Hardware/Software-Codesign : Massgeschneiderte elektronische Systeme : Teil 1 : HW/SW-Architekturen und Spezifikation

Autor(en): **Teich, Jürgen**

Objekttyp: **Article**

Zeitschrift: **Bulletin des Schweizerischen Elektrotechnischen Vereins, des Verbandes Schweizerischer Elektrizitätsunternehmen = Bulletin de l'Association Suisse des Electriciens, de l'Association des Entreprises électriques suisses**

Band (Jahr): **87 (1996)**

Heft 25

PDF erstellt am: **12.07.2024**

Persistenter Link: https://doi.org/10.5169/seals-902405

Die zunehmende Automatisierung im Bereich des Entwurfs elektronischer Schaltungen hat dazu geführt, dass man immer komplexere Systeme in immer kürzerer Zeit mit Hilfe von CAD-Werkzeugen entwickeln kann. Aus Kosten- und Effizienzgründen bestehen die betrachteten Systeme typischerweise aus einer Kombination von (programmierbaren) Hardware- und Softwarekomponenten. Unter dem Schlagwort Hardware/Software-Codesign verbirgt sich das heutige Bestreben, dem Entwurf ganzer, komplexer Systeme mit Hilfe von CAD-Werkzeugen Herr zu werden. Dieser Artikel gibt eine einführende Übersicht über die Entwurfsprobleme, mit denen sich dieses Forschungsgebiet beschäftigt.

Hardware/Software-Codesign: Massgeschneiderte elektronische Systeme

Teil 1: HW/SW-Architekturen und Spezifikation

■ Jürgen Teich

1. Was ist Hardware/Software-Codesign?

Obwohl der Begriff Hardware/Software-Codesign als neues Zauberwort in vieler Munde ist, sind bestimmte Ausprägungsformen und Problemstellungen der damit bezeichneten Technik bereits seit vielen Jahren bekannt. Ein Ingenieur beispielsweise, der ein neues System entwikkelt, das einen Mikroprozessor als Schaltungsbestandteil enthält, baut zuerst die Hardware (Platine, ASIC usw.) auf und programmiert anschliessend die programmierbaren Komponenten (Software).

1.1 Wachsende Komplexität

Wir betrachten den automatisierten Entwurf und die Optimierung komplexer digitaler Systeme, die aus Hardware- und Softwarekomponenten bestehen, sogenannte Hardware/Software-Systeme. Obwohl solche Systeme bereits seit vielen Jahren von Ingenieuren und Technikern konzipiert und gebaut werden, ist man sich heutzutage darüber einig, dass man nur durch den Einsatz rechnergestützter Entwurfsmethoden (Computer-aided Design) die Komplexität moderner Systeme bewältigen und bessere Entwürfe (Performanz, Kosten) in kürzerer Zeit schaffen kann. Dies erklärt das wachsende Interesse der Industrie und der Forschung an rechnergestützten Entwurfsmethoden.

Die Komplexität, so wie sie hier verstanden wird, entsteht nicht nur durch die Anzahl der Einzelkomponenten, aus denen ein System zusammengesetzt ist, sondern vor allem durch *Heterogenität*. In Zukunft liegen die Anforderungen gerade bei der Beherrschung heterogener technischer Systeme, die sich durch verschiedenartige Komponenten und Interaktionen auszeichnen und die für einen ganz bestimmten Anwendungsbereich zugeschnitten sind, bei den sogenannten *anwendungsspezifischen Systemen*. Solche Anwendungsbereiche sind unter anderem die Bereiche Medizintechnik, Prozess- und Industriesteuerungen, digitale Netzwerke, Telekommunikation und digitale Signal- und Bildverarbeitung. Bemerkenswerterweise lag 1991 der Marktanteil solcher Systeme bereits bei umgerechnet 31 Milliarden US-Dollar gegenüber 46,5 Milliarden US-Dollar für Vielzweckrechner (PC, Laptops, Workstations) mit einer Wachstumsrate von 18% gegenüber 10% für Vielzweckrechner [1].

Ein konkretes Beispiel eines solchen Systems ist die in Bild 1 dargestellte integrierte Schaltung, die innerhalb eines

Adresse des Autors
PD Dr.-Ing. *Jürgen Teich*, Institut für Technische Informatik und Kommunikationsnetze (TIK) ETH Zürich, 8092 Zürich

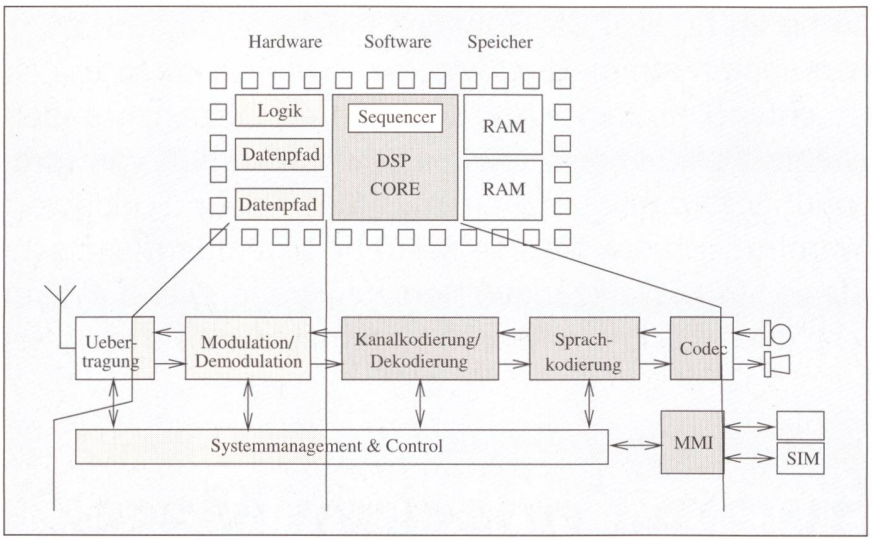

Bild 1 Blockdiagramm der Funktion eines zellularen Telefons nach dem GSM-Standard (unten) und Abbildung der Blöcke auf eine Ein-Chip-Realisierung mit einem DSP-Core (oben)

mobilen Telefongerätes (GSM-Standard) eingesetzt wird.

Beispiel 1.1

Das Bild 1 aus [2] zeigt ein typisches heterogenes Hardware/Software-System, bestehend aus einem Prozessor (DSP), anwendungsspezifischer Hardware und Peripherie. Es handelt sich um eine Ein-Chip-Realisierung des GSM-Standards für ein zellulares Telefon. Das Bild zeigt die Zuteilung der einzelnen Segmente eines Blockdiagramms auf die Architektur. Der Prozessor wird eingesetzt, um Aufgaben mit niedrigen bis mittleren Datenraten (Kodierung/Dekodierung) sowie Steuerungsfunktionen zu übernehmen. Die Blöcke, die höhere Rechenleistungen erfordern (Modulation und Demodulation), werden in anwendungsspezifischer Hardware realisiert.

