

Inhaltsverzeichnis

1	Einleitung	1
1.1	Entwicklung der feldprogrammierbaren Logikbausteine (PLD)	2
1.2	Motivation: Rechner im Jahr 2010	3
1.2.1	Organic Computing	3
1.2.2	Ambient Intelligence Devices	3
1.3	Zum Inhalt des Skripts	5
2	Operationsprinzipien und Klassifizierungen im Configurable Computing	7
2.1	Operationsprinzip im Configurable Computing	9
2.1.1	Grundlegendes	9
2.1.2	Logische Programmierung	12
2.1.3	Speicherelemente	15
2.2	Klassifizierung nach Speichertechnologie	18
2.3	Klassifizierung nach Programmierbarkeit	23
2.3.1	Formen der (physikalischen) Programmierung	23
2.3.2	Testbarkeit	24
2.3.3	Zusammenfassung zur Programmierbarkeit	27
2.4	Klassifizierung nach Granularität	27
3	Codierungsmodell für die Hardwareentwicklung	29
3.1	Deterministic Finite Automaton	30
3.2	Klassen einfacher Zustandsautomaten	31
3.3	Syntheseverfahren für Zustandsautomaten	32
3.4	Darstellungsformen für Zustandsautomaten	33
3.5	Zustandsminimierung	34
3.6	Zustandscodierung	35
4	Architekturen und Technologien im Configurable Computing	37
4.1	Industriestandard PAL/GAL22V10	38
4.2	Architekturen hochdichter PLDs	42
4.3	Konkrete Optimierungsziele	43
4.4	PLD-Klassen	45
4.5	HDPLDs in der Praxis	46
4.6	Ein (einfaches) PLD-Modell	46
4.6.1	Field-Programmable Gate Arrays	47
4.6.2	Complex Programmable Logic Devices	49
4.7	XC6200 als Forschungsarchitektur	52
4.7.1	Verwendete Multiplexerstrukturen	53
4.7.2	XC6200 Zellenstruktur	57
4.7.3	Lokale Routingstrukturen in der Zelle	58
4.7.4	XC6200 Routingstruktur	60

4.7.5	Aufwand zur Programmierung	62
4.7.6	Transistorschätzungen	62
4.8	Reale Architekturen: ispLSI5000 und Virtex II	63
4.8.1	Lattice ispLSI5000	65
4.8.2	Xilinx Virtex II	68
4.8.3	Elektrische Eigenschaften.....	71
4.8.4	Varianten	72
5	Hardware/Software-Interface und Software im Configurable Computing	75
5.1	Softwareprozess im Configurable Computing	75
5.2	Design von Programmen im Configurable Computing.....	77
5.2.1	Programmiermodell in PLDs	77
5.2.2	Prinzipien der HDL-Programmierung	80
5.3	Übersetzungsalgorithmen.....	81
5.3.1	Logiksynthese.....	81
5.3.2	Algorithmen zum Mapping	86
5.3.3	Mappingalgorithmen	93
6	Spezielle Themen des Configurable Computing	102
6.1	Klassifizierung der Hardware.....	102
6.2	XPP-Architektur (Fa. PACT)	105
6.2.1	Hardware-Objekte	105
6.2.2	Processing Array Elements und Processing Array Cluster.....	107
6.3	Programmiermodell im Vergleich zu CC (4.2)	110
6.3.1	Programmiermodell für hybride Bausteine (µP/PLD).....	110
6.3.2	Programmiermodell für Array-basierte Bausteine.....	111
6.4	Compilertechnologie	112
6.4.1	Übersetzungsprozess	112
6.4.2	Sourcecodeeinschränkungen.....	114
6.4.3	Pipeline Vectorization	114
6.4.4	Loop Transformationen	122
6.4.5	Temporale Partitionierung.....	125
Glossar		127
Literatur		130
Sachwortverzeichnis		132

1 Einleitung

Vor Jahren konnte man sie noch zu den exotischen Bausteinen zählen, doch nun stellen sie eine neue Form der Programmierbarkeit zur Verfügung. Feldprogrammierbare Bausteine und die damit verbundenen Designverfahren sind dabei, einen neuen Markt von programmierbaren Systemen aufzubauen.

Feldprogrammierbare Bausteine werden häufig als Ersatz für feste Hardware angesehen. Dies ist natürlich richtig, denn sowohl die Designverfahren für beide Zielbausteine – ASICs (Application Specific Integrated Circuits) und (F)PLDs (Field-Programmable Logic Devices – als auch die Funktionsweisen sind fast identisch. Diese Sichtweise stellt jedoch nur einen Teilaspekt dar, denn in der Feldprogrammierbarkeit steckt noch viel mehr Potenzial, wie man aus folgendem Beispiel bereits erahnen kann:

Die Größe der verfügbaren PLDs erlaubt es, einen eigenen Prozessor zu programmieren, der wiederum durch die Software auf einer anderen, höheren Ebene programmiert werden kann [10] [12]. Dies ist keineswegs utopisch, sondern bereits Realität [9]. Der Vorteil besteht natürlich darin, einen Prozessor mit bestmöglicher Anpassung an die Applikation zu erhalten. Eine Änderung des Prozessors selbst – und nicht nur des Programms – ist häufig möglich, und dadurch ist ein PLD viel mehr als nur ein Hardwareersatz.

Während die PLDs selbst nur den Hardwareaspekt und auch nur einen Ausschnitt daraus repräsentieren, umfasst das Gebiet des Configurable Computing wesentlich mehr, nämlich das (oder die) Operationsprinzip(ien), die Programmierschnittstelle, die Paradigmen zur Codierung von Applikationen, die Software und die Algorithmen zur Umsetzung auf die Bausteine sowie die Architektur und die Technologien der Bausteine zur Ausführung der Programme.

All dies soll in dieser Vorlesung behandelt werden, und zwar sowohl in der 'klassischen' Form (PLDs) als auch in der neuen Form, mittlerweile FPFA (Field-Programmable Functional Arrays) genannt. Um dies systematisch durchführen zu können, beginnt Kapitel 2 mit einer Darstellung der Operationsprinzipien im Configurable Computing, dies auch im Vergleich zum bekannten Von-Neumann-Paradigma, und fügt verschiedene Klassifizierungen bei. Kapitel 3 befasst sich mit den Aspekten der Bit-basierten Bausteine (PLD): Architekturklassen, Architekturen, Speichertechnologien und Beispiele sind Inhalt dieses Abschnitts.

Kapitel 4 wendet sich der Softwareseite im Bit-basierten Configurable Computing zu. Zwei wichtige Fragen sind zu klären: Wie sieht das Programmierparadigma aus, und welche wesentlichen Vorgänge spielen sich bei der Softwareentwicklung (insbesondere den Übersetzungsvorgängen ab?

Kapitel 5 befasst sich mit den Word-basierten Bausteinen im Configurable Computing. Hierzu zählen einige Forschungsthemen wie die Frage nach Betriebssystemen für CC-Devices sowie Architekturen, die in der Forschung diskutiert werden.

1.1 Entwicklung der feldprogrammierbaren Logikbausteine (PLD)

Wie kam es eigentlich zu der Entwicklung dieser Bausteine? In den 30er Jahren wurden erste Ideen zu Rechenmaschinen – oder wie man etwas später auch sagte: Elektronengehirnen – entwickelt. Außerhalb der zunächst wirklich eng umgrenzten Fachkreise war daran wohl eines unbegreiflich: Wie kann es eine Maschine geben, die nicht für den endgültigen Zweck produziert wird, sondern nur «halbfertig» ist? Seitdem hat man sich natürlich an Software gewöhnt, sie gibt der Rechenmaschine die für den Moment gültige Funktion und kann jederzeit gewechselt werden. In der Frühzeit der Rechner war diese Software für den Laien unbegreiflich.

Hard- und Software haben in der Welt ihren wohldefinierten Platz gefunden. Die fixierte Hardware, die zunächst (zum großen Teil in der Fabrik) entworfen wird (Hardware First), bleibt unverändert und bietet ein Interface für die Software. Software wird vom Entwickler entworfen und ist flexibel.

Ende der 60er Jahre kamen die ersten Ideen auf, auch die Hardware flexibler zu gestalten. So überraschend es klingen mag, auch hier gibt es wieder einen fabrikatorischen, fixierten Teil (der «echten» Hardware) und ein Stück Beschreibung der endgültigen Funktionalität (die Software, hier meist Konfiguration genannt). Man muss sich schon fragen, wo der Unterschied denn eigentlich noch zu suchen ist.

1977, also nur 6 Jahre nach der Markteinführung des ersten Mikroprozessors Intel 4004, erschienen die ersten kommerziellen Bausteine auf dem Markt, Hersteller war Monolithic Memories Inc. (MMI). Die Entwickler dieser Bausteine, PAL (Programmable Array Logic) genannt, haben mit ihrer Wahl den inneren Aufbau einer Vielzahl von heute verfügbaren ICs beeinflusst. 1985 wurden dann die ersten feldprogrammierbaren ICs, die in der Lage waren, ein «System» aufzunehmen, von Xilinx angeboten.

Die PLDs erleben zurzeit einen wahren Boom, Geschwindigkeit und nutzbare Größe betreffend. Für die Entwicklung von digitalen Systemen bieten sie gegenüber ASICs den Vorteil, programmierbar zu sein, während sie sich von den ebenfalls programmierbaren Mikroprozessoren dadurch unterscheiden, dass ihr Ausführungsprinzip nicht mehr sequenziell ist, sondern dass das Programm in der Struktur liegt. Programmierte PLDs funktionieren wie hergestellte Hardware.

Andererseits stehen wir derzeit an der Schwelle zur Nutzung der dynamischen Reprogrammierbarkeit. Bestimmte Speichertechnologien ermöglichen einen Wechsel des Programms, auch teilweise, zur Laufzeit. Dies bedeutet, dass zukünftig mehr Programm gespeichert wird als aktuell ausführbar ist – ein Zustand, der bei prozessorbasierten Rechner gängige Praxis ist, bei PLDs jedoch neu sein wird.

Als Konsequenz entwickelt sich gerade eine neue Bausteinklasse, derzeit mit FPFA (Field-Programmable Functional Array) bezeichnet. Diese Bausteinklasse erhält

derzeit in der Forschung höchstes Interesse, da hier das zukünftig größte Potenzial vermutet wird. Erste Bausteine sind seit Ende 2000 auf dem Markt.

1.2 Motivation: Rechner im Jahr 2010

Benötigt man eigentlich eine Motivation zur Beschäftigung mit einem neuen Rechnerparadigma? Die Antwort ist zweiteilig: In der Forschung ist das Neue Motivation genug, in der Anwendung jedoch muss es schon gute Gründe hierfür geben. Glücklicherweise gibt es diese guten Gründe auch.

Hierzu lohnt sich ein Blick auf das Jahr 2010 und die Vorhersagen für Mainstream-Applikationen [26, 27].

1.2.1 Organic Computing

Die Bezeichnung *Organic Computing* wird leider für zwei verschiedene Architekturen bzw. Technologien verwendet: Hardware-Technologen verwenden dies, um die Nutzung von organischen Molekülen für Rechnungen, Speicherung von Daten etc. zu beschreiben, in der Informatik hat sich dies als Beschreibung für eine zukünftige Architektur mit folgenden Eigenschaften etabliert:

- Selbst-konfigurierend
- Selbst-heilend
- Selbst-optimierend
- Selbst-schützend

Von diesen Eigenschaften ist die der Selbst-Konfiguration die zentrale, alle anderen basieren darauf. Das *Organic Computing* zielt natürlich insbesondere auf vernetzte Systeme, wobei die einzelnen Netzknoten allerdings einen Großteil der Eigenschaften tragen. Mit anderen Worten: Organic Computing basiert auf rekonfigurierender Hardware.

1.2.2 Ambient Intelligence Devices

Unabhängig von dem eben skizzierten Ansatz zu zukünftigen verteilten Systemen und ihren Architekturen existieren Voraussagen zu dem zukünftigen Mainstream. Hier werden die Nachfolger der jetzigen Personal Digital Assistents (PDAs) prognostiziert, meist mit Ambient Intelligent Devices (AmI) bezeichnet.

Für die AmI-Devices, die sich im Übrigen nahtlos in die Organic-Computing-Welt einfügen, existieren einige Detailschätzungen:

- Ein Batteriebetrieb auf Akkumulatorbasis wird allein aus Kosten- und Mobilitätsgründen angenommen, wobei allerdings die Entwicklung der Kapazitäten sehr konservativ angenommen werden ($< 10\%$ /Jahr).

- Typische Applikationen sind: Kommunikation jeglicher Art, hierbei Verschlüsselung, Video-on-demand bzw. live, Spiele, etc.
- Als maximale Rechenleistung wird ca. 100 Giga-Operationen pro Sekunde (100 GOPS) prognostiziert. Spitzenreiter sind dabei Spiele und Video.
- Aus der Entwicklung der Batteriekapazitäten und den Anforderungen für die Rechenleistung ergibt sich die wesentliche Forderung: 2 W Leistungsaufnahme, d.h. eine **Energieeffizienz von 10 – 100 MOPS/mW** [26].

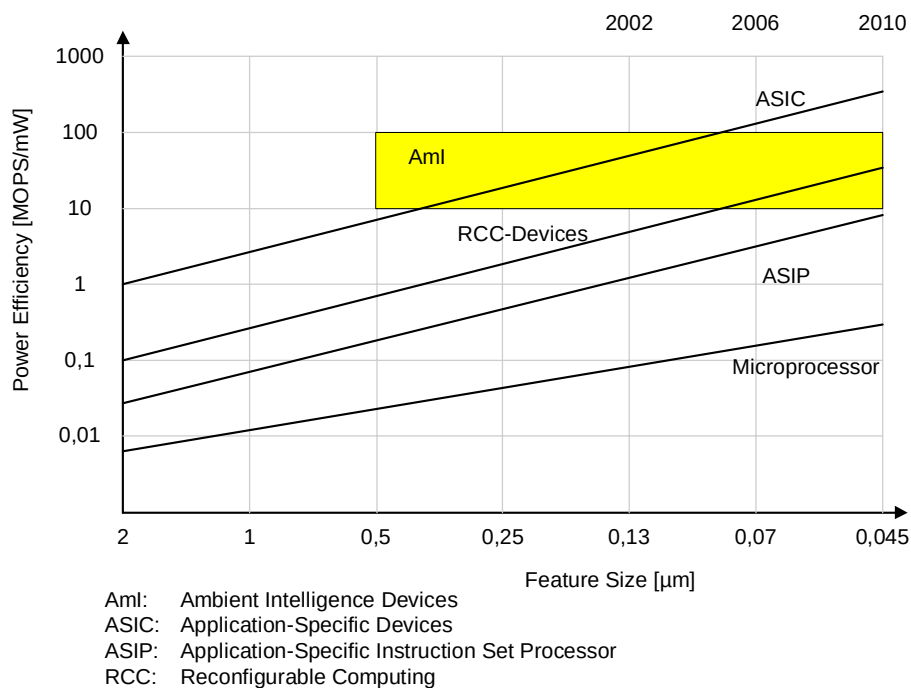


Bild 1.1 Powereffizienzbereiche [26]

Diese Zahlen sind recht erschreckend, denn programmierbare Systeme liefern andere Werte. Bild 1.1 zeigt die bisherige und prognostizierte Entwicklung für (General-Purpose-) Mikroprozessoren, applikationsspezifische Prozessoren (ASIP), Reconfigurable Computing Devices und ASICs – letztere nur zum Vergleich, denn diese sind nicht programmierbar, sondern fixiert. Tabelle 1.1 gibt einen weiteren Aufschluss zu dem Thema Energieeffizienz [28, 29].

