen einen Multiplizierer ausgetauscht. Dieser kann zwei vorzeichenlose 64-Bit-Zahlen multiplizieren. Das Ergebnis ist eine 128 Bit-Zahl. Der Multiplizierer arbeitet als Pipeline, d.h. zu jeder steigenden Taktflanke können neue Faktoren angelegt werden. Die zugehörigen Ergebnisse kommen am Ausgang mit einer Verzögerung von fünf Takten an.

Die Software `testApp_amultb` führt 10.000 Multiplikationen aus. Dies tut sie zunächst unter Verwendung des Hardware-Multiplizierers. Pro Multiplikation werden die Faktoren an die Hardware übertragen, die Berechnung des Ergebnisses wird abgewartet, und anschließend wird das 128-Bit-Produkt zurückgelesen. Die Software misst die Zeit, die insgesamt für die 10.000 Multiplikationen einschließlich Datentransfer benötigt wird und gibt das Ergebnis über die serielle Schnittstelle aus.

Anschließend misst das Programm die Zeit, die für 10.000 entsprechende Software-Multiplikationen gebraucht wird, und gibt auch hier das Ergebnis aus.

D.4.2 Test

Sie sollen nun das Programm auf dem ML310 testen. Entpacken Sie `d.4_wettlauf_zwischen_hardware_und_software.zip` in Ihr Home-Verzeichnis und öffnen Sie das XPS-Projekt. Die Hardware ist bereits synthetisiert, die Software compiliert. (Sie können diese aber gerne auch selbst noch einmal mit XPS synthetisieren und compilieren, falls Sie annehmen, wir wollen Ihnen falsche Ergebnisse vorgaukeln.) Downloaden Sie den Bitstrom auf das ML310, laden Sie mit XMD die Software testApp_amultb hoch, öffnen Sie TeraTerm und starten Sie dann die Software. Betrachten Sie die gemessenen Zeiten im Konsolenfenster von TeraTerm.

Aufgabe 5

Wie lange braucht die Hardware für 10.000 Multiplikationen? Wie lange braucht die Software? Um welchen Faktor ist die Hardware schneller als die Software?

D.5 Abschlussquiz

Frage 1

Was ist ein integrierter Logikanalysator (ILA) und wodurch zeichnet er sich aus?

Frage 2

Was bedeutet die Abkürzung DCR? Welche essenziellen Funktionen erfüllt der DCR-Bus?

Frage 3

Was ist ein IP-Core? Wofür steht die Abkürzung IP?

Frage 4

Weswegen ist eine Hardware-Auslagerung sinnvoll? Für welche Zwecke kann man sie benutzen?

D.6 Trigger Happy

Bisher haben Sie gelernt, wie man einfache Berechnungen in Hardware auslagern kann. Zugegeben, das war manchmal langweilig, aber dabei haben Sie wesentliche Fertigkeiten erworben, um nun richtig loszulegen. Zeigen Sie mit *Trigger Happy*, dass Sie ein echter Hardware-Software-Codesigner sind. Übrigens ist diese spannende Übung freiwillig.

D.6.1 Das Spiel

In Trigger Happy geht es um Reaktion, Geschwindigkeit und Ablenkung. Achten Sie auf das Zeichen. Es kommt plötzlich und unerwartet. Ziehen Sie schneller als der Gegner und holen Sie sich den Punkt. Lenken Sie Ihren Gegner ab, damit er das Zeichen später bemerkt als Sie.

Trigger Happy ist für zwei Spieler. Auf dem ML310 spielen Sie mit den beiden Tastern, die an die IO-Anschlüsse der Frontblende angeschlossen sind. Die 20 Leuchtdioden (LED) zeigen den Punktestand und fordern Ihre Reaktion heraus. Nach einer Zufallszeit wird die mittlere LED eingeschaltet. Seien Sie schneller als Ihr Gegner und drücken Sie Ihren Taster zuerst. Die mittlere LED erlischt, und wer schneller am Drücker war, bekommt einen Punkt gut geschrieben. Zwei leuchtende Balken, die sich vom linken und rechten Rand aufbauen, zeigen den aktuellen Punktestand der beiden Kontrahenten.

Schummeln wird bestraft. Wer zu früh drückt, bekommt einen Punkt abgezogen. Gewonnen hat, wer zuerst 9 Punkte erreicht.

D.6.2 Realisierung als Hardware-Software-Codesign

Die exakte Ermittlung der Reaktionszeiten der beiden Spieler erfordert eine schnelle Hardware-Lösung. Im Eifer des Gefechts kann es auf Nanosekunden ankommen! Software wäre hier nicht schnell genug, beziehungsweise eine reine Software-Lösung wäre nur mit unverhältnismäßig größerem Aufwand möglich.

Wir partitionieren die Entwurfsaufgabe deshalb wie folgt in Hardware und Software.

Hardware

1. Benutzerschnittstelle
   a. Zustand der beiden Taster (gedrückt / nicht gedrückt)
   b. Ansteuerung der 20 LEDs (ein / aus)
2. Reaktionszeitmessung
   a. Erkennung, welcher Taster zuerst gedrückt wurde

Software

3. Spielablauf-Steuerung (führe folgende Punkte nacheinander aus)
   a. Hardware initialisieren, Punktestand auf 0-0 zurücksetzen
   b. zufällige Zeitspanne warten; dabei prüfen, ob Taste gedrückt wird (Schummeln) und in diesem Fall dem Schummler einen Punkt abziehen
   c. nach der Wartezeit mittlere LED einschalten
   d. auf Tastendruck warten und Gewinner ermitteln
   e. mittlere LED ausschalten, Punktestand aktualisieren
   f. hat ein Spieler 9 Punkte erreicht, Gewinner anzeigen (z.B. durch blinkende LED); sonst gehe zu b
4. Punktestand verwalten und anzeigen
   a. Ausgabe des aktuellen Punktestands als zwei LED-Balken (der Balken für Spieler 1 baut sich von links zur Mitte auf, der Balken für Spieler 2 von rechts)

Hardware-Software-Kommunikation

Welche Arten der Hardware-Software-Kommunikation benötigen wir für die Realisierung des Spiels? Gehen wir die obige Anforderungsliste für Hardware und Software durch und leiten dabei die benötigten Kommunikationsformen ab:

- *Software sendet Daten an Hardware:* Dies ist der Fall beim Ansteuern der Leuchtdioden. Die Hardware muss hierzu ein geeignetes Register zur Verfügung stellen, in das die Software einen Wert schreiben kann, der das gewünschte Muster auf den Leuchtdioden erscheinen lässt. Beim Addierer haben wir diese Kommunikationsform eingesetzt, um die Operanden zu übertragen.
- *Software liest Daten von Hardware:* Dies ist der Fall nachdem eine Taste gedrückt wurde. Die Hardware muss ein Register zur Verfügung stellen, aus dem die Software auslesen kann, welcher Taster zuerst gedrückt wurde. Beim Addierer haben wir diese Kommunikationsform eingesetzt, um das Ergebnis der Addition auszulesen.
- *Hardware benachrichtigt Software:* Diese Kommunikationsform benötigen wir, wenn eine Taste gedrückt wurde. Die Hardware löst einen *Interrupt* aus und weist damit den PowerPC an, einen dafür bestimmten Software-Prozess, den *Interrupt-Handler* zu starten.