Da solche Systeme meistens in einen technischen Kontext eingebettet sind, spricht man auch von *eingebetteten Systemen* (Bild 2). Diese sind dazu bestimmt, Funktionen als Antwort auf bestimmte Stimuli auszuführen und Daten informationstechnisch zu verarbeiten. Neben dem Hardware/Software-Heterogenitätsaspekt sind bei diesen Systemen auch die Aspekte mechanisch/elektrisch (Sensoren, Aktoren) und Analog/Digital (A/D- und D/A-Wandler) wichtig.

Das grosse Interesse am systematischen Entwurf von eingebetteten Systemen ist zurückzuführen auf

- Fortschritte in den Schlüsseltechnologien (Mikroelektronik, formale Methoden). Dadurch ergibt sich eine
- steigende Vielfalt von Anwendungen und Leistungsanforderungen, verbunden mit
- der Notwendigkeit, Entwurfs- und Testkosten zu senken.

Um die Entwicklung in diesem Bereich genauer zu verstehen, führt man sich sinnvollerweise die historische Entwicklung kurz vor Augen. Insgesamt lassen sich drei Abschnitte erkennen:

- Nachdem die Zahl der Objekte in den unteren Entwurfsebenen (Geometrie, physikalische Ebene) aufgrund des Zeitaufwandes und der Fehleranfälligkeit ohne Automatisierung nicht mehr handhabbar war, wurden in der Industrie und in der Forschung Modelle und Methoden für die Schaltungssimulation, Plazierung und Verdrahtung entwickelt.
- In einem weiteren Schritt wurden dann auch höhere Abstraktionsebenen in die Automatisierung einbezogen, wie zum Beispiel die Simulation von Schaltungen auf Logikebene oder die Logiksynthese.
- Neue Anforderungen bezüglich der Systemkomplexität, der Zeitspanne zwischen Produktidee und Markteinführung sowie der Zuverlässigkeit und Güte führen nun zur Entwurfsautomatisierung auf der noch abstrakteren Systemebene.

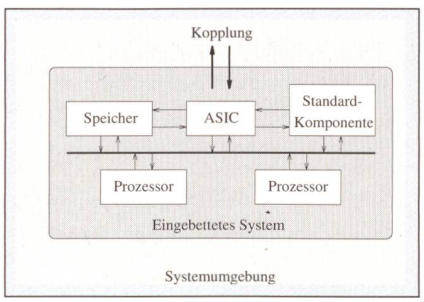

Bild 2 Schematische Darstellung eines eingebetteten Systems

Auf der Systemebene besteht eine zentrale Aufgabe darin, eine Aufteilung der Funktionalität in Hardware- und Softwarekomponenten vorzunehmen, die sogenannte *Hardware/Software-Partitionierung*.

Beispiel 1.2

Ein Netzwerk-Controller soll entworfen werden, der einen Speicher mit einer seriellen Schnittstelle koppelt (Bild 3 aus [3]). Seine Aufgabe besteht darin, Daten über die serielle Schnittstelle zu senden und zu empfangen und dabei ein bestimmtes Protokoll einzuhalten (z. B. CS/CD für Ethernet). Dabei ist die maximale Datenübertragungszeit T_{max} (in ns) für ein Kilobyte an Daten einzuhalten. Das System ist ferner einer Kostenschranke K_{max} (in Franken) unterworfen und soll nicht mehr als P_{max} (in mW) Leistung verbrauchen. Offensichtlich muss man zunächst die Entscheidung treffen, welche Aufgaben in Software und welche Aufgaben in Hardware realisiert werden. Eine exemplarische Aufteilung ist in Bild 3 dargestellt.

In den meisten Fällen erfolgt die Hardware/Software-Partitionierung durch Abschätzung von Kosten und Performanzanforderungen nach dem jeweiligen Erfahrungswissen des Entwicklungsingenieurs. Da diese Entwurfsentscheidung auf groben Schätzungen beruht, ist keine Gewähr geleistet, dass das realisierte System alle Entwurfsbeschränkungen erfüllt beziehungsweise dass es in einer gewünschten Hinsicht optimal ist. Solche Systeme sind meist in mindestens einer Eigenschaft unter- oder überdimensioniert, wie das folgende Beispiel zeigt.

Beispiel 1.3

Das Bild 4 zeigt verschiedene Lösungen zur Realisierung des Netzwerk-Controllers aus Beispiel 1.2. Die Menge von Entwurfsbeschränkungen kennzeichnet hier einen dreidimensionalen Entwurfsraum mit den Achsen Datenübertragungszeit/kByte T, Kosten K und Leistungsverbrauch P. Ein mit der Entwicklung beauftragter Softwareingenieur konstruierte ein System mit einem Mikroprozessor und erhielt ein System mit den durch den Entwurfspunkt P_1 gekennzeichneten Eigenschaften. Offensichtlich erfüllt diese Lösung zwar die Kostenanforderungen, aber nicht die Datenratenbeschränkung. Ein Hardwareingenieur entwickelte eine dedizierte integrierte Schaltung, deren Eigenschaften durch den Punkt P_2 dargestellt sind. Offensichtlich erfüllt diese Realisierung die Performanzanforderungen und Leistungsverbrauchsanforderungen, nicht aber die Kosten. Die Schaltung wurde überdimensioniert. Der Punkt P_4 entspricht

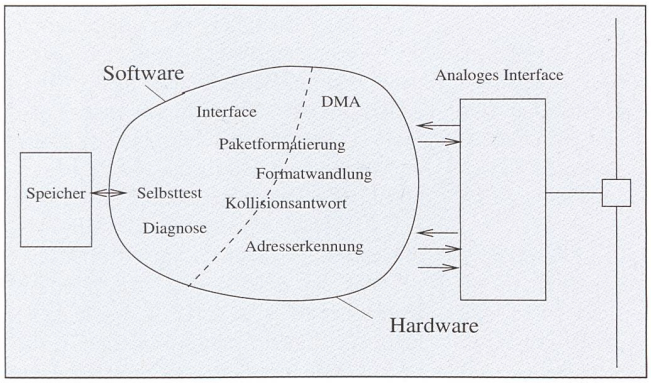

Bild 3 Netzwerk-Controller: Exemplarische Partitionierung der zu implementierenden Funktionalität in Hardware und Software

einer gemischten Hardware/Software-Lösung gemäss der Partitionierung in Bild 3.