Die darin aufgeführten Mikroprozessoren und -controller sind willkürlich ausgewählt, um die Spannweite der momentanen Effizienzwerte zu demonstrieren. Pentium Pro und Celeron sind bekannt. SH7709 stellt einen 32-Bit-Mikrocontroller von Hitachi (jetzt: Renesas) dar, DEC Alpha 21364 dürfte auch noch bekannt sein.

Der Lutonium Mikroprozessor [28] ist eine Entwicklung vom Caltech, USA. Es handelt sich hierbei um den 8051-Mikroprozessor in einer asynchronen Variante, also ohne Takt. Die beiden Werte sind durch verschiedene Betriebsspannungen entstanden, einmal für 1,8 V. der besonders effiziente Wert für 0,5 V, eine Subthreshold-Spannung, bei der der Mikroprozessor eigentlich nicht mehr arbeiten dürfte.

Mikro-Prozessor	Prozess-technologie	Frequenz	Verlust-leistung	MIPS/mW	Bemerkung
Pentium Pro	0,6 μm	150 MHz	23 W	0,0065	1995
Celeron	0,25 μm	400 MHz	23,4 W	0,017	1999
Athlon (K7)	0,25 μm	700 MHz	45 W	0,019	1999
DEC21364	0,18 μm	1 GHz	100 W	0,047	2000
SH7708	0,5 μm	60 MHz	0,6 W	0,1	1994
Lutonium	0,18 μm	200 MHz	0,1 W	1,8	Asynchron, 1,8 V, 2004
Lutonium	0,18 μm	4 MHz	0,17 mW	23	Asynchron, 0,5 V, 2004
XPP (RCC)	0,18 μm	200 MHz	10 W	20	Peak-Schätzwerte, 2003

Tabelle 1.1 Leistungsdaten einiger Mikroprozessoren [28–30]

Die Tabelle für Mikroprozessoren ist ergänzt durch ein FPFA-Device der Firma PACT, XPP (eXtreme Processing Platform). Die hier angegebenen Werte sind Schätzungen für die Peak-Performance (200 GOPS), so dass nur bedingt verglichen werden kann. Man erkennt trotzdem aus der Tabelle, dass wir mit Mikroprozessoren weit von dem 10 MIPS/mW-Punkt entfernt sind. Genau hier setzen die Hoffnungen zum Reconfigurable Computing an: Es soll damit eine bislang unerreichbare Effizienz erreicht werden.

1.3 Zum Inhalt des Skripts

Das Skript (es ist kein Buch, nur ein Vorlesungsskript!) baut auf sehr vielen Quellen auf, erkenntlich am Literaturverzeichnis. Eine sehr schöne, einheitliche Darstellung eines wesentlichen Teils des Themas (PLDs) liegt mit [22] vor; leider ist dieses Werk bereits 1992 erschienen und muss damit als veraltet bezeichnet werden – obwohl viele der dort dargestellten Prinzipien heute noch gelten.

Die Vielfalt der Quellen macht eine Verfolgung in der Literatur nicht gerade einfach. Als eine der (neben [22]) wichtigen Quellen können [12] und [24] genannt werden, obwohl beide Bücher eher dem praktischen Aspekt (Einsatz in der Praxis oder der Lehre) und nicht einer umfassenden Darstellung auch der Forschung

dienen. [25] ist als Taschenbuch gut als Überblick bzw. Repetitorium geeignet, aber natürlich nicht als Lehrbuch.

Diese Vorlesung versucht, einen Bogen zwischen Praxis und Forschung (= zukünftige Praxis?) zu spannen. Dies ist deshalb so wichtig, weil sich im Gebiet des Configurable Computing erhebliches Entwicklungspotenzial – gerade für Architekturen – befindet, wie noch darzustellen ist.

Aus diesem Grund finden sich in diesem Skript am Markt befindliche Architekturen der PLDs ebenso wieder wie Metriken zur Klassifizierung der Bausteine, Forschungsarbeiten zu Betriebssystemansätzen und ein neues Architekturparadigma: Reconfigurable Computing.

2 Operationsprinzipien und Klassifizierungen im Configurable Computing

Man kann sich allgemein die Frage stellen, wie programmierbare Systeme eigentlich arbeiten. Diese Systeme sind im Kern halbfertig, zur tatsächlichen Funktion benötigen sie zusätzliche Informationen, die beim Ablauf interpretiert werden. Programmierbare Systeme interpretieren also zur Laufzeit das Programm (unabhängig davon, welcher Art diese Informationen sind).

Wenn diese Annahme stimmt, dann muss man sich fragen, worin denn eigentlich die Unterschiede bestehen. Ein Mikroprozessor nimmt eine Programminformation (genannt Instruktion oder Maschinenbefehl) auf, interpretiert sie und arbeitet nach den dort codierten Angaben. Ein programmierbarer Logikbaustein (PLD) nimmt ebenfalls eine Programminformation (genannt Konfiguration) auf, interpretiert sie und arbeitet gemäß der Codierung.

Der Unterschied liegt in dem zeitlichen Ablauf, besser in dem "danach". Tabelle 2.1, ursprünglich in [4] publiziert und weiterentwickelt, gibt Auskunft darüber, wie sich die Zeiten bei den beiden programmierbaren Architekturen verhalten.

	PLD, one-time progr./Flash	PLD, SRAM	Reconf. Computing	µP
<i>Bindungszeit</i>	Fuseprogrammierung	Laden Konfig.	Laden Konfig.	Takt
<i>Bindungsdauer</i>	unendlich/ Reprogrammierung	Reset	user- definiert	Takt
<i>Programmierzeit</i>	Sekunden	µs-ms (s)	Takt	Takt
<i>Typische Anzahl Instruktionen</i>	1	1	> 1	>> 1

Tabelle 2.1 Charakteristische Zeiten für digitale Systeme

Diese Tabelle vergleicht zunächst drei Formen aus dem Bereich des Configurable Computing, ein PLD, das einmal programmierbar ist bzw. auf einer Flash-EEPROM-Technologie basiert, ein SRAM-basiertes PLD und ein (virtuelles) Device aus dem Bereich des Reconfigurable Computing mit einem Mikroprozessor auf die charakteristischen Zeiten.

Für den Mikroprozessor verlaufen alle wesentlichen Vorgänge im Rahmen von (wenigen) Takten: Die aktuelle Arbeitsweise wird in der Fetchphase festgelegt, die Einstellung auf die Arbeitsweise (Decode/Load) dauert ebenfalls ca. einen Takt,

und abschließend wird ausgeführt (Execute). Auch die letzte Phase nimmt wenige Takte (im Idealfall einen) in Anspruch. Mit anderen Worten: Der Mikroprozessor arbeitet etwa in einem Taktschema, die Idealbild einer RISC-Architektur). Was aber wesentlich wichtiger ist: Der Mikroprozessor arbeitet nach einer Instruktion die nächste ab: Der Instruktionswechsel ist Maschinen-definiert!

Beim den PLDs in Tabelle 2.1 fällt zunächst einmal auf, dass die Speichertechnologie viel Einfluss auf die Anwendungsweise haben kann: One-Time Programmable und Flash-EEPROM-basierte Bausteine werden programmiert und arbeiten dann, ohne einen weiteren Support bekommen zu müssen. Diese Fuseprogrammierung ist ein Vorgang, der außerhalb des eigentlichen Betriebs durchlaufen wird. Aus diesem Grund spielt auch die Programmierzeit im Bereich von Sekunden nur eine untergeordnete Rolle.

Bei SRAM-basierten PLDs muss aufgrund der Flüchtigkeit der Information bei Ausfall der Betriebsspannung die Konfiguration bei jedem Anschalten geladen werden – Vor- und Nachteil zugleich. Der Vorteil ist derjenige, dass man zumindest bei jedem Anschalten auswählen kann, welche Funktion ausgeführt werden soll, der Nachteil besteht darin, dass man es auch muss (Boot-Vorgang).

Die Programmierzeit liegt bei Forschungsarchitekturen im Rahmen von Mikrosekunden, bei produzierten Architekturen eher im Rahmen von Sekunden. Dies ist im Wesentlichen eine Frage des Interface und dessen Parallelität. In rekonfigurierbaren Architekturen wird man Wert darauf legen, dass die Rekonfiguration, ggf. partiell, nur wenige Betriebstakte in Anspruch nehmen wird.

Ein wichtiger Aspekt ist auch, dass die Anzahl der verschiedenen Instruktionen (verschieden in Typ, Operanden oder der Stelle im Programm), die ein Algorithmus benötigt und die in einer sequenziellen Form bearbeitet werden müssen, gleich 1 ist für die PLDs und $\gg 1$ für Mikroprozessoren. Mit anderen Worten: In PLDs wird *eine* Instruktion spezifisch für den Algorithmus konfiguriert, und zwar in der Fläche, bei Mikroprozessoren werden unterschiedliche Instruktionen in einer zeitlichen Sequenz benutzt.

Bei rekonfigurierbaren Architekturen wird eine Anzahl von >1 (typischerweise 8–16) benutzt, um den Algorithmus darzustellen. Für diese Architekturen wird sowohl die Programmierung in der Fläche als auch die in der Zeit genutzt. Bei den Zeiten drückt sich dies so aus, dass die Bindungsdauer hier nicht Maschinen- sondern User- oder Applikations-definiert ist.

Mithilfe der Tabelle 2.1 können also sowohl das Operationsprinzip als auch Klassifizierungsmerkmale im Bereich des Configurable Computing hergeleitet werden. Die genaue Beschreibung ist Aufgabe der nachfolgenden Abschnitte dieses Kapitels.

2.1 Operationsprinzip im Configurable Computing

Wie schon im einleitenden Abschnitt erörtert wurde, besteht das Prinzip einer programmierbaren Architektur darin, eine Programminformation aufzunehmen, zu interpretieren und dementsprechend zu arbeiten. Da gemäß Tabelle 2.1 außer im Bereich des Reconfigurable Computing nur eine Instruktion pro Algorithmus zur Verfügung steht, muss diese Instruktion zusammensetzbar sein.

Wie kann diese Zusammensetzung erfolgen? Hierzu müssen einige grundlegende Bemerkungen gemacht werden, die sich auf die Darstellbarkeit beliebiger binärer Funktionen durch Grundelemente der Schaltalgebra beziehen. Sind diese Grundelemente miteinander durch programmierbare Leitungen verbindbar und ggf. auch in ihrer Funktionalität konfigurierbar, erhält man eine beliebige Funktion – in der Fläche.

Diese Programmierung in der Fläche führt zu einer Rechnung 'in der Fläche', als *Computing in Space* (im Gegensatz zum *Computing in Time* der Von-Neumann-basierten Architekturen) bezeichnet.

2.1.1 Grundlegendes

Die Boolesche Algebra (oder auch Schaltalgebra genannt) befasst sich mit Verknüpfungen und Rechenregeln für Variable, die ausschließlich 2 Werte annehmen können: 0 und 1. Die zugehörige mathematische Disziplin, die Aussagenlogik, besitzt in diesem Fall eine technische Realisierung. Vergleicht man dies einmal mit der Welt der Mikroprozessoren, so ist festzustellen, dass die Aussagenlogik mit ihrer Darstellungsform die Assemblersprache der Digitaltechnik darstellt.

In der Theorie reicht es aus, einen beliebigen Satz von logischen Operationen zu finden, mit dessen Hilfe jeder funktionale Zusammenhang zwischen dem Zielsignal und den Eingangssignalen formuliert werden kann. Diese Menge von logischen Operationen (tatsächlich genügt eine hierfür: ein NAND mit 2 Eingängen) wird als logische Basis bezeichnet.

In der Technik werden weitere Forderungen an das Basissystem gestellt. Die logischen Operationen werden schließlich auf Transistorschaltungen zurückgeführt, und Transistoren haben Laufzeiten, setzen elektrische Leistung in Wärme um (Verlustleistung) und kosten Platz. Allgemein fordert man also ein Basissystem mit minimalen, möglichst immer gleichen Laufzeiten, einer hohen Verknüpfungsdichte (zur Energieeinsparung) und wenig Platzverschwendung (im statistischen Mittel).

Dieses System ist in der disjunktiven Normalform (DNF, englisch: Sum-of-Products, SOP) vorhanden. Hierfür werden 3 Operationen benötigt (siehe auch Bild 2.1): Negation (NOT), Konjunktion (AND) und Disjunktion (OR). Die entsprechenden Verknüpfungszeichen der Aussagenlogik ($\neg$, $\wedge$, $\vee$) werden in textuellen Entwicklungssystemen durch / für NOT, * für AND und + für OR ersetzt, wobei auch einige andere genutzt werden.

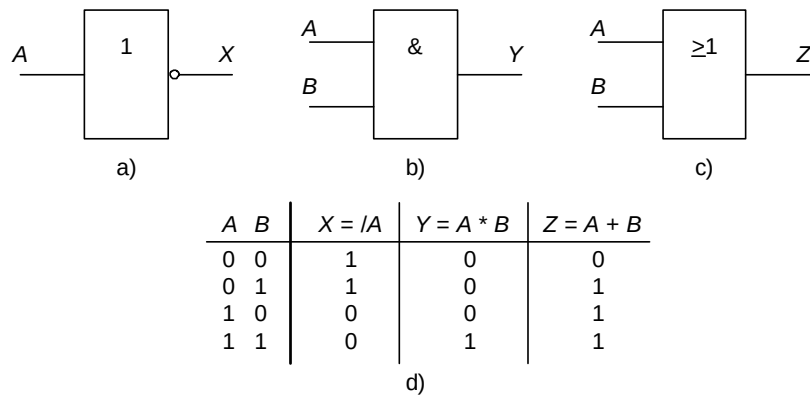


Bild 2.1 Basisoperationen für disjunktive Normalform
a) Schaltzeichen NOT b) Schaltzeichen AND c) Schaltzeichen OR
d) Wahrheitstabelle

Die Wahrheitstabelle in Bild 2.1d dient hier zur Definition der Operationen, die sich im Übrigen leicht auf mehrere Eingangsvariablen ausdehnen lassen (was noch dringend benötigt wird). Eigentlich werden diese Tabellen jedoch zur Definition der anwendungsspezifischen Funktion wie in dem Beispiel für die XOR- (Exclusive-OR-) Funktion genutzt:

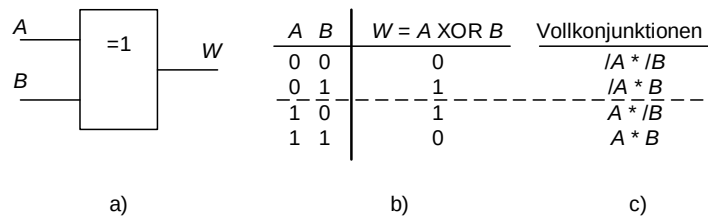


Bild 2.2 a) Schaltzeichen Exclusive-OR-Funktion mit 2 Eingängen
b) Wahrheitstabelle c) zugehörige Vollkonjunktionen

Die erste Aufgabe für den Entwickler (oder für die Entwicklungssoftware) besteht nun darin, diese neue Funktion durch die Basisoperationen auszudrücken. Hierfür gibt es ein Standardverfahren, für das in Bild 2.2 c) bereits die sogenannten Vollkonjunktionen angegeben sind. Diese bestehen jeweils aus der AND-Verknüpfung aller Variablen oder der negierten Größen. Jede Vollkonjunktion erhält den Wert 1, wenn die Eingangswerte der zugehörigen Zeile eingesetzt werden, ansonsten jedoch 0.