Wir haben es also mit drei typischen Verfahren der Hardware-Software-Kommunikation zu tun. Die ersten beiden sind Ihnen schon bekannt, wir können sie auf dem ML310 leicht über den DCR-Bus realisieren.

Ein Interrupt als drittes Verfahren ist Neuland. Der PowerPC bietet hierzu einen Interrupt-Eingang als speziellen Pin. Mit der Methode `initInterrupt` in Beispiel D.35 wird eine Software-Methode als Behandlung von Interrupts deklariert (*Interrupt-Handler*). Der PowerPC ruft anschließend automatisch die Methode `irqHandler` auf, wenn das Signal an seinem Interrupt-Pin von `0` auf `1` wechselt.

```
// Die Interrupt-Handler-Methode
void irqHandler(void* not_in_use);

// Initialisieren des PowerPC-Interrupt
void initInterrupt() {

  // Initialisieren des Exception-Handling
  XExc_Init();

  // Register Interrupt-Handler
  XExc_RegisterHandler(XEXC_ID_CRITICAL_INT,
    (XExceptionHandler)irqHandler, NULL);

  // Aktiviere unkritische Interrupts
  XExc_mEnableExceptions(XEXC_CRITICAL);
}

// Implementierung des Interrupt-Handlers
void irqHandler(void* not_in_use) {
  // Hardware anweisen, das Interrupt-Signal zu loeschen
  mtdcr(HW_DCR_ADDR, 0);
  printf("He, die Hardware hat gerade einen Interrupt ausgeloest!");

  // hier steht sinnvoller Quelltext
}
```

Beispiel D.35 Eine Software-Methode als Interrupt-Handler deklarieren

In der Methode `irqHandler` ist die erste Amtshandlung besonders wichtig: durch einen DCR-Schreibzugriff wird der Hardware mitgeteilt, dass der Interrupt in der Software angekommen ist. Die Hardware sollte nun das Interrupt-Signal löschen, d.h. auf `0` zurücksetzen. Ansonsten würde nämlich nach Verlassen des Interrupt-Handlers dieser gleich wieder aufgerufen.

Aufgabe 6

Entpacken Sie das Archiv `d.6_trigger happy.zip` und öffnen Sie das XPS-Projekt. Es enthält eine Basisumgebung, mit der Sie die Spielentwicklung starten können. *Wichtig:* Bei dieser komplexen Aufgabe ist es wichtig, jede Teilaufgabe separat zu lösen und diese dem HiWi zu präsentieren. Nur so lassen sich Folgefehler vermeiden, die Sie in eine Sackgasse führen könnten.

6.1 Schauen Sie sich zunächst die Hardware-Plattform an. Welche IP-Cores sind enthalten, und wie sind sie untereinander verbunden? Fertigen Sie eine Skizze (Blockschaltbild) der Hardware-Plattform an und zeigen Sie sie Ihrem HiWi.

Der für das Spiel interessante Teil des XPS-Projekts befindet sich in dem IP-Core `gamecontroller_0`. Um Ihnen etwas Arbeit zu sparen, haben wir schon mal ein paar Zeilen VERILOG geschrieben. Sehen Sie sich die Quelltexte an. Das Hauptmodul ist `gamecontroller` in `gamecontroller.v`. Darin werden drei Submodule instanziert:

- `dcr_if`: Dies ist die Schnittstelle zum DCR-Bus. Über die Anschlüsse `CPU2SLAVE` und `SLAVE2CPU` stellt es zwei 32-Bit-Register zur Verfügung, deren Inhalt von der Software gelesen (`SLAVE2CPU`) bzw. durch die Software geschrieben werden kann (`CPU2SLAVE`).
- `debouncer`: An dieses Modul werden die beiden Taster angeschlossen. Die Wires `PLAYER1` und `PLAYER2` teilen mit, ob Taster 1 oder Taster 2 gedrückt wurde. Intern sorgt `debouncer` dafür, dass die Taster entprellt werden. Bei mechanischen Tastern typische Schwingungen werden herausgefiltert, um einen Tastendruck nicht versehentlich als mehrere Tastendrücke zu interpretieren. Benutzen Sie als `debouncer` ihr eigenes Modul aus Aufgabe B.7. Sollten Sie diese Aufgabe nicht bearbeitet haben oder entspricht ihr Modul nicht den genauen Vorgaben, so finden Sie im Archiv auch einen vorgefertigten Modul, den Sie statt einer eigenen Lösung benutzen können.
- `gamelogic`: In diesem noch leeren Modul soll die fehlende Hardware implementiert werden. Wir haben es bereits mit den Anschlüssen von `dcr_if` und `debouncer` verbunden, so dass sie auf die Taster und den DCR-Bus zugreifen können. Des Weiteren stehen ein `IRQ`-Signal und 20 LED-Leitungen zur Verfügung, die wir im XPS-Projekt bereits mit den richtigen Anschlüssen des FPGAs verbunden haben.

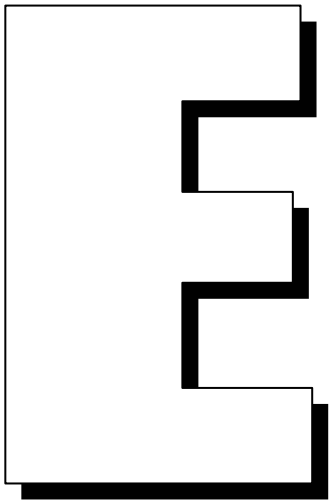
6.2 Haben Sie erkannt, welche Funktion der `gamecontroller` im momentanen Zustand bereits ausführt? Erzeugen Sie einen FPGA-Bitstrom und laden Sie ihn mit Chipscope auf das ML310. Stimmt Ihre Annahme?

Wechseln Sie nun in die Ansicht Applications. Im XPS-Projekt ist bereits ein minimales Software-Grundgerüst enthalten. Die Hauptdatei ist `game_control.c`. Sehen Sie sich den Quelltext an. Welche Methoden sind enthalten und welche Aufgaben erfüllen sie?

6.3 Entwickeln Sie ein Konzept für die Implementierung des Spiels auf Basis der vorgegebenen Hardware- und Software-Quellen. Planen Sie den genauen Ablauf des Spiels. Welche Funktionen werden benötigt und durch welche Funktionen in Hardware und Software werden sie ausgeführt? Sprechen Sie Ihre Ergebnisse mit Ihrem HiWi durch.

6.4 Implementieren Sie das Spiel Trigger Happy als Hardware-Software-Codesign für das ML310.

6.5 Testen Sie Ihr Spiel ausführlich auf dem ML310. Spielen Sie gegen Ihre Team-Mitglieder, Ihre HiWis und gegen andere Teams. Werden Sie Trigger-Happy-Champion 2009!