Viele Lösungen, bei denen die Hardware/Software-Partitionierung ad hoc bestimmt wurde, erfüllen die Entwurfsbeschränkungen nicht oder sind suboptimal.

Aus diesen Beispielen sollte deutlich geworden sein, dass automatische Syntheseverfahren von heterogenen Systemen unbedingt erforderlich sind, damit die Systementwicklung mit dem Technologiefortschritt Schritt halten kann und der Entwurf effizienter Systeme möglich ist.

1.2 Problemstellungen

Die Silbe «Co» im Wort Codesign (deutsch: Co-Entwurf) erlaubt zahlreiche Interpretationen, die zusammen gesehen die wichtigsten Problemstellungen dieses Forschungsgebietes umfassen.

- Complexity (Komplexität der betrachteten Systeme): Wie bereits erwähnt, zeichnet sich die Komplexität in unserem Zusammenhang vor allem durch die Heterogenität der betrachteten Komponenten aus.
- Concurrent Design: HW/SW-Codesign nutzt die Synergie von Hardware und Software durch einen gemeinsamen Entwurf aus.
- Co-Specification (Spezifikation: heterogen/homogen): Offensichtlich besteht ein grosses Problem darin, wie man ein komplexes Hardware/Software-System spezifiziert.
- Co-Synthesis: Hierunter versteht man das gemeinsame Synthetisieren von Hardware und Software, wobei das Problem der Entwurfsraumexploration von gemischten Lösungsformen sowie die Optimierung im Vordergrund stehen.
- Correctness: Aufgrund der Heterogenität der Komponenten ist die Validierung eines Entwurfs auf Korrektheit (Simulation, Verifikation) schwierig, da existierende Werkzeuge entweder nur auf Hardware- oder nur auf Softwarebereiche zugeschnitten sind. Die Probleme der Kopplung von Simulatoren und des Beweisens von Systemeigenschaften sind mit den Problemen der Spezifikation verwandt.
- Coordination: Schliesslich zielt man im Bereich des Hardware/Software-Codesigns auf eine Automatisierung der Entwurfsabläufe. Dazu gehört die Kopplung existierender Werkzeuge, die Verwaltung von Versionen sowie die Möglichkeit, den Entwurfsprozess über grafische Benutzeroberflächen zu steuern usw.

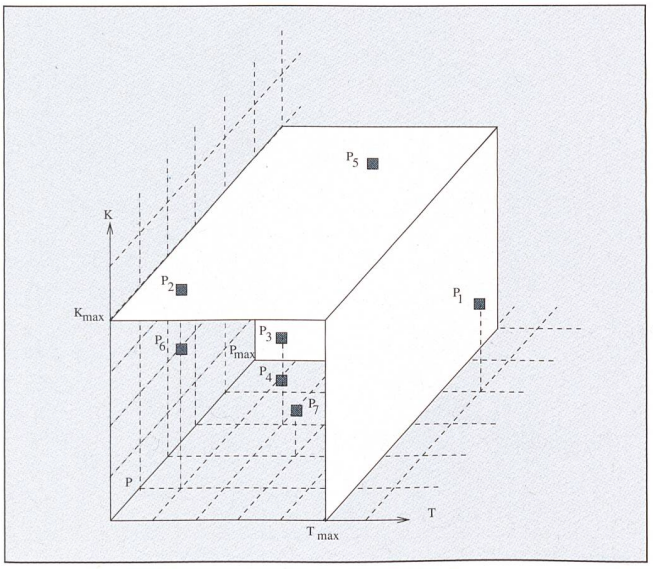

Bild 4 Verschiedene Realisierungsvarianten eines Netzwerk-Controllers aus Beispiel 1.2

Obwohl die hier beschriebenen Problemstellungen schon zum Teil bekannt oder zumindest unbewusst als Probleme wahrgenommen worden sind, versucht man jetzt – da die Werkzeuge für untere Entwurfsebenen zunehmend ausgereift sind – verstärkt, deren Lösung in Angriff zu nehmen und die letzte Entwurfsstufe der Systemebene zu automatisieren.

Im folgenden möchten wir die Vielfalt von Realisierungsvarianten und Komponenten vorstellen, die ein Systementwickler gezielt einsetzen sollte. Neben der Übersicht werden Faustregeln angegeben, wann sich welche Realisierungsform beziehungsweise Komponente am besten einsetzen lässt. Danach wollen wir uns mit der automatischen Auswahl der Hardware- und Softwarekomponenten, der sogenannten Hardware/Software-Partitionierung, beschäftigen. Sie ist Teil der Systemsynthese. Aufgrund von Platzbeschränkungen werden wir uns nur am Rande mit den Problemen der Spezifikation, Validierung und Entwurfsablaufssteuerung beschäftigen.

2. HW/SW-Architekturen

Im folgenden wollen wir die wichtigsten typischen Realisierungsformen von HW/SW-Systemen und deren Komponentenarten klassifizieren. Dabei zeigt es sich, dass unterschiedliche Komponententypen auf unterschiedliche Aufgaben und Anwendungsbereiche zugeschnitten sind. Eine Übersicht soll dem Entwicklungsingenieur helfen, die für seine Bedürfnisse notwendigen Komponenten beurteilen und auswählen zu können.

2.1 Komponenten

Zunächst sollen einige Gesichtspunkte von unterschiedlichen, im Bereich von Hardware/Software-Systemen verbreiteten Prozessortypen dargestellt werden.

2.1.1 Klassifikation von Prozessoren

In eingebetteten Hardware/Software-Systemen, wie sie zum Beispiel in Automobilen, Audio- und Videoprodukten der Telekommunikation eingesetzt werden, ist zunächst ein geeigneter Prozessortyp für das zu entwerfende System auszuwählen. Nach dem heutigen Stand der Technik kann diese Wahl durch folgende Kriterien beeinflusst sein:

Vielzweck – anwendungsspezifisch: Für bestimmte Anwendungsgebiete, wie zum Beispiel für die digitale Signalverarbeitung, sind anwendungsspezifische Prozessoren nötig, weil nur sie den geforderten Leistungsanforderungen genügen. Digitale Signalprozessoren (DSP) unterstützen beispielsweise Spezialoperationen wie Multi-

plizier/Addierinstruktionen in einem Maschinenzyklus und besitzen spezielle Adressierungsarten sowie heterogene Registersätze. Ein anderer Anwendungsbereich, der spezielle Architekturen hervorgebracht hat, ist der Bereich der Prozesssteuerungen (Embedded Control). Dort spielen Mikrocontroller-Architekturen, die auf minimale Kontextwechselzeiten, minimale Interruptlatenzen oder auf minimale Kosten optimiert sind (Bitbreite 8 Bit oder 16 Bit, minimale Speichergrösse), eine wesentliche Rolle. Sie besitzen in den meisten Fällen eine CISC-Architektur (Complex Instruction Set Computer) im Gegensatz zu heutigen Vielzweckrechnern, die fast alle RISC-(Reduced Instruction Set Computer) oder superskalare Rechner sind, da bei CISC-Rechnern im allgemeinen eine wesentlich höhere Codedichte (d. h. weniger Programmspeicher) erzielt werden kann.