Wählt man jetzt nur die Vollkonjunktionen aus, bei denen in der XOR-Spalte der Wahrheitstabelle auch eine 1 steht, und verknüpft diese mit OR, so erhält man

$$W = \neg A * B + A * \neg B \quad (\text{Gl. 2.1})$$

Dieses Verfahren liefert immer ein Ergebnis, indem die Vollkonjunktionen aufgestellt und alle mit Ergebnis 1 in der Wahrheitstabelle mit OR verknüpft werden. Die Lösung ist im Allgemeinen nicht optimal (im obigen Beispiel des XOR allerdings schon), stellt aber eine Lösung dar, die in der Praxis noch minimiert werden kann. Mit Minimierung wird das systematische Suchen nach einer Lösung mit gleichen Eigenschaften, aber weniger Ressourcenverbrauch bezeichnet.

Bild 2.3 zeigt die Schaltungsrealisierung des XOR-Gatters. Jede Signaländerung durchläuft 3 Gatterebenen: Pufferung/Invertierung, AND- und OR-Ebene. Dies gilt für jede beliebige Funktion in DNF-Darstellung, und hierin ist der große Vorteil dieses Basissystems zu sehen: garantierte maximale Laufzeiten, unabhängig von der Komplexität der Funktion.

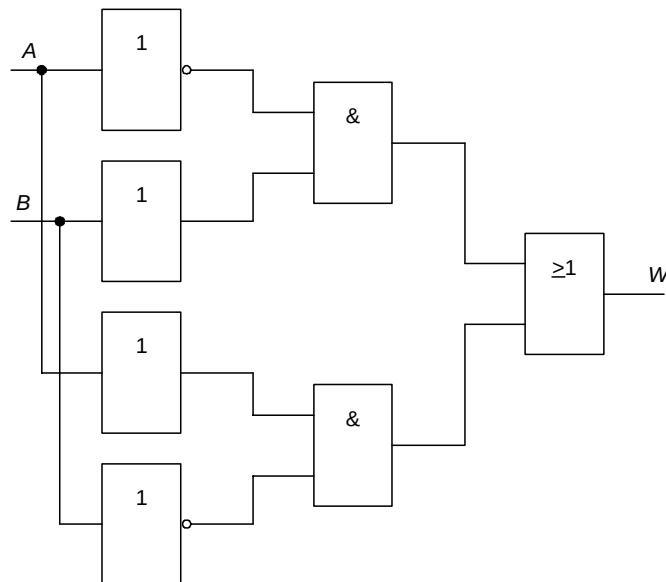


Bild 2.3 Schaltungstechnische Realisierung der XOR-Funktion durch DNF

Leider hat dies auch einen Preis, denn für eine Dreistufigkeit in der Lösung muss es auch AND- und OR-Gatter geben, die eine große Anzahl von Eingängen enthalten (zur Darstellungsweise solcher Gatter siehe auch Bild 2.4). Die meisten Funktionen benötigen als DNF ca. 3–5 Eingänge im OR-Gatter (man spricht hier auch von Termen); nach oben hin gibt es aber keine Beschränkung, es könnten also im Einzelfall auch 16 oder 20 Terme sein. Wie soll ein PLD, das die DNF implementiert, nun ausgerichtet sein?

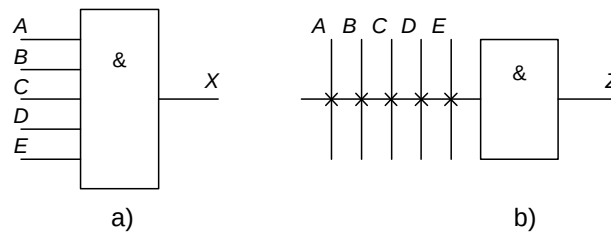


Bild 2.4 Darstellung AND-Gatter mit 5 Eingängen
a) normgerechte Darstellung b) Darstellung in PLDs (die Verbindungen an den Schnittstellen können fest oder programmierbar sein)

Nimmt man einmal an, jedes OR-Gatter hätte 20 Eingänge, dann würden im Durchschnitt 20% genutzt werden. Es gäbe keinerlei Probleme, auch komplexe Funktionen ohne Geschwindigkeitsverlust zu implementieren. Natürlich kann eine komplexe Funktion auch auf mehrere kleinere DNFs verteilt werden, aber dann wächst die Laufzeit. Und genau vor diesem Problem stehen die IC-Designer: Geschwindigkeit oder Nutzungsdichte, was ist wichtiger? Die Antwort kann allgemein nicht gegeben werden, denn dies hängt von der Applikation ab. Die am Markt erhältlichen PLDs bieten verschiedene Optimierungen, so dass für den jeweiligen Einsatzfall entschieden werden kann, welche Form tatsächlich gewählt wird.

Aussagen über die im Allgemeinen beste Lösung kann man nur treffen, wenn ein zutreffendes Modell für die verschiedenen Varianten vorliegt und über möglichst viele Applikationen gemittelt wird. Diese Aussagen sind schwierig zu erhalten, im weiteren Verlauf dieses Skripts wird noch ein Ansatz diskutiert werden.

2.1.2 Logische Programmierung

Jede Basisarchitektur erfordert die Möglichkeit zur Programmspeicherung, in der die Struktur auf ihre endgültige Funktionalität festgelegt wird. Das Prinzip sieht viele mögliche Verbindungen vor, die während der Programmierung belassen oder gelöscht werden.

Die Programmierung wird durch die Architektur und die zugrundeliegende Technologie ($\rightarrow$ 2.2) bestimmt. Bild 2.5 zeigt einen Ausschnitt aus der PAL-Struktur mit zwei AND-Gattern. Diese Darstellung verwendet die Notation aus Bild 2.4 für Gatter mit vielen Eingängen.

Der hier dargestellte Ausschnitt stammt aus einer klassischen (aber kleinen) Architektur, einem PAL (Programmable Array Logic) oder GAL (Generic Array Logic). Die Eingänge auf der linken Seite werden normal und invertiert im Baustein auf die senkrechten Verteilerleitungen gegeben. An jedem Kreuzungspunkt einer solchen Verteilerleitung mit der (scheinbar einzigen) Eingangsleitung jedes AND-Gatters besteht nun eine programmierbare Verbindung.

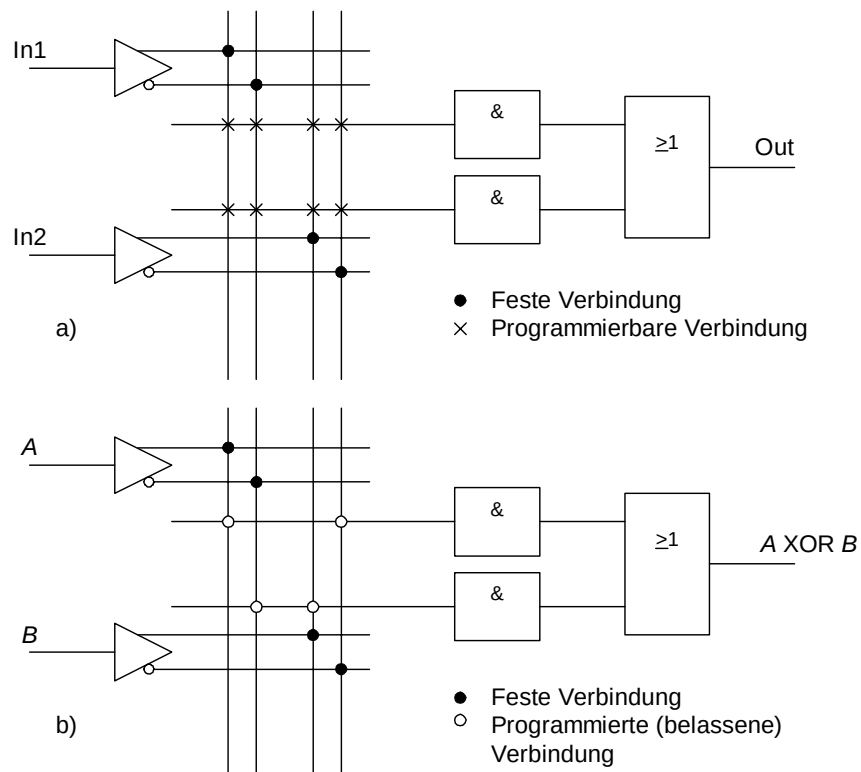


Bild 2.5 Ausschnitt aus PAL-Architektur
a) unprogrammiert b) programmiert für A XOR B

In der Realität besitzt jedoch das AND-Gatter nicht 1, sondern exakt so viele Eingangsleitungen, wie Verteilerleitungen im Baustein existieren (in Bild 2.5 a) also 4, in Bild 2.4 b) 5 Eingänge). Für den sogenannten Industriestandard, den GAL/PAL 22V10, hieße dies, dass man 44 Eingänge am AND zu zeichnen hätte – womit die Übersichtlichkeit verloren gegangen wäre. Aus diesem Grund werden Gatter mit sehr vielen Eingängen in der PLD-Welt wie in Bild 2.5 dargestellt.

In Bild 2.5 b) ist das XOR-Beispiel programmiert. Je 2 der Eingänge sind pro AND-Gatter belassen worden, die anderen wurden unterbrochen. Offene Eingänge am AND-Gatter führen in derartigen PLDs den Wert 1, gehen also nicht in die Schaltlogik ein. Im Gegensatz zu diesen Eingängen besitzen diejenigen am OR-Gatter jedoch keine programmierbare Eigenschaft, hier wurde fest verdrahtet. Genau dies ist die Philosophie der PAL-Architektur: Programmierbare AND-Eingänge, aber feste OR-Gatter. Dies ist eine spezielle Form der Realisierung einer DNF.

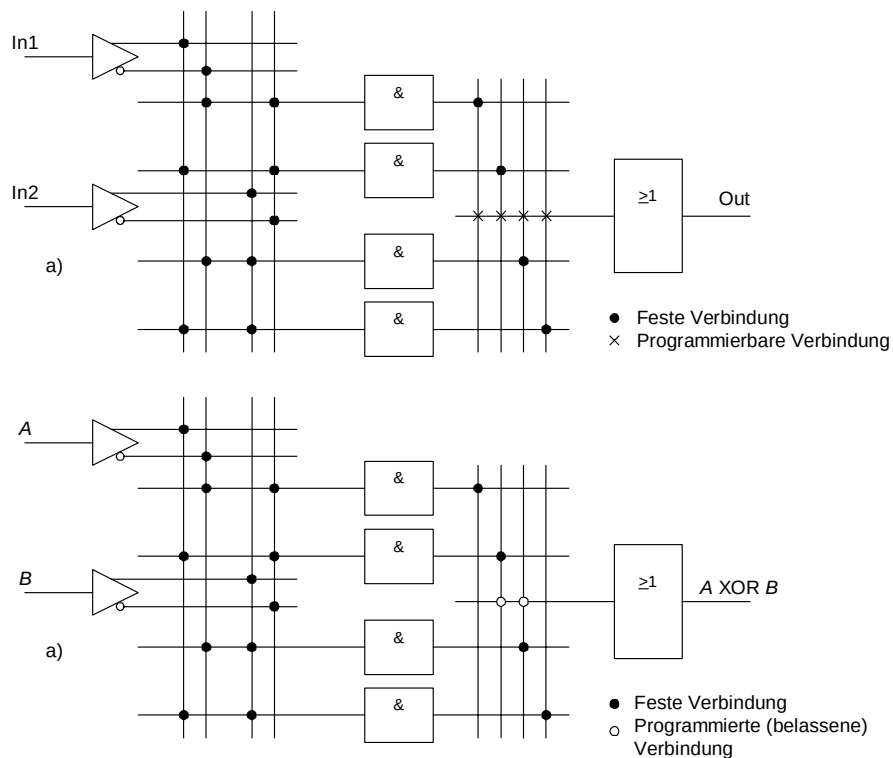


Bild 2.6 Look-Up-Table für 2 Eingänge und 1 Ausgang
 a) prinzipieller Aufbau b) Programmierung für XOR-Funktion

Eine andere Strategie verfolgt man bei der Look-Up-Table (LUT). Hierunter wird eine Tabellenstruktur verstanden, die an Datenausgängen die Tabelleneinträge anzeigt, die durch die Adresseingänge adressiert werden. Dies stellt die «klassische» ROM-Struktur dar, und Bild 2.6 gibt einen genaueren Einblick.

Für die Darstellungsweise des OR-Gatters mit vielen (hier: 4) Eingängen gilt wieder Entsprechendes wie beim AND-Gatter, nur dass nun offene Eingänge eine 0 führen müssen. Worin besteht denn nun der praktische Unterschied zwischen PAL-Struktur und LUT? Bei der PAL-Struktur kann man mit relativ wenigen Stellen zur Programmierung viele, aber keineswegs alle logischen Funktionen darstellen. Wenige Stellen zur Programmierung bedeutet, dass hier mit Siliziumressourcen vorsichtig umgegangen wird, denn jeder Programmierungstransistor (Anzahl hier: proportional zu k , wenn k Eingänge vorhanden sind) hat einige weitere an «Verwaltung» um sich herum. Die Einschränkung bei den möglichen Funktionen bedeutet jedoch, dass man im Extremfall auch die Überraschung erleben kann, dass eine Funktion plötzlich nicht mehr passt!

Eine Realisierungsweise für eine Look-Up-Table zeigt Bild 2.7 bei Verwendung eines 1-aus- 2^k -Multiplexers. Ein derartiger Multiplexer besitzt k Steuereingänge und 2^k Dateneingänge. Die Steuereingänge wählen mit dem Bitvektor aus, welche Datenleitung am Ausgang erscheint. Dies kann so genutzt, dass an den Dateneingängen Speicherzellen liegen, die den jeweiligen Wert aufnehmen, während die Steuereingänge wie "Adressen" wirken. Auf diese Weise lässt sich jede beliebige Funktion mit k Eingängen realisieren (z.B. XOR, Bild 2.7b).

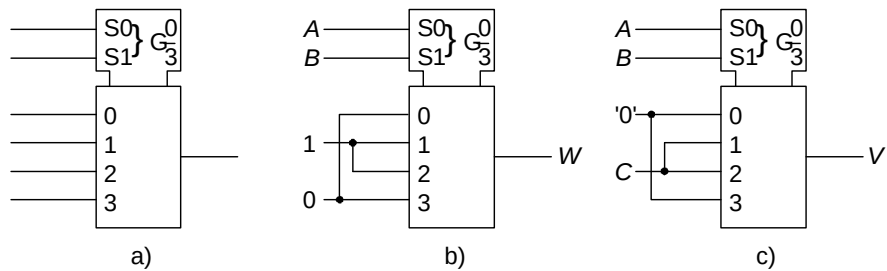


Bild 2.7 a) 1 aus 4 Multiplexer b) $W = A \text{ XOR } B$ c) $V = C * (A \text{ XOR } B)$

Bild 2.7 c) zeigt eine weitergehende Nutzung des Multiplexers, indem die Dateneingänge ebenfalls mit variablen Signalen verbunden werden. Auf diese Weise können keineswegs alle, aber einige Funktionen mit mehr als 2 Eingangsvariablen auf die Hardware abgebildet werden. Multiplexer mit dieser Möglichkeit bieten im Durchschnitt eine etwas erhöhte Flächennutzung gegenüber der LUT-Variante, allerdings muss auch die Belegung der Dateneingänge wieder programmiert werden.