Zusammenfassung der drei Labs

In diesem Kapitel finden Sie Informationen zu den folgenden Stichworten:

E.1 Zusammenfassung von Lab 1

Benutzeroberfläche des Projekt-Navigators

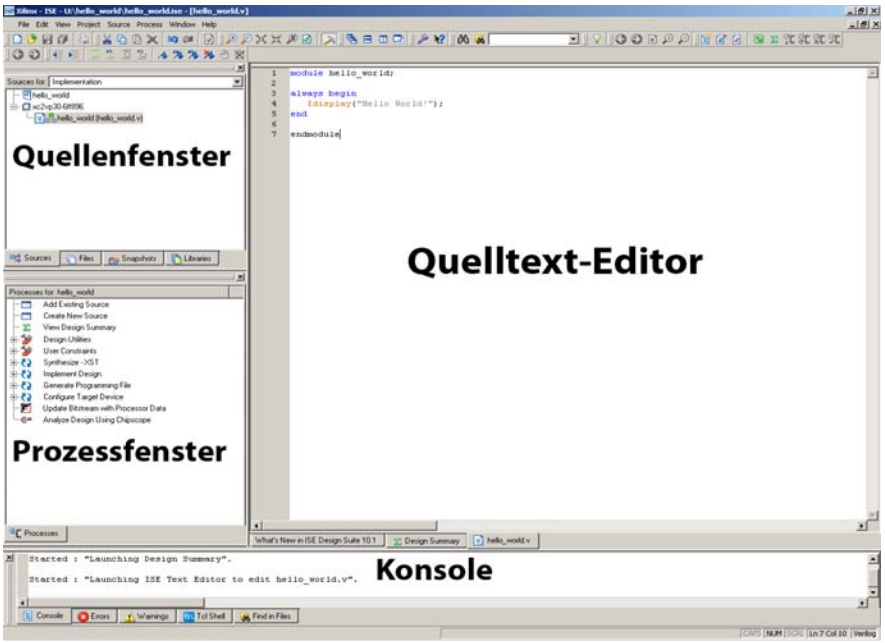


Bild E.1
Überblick zum Projekt-Navigator

- Quelltext-Editor: Hier schreiben Sie ihren Quelltext (Bild E.1).
- Quellenfenster: Ein Rechtsklick erlaubt ihnen hier, mit New Source… neue Quellen hinzuzufügen, wie z. B. Module und Testrahmen. Unter Sources for: lassen sich verschiedene Modi auswählen, z. B. Implementation.
- Prozessfenster: In diesem Fenster werden alle für die momentan ausgewählte Quelle aus dem Quellenfenster verfügbaren Prozesse angezeigt. Beachten Sie also, dass Sie stets die richtige Quelle angewählt haben, um den gewünschten Prozess auszuführen.
- Konsole: Hier erscheinen die Konsolenausgaben, die über erfolgreiche Compilierung oder gefundene Errors etc. Auskunft geben.

Ein neues Projekt anlegen

- Klicken Sie: File → New Project…
- Wählen Sie ein Projektverzeichnis und einen Projektnamen.
- Nehmen Sie die Einstellungen wie in Bild E.2 vor.
- Die nachfolgenden Dialogfelder können mit Next bzw. Finish übersprungen werden.

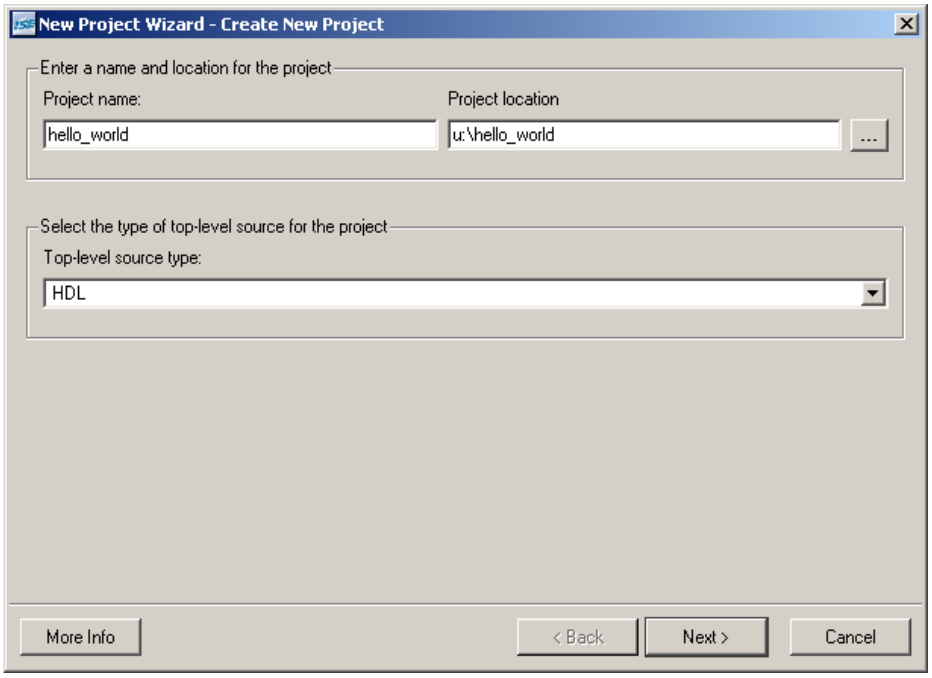


Bild E.2
Neues Projekt: Einstellungen

Quelltext erzeugen und ins Projekt einfügen

- Rechtsklick im Quellenfenster, New Source…
- Wählen Sie ihre gewünschte Datei und benennen Sie diese.
- Für ein normales Modul wählen Sie Verilog Module. Für einen Testrahmen wählen Sie Verilog Test Fixture.
- Falls Sie einen Testrahmen gewählt haben, müssen Sie im folgenden Schritt auswählen, zu welchem ihrer Module der Testrahmen gehört.
- Die nachfolgenden Dialogfelder können mit Weiter bzw. Fertig stellen übersprungen werden.

Simulator starten (ohne expliziten Testrahmen)

- Wählen Sie unter Sources for im Quellenfenster den Modus Behavioral Simulation.
- Selektieren Sie das gewünschte Modul im Quellenfenster.
- Öffnen Sie im Prozessfenster den Punkt ModelSim Simulator.
- Doppelklicken Sie den Prozess Simulate Behavioral Model.
- Stellen Sie sicher, dass für die simulierte Zeit ein genügend großer Wert eingestellt ist.

Simulator starten (mit explizitem Testrahmen)

- Selektieren Sie den Testrahmen im Quellenfenster.
- Doppelklicken Sie im Prozessfenster Simulate Behavioral Model.