Chip – Layoutzelle (Core): Ein Prozessor kann entweder als Chip in einem Gehäuse oder als Layoutzelle (Processor Core) verfügbar sein. Falls die Layoutzelle von einer Firma entworfen wurde und nicht nach aussen hin verkäuflich ist, spricht man von sogenannten In-house Cores. Ein Beispiel eines Cores ist in Bild 7 dargestellt.

Konfigurierbarkeit: Die interne Architektur eines Prozessors kann entweder fest (Off-the-Shelf Processor) oder konfigurierbar (ASIP, Application-specific Instruction Set Processor) sein. Erstere haben den Vorteil, dass Compiler verfügbar sind; sie haben allerdings auch eine Menge von Nachteilen: Sie sind entweder zu teuer (Fläche, Kosten) oder für gewisse Anwendungen nicht einsetzbar. Bei portablen Geräten ist beispielsweise der Leistungsverbrauch des Prozessors entscheidend, so dass Standardrealisierungen nicht eingesetzt werden können. ASIP besitzen im allgemeinen eine Menge generischer Parameter, die ein Anwender individuell festlegen kann. Dazu gehören zum Beispiel die Grösse von Speichern, die Anzahl und Wortbreiten der funktionalen Einheiten, der Instruktionssatz, die Anzahl von Interruptleitungen, Technologieparameter (z. B. Versorgungsspannung, Taktrate) und viele andere mehr. Die optimale Auswahl von Instruktionen und Parametern von ASIP für bestimmte Anwendungsgebiete ist Gegenstand zahlreicher Forschungsprogramme, siehe zum Beispiel [4, 5, 6].

Die Architekturparameter von Prozessoren (insbesondere ASIP) unterscheidet man nach folgenden Gesichtspunkten:

Datentyp: Festpunkt- oder Fliesspunktarithmetik. Bei ASIP wird fast ausschliesslich mit Festpunktarithmetik gerechnet.

Codetyp – Mikrocode oder Makrocode: Bei Mikrocode – gilt für die meisten existierenden Typen von ASIP – benötigen alle Instruktionen einen Maschinenzyklus, siehe zum Beispiel [7]. Bei Makrocode kann eine Instruktion mehrere Zyklen benötigen (z. B. bei Einheiten mit Fliessbandverarbeitung).

Speicherorganisation – Load-Store oder Mem-Reg: ASIP sind üblicherweise Load-Store-Architekturen, das heisst, alle Maschinenoperationen arbeiten mit Register-Operanden, welche über einen Load-Befehl aus dem Speicher geladen (falls sie nicht bereits dort vorliegen) beziehungsweise aus einem Register über einen Store-Befehl in den Speicher abgelegt werden. ASIP besitzen im allgemeinen keinen Cache; der Speicher (RAM, ROM, Register) wird in den meisten Fällen auf dem Chip realisiert. Zur Speicherorganisation gehört auch die Registerstruktur, die entweder heterogen oder homogen sein kann. Bei homogenen Registersätzen kann im Prinzip jedes Register universell eingesetzt werden. Das Bild 5 zeigt eine Architektur mit heterogenem Registersatz.

Instruktionsformat – vertikal oder horizontal (bei VLIW-Maschinen [8, 9]): Alle uns bekannten ASIP-Typen besitzen ein vertikales Instruktionsformat.

Besonderheiten: Im weiteren besitzen ASIP häufig eine Reihe weiterer Besonderheiten wie beispielsweise spezielle arithmetische Einheiten, besondere Adressierungsarten, Unterstützung von Schleifenkonstrukten (z. B. Zero-Overhead Looping) usw.

Beispiel 2.1

Bild 5 zeigt eine aus einem Operationswerk und einem Steuerwerk bestehende ASIP-Architektur. Eine besondere Eigenschaft des Datenpfads sind einige besondere Verbindungswege sowie eine gekoppelte Multiplizier/Addier-Einheit (MUL-ADD). Es handelt sich um eine Load-Store-Architektur mit Festpunktarithmetik und vertikalem Mikrocode sowie heterogenem Registersatz (Adressregister A1, A2, AR, Datenregister R1, R2, MR). Das Steuerwerk dekodiert im Instruktionsdekoder die Befehlsworte. Als periphere Komponenten sind A/D(Analog/Digital)- und D/A(Digital/Analog)-Wandler, Timer, serielle Schnittstellen oder ein DMA-Controller (Direct Memory Access) konfigurierbar.

2.1.2 Klassifikation von Hardwarekomponenten

Als Hardwarekomponenten betrachtet man zum einen *anwendungsspezifische integrierte Schaltungen* (ASIC). Diese in VLSI (Very Large Scale Integration) realisierten Komponenten können in verschiedenen Technologien ausgeführt sein. Heutzutage kann man auf einem ASIC gemischt digital/analoge Schaltungen realisieren und verschiedene Technologien koppeln (z. B. Bi-CMOS). Als Realisierungsformen unterscheidet man unter anderem Full Custom (vollkundenspezifisch), Standardzellen und Gate-Array-Entwürfe.

Zum anderen gibt es eine Menge von Peripheriebausteinen, zum Beispiel DMA-Bausteine, Interrupt-Controller, Busarbiter, Pulsweitenmodulatoren, Bausteine serieller Schnittstellen, die nützliche Schnittstellen zur Kopplung von Prozessoren und ASIC und zur Aussenwelt eines Systems (Sensoren, Aktoren) ermöglichen.