In der Praxis erfolgt die Abbildung, man spricht auch von Mapping, durch Software, nicht durch den Entwickler. Dementsprechend ist es mehr eine Philosophie des Herstellers, welche Form der Logikrealisierung konkret gewählt wird, für den Entwickler bleiben die Unterschiede meist verborgen.

Als Fazit kann zusammengefasst werden, dass sich die LUT-Struktur mit hoher Flexibilität gut eignet für kleine Blöcke, die PAL-Struktur mit Ressourcenschonung gut für größere. Der zugehörige Speicher wird als *Konfigurationsspeicher* bezeichnet, er entspricht dem Codespeicher bei Von-Neumann-Architekturen.

2.1.3 Speicherelemente

Die bisherigen Darstellungen der Programmierung von logischen Schaltungen berücksichtigen nur eine Klasse von Funktionseinheiten, die Schaltnetze. Sie sind durch folgende Definition charakterisiert:

Definition 2.1

Ein **Schaltnetz** ist eine Funktionseinheit zum Verarbeiten von Schaltvariablen, deren Wert am Ausgang zu irgendeinem Zeitpunkt nur vom Wert am Eingang zu diesem Zeitpunkt abhängt.

Diese Definition bedeutet, dass hier keine Zeit notwendig ist – abgesehen von der Signallaufzeit im Baustein, von der ganz bewusst abgesehen wird –, um eine Eingangsänderung am Ausgang bemerkbar zu machen. Dies entspricht den Funktionseinheiten, die in den programmierbaren Strukturen aus Bild 2.5 und 2.6 gebildet werden können.

Viele technische Systeme benötigen allerdings einen Zusatz, in dem sie den bisherigen Zustand speichern können, weil die Ausgänge vom bisherigen Weg abhängig sind. Ein einfaches Beispiel sind Ampelschaltungen, bei denen die verbleibende Zeit einer Phase natürlich von der bislang verstrichenen abhängt. Die (formale) Größe Zeit erlangt hierbei über die (funktionalen) Größen Zustände Zugang zu der Applikation. Systeme, die von der Vorgeschichte abhängig sind, werden als Schaltwerke bezeichnet:

Definition 2.2

Ein **Schaltwerk** ist eine Funktionseinheit zum Verarbeiten von Schaltvariablen, deren Wert am Ausgang zu irgendeinem Zeitpunkt vom Wert am Eingang zu diesem Zeitpunkt sowie zu endlich vielen vorangegangenen Zeitpunkten abhängt.

Zur Realisierung eines Schaltwerks ist eine zunächst beliebige Speicherfunktion notwendig. In der Praxis werden natürlich genormte Bauteile, die Flipflops, hierfür verwendet:

Definition 2.3:

Ein **Flipflop** (bistabiles Kippglied) ist ein Speicherglied mit 2 stabilen Zuständen, das aus jedem der beiden Zustände durch eine geeignete Ansteuerung in den anderen Zustand übergeht.

Die Funktion dieses Flipflop wird durch 2 Teilfunktionen bestimmt:

- das Zulassen bzw. Sperren einer Veränderung und
- die Art der Änderung in Abhängigkeit von Eingängen.

Die Zulassungsfunktion wird meist als *Takt (Clock)* beschrieben. Taktzustands-gesteuerte Flipflops kombinieren die Übergänge mit einem der beiden möglichen *Taktzustände*. Während einer der beiden Taktzustände sind Eingangsänderungen am Ausgang sichtbar, während des anderen nicht, das Flipflop sperrt. Bild 2.8 zeigt ein taktzustandsgesteuertes Datenflipflop, auch als Latch bezeichnet, aufgebaut als DNF.

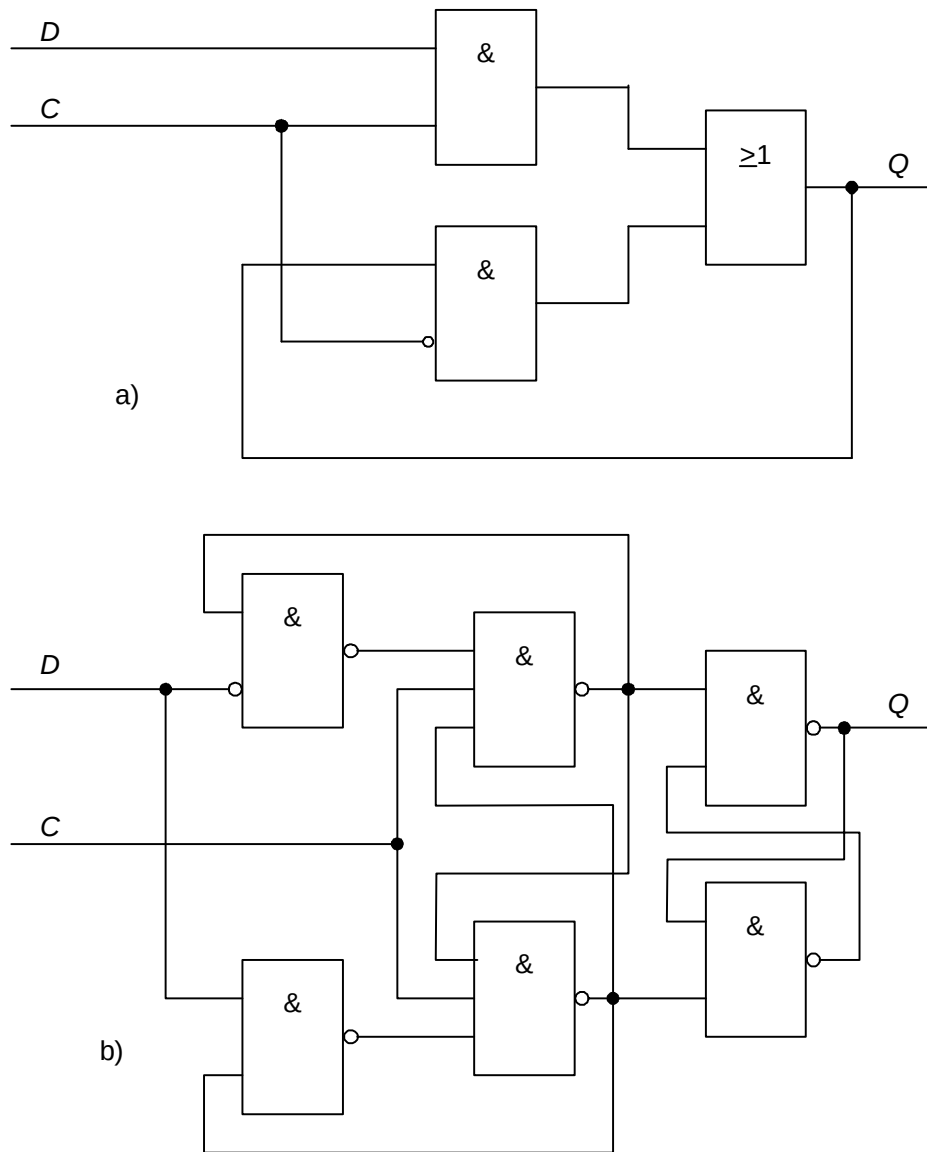


Bild 2.8 Datenflipflop a) Taktzustandsgesteuert (Latch) b) Taktflankengesteuert

Funktionell ist das D-Flipflop einfach zu beschreiben, da der Wert am Eingang D am Ausgang Q transparent erscheint, wenn $C = 1$ ist, während im Fall $C = 0$ der

zuletzt gespeicherte Wert gehalten wird. Gl. 2.2 beschreibt diesen Zusammenhang als DNF.

$$Q = D * C + Q * /C \quad (\text{Gl. 2.2})$$

Im Unterschied zu einer DNF wie in Bild 2.3 bzw. in Gl. 2.1 wird bei diesem Flipflop der Ausgang auf einen Eingang rückgekoppelt. Dies ist notwendig, da ja bei $C = 0$ gespeichert werden soll, zugleich im Allgemeinen aber gefährlich, denn eine gegenphasige Rückkopplung kann zu einer «Schwingung» führen.

Der Aufbau eines taktzustandsgesteuerten Flipflops ($\rightarrow$ Bild 2.8a) ist in den feldprogrammierbaren Logikbausteinen immerhin möglich, der eines taktflankengesteuerten ist schon wesentlich aufwendiger ($\rightarrow$ Bild 2.8b). Taktflankensteuerung bedeutet, dass der Eingangswert des D-Flipflops nur mit dem Wechsel des Taktsignals ($0 \rightarrow 1$: positive Flanke, $1 \rightarrow 0$: negative Flanke) vollzogen wird. Diese Kopplung an die Flanke bedeutet, dass hier keine ausschließlich logische Funktion mehr vorliegt. In der Realisierung wird hier auf direkte Transistorverschaltungen zurückgegriffen.

Die Art der Ansteuerung unterscheidet die einzelnen Flipfloptypen. Folgende charakteristische Gleichungen beschreiben dies:

$$\text{RS-Flipflop: } Q_{t(n+1)} = [S + (/R * Q)]_{t(n)} \quad (\text{Gl. 2.3})$$

$$\text{D-Flipflop: } Q_{t(n+1)} = [D]_{t(n)} \quad (\text{Gl. 2.4})$$

$$\text{T-Flipflop: } Q_{t(n+1)} = [(Q * /T) + (/Q * T)]_{t(n)} \quad (\text{Gl. 2.5})$$

$$\text{JK-Flipflop: } Q_{t(n+1)} = [(J * /Q) + (/K * Q)]_{t(n)} \quad (\text{Gl. 2.6})$$

Das D-(Daten-) und das T-(Toggle-)Flipflop besitzen dabei lediglich einen Eingang (D als zu speicherndes Datenbit bzw. T als Umschalter in den anderen Zustand), während RS (Reset/Set) und JK (Jump/Kill) zwei besitzen. Die charakteristischen Gleichungen 2.3 bis 2.6 sind so zu lesen, dass auf der rechten Seite die logischen Werte vor dem Taktereignis stehen und diese einem Ausgang mit dem Taktereignis zugewiesen werden.

Die hier aufgeführten Speicherelemente werden im Allgemeinen als Zustandsspeicher bezeichnet. Die Technologie ist fast ausschließlich SRAM, unabhängig von der Technologie des Konfigurationsspeichers. Dieser Speicher entspricht in etwa dem Datenspeicher des Von-Neumann-Modells.

2.2 Klassifizierung nach Speichertechnologie

Neben der bislang behandelten Architektur der PLDs ist es auch wichtig, die Wirkungsweise der Programmierung zu verstehen und die einsetzbaren Speichertechnologien zu betrachten, da sich hieraus einige Konsequenzen für den Betrieb ergeben. In diesem Zusammenhang sei nochmals erwähnt, dass sich die Speichertechnologien ausschließlich auf Konfigurationsspeicher beziehen.

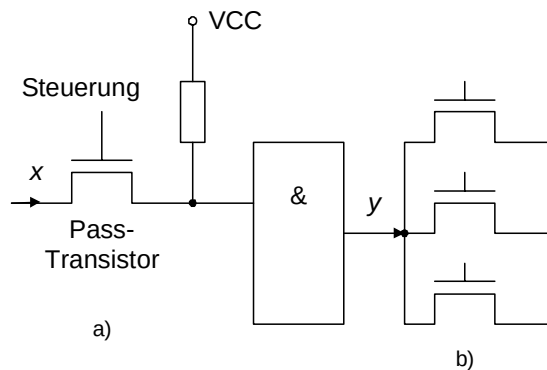


Bild 2.9 Nutzung von Pass-Transistoren in PLDs
a) Beschaltung von Gattereingängen b) Nutzung im Routing der Bausteine

Zentrales Element bei der Programmierung ist der Pass-Transistor (Bild 2.9). Dies ist meist ein selbstsperrender N-Kanal MOSFET-Transistor, der mit Hilfe einer geeigneten Ansteuerung die Zustände sperrend und leitend annehmen kann. In Bild 2.9 a) wird ein potenzielles Eingangssignal x an ein AND-Gatter über einen Pass-Transistor geführt. Wird dieser derart angesteuert, dass er sperrt, ist das Eingangssignal nicht sichtbar, und das AND-Gatter erhält via Pull-Up-Widerstand ein High-Signal am Eingang. Für ein OR-Gatter wäre hier entsprechend ein Pull-Down-Widerstand notwendig.

Der Ausgang des AND-Gatters y wird mit Hilfe von 3 Pass-Transistoren, die prinzipiell alle leitend sein können, im Baustein geroutet, d.h. elektrisch weitergeführt. Die Programmierung des Bausteins entspricht damit exakt der Steuerung dieser Pass-Transistoren, und diese Steuerinformationen müssen im Baustein gespeichert werden. Prinzipiell dürften bezüglich der Speichertechnologie damit keine großen Differenzen zu Mikrocontrollern und deren Speicher existieren.

Die 3 aktuellen Speichertechnologien, die für Konfigurationsspeicher zum Einsatz kommen, sind Anti-Fuse, (E)EPROM und SRAM. Anti-Fuse, aus Sicht der Mikroprozessoren und -controller der Exot, entstammt der alten Fuse-Technologie. Innerhalb der bipolaren Bausteine waren an den Kreuzungspunkten (z.B. aus Bild 2.5 oder 2.6) kleine Metallverbindung, die so genannten Fusable Links, angebracht. Programmierung bedeutete Zerstörung dieser «Schmelzsicherungen», wodurch der Kontakt dann unterbrochen wurde.

Der wesentliche Nachteil dieser Fuses bestand in hohen Übergangswiderständen oder großen Strömen bei der Programmierung. Für den Einsatz in PLDs wurde der Programmierprozess umgedreht, Verbindungen können nunmehr einmalig hergestellt (und nicht mehr zerstört) werden. Man nennt dies *Antifuse*, und der Grund zum Wechsel der Programmierrichtung ist leicht in den gegenüber den bipolaren

Fuses verbesserten elektrischen Eigenschaften (Widerstand) zu finden. Bild 2.10 zeigt das Prinzip dieser Zelle.