Benutzeroberfläche von ModelSim

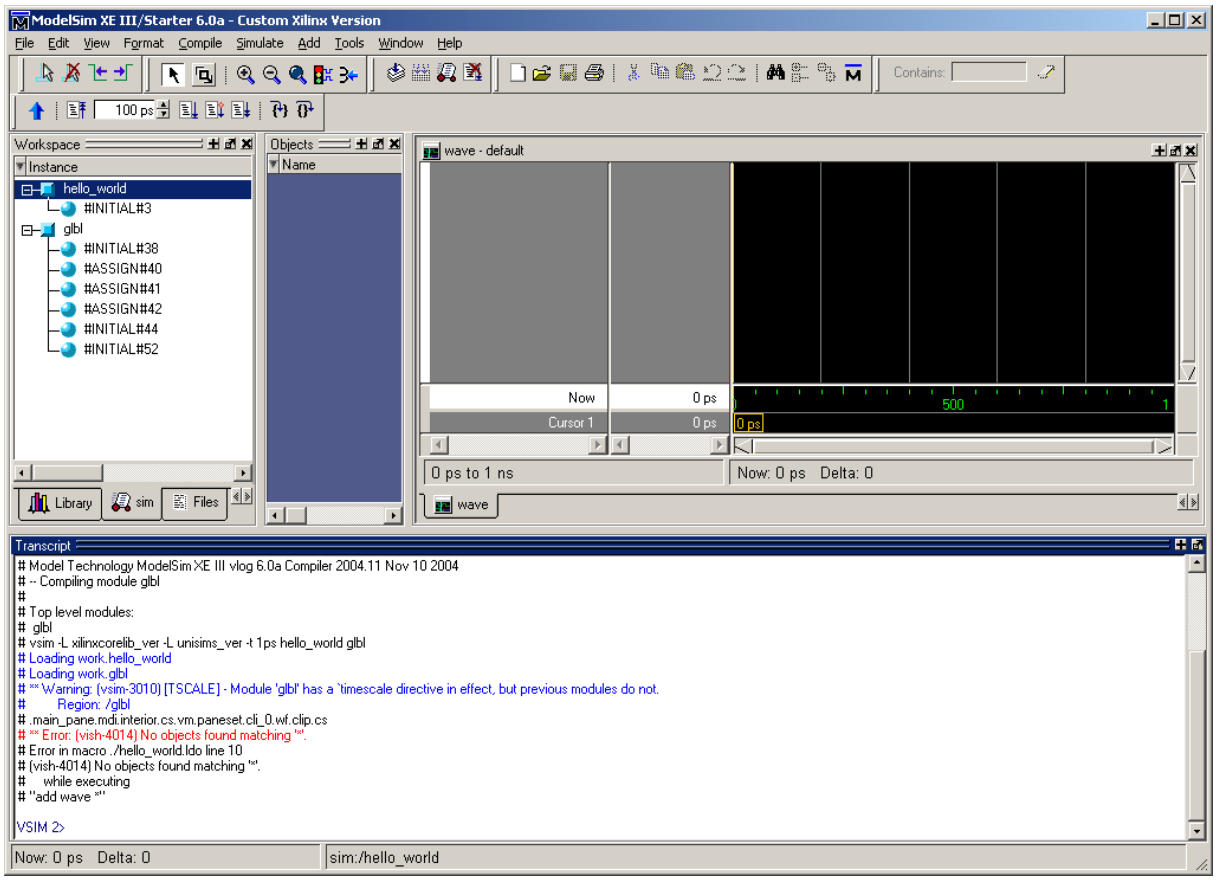


Bild E.3 Benutzeroberfläche des Simulators ModelSim

- Workspace-Fenster, links oben: Dieser Teil zeigt alle Module, Modulinstanzen, initial- und always-Blöcke sowie Continuous Assignments an (Bild E.3).
- Waveform-Fenster, rechts oben: Im rechten Fenster werden die simulierten Waveforms angezeigt. Mittels „I“ und „O“ kann herein- und herausgezoomt werden. Ein Klick mitten ins Waveform setzt die gelbe Linie an den gewünschten Punkt und zeigt die genaue Simulationszeit an.
- Objects-Fenster, mittig oben: Aus diesem Fenster können weitere Signale in das Waveform-Fenster per Drag & Drop eingefügt werden, wenn man zuvor im Workspace-Fenster eine andere Modulinstanz anklickt. Die Simulation muss allerdings neu gestartet werden, bevor das neue Signal aufgezeichnet wird.

- Erneutes Starten der Simulation: Stellen Sie mittig in der Zeile unter der Menüleiste in dem dafür vorgesehen Feld eine angebrachte Simulationszeit mit Zeiteinheit ein, z.B. 200 ns. Klicken Sie dann direkt links daneben auf das Icon mit dem aufwärts zeigenden Pfeil und bestätigen Sie den folgenden Dialog mit Restart. Klicken Sie danach direkt rechts neben dem Simulationszeitfeld auf das Icon mit dem abwärts zeigenden Pfeil.
- Alternativ können Sie in der Konsole auch den Befehl restart eingeben, den Dialog mit Restart bestätigen, und dann run eintippen.
- Ändern des Zahlenformats von Signalen: Rechtsklicken Sie auf das gewünschte Signal, und wählen Sie dann unter Radix die gewünschte Basis aus.
- Recompilieren von Modulen: Wechseln Sie im Workspace-Fenster in den Tab Library und öffnen Sie den Punkt work. Rechtsklicken Sie auf die Dateien, welche Sie mit dem Befehl Recompile recompilieren möchten.
- Schriftart ändern: Klicken Sie in der Menüleiste auf Tools | Edit Preferences | Main Window und wählen Sie am besten Courier New.

E.2 Zusammenfassung von Lab 2

Alle Vorgänge sind aus dem Xilinx Project Navigator heraus auszuführen.

Logiksynthese

- Selektieren Sie das gewünschte Modul im Quellenfenster.
- Doppelklicken Sie im Prozessfenster den Punkt Synthesize – XST.

Schematische RTL-Darstellung

- Wählen Sie im Quellenfenster unter Sources for: den Modus Implementation.
- Selektieren Sie das gewünschte Modul im Quellenfenster.
- Öffnen Sie im Prozessfenster den Punkt Synthesize – XST.
- Doppelklicken Sie View RTL Schematic.

Technology-Mapping

- Wählen Sie im Quellenfenster unter Sources for: den Modus Implementation.
- Selektieren Sie das gewünschte Modul im Quellenfenster.

- Öffnen Sie im Prozessfenster den Punkt Synthesize – XST.
- Doppelklicken Sie View Technology Schematic.

Synthese-Bericht

- Wählen Sie im Quellenfenster unter Sources for: den Modus Implementation.
- Selektieren Sie das gewünschte Modul im Quellenfenster.
- Öffnen Sie im Prozessfenster den Punkt Synthesize – XST.
- Doppelklicken Sie View Synthesis Report.
- Hier finden Sie unter anderem auch Angaben über die kritischen Pfade der Schaltung, also über die maximale Taktrate, mit der sie betrieben werden kann.

E.3 Zusammenfassung von Lab 3

Chipscope: Bitstrom herunterladen

- Schalten Sie das ML310 ein.
- Betätigen Sie den Button Open Cable/Search JTAG Chain. Bestätigen Sie das folgende Fenster mit OK.
- Klicken Sie mit der rechten Maustaste links oben auf DEV:1 MyDevice1 (XC2VP30) und wählen Sie Configure...
- Wählen Sie mit Select New File ihren Bitstrom `system.bit`, der sich in dem Unterordner `implementation` Ihres Projekts befindet, und bestätigen Sie mit OK.
- Warten Sie den Download-Fortschritt ab, der ganz rechts unten angezeigt wird.