Schliesslich hat man ebenfalls den Markt für programmierbare Logikbausteine (z. B. FPGA, PLA) erkannt. Solche

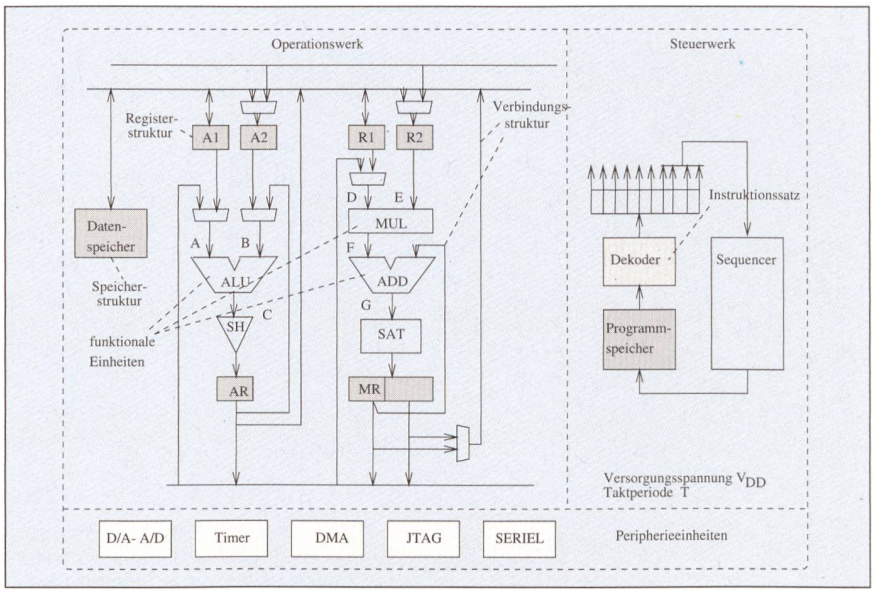

Bild 5 Beispiel einer ASIP-Architektur

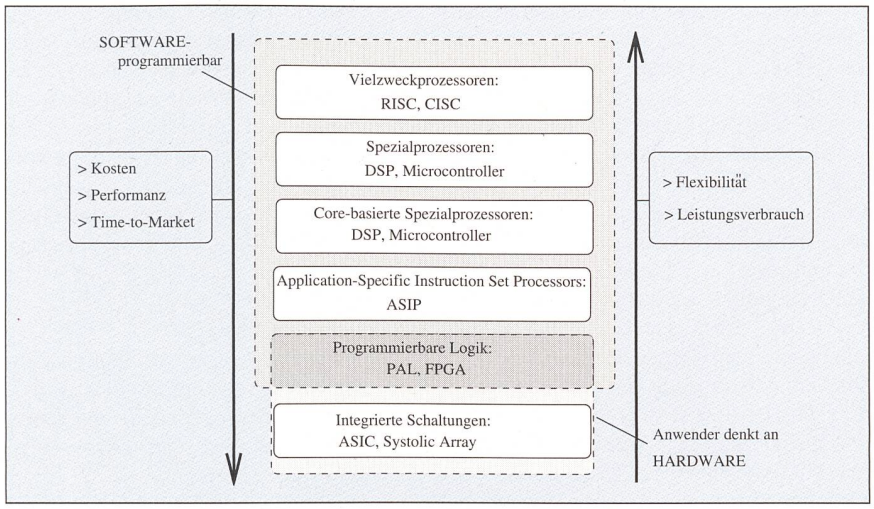

Bild 6 Spezialisierungsformen und Kriterien für Hardware/Software-Entscheidungen

Bausteine erlauben die Implementierung von Logikfunktionen und Zustandsmaschinen durch Programmierung und bieten daher eine ausgezeichnete Möglichkeit, ein System durch Umprogrammierung der Hardware flexibel an andere Systeme zu koppeln (z. B. Realisierung eines flexiblen Kommunikationsbausteins).

2.1.3 Wann welche Komponente?

ASIP stellen offensichtlich bezüglich Flexibilität und Performanz die Nahtstelle von der Softwareseite zur Hardwareseite her. Aus Kostengründen ist ein ASIP oft nur ein «abgespeckter» Prozessor und damit günstiger als ein Vielzweckprozessor, aber aufgrund seiner (wenn auch beschränkten) Programmierbarkeit immer noch flexibler als dedizierte Hardware.

Ein ASIC ist oft zu teuer, nicht flexibel genug oder bedarf einer zu hohen Entwicklungszeit. Die Nahtstelle zwischen Softwarewelt und Hardwarewelt von der Hardwareseite her bilden die programmierbaren Logikbausteine (Bild 6). Der Anwender denkt dabei an Hardware, obwohl er den FPGA-Baustein programmiert. FPGA besitzen als Hardware-Realisierungsvariante die Flexibilität von Softwarelösungen bei hoher Performanz. Allerdings sind die Performanz und die Auslastung der Ressourcen lange nicht so hoch wie bei ASIC, insbesondere nicht wie bei VLSI-Rechenfeldern (Systolic Arrays) [10]. Deshalb liegt der Anwendungsbereich von programmierbarer Logik vornehmlich bei der Realisierung von «kleinen Inseln» eines komplexen Systems, die hohe Flexibilitäts- und Performanzanforderungen erfüllen müssen.

2.2 Realisierungsformen von HW/SW-Systemen

Die oben beschriebenen Komponenten können nun entweder als *Ein-Chip-System, Ein-Platinen-System* oder *Mehr-Platinen-System* entworfen werden, denn die meisten Komponenten werden von Halbleiterherstellern bereits neben der üblichen Form eines ASIC in einem Gehäuse auch als Layout in Form einer Makrozelle für den VLSI-Entwurf angeboten.

Beispiel 2.2

Bild 7 zeigt eine physikalische Sicht einer Ein-Chip-Realisierung. Der Chip enthält einen Prozessor als Makrozelle (Core) sowie eine Menge von auf der Chipfläche integrierten programmierbaren Logik-, Speicher- und Peripherieblöcken (z. B. Timer, D/A- und A/D-Wandler). Bei einigen Anbietern sind die einzelnen Komponenten in Anzahl und Grösse individuell konfigurierbar.

Andere Realisierungsformen von hier betrachteten Hardware/Software-Systemen sind Ein- und Mehr-Platinen-Entwürfe. Jede Realisierungsform hat ihre Vor- und Nachteile:

Ein-Chip-Realisierungen haben den Vorteil, dass sie aufgrund ihres relativ geringen Gewichts und ihrer kleinen Grösse gut in mobilen Geräten und unabhängigen Systemen eingesetzt werden können. Gegenüber Vielzweckkomponenten können sie auch auf geringen Leistungsverbrauch optimiert werden. Ein Nachteil dieser Realisierungsform sind die relativ hohen Kosten bei kleinen Stückzahlen. Folglich werden sie bisher vor allem da eingesetzt, wo Massenproduktionen zu erwarten sind (bestes Beispiel: Mobiltelefon, Videotelefon).