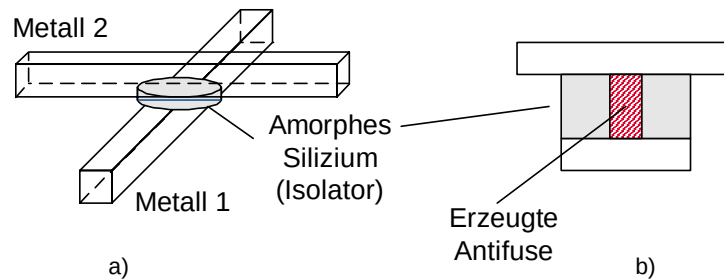


Bild 2.10 Antifuse-Technologie a) Kreuzung zweier Metalleiter im IC
b) Kreuzungspunkt nach Programmierung

Der metallische Leiter zwischen den eigentlichen Metalleitern (in Bild 2.10 b) schraffiert) wird durch kontrollierte Überspannung und einen Überschlag im Baustein erzeugt. Hierdurch kann man festhalten, dass diese Programmierung irreversibel ist, der Kreuzungspunkt jedoch sehr wenig Fläche bei geringem Widerstand und Kapazität beansprucht – Eigenschaften, die diese Technologie für den Betrieb ideal, für die Programmierung und Entwicklung eher unhandlich erscheinen lassen. Der Hauptvorteil der Antifuse-Technologie besteht in der hohen Geschwindigkeit der programmierten Bausteine, als wesentlicher Nachteil wird meist der nicht-reversible Prozess angesehen. Dieser Nachteil wird übrigens manchmal auch als Vorteil gesehen, denn was nicht mehr änderbar ist, kann auch nicht verfälscht werden.

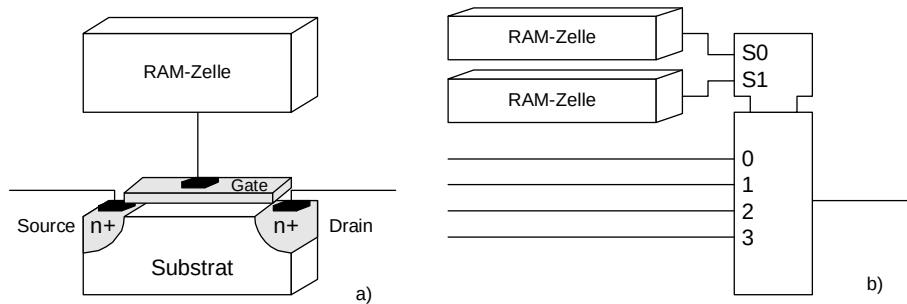


Bild 2.11 Nutzung von SRAM-Zellen für PLDs
a) mit Pass-Transistor b) 2 RAM-Zellen für 1-aus-4-Multiplexer (im Routing)

Das andere Extrem der Programmier Technologie besteht in statischen RAM-Zellen, die die Strukturinformation tragen. Statisches RAM (SRAM) entspricht der Verschaltung von 4–6 Transistoren zu einem (D-)Flipflop. Hierin kann die Programmierinformation gespeichert werden, die dann mittels eines Pass-Transistors als Verbindung/Trennung (Bild 2.11 a)) oder eines Multiplexers (Bild 2.11 b)) auswertbar ist (Anmerkung: Die Verwendung des 1-aus-4-Multiplexers entspricht nicht derjenigen für eine Look-Up-Table, → Bild 2.7, sondern derjenigen im Routing, → 3). Die SRAM-Technologie ist daher beliebig wiederverwendbar, selbst der Löschvorgang wird sehr einfach: Ein Abschalten der Betriebsspannung reicht, der Baustein ist danach unprogrammiert.

Der Programmiervorgang besteht aus einem Ladevorgang, häufig auch als Boot-Vorgang bezeichnet, und ist vergleichbar mit dem eines Programms zur Ausführung in einem PC. Dies ist natürlich jederzeit reversibel, was zugleich Vor- und Nachteil ist: Mit jedem Spannungsverlust ist die Programmierung des PLDs schließlich verloren. Um ein Stand-Alone-System zu erhalten, muss ein zusätzlicher Speicherbaustein (meist seriell es EPROM) hinzugefügt werden, der bei Reset oder Einschalten das Programm in den Baustein lädt.

Andererseits ermöglichen SRAM-basierte PLDs eine unbegrenzte Fähigkeit zur Aufnahme neuer Programme, was insbesondere bei der Entwicklung (Rapid Prototyping) zum Tragen kommt. Dynamisch rekonfigurierbare (auch partiell) Bausteine sind ebenfalls erhältlich, eine Fähigkeit, die zurzeit noch kaum in Produkten genutzt wird, die aber große Flexibilität verspricht.

Als Hauptvorteil von SRAM gilt die Flexibilität, die Hauptnachteile bestehen in der relativ niedrigen Geschwindigkeit (hohe Kapazitäten der Pass-Transistoren), in der Anzahl der Transistoren pro Speicherung des Werts sowie im Aufwand beim Einschalten der Spannung.

Die dritte Technologie liegt in gewissem Sinn in der Mitte: EPROM-Transistoren werden programmiert und sind hierdurch entweder durchschaltbar (1) oder nicht (0). Wie bei SRAM ist auch diese Technologie aus der Rechner Technik bekannt, sie wurde mit Erfolg in die PLD-Welt transferiert.

Die EPROM/Flash-EEPROM/EEPROM-Technologie basiert auf speziellen MOS-Transistortypen mit Floating Gate (→ Bild 2.12). Bei ungeladenem Floating Gate kann ein positives Selektierungssignal den sonst selbstsperrenden MOS-Transistor leitend schalten; dies wird verhindert, wenn das Floating Gate Elektronen trägt.

Meist werden jetzt EEPROM- oder Flash-EPROM-Zellen genutzt, die elektrisch löscher sind. Als weitere Entwicklung sind viele Bausteine In-System-Programmable (ISP), d.h., ein spezielles Programmiergerät entfällt, da die Programmierung über einfache digitale Leitungen (4–5#) durchführbar ist. Die (E)EPROM-Zellen sind durch hohe Kapazität und hohen Widerstand relativ langsam, was sich eigentlich in geringeren Geschwindigkeiten bemerkbar machen sollte. Da diese Technologie jedoch meist bei den CPLDs (Complex Programmable Logic Devices, → 3),

die – wie noch diskutiert wird – auf Geschwindigkeit optimiert sind, fällt dieser Nachteil jedoch kaum ins Gewicht.

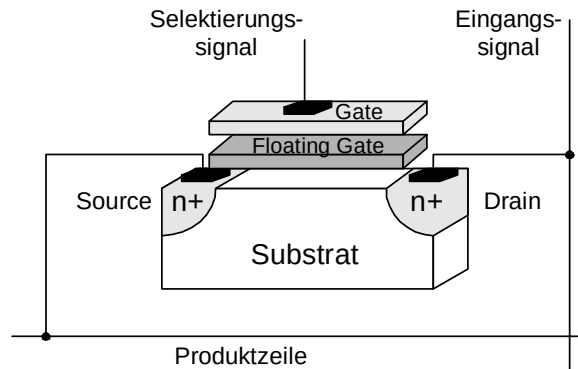


Bild 2.12 EPROM-Transistorzelle

Der Hauptvorteil der (E)EPROM-Technologie besteht in einem guten Kompromiss zwischen Flexibilität und Programmspeicherung. Eigentlich ist kein wesentlicher Nachteil zu erwähnen – außer vielleicht der Tatsache, dass keine herausragende Eigenschaft vorhanden ist. Ein wesentlicher Nachteil wird aber zukünftig bemerkbar sein: (E)EPROM-Zellen sind nicht mehr skalierbar, d.h., das durch den Technologiefortschritt mögliche Downscaling der Transistorgrößen ist unterhalb von $0,25\ \mu\text{m}$ für diese Transistoren nicht mehr möglich, ohne dass es zu drastischen Problemen mit der Zuverlässigkeit kommen würde.

	Anti-Fuse	(E)EPROM	SRAM
<i>Geschwindigkeit</i>	hoch	niedrig	niedrig
<i>Reprogrammierbarkeit</i>	keine	begrenzte Anzahl	beliebig oft
<i>Mögliche Besonderheiten</i>		In-System-Programmable	(partiell) dynamisch rekonfigurierbar

Tabelle 2.2 Zusammenfassung Speichertechnologien im Configurable Computing

Tabelle 2.2 fasst die Speichertechnologie zusammen. Grundsätzlich sind andere Technologien wie DRAM (dynamisches RAM), FRAM (ferroelektrisches RAM), MRAM (magnetoresistives RAM) und OUM (Ovonic Unified Memory) möglich, derzeit jedoch nicht in Gebrauch.

2.3 Klassifizierung nach Programmierbarkeit

Die Speichertechnologie hat schon Auswirkungen auf die Praxis, z.B. die Laufzeiten und die Verlustleistungen in den PLDs. Dennoch fallen diese Unterschiede kaum auf, denn es sind keine Größenordnungen, in denen sich die Bausteine voneinander differenzieren. Interessanter ist vielmehr, wie sich ein Programmierablauf gestaltet, welcher Aufwand an Zeit und Geräten getrieben werden muss.

2.3.1 Formen der (physikalischen) Programmierung

Die Antifuse-Technologie ist irreversibel und benötigt relativ hohe Spannungen und Ströme, damit die Zellen ihre Programmierung erhalten (jedoch nicht im eigentlichen Betrieb). Hier ist ständig ein Programmiergerät notwendig, und die einzelnen Hersteller spezifizieren diese auch dementsprechend. Antifuse-basierte PLDs entsprechen am ehesten den (in der Fabrik hergestellten) ASICs, und es macht kaum Sinn, einen Aufwand in Richtung einer Programmierung im System, also ohne Programmiergerät zu betreiben, wie es für die anderen Speichertechnologien beschrieben wird. Die Stärken der Antifuse-Technologie liegen eben in der Geschwindigkeit, in der Unlösbarkeit (und damit Anwendungssicherheit in Extremfällen), des sehr guten Schutzes gegen unberechtigtes Auslesen (Schutz der *Intellectual Properties* gegen Reengineering) und des Verzichts auf externes Boot-EPROM (wie bei SRAM notwendig).

Für EPROM-basierte PLDs gibt es jedoch Variationsmöglichkeiten in der Programmieretechnik. War es ursprünglich so, dass auch diese Bausteine ausschließlich in Programmiergeräten ihr Programm erhalten konnten (und dass bei vielen der Löschvorgang mittels UV-Licht durchzuführen war), so hat sich dies grundlegend geändert. Mittlerweile hat sich die In-System-Programmierung durchgesetzt.

Mit *In-System-Programmable (ISP)* wird eine Technologie und Architektur bezeichnet, die die Programmierung von bereits eingebauten Bausteinen, also im System, ermöglicht. Sollte dabei die Erzeugung der Spannung zur Programmierung auf dem Chip erfolgen, dann werden weder besondere Spannungen (z.B. 16,5 V) noch eine kostspielige Datenzuführung benötigt.

Für diese In-System-Programmierung werden die Daten in der Regel seriell in den Baustein übertragen, und somit benötigt man nur 4 Anschlüsse dafür: *Mode Select* zum Umschalten zwischen Programmierung und Normalbetrieb, *Clock* zum seriellen Eintakten, *Data In* für die Eingangsdaten und *Data Out* zum Auslesen, z.B. für ein Verifizieren der Programmierung. Durchaus nicht zufällig entsprechen Anzahl und Funktion denjenigen Anschlüssen, die in der Norm IEEE 1149.1 als Testinterface für den Boundary-Scan-Test (→ 2.3.2) genannt sind, und mittlerweile haben einige IC-Hersteller Programmier- und Testinterface vereinigt.

In jedem Fall ist es wichtig, bei den EEPROM-basierten Systemen die endliche Anzahl der Reprogrammierungen und den Datenerhalt zu beachten (mindestens

100 Zyklen, etwa 20 Jahre Datenerhalt). Die Werte sind meist nicht kritisch, da in der Praxis nicht sehr oft reprogrammiert wird. Die wesentlichen Stärken der EEPROM-basierten Bausteine ist ihre Flexibilität für Updates sowie der Verzicht auf einen Bootspeicher.

Die *SRAM-basierte* Speichertechnologie kann nur «In-System» betrieben werden, da die Informationen so gespeichert werden, dass sie bei Spannungsverlust flüchtig sind. Da die RAM-Zellen nach jedem Einschalten neu konfiguriert werden müssen, kann hier die Programmierzeit eine Rolle spielen. Aus diesem Grund werden meist verschiedene Interfaces (seriell, parallel) angeboten. Ein aktives Interface erzeugt den Auslesetakt für z.B. ein serielles oder paralleles EPROM selbst, ein passives wird von außen gesteuert und z.B. bei Konfigurationssteuerung durch einen Mikroprozessor verwendet.

Hier gibt es jedoch noch eine andere Spezialität: die partielle dynamische Rekonfigurierbarkeit. Die Grundidee stammt aus der Rechnerwelt, denn dort können nicht benötigte Programme oder Daten aus dem Speicher entfernt und dafür andere geladen werden. Diese Aufgabe gehört in die Bereiche Task Management sowie Memory Management und wird vom Betriebssystem meist mit Hardwareunterstützung durchgeführt.

Ein Betriebssystem für PLDs wird man nun vergebens suchen (obwohl derartige Ansätze in der Forschung durchaus diskutiert werden, → 5). Somit verbleibt die Organisation der partiellen Rekonfiguration beim Anwender, und es wird noch sehr selten genutzt – noch!

2.3.2 Testbarkeit

Mitte der 80er Jahre zeigte sich die wachsende Komplexität von integrierten Schaltungen und den daraus aufgebauten Boards in aller Deutlichkeit. Das Problem, wie die gesamte Schaltung zu testen sei, war zwar zu lösen, aber der Aufwand stieg enorm an, denn letztendlich musste ein Messgerät mit vielen kleinen Messspitzen die Signale auf dem Board aufnehmen.

Und noch eine Fragestellung trat auf: Um z.B. eine fehlertolerante Schaltung testen zu können, müssen Fehler erst einmal erzeugt werden – Fehler, die eigentlich nicht auftreten dürfen. Um dies planvoll gestalten zu können, wird ein Mechanismus außerhalb des eigentlichen Betriebs benötigt. Um den einsetzenden Wildwuchs an firmenspezifischen Lösungen einzudämmen, wurde in Europa die Joint Test Action Group (JTAG) gegründet, der sich später auch amerikanische und asiatische Firmen anschlossen und deren Aktivitäten 1990 zum IEEE 1149.1-Standard führte. 1993 wurde dies u.a. durch eine Boundary Scan Definition Language (BSDL, Untermenge von VHDL) zur Beschreibung der im IC integrierten Testfunktionen ergänzt.