Chipscope: mehrere Bits zu einem Bus zusammenfassen

- Selektieren Sie die Bits, die Sie zusammenfassen wollen.
- Rechtsklicken Sie auf ihre Auswahl und wählen Sie Add to Bus | New Bus. Benennen Sie diesen Bus sinnvoll mit rechte Maustaste | Rename.

Chipscope: sauberes Beenden

- Um die Verbindung zum ML310 wieder sauber zu trennen, wählen Sie im Menü JTAG Chain den Befehl Close Cable.

Chipscope: Signalverläufe des FPGAs aufzeichnen

- Betätigten Sie den Button Trigger Now in der Button-Zeile, der mit „T!“ beschriftet ist.
- Nun wird für jedes Bit ein Signalverlauf im Waveform-Fenster angezeigt.

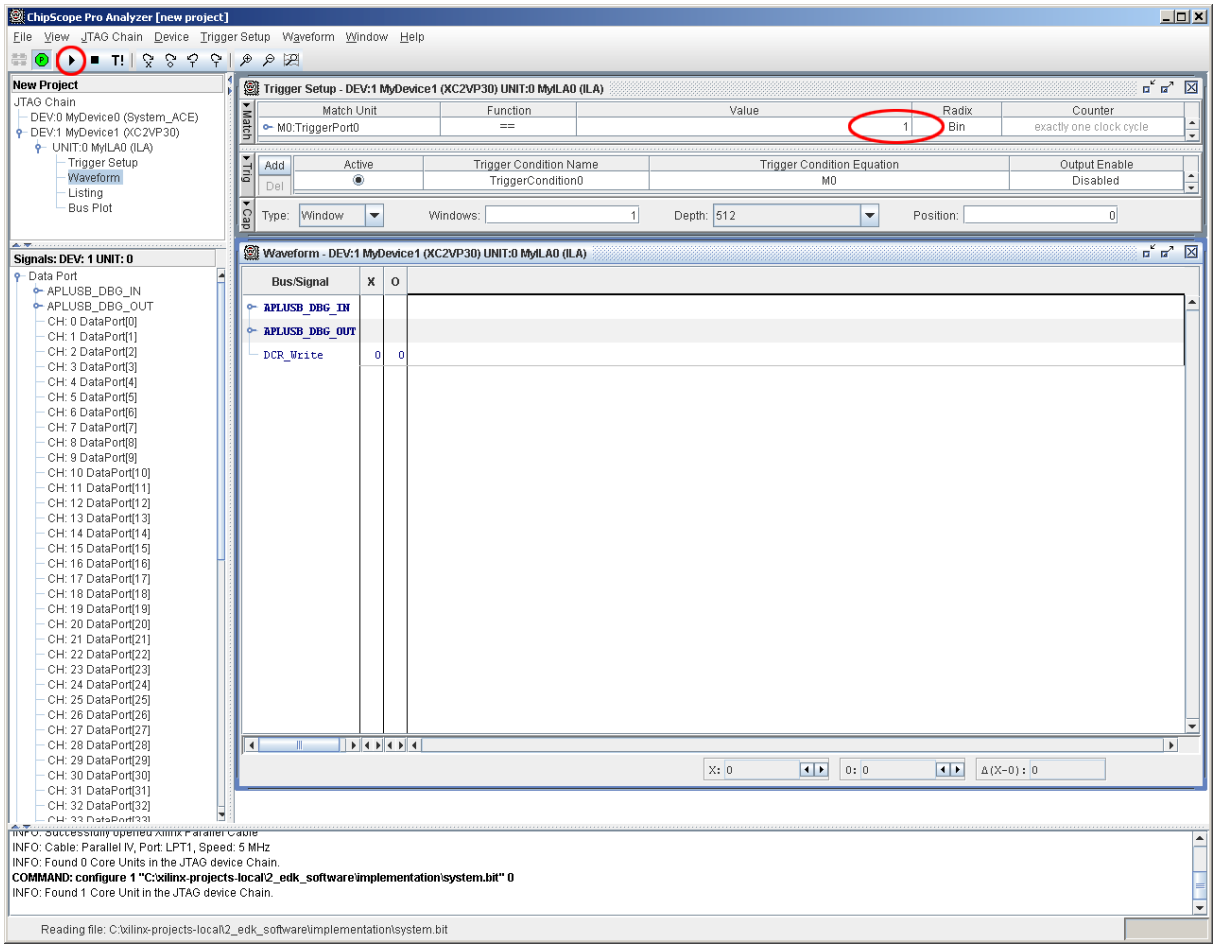


Bild E.4 Triggern in Chipscope

Chipscope: Trigger-Signal festlegen

- Wenn in der der `system.mhs` ein Signal als so genanntes *Trigger*-Signal bereits festgelegt worden ist, müssen Sie im Trigger Setup von Chipscope lediglich Value

von X auf 1 setzen und den Button Apply Settings and Arm Trigger (Bild E.4) betätigen.

TeraTerm

- Wählen Sie beim Start von TeraTerm als Schnittstelle Serial und als Port COM1.
- Im Menü Setup | Serial Port... können Sie diese Einstellungen kontrollieren (Bild E.5).

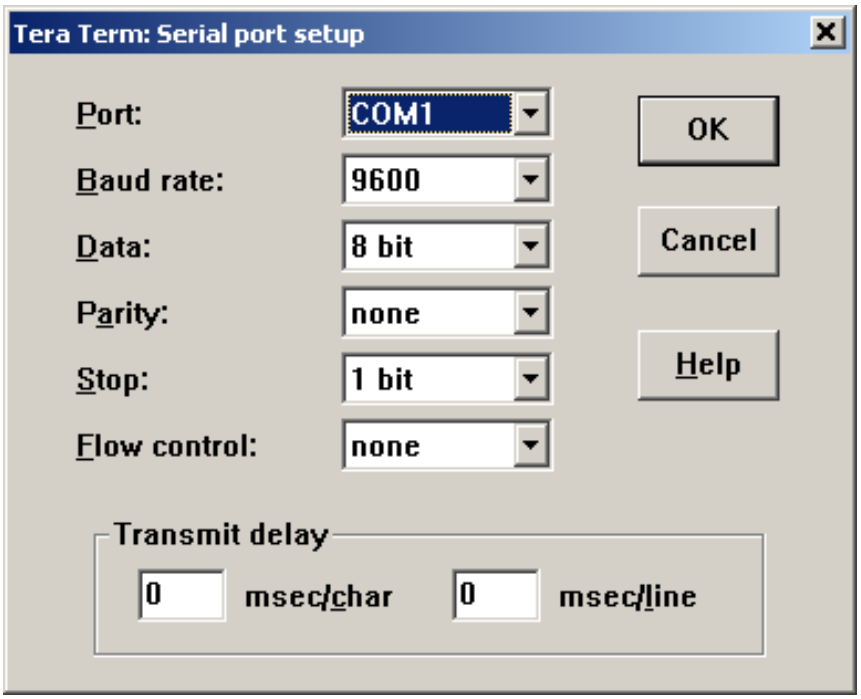


Bild E.5
TeraTerm:
Serial port setup

XMD: Konsole öffnen/starten

- Wählen Sie im Menü Debug den Eintrag Launch XMD.
- Als Prozessorinstanz wird ppc405_0 ausgewählt.
- Beim ersten Starten werden Sie aufgefordert, die XMD-Optionen einzustellen. Orientieren Sie sich hierzu an Bild E.6.