Gegenüber Ein-Chip-Lösungen bieten sich Ein-Platinen-Entwürfe beziehungsweise Mehr-Platinen-Entwürfe dann an, wenn das zu realisierende System nicht auf einen Chip passt, wenn niedrige Stückzahlen die Verwendung von Standardkomponenten kostengünstiger machen und/oder falls eine gewisse Flexibilität bezüglich zu erwartender Änderungen erforderlich ist. Ausserdem sind die Fertigungszeiten für einen Platinenentwurf erfahrungsgemäss niedriger als jene für einen Chip. Nachteil dieser Varianten sind ein zu erwartender Performanzverlust aufgrund längerer Verdrahtungswege zwischen den Komponenten und ein höherer Leistungs- und Platzverbrauch.

Mehr-Platinen-Entwürfe werden im allgemeinen so konzipiert, dass sie erweiterbar beziehungsweise skalierbar sind. Sie sind dadurch im allgemeinen leicht wartbar und fehlertolerant.

3. Spezifikation von HW/SW-Systemen

Die Spezifikation von Hardware/Software-Systemen stellt aufgrund der Heterogenität der Komponenten ein grosses Problem dar, denn bekannte Spezifikationsformen sind stark auf einen Anwendungsbereich (z. B. kontrollflussdominant oder datenflussdominant) beziehungsweise entweder auf die Modellierung von Hardware (z. B. VHDL, Verilog) oder die Modellierung von Software zugeschnitten.

Im allgemeinen unterscheidet man an Spezifikationsformen *Berechnungsmodelle* und *Spezifikationssprachen*. Berechnungsmodelle besitzen eine formale, mathematische Struktur (z. B. Petri-Netze, endliche Automaten). Spezifikationssprachen sind im wesentlichen Programmiersprachen. Manche Sprachen vermögen ein oder mehrere Modelle auszudrücken. Einige Sprachen besitzen keine formale Semantik.

3.1 (Formale) Berechnungsmodelle

Ein *Kontrollflussmodell*, wie beispielsweise das Modell des endlichen Automaten, im weiteren FSM (Finite State Machine) genannt, repräsentiert ein System als

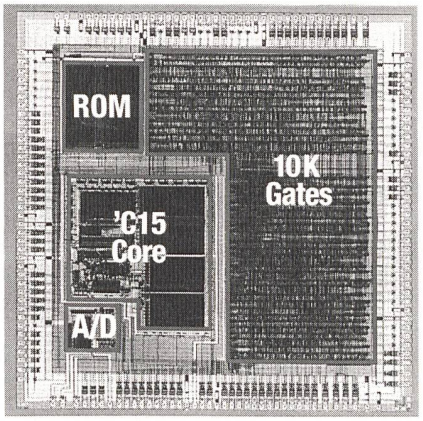

Bild 7 Physikalische Sicht einer Ein-Chip-Realisierung eines Hardware/Software-Systems
Quelle: Texas Instruments, cDSP

eine Menge von Zuständen und Zustandsübergängen. Ein kontrollflussdominantes Modell ist am besten zur Modellierung von Steuerungsaufgaben geeignet, wie sie beispielsweise in reaktiven Echtzeitsystemen vorkommen. Hierzu zählen auch erweiterte, hierarchische Automatenmodelle, wie zum Beispiel Statecharts [11].

Datenflussmodelle basieren auf Datenflussgraphen, in denen Aktoren als Knoten und deren Datenabhängigkeiten als Kanten dargestellt sind. Solche Modelle sind am besten zur Modellierung *transformationaler Systeme* geeignet, wie sie beispielsweise in der digitalen Signalverarbeitung vorkommen. Dort werden Daten einer Reihe von Transformationen unterworfen. Dazu gehört auch das bekannte Modell des *Synchronen Datenfluss-Graphen* [12], im folgenden SDF-Modell genannt.

3.2 Spezifikationssprachen

Programmiersprachen drücken meist heterogene Modelle aus, die gleichzeitig datenfluss- sowie kontrollflussorientiert sein können. Grundsätzlich unterscheidet man zwei Arten von Sprachen: *imperative* und *deklarative Sprachen*.

Imperative Programmiersprachen, wie beispielsweise C und Pascal, besitzen ein Ausführungsmodell, in dem Anweisungen in der Reihenfolge ausgeführt werden, wie sie im Programmtext erscheinen (Control-driven). Lisp und Prolog hingegen sind deklarative Sprachen. Für diese Sprachen ist charakteristisch, dass sie keine explizite Ausführungsreihenfolge spezifizieren. Das Ziel der Berechnung wird durch eine Menge von Funktionen oder logische Regeln ausgedrückt.

Imperative Programmiersprachen wie C bieten den Vorteil, dass komplexe Datenstrukturen wie Verbundtypen (Arrays, Records usw.) leicht modelliert werden können. Prozeduren und Funktionen erlauben die Bildung von Hierarchie. Im weiteren besitzen diese Sprachen zahlreiche Kontrollstrukturen wie Sequenzen von Anweisungen, Verzweigungen (z. B. IF, CASE), Schleifenkonstrukte (WHILE, FOR, REPEAT) und Unterprogrammaufrufe. Auch haben imperative Programme den Vorteil, dass sie weitverbreitet sind und durch Compilierung auf einem Mikrocomputer direkt ausgeführt werden können. Sie eignen sich folglich gut zur funktionalen Simulation von Verhalten, das später in Software oder in Hardware verfeinert werden soll. Die meisten programmierbaren Rechnerarchitekturen, darunter die meisten Mikroprozessoren, werden in imperativen Programmiersprachen programmiert. Imperative Programmiersprachen haben jedoch den Nachteil, dass sie nebenläufige Operationen nicht beschreiben können. Da Hardware inhärent parallel arbeitet, sind Programmiersprachen mit sequentiellem Programmfluss wie C (u. a. auch wegen fehlender Modellierbarkeit zeitlichen Verhaltens) nur zur Spezifikation und Simulation funktionalen Verhaltens geeignet.

Diese Probleme führten zur Einführung zahlreicher neuer Programmiersprachen, darunter Occam, ADA, Parallel C und VHDL [13]. VHDL hat sich dabei als Hardwarebeschreibungssprache durchgesetzt. Zahlreiche verschiedene Kommunikationsmechanismen wie das sogenannte *Message Passing* in CSP [14], der *Rendezvous*-Mechanismus in ADA und die Kommunikation über globalen Speicher können in VHDL modelliert werden. Während die oben genannte Sprachen sehr allgemein sind, soll hier erwähnt werden, dass zur Spezifikation von *reaktiven Echtzeitsystemen* weitere Sprachen entwickelt wurden, darunter SDL, Esterel [15], Lustre [16], Lucid [17] und Signal [18]. Diese Sprachen ermöglichen nicht nur die Synthese von Hardware ausgehend von einer Verhaltensspezifikation, sondern bieten elegante formale Verifikationsmethoden zur Überprüfung der Korrektheit. Einige der obigen Sprachen bezeichnet man als *synchron*, was der Vorstellung entspricht, dass die Antwort eines Systems auf externe Ereignisse ohne zeitliche Verzögerung erfolgt.