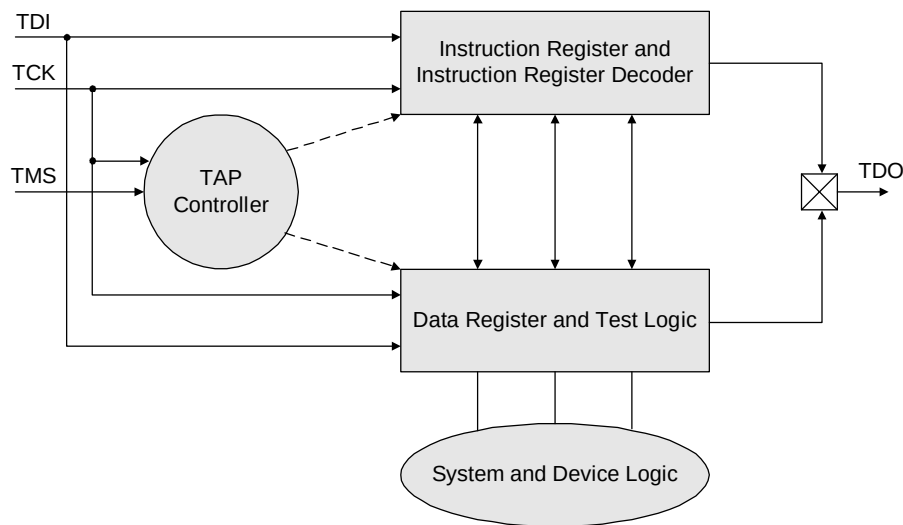


Bild 2.13 Architektur der Boundary-Scan-Schnittstelle

IEEE 1149.1 (für eine ausführliche Einführung siehe [1]) definiert das Interface zum Testen von Bausteinen und Boards, nicht jedoch, wie spezifische Applikationen zu testen sind. Beim Interface sind wie üblich Hardware- und Softwareteil zu unterscheiden, und für den Hardwareteil hat man einen minimalen Ansatz gewählt. Der *Test Access Point* (TAP) umfasst die obligatorischen Anschlüsse *Test Clock* (TCK), *Test Mode Select* (TMS), *Test Data In* (TDI) und *Test Data Out* (TDO) sowie den optionalen Anschluss *Test Reset* (/TRST). Test Reset (aktiv low) setzt dabei alle Testregister (Zustand und Instruktion) zurück, hat natürlich keine Wirkung auf die eigentlichen Daten im Baustein.

Die Softwareseite besteht aus dem TAP-Controller, einem synchronen, endlichen Automaten (FSM, Finite State Machine), und 2 Shiftregistern zur Aufnahme einer Instruktion und eines Datensatzes (Bild 2.13). Hiermit ist die Arbeitsteilung naheliegend: Im TAP-Controller wird entschieden, was momentan überhaupt zu machen ist (keine Aktion, Laden von Registern, Ausführen von Aktionen), während in den Registern die Feinspezifikation gespeichert wird.

Im TAP-Controller selbst wird zwischen 4 Basisaktionen und darin mehreren Unteraktionen unterschieden. Bild 2.14 zeigt den Zusammenhang zwischen diesen Zuständen.

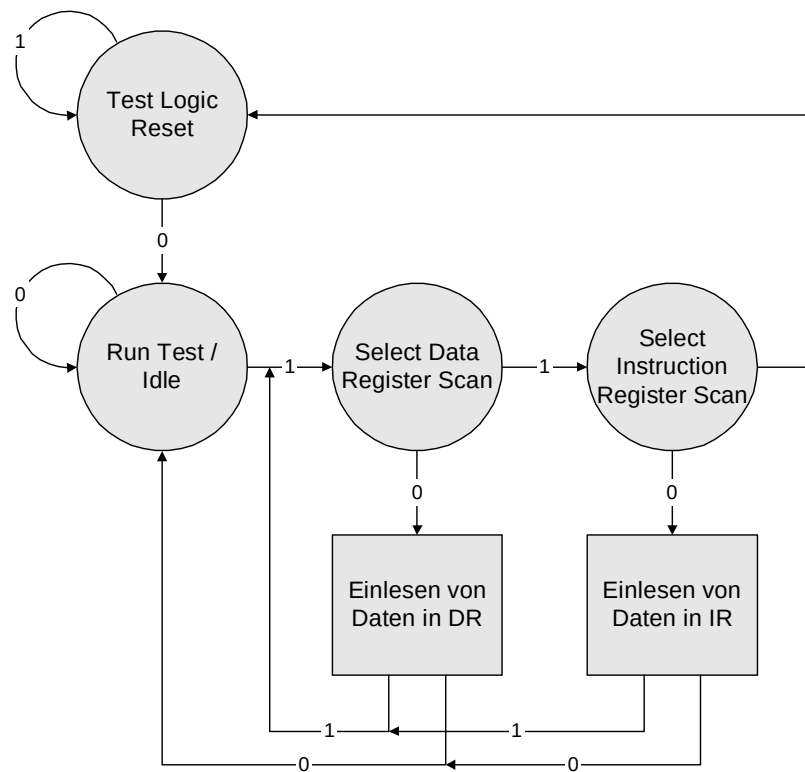


Bild 2.14 Zustandsdiagramm TAP-Controller

Drei Instruktionen sind in IEEE 1149.1 obligatorisch: BYPASS, SAMPLE und EXTEST. BYPASS schaltet das Datenregister schlicht ab, sodass alle Daten, die in den Baustein über TDI seriell eingelesen werden (Takt: TCK), sofort am Ausgang TDO wieder erscheinen. Der Baustein wird quasi transparent.

SAMPLE bewirkt das Einlesen der Daten in das eigentliche Boundary Scan Register, das ein Abbild für die In- und Outputs des Bausteins darstellt. Der gleiche Befehl kann zum Auslesen via TDO genutzt werden. EXTEST lässt dann den Baustein mit den im Datenregister geladenen Werten laufen.

Optionale Befehle wie HIGHZ (alle Ausgangstreiber werden ausgeschaltet), IDCODE und USERCODE (Baustein- bzw. Usercode wird ausgelesen) runden die Möglichkeiten ab, und auch das Programmieren kann in In-System-Programmable-ICs über (herstellerspezifische) Befehle erfolgen.

2.3.3 Zusammenfassung zur Programmierbarkeit

Alle Devices aus dem Bereich Configurable Computing sind programmierbar in dem Sinn, dass die endgültige Funktionalität erst durch das "Einspielen" des Programms erreicht wird. Tabelle 2.3 fasst die unterschiedlichen Versionen der Programmierbarkeit mit den besonderen Eigenschaften zusammen.

Programmierbarkeit	Speichertechnologie	Spezifische Eigenschaft
One-Time-Programmable	Anti-Fuse	besondere Sicherheit, ASIC-Ersatz
Erasable	EPROM, Flash-EEPROM	In kleinem Maß reprogrammierbar, z.B. für Service
In-System-Programmable	Flash-EEPROM	programmierbar nach IC-Montage, Testinterface
Rekonfigurierbar	SRAM	Beliebig oft reprogrammierbar, mit Aufwand auch zur Laufzeit
Partiell rekonfigurierbar	SRAM	wie rekonfigurierbar, zusätzlich Betrieb möglich bei partieller Reprogrammierung

Tabelle 2.3 Zusammenfassung zur Programmierbarkeit

2.4 Klassifizierung nach Granularität

Die bisherigen Darstellungen erwecken den Eindruck, dass alle Bausteine aus dem Bereich des Configurable Computing aus einer großen Matrix mit logischen Operationselementen (AND, OR, NOT) bestehen würden. Dies gilt jedoch nur im einfachsten Fall (bei den sogenannten SPLDs, Simple Programmable Logic Devices), im allgemeinen Fall ist die Architektur komplexer.

Mehr noch als das, auch der Typ der Grundoperation wechselt zwischen verschiedenen Klassen, und zwar zwischen logisch und arithmetisch. In Kapitel 3 werden konkrete Architekturen vorgestellt. Die Grundidee besteht darin, nicht einen, sondern mehrere Operationsblöcke vorzusehen und diese dann durch ein ebenfalls konfigurierbare Netzwerke miteinander zu verbinden. Daraus ergeben sich folgende Klassifizierungsmerkmale:

- Ausführung des Netzwerks (vollständig, Bussystem, Nearest Neighbour, Mischformen)
- Ausführung der Operationsblöcke (arithmetische Operationen, logische Operationen, Bitbreite)

- Konfigurierbarkeit der Operationsblöcke (vollständiger Designraum = LUT, PAL-Struktur, Auswahl vordefinierter Operationen)

Die Unterscheidung arithmetisch/logisch ist hierbei vielleicht ungewöhnlich, hat jedoch erhebliche Auswirkungen:

Definition 2.4:

Eine Operation wird als logisch bezeichnet, wenn für jedes Ergebnisbit dieser Operation gilt, dass es nur von einem einzigen Bit eines jeden Eingangsoperanden abhängt. Im anderen Fall wird die Operation als arithmetisch bezeichnet.

Beispiele für logische Operationen sind die Verknüpfungen AND, OR, XOR und NOT, auch für Bitvektoren (z.B. 8 Bit), sowie die Shift- und Rotate-Operationen (wobei hier Shift oder Rotate um eine variable Anzahl der Bits als weiterer Eingangswert schon als arithmetisch zählt). Typische Beispiele für arithmetische Operationen sind Addition und Multiplikation. Die logischen Operationen steigen im Hardwareaufwand (Fläche) im Allgemeinen linear mit der Anzahl der Eingangsbits, während die arithmetischen Operationen exponentiell (paralleler Addierer: 4^k , Carry Look-Ahead Adder: 3^k bei k bit pro Eingang) in der Fläche ansteigen.

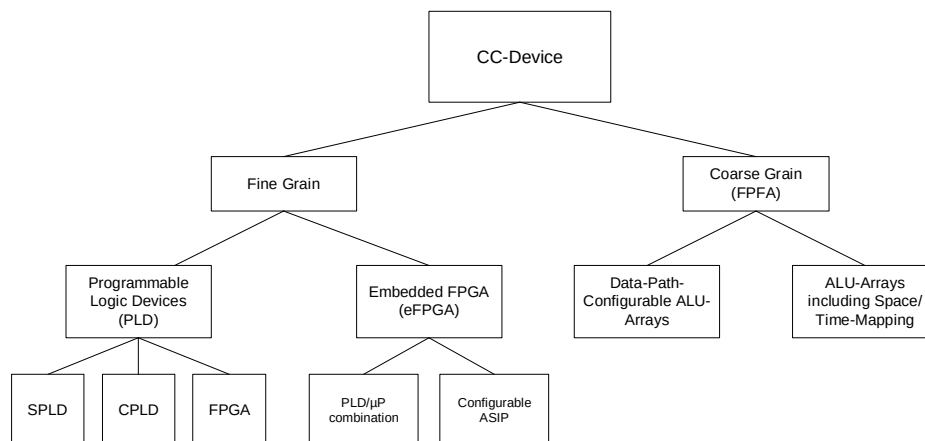


Bild 2.15 Klassifizierung der Architekturen im Bereich Configurable Computing

In der Praxis ergibt dies nun, dass zwischen bit- (= logische Operationen) und wort-basierten (= arithmetische Operationen) Bausteinen unterschieden wird (zumindest in der Forschung). In den einzelnen Klassen werden dann weitere Subklassen unterschieden, die die Netzwerkstruktur betreffen. Im Kapitel 3 wird die Designmethodik, die auf die bit-orientierten Bausteine (Fine Grain) angewendet wird, beschrieben. Kapitel 4 und 5 behandeln dann die bit-basierten Bausteine, Kapitel 6 zieht dann Resümee und führt die wort-basierten Bausteine ein (soweit aktuell möglich).

3 Codierungsmodell für die Hardwareentwicklung

Für die binärwertige, digitale Hardwareentwicklung sind allgemein – von Spezialfällen abgesehen – zwei Codierungsmodelle in Nutzung: Einfache Boolesche Netzwerke, als Schaltnetz bezeichnet, sowie Schaltwerke, die eine Gedächtnisfunktion besitzen und damit Vorgeschichte kennen.

Definition 3.1

Ein **Schaltnetz** ist eine Funktionseinheit zum Verarbeiten von Schaltvariablen, deren Wert am Ausgang zu irgendeinem Zeitpunkt nur vom Wert am Eingang zu diesem Zeitpunkt abhängt.

Definition 3.2

Ein **Schaltwerk** ist eine Funktionseinheit zum Verarbeiten von Schaltvariablen, wobei der Wert am Ausgang zu einem bestimmten Zeitpunkt abhängt von den Werten am Eingang zu diesem und endlich vielen vorangegangenen Zeitpunkten.

Die Schaltnetze besitzen einen einfachen funktionalen Zusammenhang zwischen Ein- und Ausgängen, wobei die Eingangswerte ausschließlich zum Zeitpunkt der Beobachtung unter Vernachlässigung der endlichen Signalausbreitungsgeschwindigkeit im Baustein betrachtet werden. Hierfür können Syntheseverfahren angegeben werden (→ 5.3), die bei ausreichenden Ressourcen immer zum Erfolg führen.

Schaltwerke werden im Folgenden näher betrachtet, sie sind die Implementierung der deterministischen endlichen Automaten (DEA, DFA). In der Technik werden dabei fast ausschließlich synchrone Schaltwerke genutzt, deren zeitliche Steuerung einem ausgezeichneten Signal, mit *Takt* (*Clock*) bezeichnet, obliegt. Die ist das Codierungsmodell für die Hardwareentwicklung.

Eine besondere Implementierung der Schaltwerke, mit immer noch geringer Bedeutung aber stark zunehmender Beachtung, sind asynchrone Schaltwerke, besser als selbst-synchronisierende Schaltwerke bezeichnet. Der Verzicht auf den Takt hat besondere technische Gründe wie die Minimierung der Verlustleistung und der elektromagnetischen Störungen.

Diese Variante ist nicht Gegenstand dieser Vorlesung (weil diese Vorlesung nicht den Designverfahren, sondern den Architekturen gewidmet ist), geneigte Leser finden jedoch entsprechende Literatur in [31].

Der Zusammenhang zwischen diesem Kapitel und dem Configurable Computing ist im Übrigen leicht hergestellt. Die programmierbare, bit-orientierte Logik der PLDs, die im nächsten Kapitel dargestellt werden, entstand als Ersatz für fixierte Hardware (ASICs). Dementsprechend war und ist die Designmethodik für diese Bausteine die gleiche wie für Hardware, und der interne Aufbau der PLDs muss sich an dieser Tatsache orientieren.

Für die wort-orientierten Devices der FPFA-Klasse ($\rightarrow$ Kapitel 6) gelten jedoch andere Gesetze: Diese stammen eher aus den Mikroprozessorarchitekturen, und dementsprechend fordert man hierfür auch Designmethodik und Sprachen wie für Mikroprozessoren (sprich: sequenzielles Programmiermodell, typischen Sprachen C, C++, Java).

3.1 Deterministic Finite Automaton

Das Konzept der deterministischen endlichen Automaten (Deterministic Finite Automaton, DFA) stellt eine starke Brücke zwischen Theoretischer Informatik und angewandter Ingenieurwissenschaft dar. In der Praxis unterscheidet man dann zwischen einfachen und komplexen (kooperierenden) Automaten.

Definition 3.3

Ein **deterministischer endlicher Automat** (DFA) M wird durch ein 5-Tupel

$$M = (Z, \Sigma, \delta, z_0, E)$$

spezifiziert. Hierbei bezeichnet Z die Menge der Zustände (endlich), Σ ist ein endliches Eingabealphabet, $z_0 \in Z$ ist der Startzustand, $E \subseteq Z$ ist die Menge der Endzustände und $\delta: Z \times \Sigma \rightarrow Z$ heißt Überföhrungsfunktion.

Der DFA stellt ein theoretisches Konzept als grundlegendes Modell dar, für das eine sehr häufig genutzte technische Ausprägung existiert. Die Menge der Zustände eines DFAs werden im Rahmen der Technik durch Speicherbelegungen, d.h. in Registern gespeicherte Binärwerte realisiert. Dies ist der für (*synchrone*) *Schaltwerke* (Definition 3.3) notwendigen Zwischenspeicherung zu endlich vielen vorangegangenen Zeitpunkten äquivalent, benötigt jedoch mindestens ein zusätzliches Taktsignal. Das Eingabealphabet Σ entspricht den Eingängen des Schaltwerks, die Überföhrungsfunktion dem Schaltnetz zwischen den Eingängen, den Speicherfunktionen und den Ausgängen.