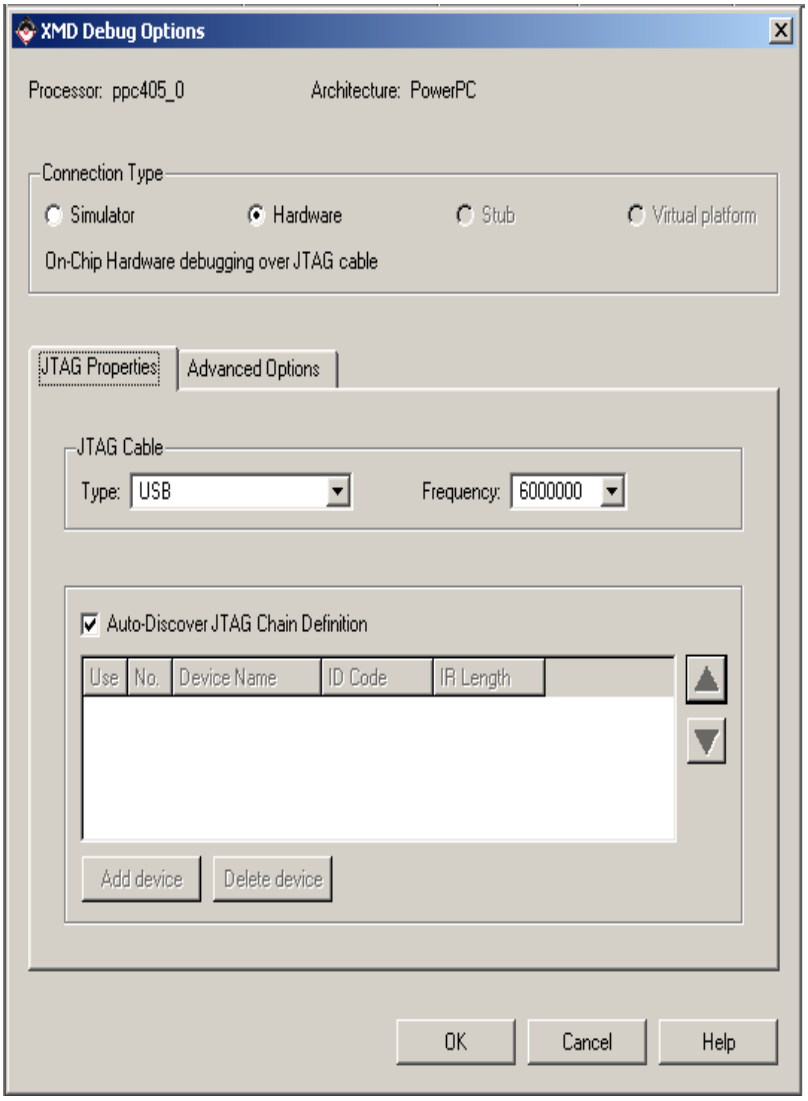


Bild E.6
XMD-Optionen

XMD: Software auf das Board laden, ausführen und stoppen

- Um das Board in einen definierten Zustand zu bringen, führen Sie als erstes mit dem Befehl `rst` einen Reset durch.
- Wechseln Sie mit `cd Projektname` in das Unterverzeichnis der compilierten Software.
- Laden Sie mit `dow executable.elf` die Software auf das Board.
- Überprüfen Sie mit dem Befehl `srrd`, ob sich der Programmzähler (Register `pc`) an der richtigen Stelle `fffffffc` befindet. Ist dies nicht der Fall, müssen Sie den Reset-Befehl erneut durchführen und die Software wieder hochladen.
- Mit dem Befehl `con 0xfffffffc` starten sie die das Programm.
- Abschließend wird die CPU mit dem Befehl `stop` angehalten und mit `exit` die Konsole verlassen.