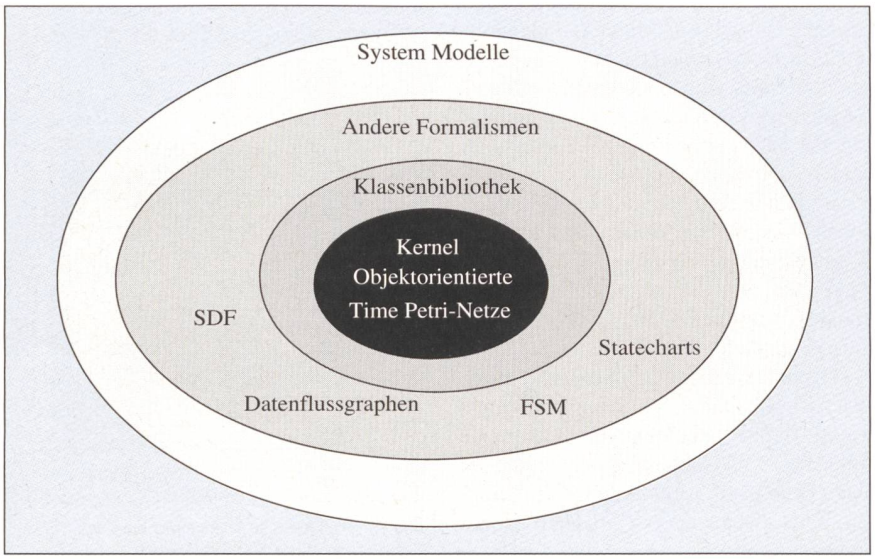

Bild 8 Modellarchitektur im Framework CodeSign

3.3 Entwurfspraktiken

In heutigen Systemen zur Spezifikation von Hardware/Software-Systemen tauchen nun verschiedene Modelle und Spezifikationssprachen auf, und zwar oft in einer gemischten Form.

Ptolemy [19] von der UC Berkeley erlaubt die Modellierung verschiedener Klassen von Datenflussgraphen, wobei die Funktionalität der Knoten in einer Programmiersprache (C++) beschrieben wird. Aus der Sicht des Berechnungsmodells sind die implementierten Knoten auf der Ebene der Eingabe hierarchisch, das heisst, ein Knoten kann wieder durch einen Datenflussgraphen auf niederer Ebene beschrieben sein.

Eine gute Übersicht über hierarchische, nebenläufige Automaten gibt der Artikel [11] von Harel. Als Entwurfssysteme zum Entwurf von hierarchischen, nebenläufigen Zustandsmaschinen existieren die Werkzeuge Statemate und Speedcharts. Beide bieten Möglichkeiten zur Codegenerierung, zum Beispiel in VHDL, Verilog und C.

3.3.1 Spezifikation in CodeSign

Schliesslich möchten wir stichwortartig die Philosophie der Spezifikation im Framework CodeSign der ETH Zürich beschreiben: In CodeSign dienen objektorientierte, zeitbehaftete *Petri-Netze* dazu, Systeme zu beschreiben. Die Modellarchitektur ist in Bild 8 beschrieben.

Der Ansatz baut auf einem formalen Berechnungsmodell (High-Level-Petri-Netze, erweitert um objektorientierte Mechanismen) auf. Durch Abstraktion und Verfeinerung können komplexe Systeme modelliert und schrittweise zu einer Implementierung verfeinert werden. Dabei wird die Verfeinerung durch Überschreiben von Komponenten durch Komponenten mit höherem Detaillierungsgrad erreicht.

Spezielle Formalismen wie Zustandsmaschinen (FSM) oder verschiedene Typen von Datenflussgraphen lassen sich dank Objektorientiertheit für den Anwender in einem dem Formalismus spezifischen Editor darstellen, werden aber im System einheitlich als Petri-Netz repräsentiert.

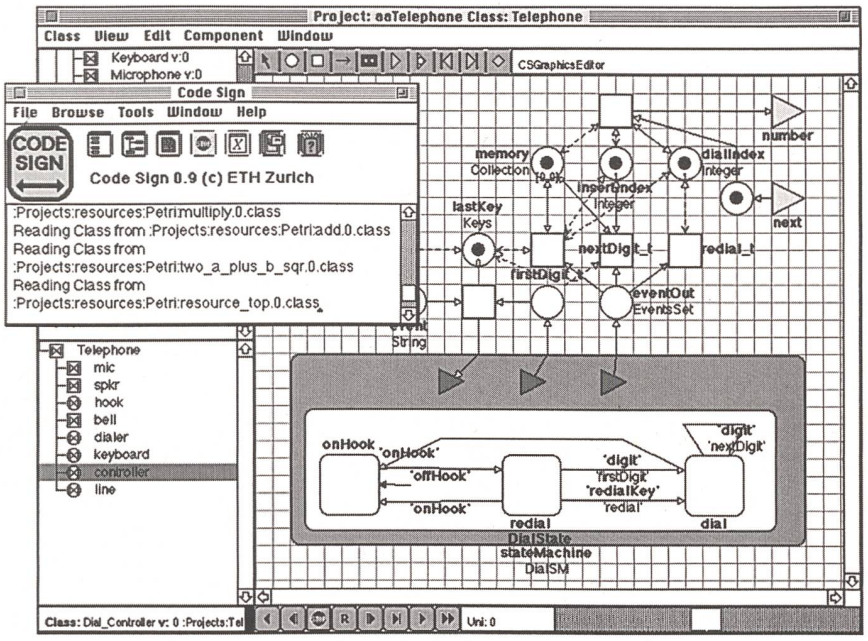

Bild 9 Bildschirmauszug des Frameworks CodeSign

Dargestellt ist die Simulation eines Telefons. Die Zustandsmaschine für die Wiederwahlfunktion des Telefons ist im Statechart-Formalismus (abgerundetes Rechteck) dargestellt; sie kommuniziert mit dem Petri-Netz, welches die Umgebung darstellt.