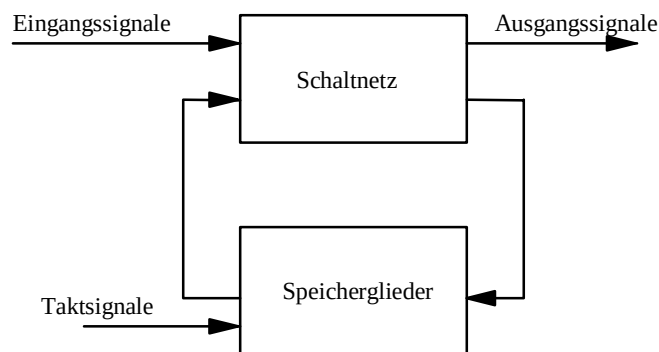


Bild 3.1 Aufbau eines synchronen Schaltwerks

Die synchronen Schaltwerke, die im Rahmen dieses Skripts ausschließlich betrachtet werden, lassen sich in einfache und kooperierende Schaltwerke unterteilen. Die *einfachen Schaltwerke*, auch als Einregister-Automat bezeichnet, bestehen dabei aus einer Struktur gemäß Bild 3.1, wobei die Speicherglieder von *genau einem Taktsignal* gesteuert sind.

3.2 Klassen einfacher Zustandsautomaten

Zur weiteren Klassifizierung wird der in Bild 3.1 gezeigte Aufbau eines Schaltwerks weiter spezifiziert. Nach [3] wird im allgemeinen folgende Definition für die technische Ausführung in Form eines endlichen Zustandsautomaten (Finite State Machine, FSM) gegeben:

Definition 3.4:

Eine FSM besteht aus einem 6-Tupel $\{A, X, Y, f, g, a_0\}$. $A = \{a_0, a_1, \dots, a_M\}$ ist hierbei die endliche Menge der Zustände, a_0 der Startzustand. $X = \{X_1, \dots, X_K\}$ ist die endliche Menge der Eingangsvektoren mit $X_I = (x_1, \dots, x_L)_i$, $x_i \in \{0, 1, -\}$. $Y = \{Y_1, \dots, Y_N\}$ ist die endliche Menge der Ausgangsvektoren mit $Y_I = (y_1, \dots, y_H)_j$, $y_h \in \{0, 1, -\}$. $f: A \times X \rightarrow A$ heißt Transitionsfunktion (Next State Decoder), $g: A \times X \rightarrow Y$ heißt Ausgangsfunktion (Output Decoder).

Die in dieser Definition genannten Funktionen werden durch Schaltnetze realisiert, die den algorithmischen Zusammenhang zwischen den (im wesentlichen binären) Eingangs- und Zustandsvektoren darstellen. Hierzu ist allerdings notwendig, dass auch die Zustände binär codiert werden, was in Definition 3.4 noch nicht der Fall war.

Bild 3.2 zeigt die allgemeine Definition sowie drei spezifizierte FSM-Typen, wobei die Zustandskodierung mit u bezeichnet wird. $g(u, x)$ stellt die Ausgangsfunktion (Output Decoder) für die Ausgangsvariablen y dar, $f(u, x)$ die Übergangsfunktion (Next State Decoder) für den inneren Zustand u , der zum Zeitpunkt t_{n+1} in die Speicherglieder übernommen wird. t_n und t_{n+1} stehen dabei in enger Korrelation mit dem (einzigen) Takt C und markieren die aktiven Zeitpunkte der Speicherung.

Die Unterschiede zwischen Medwedjew- und Moore-Automat bestehen in einem zusätzlichen Schaltnetz $g(u)$ am Ausgang des Moore-Automaten, sodass die Ausgangssignale beim Moore-Automat nicht eine einfache Teilmenge des inneren Zustands darstellen. Medwedjew- und Moore-Automat können die Änderung einer Eingangsvariablen grundsätzlich erst bei der nächsten Taktflanke am Ausgang des Schaltwerks zeigen. Beim Mealy-Automat besteht jedoch eine asynchrone Kopplung zwischen Ein- und Ausgang, sodass im Unterschied zu den erstgenannten die Änderung von Eingangsvariablen auch ohne Takt Ausgänge beeinflussen kann. Moore- und Medwedjew-Automat werden aufgrund des Fehlens der asynchronen Kopplung auch als zustandsorientierte Schaltwerke bezeichnet.

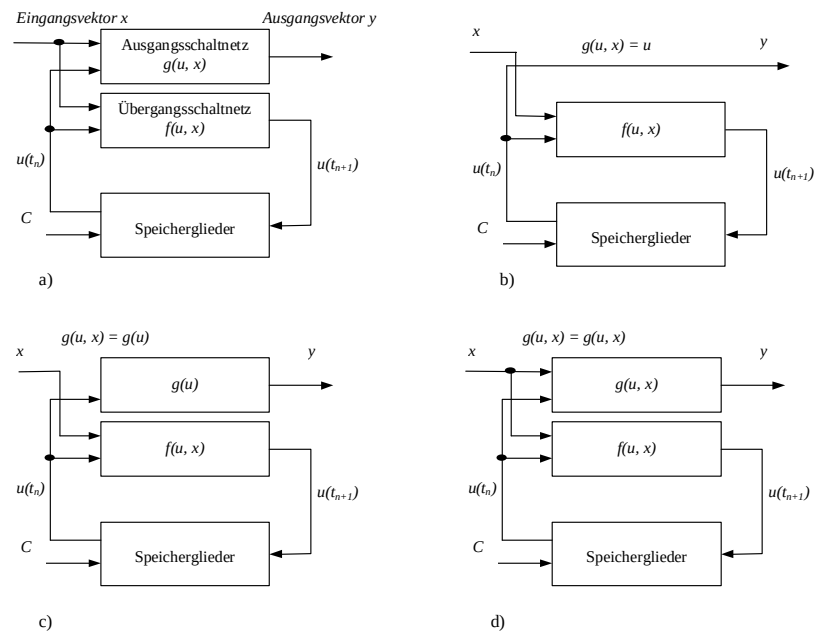


Bild 3.2 Einfache Automatentypen
a) Allgemeiner struktureller Aufbau b) Medwedjew-Automat
c) Moore-Automat d) Mealy-Automat

3.3 Syntheseverfahren für Zustandsautomaten

Die Synthese von Zustandsautomaten ist keineswegs mehr so einfach zu beschreiben, wie dies noch bei Schaltnetzen der Fall war. Der Grund hierfür liegt in der zusätzlichen Ebene der Codierung der Zustände aus Z gemäß den Definitionen 3.2 und 3.3; die Wahl der Codierung hat maßgeblichen Einfluss auf die Ausführung der verbindenden Schaltnetze.

Nach [6] müssen für die Synthese eines Zustandsautomaten folgende Schritte durchlaufen werden:

1. **Festlegen der Signale:** Das Schaltwerk wird als "Black Box" mit den erforderlichen Eingangs- und Ausgangssignalen skizziert.
2. **Entwerfen des Zustandsdiagramms:** Dieser Schritt ist der eigentliche Kern der Synthese, da hier das zu lösende Problem formal beschrieben wird. Als Zustandsdiagramm sind Zustandsgraphen (siehe Bild 3.3), Programmablaufpläne oder auch Schaltwerkstabellen (siehe unten) möglich.

3. *Aufstellen der Schaltwerktafel* als formaler Ausgangspunkt für alle weiteren Operationen.
4. *Zustandsminimierung*: Die Minimierung der Anzahl der Zustände soll eine Vereinfachung des Designs erreichen. Hierzu ist ein formales Verfahren entwickelt worden [6], auf das noch kurz eingegangen wird.
5. *Zustandscodierung*: Bei synchronen Schaltwerken mit synchronisierten Eingängen (nicht Mealy-Automat) lassen sich beliebige Codierungen für die Zustände aus Z angeben. Beim Medwedjew-Automat müssen die Codierungen allerdings mit den gewünschten Ausgangssignalen übereinstimmen; beim Moore-Automat hingegen können die Codierungen so gewählt werden, dass sich vereinfachte Schaltnetze für Next State Decoder (f) und Output Decoder (g) ergeben.
6. *Berechnung der Folgezustands- und Ausgangsschaltnetzen*: Zur konkreten Berechnung des Folgezustandsschaltnetzes muss ein Register- bzw. Flipfloptyp gemäß (2.3) bis (2.6) gewählt werden, da deren Eingänge (T, D, RS oder JK) durch dieses Schaltnetz belegt werden, aber unterschiedliche Funktionalitäten zeigen.
7. *Realisierung des Schaltwerks und Test*

3.4 Darstellungsformen für Zustandsautomaten

Es existiert eine Reihe von Darstellungsformen für Zustandsautomaten, von denen zwei dargestellt werden sollen: Zustandsgraphen und Schaltwerktafel.

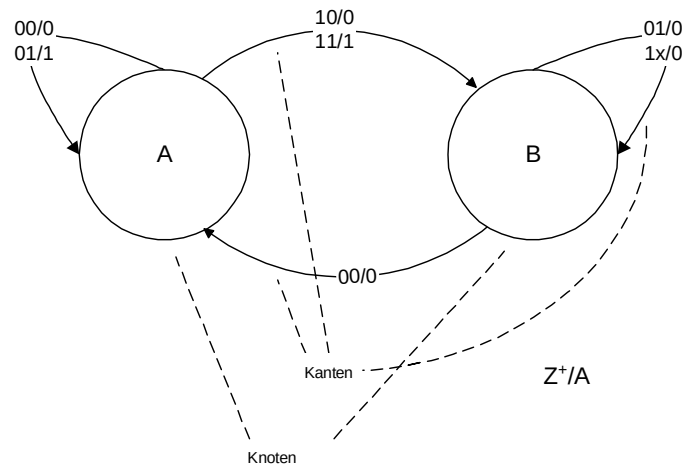


Bild 3.3 Zustandsgraph

Der Zustandsgraph (Bild 3.3) besteht aus Knoten und Kanten mit Zusatzinformationen. Die Knoten repräsentieren die Zustände, während die Kanten mit den Übergängen assoziiert werden. Die Zusatzinformationen beschreiben die Bedingungen, bei denen ein Übergang erfolgt, sowie ggf. die Ausgangswerte.

Die andere Version, die insbesondere unter Punkt 3 der Synthesemethode für Schaltwerke erwähnt ist, ist die Schaltwerktafel:

Definition 3.5:

Die **Schaltwerktafel** besteht aus der tabellarischen Darstellung der Folgezustände und der Ausgangssignale in Abhängigkeit von den Zuständen und Eingangssignalen. Hierzu wird jedem Zustand eine Zeile, jeder Eingangswertkombination eine Spalte zugeordnet. An den Kreuzungspunkten wird der Folgezustand und der Ausgangswert, getrennt durch einen Schrägstrich, eingetragen.

Bild 3.4 zeigt beispielhaft eine Schaltwerktafel:

Z	a b	00	01	10	11 (Eingänge)
A		A/0	A/1	B/0	B/1
B		A/0	B/0	B/0	B/0

Z^+/A

Bild 3.4 Schaltwerktafel

3.5 Zustandsminimierung

Folgendes Verfahren zur Minimierung der Anzahl der Zustände wird in [6] angegeben:

1. Die Tabelle wird nach Ausgangsgruppen geordnet. Eine Ausgangsgruppe ist durch den Vektor aller Ausgangswerte, die den Eingangswerten zugeordnet sind, definiert. Zur Klassifizierung werden die Ausgangswerte nebeneinander geschrieben. Folgerung: Man benötigt mindestens so viele Zustände, wie es Ausgangsgruppen gibt.
2. Für jede Ausgangsgruppe werden separat die Folgezustandsgruppen gebildet, d.h., zu jeder Eingangswertkombination werden für alle Zustände die Folgezustände betrachtet. Alle Folgezustände bilden die Folgezustandsgruppe bezüglich dieses Eingangszustands und müssen innerhalb derselben Ausgangsgruppe liegen. Ist diese Bedingung erfüllt, können die Zustände zu einem verschmolzen werden, ansonsten nicht.

Beispiel:

Z	e	0	1
A		D/1	C/0
B		D/0	B/1
C		B/1	A/1
D		A/1	C/0

Z	e	0	1
B		D/0	B/1
A		D/1	C/0
D		A/1	C/0
C		B/1	A/1

Wert haben müssen. Diese Codierung ist sehr implementierungsintensiv (bei k Zuständen k Register), lässt jedoch umfangreiche Fehlererkennungsmaßnahmen zu. In der Binärcodierung hingegen lassen sich n Zustände mit $2^{k-1} < n \leq 2^k$ durch k Bits darstellen, wodurch die Anzahl der Register drastisch minimiert wird, zugleich jedoch auch die Möglichkeiten zur Detektierung von Fehlern. In der Praxis wird meist eine Codierung gewählt, die den Bedürfnissen der jeweiligen Applikation entspricht.

Für weitere Regeln zur Zustandskodierung, d.h. zur konkreten Wertzuweisung an Zustände, siehe [6].

4 Architekturen und Technologien im Configurable Computing

Die ersten feldprogrammierbaren Logikbausteine gehörten zu denjenigen, die man aus jetziger Sicht als Simple Programmable Logic Devices (SPLDs) bezeichnet. Damit ist eine Klasse von programmierbaren ICs definiert, die der Struktur in Bild 4.1 entspricht und eine Art monolithischen Aufbau vorweist.

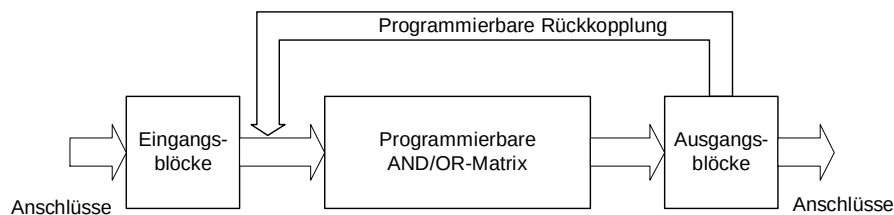


Bild 4.1 Aufbau Simple Programmable Logic Devices (SPLD)

Mit monolithischem Aufbau ist gemeint, dass alle Eingänge einschließlich der rückgekoppelten Ausgänge auf den (einzigen) Logikblock gekoppelt sind und damit dort in der programmierten Funktion werden können.

Selbst innerhalb der SPLD-Klasse waren die ersten Bausteine natürlich einfach. Etwa Mitte der 80er Jahre wurden PLDs erhältlich, die einen Industriestandard bildeten und bis heute als solcher gelten. Besonders ist hier der PAL22V10 in verschiedenen Varianten (PALCE22V10, GAL22V10) zu nennen: Mit maximal 22 Eingangs- und 10 Ausgangssignalen (synchronisiert, asynchron), 120 Konjunktionen (Terme) sowie einigen Steuersignalen ist dieser Baustein recht variabel einsetzbar.