XPS: Die Benutzeroberfläche

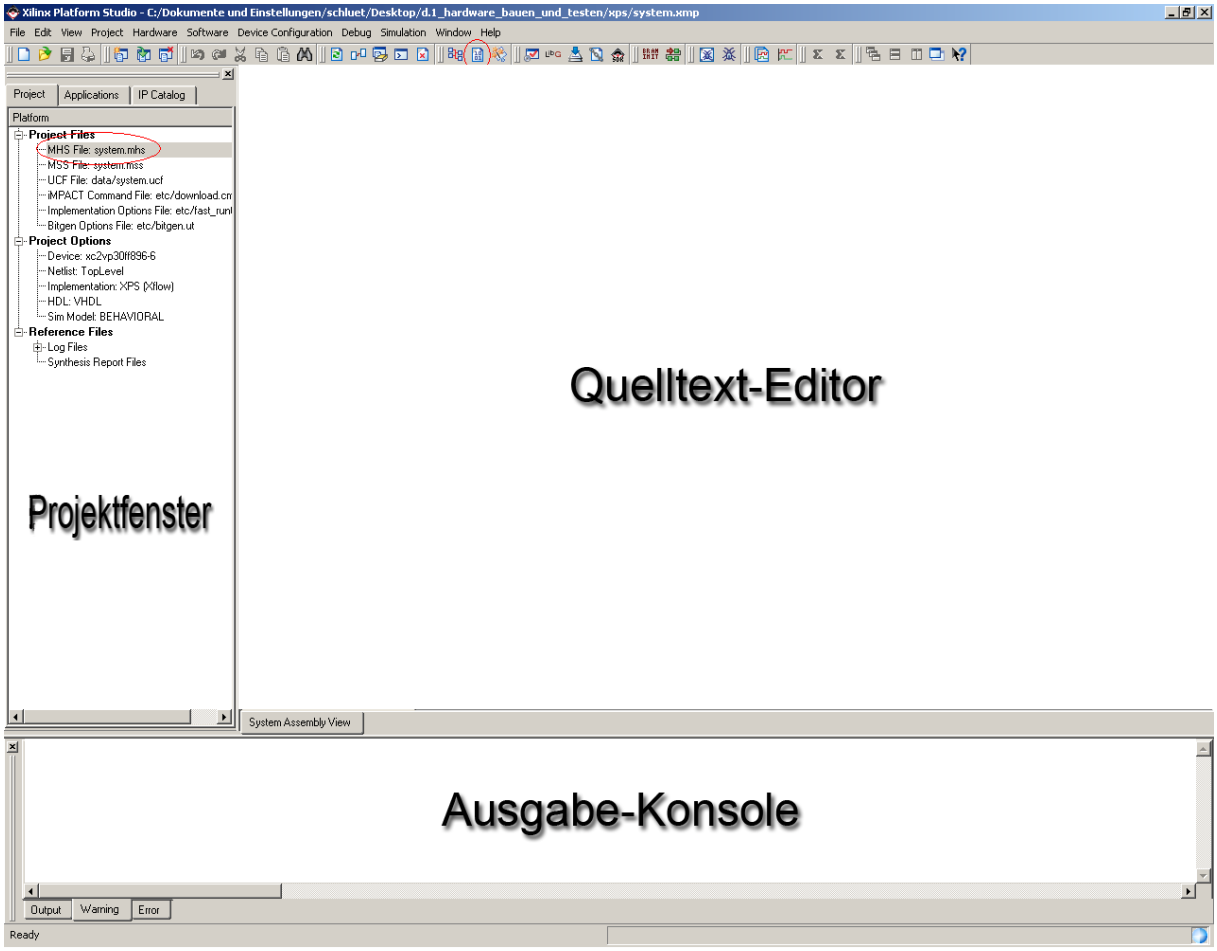


Bild E.7 Benutzeroberfläche von XPS

- Projektfenster: Hier haben sie Zugriff auf die Hardware-Komponenten ihres Systems, können Module miteinander verdrahten und Projektoptionen einstellen (Bild E.7).
- Quelltext-Editor: Hier können sie projektspezifische Dateien betrachten und verwalten.
- Ausgabe-Konsole: Die Konsole gibt Statusmeldungen, Fehler und Warnungen z.B. bei der Synthese aus.
- Zwei wichtige Funktionen sind in der Grafik extra hervorgehoben. Mit dem oberen Button kann der Bitstrom erzeugt werden und hinter der system.mhs-Datei verbirgt sich die Anzeige der Hardware-Modulinstanzen und ihrer Verdrahtungen.

XPS: Neues Projekt anlegen

- Nachdem Sie XPS und den Base System Builder Wizard gestartet haben, wählen Sie ein Verzeichnis für Ihr Projekt, bennenen Sie dieses mit der Endung *.xmp und bestätigen Sie mit OK. Der Base System Builder wird gestartet:
- Wählen Sie im Welcome-Dialog I would like to create a new design.
- Im Dialog Select Board wählen Sie als Board Vendor *Xilinx*, als Board Name *Virtex-II Pro ML310 Evaluation Platform*, und als Board Revision *D* aus. Klicken Sie auf Weiter.
- Im Dialog Select Processor wählen Sie *PowerPC*. Klicken Sie auf Weiter.
- Nehmen Sie keine Änderungen an den Default-Einstellungen des Prozessors vor.
- Lassen Sie im folgenden Dialog lediglich das Häkchen vor RS232_Uart stehen und entfernen Sie die übrigen Häkchen. Klicken Sie auf Weiter.
- Im anschließenden Dialog soll nur LCD-OPTIONAL ausgewählt bleiben. Klicken Sie auf Weiter.
- Wählen Sie im Dialog Add Internal Peripherals auf für den XPS_BRAM_IF_CNTLR eine Größe von 64 KB. Klicken Sie auf Weiter.
- Im Dialog Software Setup entfernen Sie die Häkchen vor Memory Test und Peripheral Self Test. Klicken Sie auf Weiter.
- Klicken Sie abschließend auf Generate – Finish.

XPS: IP-Core erstellen ohne DCR-Anschluss

- Klicken Sie auf den Button Create or Import Peripheral.
- Klicken Sie auf Next.
- Wählen Sie Import existing peripheral und klicken Sie zwei mal auf Next.
- Step 1: Geben Sie den Namen Ihres Top-Level-Moduls Ihres einzubindenden Codes an und aktivieren sie den Haken bei Use Version. Klicken Sie dann auf Next.
- Step 2: Wählen Sie den Punkt HDL Source Files (*.vhd, .v) und klicken Sie auf Next.
- Step 3: Stellen Sie VERILOG als Sprache ein und wählen den Punkt Browse to your existing HDL source and dependent library files (.vhd, *.vhdl, *.vh, *.v) in next step. Klicken Sie auf Next.

- Step 4: Fügen Sie die Verilog-Dateien hinzu, von denen ein IP-Core erstellt werden soll und klicken Sie auf Next.
- Step 5: Entfernen Sie den Haken bei Select Bus Interface(s). Klicken Sie dann auf Next.
- Step 6: Entfernen Sie den Haken bei Select and configure Interrupts und klicken Sie auf Next.
- Step 7: Klicken Sie auf Next.
- Klicken Sie auf Finish.
- Die IP-Cores sind nun im Unterverzeichnis pcores des Projektverzeichnisses gespeichert.

XPS: IP-Core erstellen mit DCR-Lese- oder -Lese-Schreibzugriff

- Klicken Sie auf den Button Create or Import Peripheral.
- Klicken Sie auf Next.
- Wählen Sie Import existing peripheral und klicken Sie zwei mal auf Next.
- Step 1: Geben Sie den Namen Ihres Top-Level-Moduls Ihres einzubindenden Codes an und aktivieren sie den Haken bei Use Version. Klicken Sie dann auf Next.
- Step 2: Wählen Sie den Punkt HDL Source Files (.vhd, *.vhdl, *.vh, *.v) und klicken Sie auf Next.
- Step 3: Stellen Sie VERILOG als Sprache ein und wählen den Punkt Browse to your existing HDL source and dependent library files (.vhd, *.vhdl, *.vh, *.v) in next step. Klicken Sie auf Next.
- Step 4: Fügen Sie die Verilog-Dateien hinzu, von denen ein IP-Core erstellt werden soll und klicken Sie auf Next.
- Step 5: Markieren Sie das Feld DCR Slave und klicken Sie auf Next.
- Step 6: Klicken Sie auf Next.
- Step 7: Klicken Sie auf Next.
- Step 8: Entfernen Sie den Haken bei Select and configure Interrupts und klicken Sie auf Next.
- Step 9: Klicken Sie auf Next.
- Step 10: Klicken Sie auf Next.

- Klicken Sie auf Finish.

XPS: Instanz vom IP-Core ohne DCR-Anschluss hinzufügen

- Wählen Sie im Projektfenster den Reiter IP Catalog, öffnen Sie das Menü USER
- Wählen Sie ihren IP-Core mit der rechten Maustaste aus und klicken Sie auf Add IP.

XPS: Instanz vom IP-Core mit DCR-Lesezugriff hinzufügen

- Wählen Sie im Projektfenster den Reiter IP Catalog, öffnen Sie das Menü USER
- Wählen Sie ihren IP-Core mit der rechten Maustaste aus und klicken Sie auf Add IP.
- Wechseln Sie im Quelltext-Editor den Reiter Bus Interfaces.
- Wählen Sie unter Ihrem IP-Core für SDCR den Eintrag dcr_v29_0
- Wechseln Sie in den Tab Addresses.
- Tragen Sie für Ihren IP-Core als Base Address 0b0000000000 ein.
- Stellen Sie unter Size den Wert 4 ein.