Beispiel 3.1

Das Bild 9 zeigt einen Bildschirmauszug des CodeSign-Editors. Gezeigt ist die Modellierung eines Telefons und von dessen Umgebung. Während das Verhalten der Umgebung mittels eines konventionellen Petri-Netzes beschrieben ist, möchte der Entwicklungsingenieur die Zustandsmaschine des Telefons in der gewohnten Darstellung eines Zustandsdiagramms (z. B. als Statechart) eingeben. Die Statechart-Umgebung ist in Bild 9 als abgerundetes Rechteck zu erkennen.

Die Erweiterung des Systems auf einen neuen Formalismus basiert in CodeSign auf Graphgrammatiken, bei denen der Benutzer eine Menge elementarer Komponenten kreiert und durch eine Menge von Zusammensetzungsregeln beschreibt, wie diese Komponenten dann zusammengesetzt werden können und interagieren.

Da alle Komponenten Instanzen einer Klasse sind, folgt, dass jeder Formalismus, der aus einer Menge von Komponenten aufgebaut ist, ebenfalls objektorientiert ist. Als spezielle Formalismen wurden bisher die Modelle SDF, FSM und Statecharts implementiert. Die objektorientierten Vorteile der Wiederverwendung von Code sowie der Abstraktion und Verfeinerung des Petri-Netz-Kerns können damit auf alle neu kreierten Formalismen angewendet werden. Ein weiterer Vorteil des Ansatzes ist die Wahrung eines formalen Modells.

Zurzeit beschäftigt sich unsere Gruppe an der ETH Zürich mit Problemen der Kopplung verschiedener Formalismen. Zum einen betrifft dies die zurzeit recht aufwendige Simulation der unterliegenden Petri-Netz-Modelle. Zum anderen arbeitet die Gruppe an effizienten Analyseverfahren, die Eigenschaften von Formalismen ausnutzen, um Systemeigenschaften effizienter beweisen zu können als unter der Annahme eines allgemeinen Petri-Netzes.

Fragen im Zusammenhang mit der Aufteilung einer Spezifikation in Software- und Hardwarekomponenten (Hardware/Software-Partitionierung) und Fragen der Optimierung von Hardware/Software-Systemen wollen wir im zweiten Teil beschreiben (folgt in Bulletin 3/97).

Literatur

[1] *K. P. Juliussen* and *E. Juliussen:* The 6th annual computer industry almanac 1993, 1993.

[2] *G. Goossens* et al.: Integration of medium-throughput signal processing algorithms on flexible instruction-set architectures. Journ. VLSI Signal Proc., 9(1995)1, pp. 49–65

[3] *R. Gupta:* Co-Synthesis of Hardware and Software for Digital Embedded Systems. PhD thesis, Stanford University, Department of Electrical Engineering, December 1992.

[4] *I.-J. Huang* and *A. Despain:* Generating instruction sets and microarchitectures from applications. Proc. IEEE/ACM Int. Conf. Comp.-Aided Design, pp. 391-396, San Jose (Calif., USA), Nov. 1994.

[5] *J. Van Praet, G. Goossens, D. Lanneer* and *H. De Man:* Instruction set definition and instruction selection for ASIPs. Proc. of the 7th International Symposium on High-Level Synthesis, pp. 11–16, Niagara-on-the-Lake (Ontario, Canada), May 1994.

[6] *C. Liem, T. May* and *P. Paulin:* Instruction-set matching and selection for DSP and ASIP code generation. Proc. Europ. Design and Test Conf., pp. 31–37, Paris (France), Feb. 1994.

[7] *D. Lanneer, M. Cornero, G. Goossens* and *H. De Man:* Data routing: a paradigm for efficient data-path synthesis and code generation. Proc. 7th ACM/IEEE Int. Symp. on High-Level Synthesis, pp. 17 to 22, Niagara-on-the-Lake (Ont., Canada), May 1994.

[8] *S. Davidson, D. Landskov, B. D. Shriver* and *P.W. Mallett:* Some experiments in local microcode compaction for horizontal machines. IEEE Trans. on Computers, C-30(1981)7, pp. 460–477.

[9] *J. R. Ellis:* Bulldog – A compiler for VLIW architectures. MIT Press, Cambridge, 1986.

[10] *J. Teich:* A Compiler for Application-Specific Processor Arrays. Shaker (Reihe Elektrotechnik). Zugl. Saarbrücken, Univ. Diss, ISBN 3-86111-701-0, Aachen, Germany, 1993.

[11] *D. Harel:* Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8, 1987.

[12] *E. A. Lee* and *D.G. Messerschmitt:* Synchronous dataflow. Proc. of the IEEE, 75(1987)9, pp. 1235 to 1245.

[13] IEEE Standard VHDL Language Reference Manual. IEEE, IEEE Std. 1076–1987, 1987.

[14] *C. A. R. Hoare:* Communicating Sequential Processes. Prentice Hall, Englewood Cliffs, NJ, 1985.

[15] *G. Berry* and *G. Gonthier:* The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming, 19(1992)2, pp. 87–152.

[16] *N. Halbwachs, P. Caspi, P. Raymond* and *D. Pilaud:* The synchronous data flow programming language Lustre. Proc. of the IEEE, 79(1989)9.

[17] *E. A. Ashcroft:* Proving assertions about parallel programs. Journ. of Computer and Systems Science, 10(1975)1, pp. 110–135.

[18] *A. Benveniste* and *P. Le Guernic:* Hybrid dynamical systems theory and the Signal language. IEEE Trans. on Automatic Control, 35(1990)5, pp. 535 to 546.

[19] *J. Buck, S. Ha, E. A. Lee* and *D.G. Messerschmitt:* Ptolemy: A framework for simulating and prototyping heterogeneous systems. International Journ. on Comp. Simulation, (1991)4, pp. 155–182.

Hardware/Software-Codesign

L'automatisation croissante dans le domaine de la conception de circuits électroniques a fait que l'on peut développer des systèmes toujours plus complexes dans un temps toujours plus court à l'aide d'outils CAD. Pour des raisons d'efficacité et de frais les systèmes considérés sont constitués en règle générale d'une combinaison de composants logiciels et matériels (programmables). Sous le slogan Hardware/Software-Codesign se cache l'objectif d'aujourd'hui de réaliser la conception de systèmes complexes entiers à l'aide d'outils CAD. La première partie de cet article donne un aperçu d'introduction des problèmes de conception dont s'occupe ce secteur de la recherche.


ONE
KIW NETWIL® 2600 S/STP 4x2xAWG 23 Kat.6 DIN E 44312-5 LS-FRNC
NETWIL® 2600
übertrifft Kategorie 6 Werte
600 MHz
sofort lieferbar
KIW
Kupferdraht-Isolierwerk AG
CH-5103 Wildegg
Tel. ++41 (0)62 887 87 02
Fax ++41 (0)62 887 87 12
http://www.kiw.ch