XPS: Instanz vom IP-Core mit DCR-Lese-und-Schreibzugriff hinzufügen

- Wählen Sie im Projektfenster den Reiter IP Catalog, öffnen Sie das Menü USER
- Wählen Sie ihren IP-Core mit der rechten Maustaste aus und klicken Sie auf Add IP.
- Wechseln Sie im Quelltext-Editor den Reiter Bus Interfaces.
- Wählen Sie unter Ihrem IP-Core für SDCR den Eintrag dcr_v29_0
- Wählen Sie unter dem PowerPC-IP-Core für MDCR den Eintrag dcr_v29_0
- Wechseln Sie in den Reiter Addresses.
- Tragen Sie für Ihren IP-Core als Base Address 0b0000000000 ein.
- Stellen Sie unter Size den Wert 4 ein.

XPS: IP-Core ohne DCR-Anschluss verdrahten

- Zur Verdrahtung klicken Sie im Projektfenster doppelt auf den Eintrag MHS File: system.mhs. Instanzen neu hinzugefügter IP-Cores befinden sich ganz unten im Quelltext. Sie müssen anderen Instanzen aber noch die Ein- und Ausgänge Ihres IP-Cores zur Verfügung stellen. Das geschieht für jedes Signal mit folgender Code-Zeile:

```
PORT DBG_INOUT = NAME
```

- `INOUT` ersetzen Sie durch `IN` oder `OUT`. Für `NAME` wählen Sie eine Bezeichnung, mit der das Signal anderen Instanzen zur Verfügung steht.

XPS: IP-Core mit DCR-Lesezugriff verdrahten

- Öffnen Sie die Datei `system.mhs`, indem Sie im Projektfenster doppelt auf den Eintrag MHS File: system.mhs klicken.
- Fügen Sie in der Sektion name_des_IP_cores zwischen den beiden Zeilen

```
BUS_INTERFACE SDCR = dcr_v29_0
END
```

folgendes ein:

```
PORT nRESET = sys_rst_s
PORT CPU_CLK = sys_clk_s
PORT DBG_IN = APLUSB_DBG_IN
PORT DBG_OUT = APLUSB_DBG_OUT
PORT DBG_DCR_WRITE = DBG_DCR_WRITE
```

- Korrigieren Sie die Zeile

```
PARAMETER C_DCR_HIGHADDR = 0b000000000011
```

indem Sie zwei der Nullen hinter dem „b“ löschen.

XPS: IP-Core mit DCR-Lese-und-Schreibzugriff verdrahten

- Öffnen Sie die Datei `system.mhs`, indem Sie im Projektfenster doppelt auf den Eintrag MHS File: system.mhs klicken.
- Verdrahten Sie den Reset- und Clock-Port durch Hinzufügen der folgenden zwei Zeilen in der Sektion name_des_IP_cores :

```
PORT nRESET = sys_rst_s
PORT CPU_CLK = sys_clk_s
```

- Korrigieren Sie die Zeile

```
PARAMETER C_DCR_HIGHADDR = 0b000000000011
```

indem Sie zwei der Nullen hinter dem „b“ löschen.

XPS: Bitstrom generieren

- Klicken Sie in der dritten Button-Zeile auf den zwanzigsten Button, der einen Zettel mit Einsen und Nullen darstellt oder klicken Sie auf Tools – Generate Bitstream.
- Es ist sehr wichtig, dass Sie bei dem 5- bis 10-minütigen Vorgang auf Fehlerausgaben in der Konsole achten! Eine erfolgreiche Bitstromgenerierung endet mit einer Meldung wie in Beispiel E.8.

```
DRC detected 0 errors and 6 warnings.
Saving bit stream in "system.bit".
Creating bit mask...
Saving mask bit stream in "system.msk".
Bitstream generation is complete.
Done.
```

Beispiel E.8 XPS-Konsole: erfolgreiche Bitstrom-Erzeugung

- Ein fehlerhafter Bitstrom darf auf keinen Fall auf das FPGA geladen werden, da es sonst beschädigt werden könnte. Eine häufige Fehlerursache sind falsche Verdrahtungen in der Datei `system.mhs`. Überprüfen Sie diese im Fehlerfall und wenden Sie sich an ihren Hiwi, falls Sie keinen Fehler entdecken können.
- Ihr fertig erzeugter Bitstrom `system.bit` befindet sich in ihrem Unterverzeichnis `implementation` ihres Projekts.

XPS: Software zum Projekt hinzufügen

- Wählen Sie im XPS-Projektfenster den Tab Applications.
- Doppelklicken Sie auf Add Software Application Project...
- Geben Sie einen Projektnamen ein und wählen Sie als Prozessor-Instanz ppc405_0 aus.
- Mit einem Rechtsklick auf Sources | Add File… können Sie nun Quelltextdateien zum Projekt hinzufügen.

XPS: Software bearbeiten

- Wählen Sie im XPS-Projektfenster den Tab Applications.
- Unter Sources finden Sie die C-Quellcode-Dateien Ihres Projektes.
- Mit einem Doppelklick auf die gewünschte Datei öffnen Sie diese.

XPS: Programm compilieren

- Klicken Sie im Projektfenster mit der rechten Maustaste auf Ihr Software-Projekt und wählen Sie Build Project.
- Die Ausgabe-Konsole sollte nach fehlerfreiem Ablauf der aus Beispiel E.9 ähneln.

```
Text     data      bss      dec      hex   filename
2150      808     8196    11154     2b92   projektname/executable.elf
Done.
```

Beispiel E.9 Erfolgreiche Software-Compilierung

- Muss für Ihr Projekt ein Linker-Skript erzeugt werden, führen Sie folgende Schritte zusätzlich aus:
- Klicken mit der rechten Maustaste auf Ihr Software-Projekt und wählen Sie Generate Linker Script...
- Bestätigen Sie mit Generate.
- Compilieren Sie die Software erneut.

Lehrbücher

Einige der folgenden Lehrbücher haben wesentlich zum vorliegenden Skript beigetragen. Sie enthalten auch Hinweise auf die sehr umfangreiche Fachliteratur.

Baue02 Bauer, L., Perspektiven des modernen ASIC-Designs, Dissertation, TU Berlin, 2002.

Bark02 Barke, E., Eine kleine Einführung in die Mikroelektronik, IMS, Universität Hannover, 2002.

Berg01 Berger, A.S., Embedded Systems Design, McGraw-Hill, 2001.

Bhat01 Bhatnagar, H., Advanced ASIC Chip Synthesis, Kluwer, 2001.

Buch99 Buchenrieder, K.J. (Hrsg.), Hardware-Software-Codesign; ITpress, 1999.

Cava07 Cavanagh, J., Verilog HDL, Digital Design and Modeling, CRC Press, 2007.

HaSa01 Hassoun, S., Sasao, T., Logic Synthesis and Verification, Kluwer, 2001.

HePa94 Hennessy, J.L., Patterson, D.A., Rechnerarchitektur, Vieweg, 1994.

Marw07 Marwedel, P., Eingebettete Systeme, Springer, 2007.

Plat01 Platzner, M., Hardware-Software-Codesign; Vorlesungsskript, Institut für Technische Informatik und Kommunikationsnetze, ETH Zürich, 2001.

Suthe01 Sutherland, S., Verilog-2001, Kluwer, 2001.

ThMo02 Thomas, D.E., Moorby, P.R., The Verilog Hardware Description Language, Springer, 2002.

Index

Allgemeine Begriffe

VERILOG-Begriffe

Namen und Abkürzungen