Die Weiterentwicklung der monolithischen Bausteine wurde hier gestoppt, weil verschiedene Gründe für neue Architekturen sprachen:

- Die Vergrößerung einer LUT-Matrix wird ab wenigen Signalen (ca. 5) unwirtschaftlich, da sehr viele Verbindungen (2^k bei k Eingängen) pro intern neu erzeugtem Signal zu programmieren sind. Viele Transistoren bedeuten auch eine Summe vieler kleiner Verlustströme, neben der erforderlichen Siliziumfläche.
- Die Vergrößerung einer PAL-Matrix wird ab einer mittleren Zahl von Signalen (ca. 20–64) ebenfalls unwirtschaftlich. Hier steigt die Anzahl der Programmstellen – bei etwa gleichbleibender Qualität der Nutzbarkeit eines Blocks – zwar nur mit k (exakt $k + O(1/k)$, $\rightarrow 3.6$), aber auch dies wirkt sich aus.
- Beiden Strukturen ist gemeinsam, dass sie direkt keinen allgemeinen endlichen Automaten (FSM, $\rightarrow 3$) (mit Ausnahme des Medwedjew-Automaten, $\rightarrow$ Bild

3.2) integrieren können, denn beide verfügen nicht über die eigentliche Zweiteilung der Schaltnetze mit Speicherung zwischen diesen (→ Bild 3.2).

- Auch für asynchrone Schaltnetze gilt, dass sich diese häufig in mehrteiliger Form, also als Folge von disjunktiven Normalformen (und nicht als eine einzige) wesentlich optimaler implementieren lassen. Optimal gilt hier im Sinn einer Ressourceneffizienz, nicht der Geschwindigkeit.

Als Folge dieser Überlegungen entstanden dann ab Mitte der 80er Jahre die ersten hochdichten PLDs (HDPLDs, High-Density PLDs), deren Hauptmerkmal in einer geblockten Architektur lag. Diese HDPLDs – eigentlich sind es 2 Klassen, die *Field-Programmable Gate Arrays* «FPGA» und die *Complex PLDs* «CPLD» – haben sich seitdem natürlich weiterentwickelt, und mittlerweile können 3 Charakteristika zumindest einiger HDPLDs festgestellt werden:

1. Der blockweise (segmentierte) Aufbau der HDPLDs mit programmierbaren Logikarrays und programmierbaren Verbindungen zwischen diesen.
2. Die Integration von (statischem) RAM zur Datenspeicherung, unabhängig vom Programmspeicher und von den Registern.
3. Die Anpassung der Bausteine an verschiedene I/O-Standards.

Während Punkt 1 obligatorisch ist, sind die übrigen optional und erst in letzter Zeit erhältlich. In den folgenden Abschnitten wird zunächst der Industriestandard PAL/GAL22V10 vorgestellt, um einen Vergleich der HDPLDs zu den SPLDs zu ermöglichen.

4.1 Industriestandard PAL/GAL22V10

Bild 3.1 zeigte bereits eine Struktur, die für die kleineren Bausteine unverändert gültig ist. Bei einem der größten Bausteine dieses Typs der SPLD-Klasse, dem PAL/ GAL22V10, führt dies zu 12 Eingängen und 10 rückkoppelbaren Ausgängen. Für die Logikmatrix sind dies gemäß Bild 2.6 22 Quellen, jeweils verstärkt und invertiert. Jedes AND-Gatter hat also 44 Eingänge, und im Baustein existieren hiervon 132 Gatter (im Schnitt 12 pro Ausgang plus einiger Sonderfunktionen). Dies bedeutet, dass 5808 Sicherungen zu berechnen und ggf. zu programmieren sind, und die meisten hiervon werden gelöscht.

Bei einer LUT-Struktur (→ Bild 2.7) würde diese Rechnung wesentlich schlimmer ausfallen, denn hier wären für 22 Eingänge 2^{22} Programmstellen (4 194 304) zu setzen oder zu löschen. Dies bedeutet, dass bei SPLDs keine Alternative zur PAL-Architektur der Logikmatrix besteht.

Die Bilder 4.2 a) und b) zeigen den kompletten Aufbau der Logikmatrix im PAL/-GAL22V10. Der Baustein ist im 24-poligen DIL- oder 28-poligen PLCC-Gehäuse erhältlich; in der 24-poligen Version sind neben GND/VCC 22 freie Pins nutzbar: 12 Eingänge und 10 Ein-/Ausgänge.



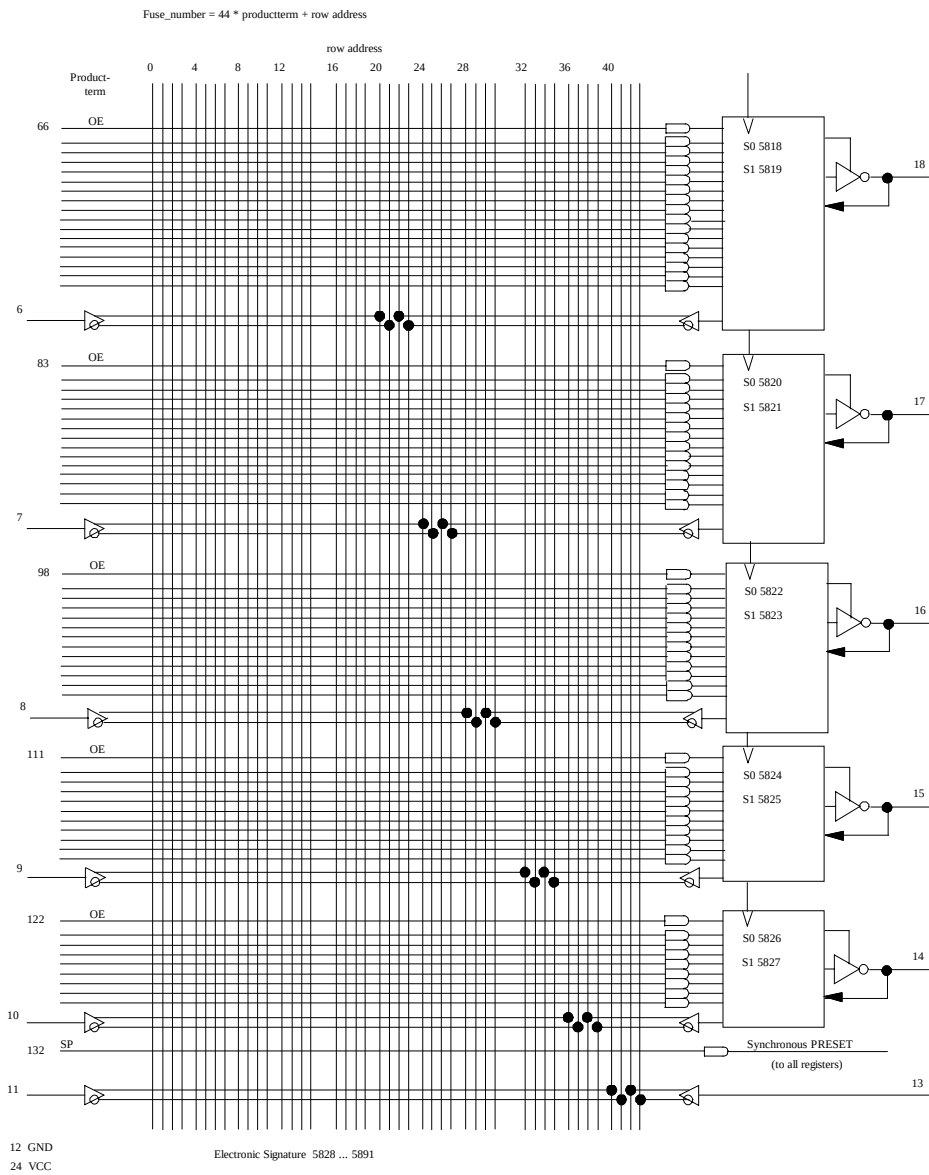


Bild 4.2b Logikmatrix PAL/GAL22V10 (2. Teil)

Pro Ausgangssignal stehen in der Matrix mindestens 8, maximal 16 Terme zur Verfügung. Bild 4.3 zeigt die Output Logic Macro Cell (OLMC), die den Ausgangsblock aus Bild 4.1 darstellt. Jedes der AND-Gatter besitzt 44 Eingänge, die

nach Bild 2.6 programmierbar sind. Die OR-Gatter hingegen haben feste Verbindungen.

Die weitere Verarbeitung – die eigentliche Aufgabe der OLMC – besteht aus Invertierung und Speicherung. Zu diesem Zweck sind die beiden Programmierbits S0 und S1 notwendig, die asynchron/synchron und eine mögliche Invertierung des Signals auswählen. S1 bestimmt zusätzlich die Art des rückgekoppelten Signals: Im synchronen Fall wird das Signal vor dem Ausgangsverstärker abgenommen, im asynchronen hinter dem Puffer.

Die Signale Asynchroner Reset (AR) und Synchroner Preset (SP) werden chipweit für alle Ausgangszellen erzeugt. Hier ist ein Term möglich, der bestimmt, wann das Signal jeweils aktiv wird oder inaktiv bleibt. AR/SP haben natürlich nur bei synchronen Ausgangssignalen Wirkung. Das Signal Takt ist wieder für alle Signale identisch und entspricht dem CLK-Signal (Pin 1).

Letztes Steuersignal ist das Tristatesignal, das die Ausgangstreiber aktiviert. Auch hier steht für jeden Ausgangspin ein individuelles Signal, generierbar durch einen Term, zur Verfügung, so dass Pins auch dynamisch zwischen Ein- und Ausgang umschaltbar sind.

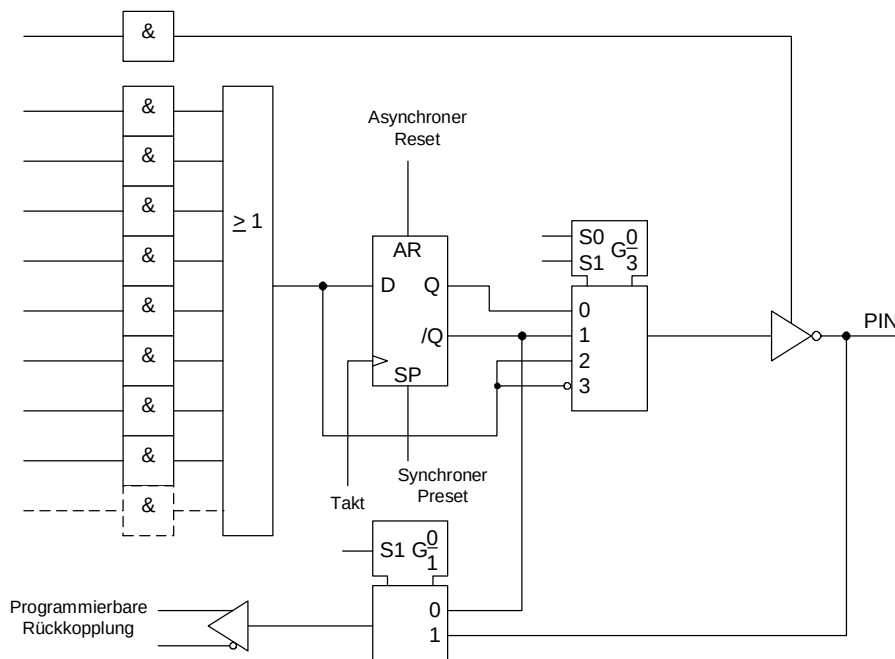


Bild 4.3 Output Logic Macro Cell (OLMC) PAL/GAL22V10

Bezüglich der Auslegung der OLMC lassen sich bei anderen SPLD-Typen weitere Varianten finden, die z.B. die Rückkopplung der Signale, weitere Steuersignale oder die Auswahl des Registertyps, der beim PAL/GAL22V10 auf das D-Flipflop beschränkt ist, finden. Diese Variationen können im Einzelfall helfen, dennoch bleibt die SPLD-Architektur erhalten. Die HDPLDs hingegen stellen eine grundsätzliche Weiterentwicklung der programmierbaren Logik dar.

4.2 Architekturen hochdichter PLDs

Die kleinsten HDPLDs oberhalb der einfachen Bausteine besitzen etwa 32 Ein- und Ausgänge und 4 zusätzlichen Eingänge. Bei gleicher durchschnittlicher Anzahl von 12 AND-Gattern pro Ausgang wären dies für die PAL-Struktur 27648 Sicherungen (ohne Sonderfunktionen), und für die LUT-Struktur von 2^{36} . Diese Architektur hätte eine sehr große Siliziumfläche mit vielen programmierbaren Stellen, aber sehr wenig Nutzungsgrad für den Betrieb zur Folge. Nachteile bei Verlustleistung und auch Geschwindigkeit wären ebenfalls in Kauf zu nehmen.

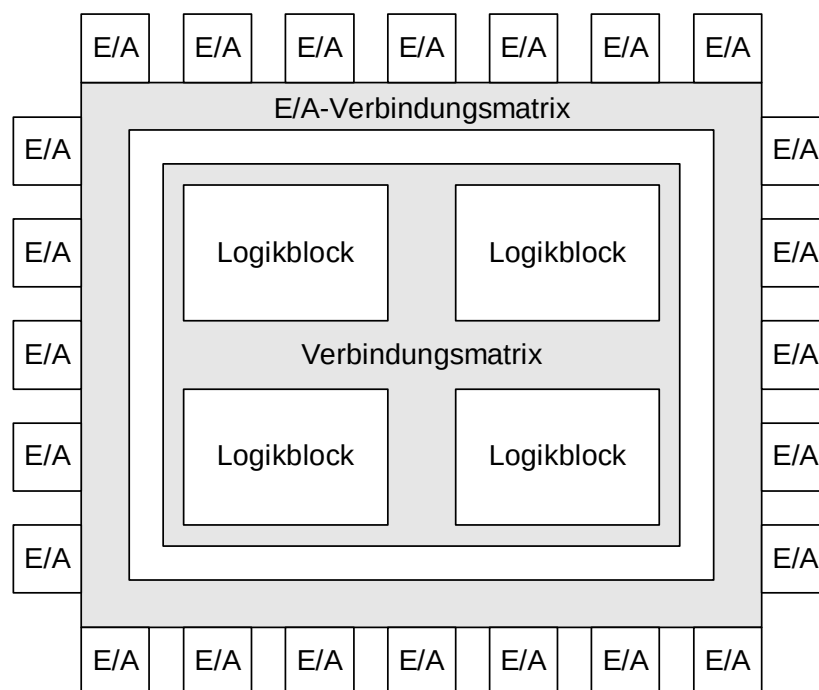


Bild 4.4 Prinzipieller Aufbau von HDPLDs

Die Lösung besteht in einer Mehrschichtigkeit im Baustein. Bei praktischen Problemen stellt sich häufig heraus, dass eine 1-schrittige Lösung (Realisierung in einer DNF) natürlich möglich, aber mit sehr viel Ressourcenverbrauch verbunden ist. Ein 2- oder mehrschrittiger Ansatz erweist sich meist als viel optimaler.

Im Baustein selbst findet man daher mehrere kleinere Teileinheiten, die ihrerseits der PAL- oder LUT-Struktur entsprechen. Diese Teileinheiten sind untereinander verbindbar (auch dies ist programmierbar), und so erhält man durch die Aufspaltung in mehrere Blöcke eine optimale Implementierung – optimal im Sinn von Verlustleistung und Ausnutzbarkeit.

Bild 4.4 zeigt eine schematische Darstellung der neuen Anordnung für HDPLDs (High-Density Programmable Logic Devices). Eine Außenschicht mit Ein- und Ausgangsblöcken und ein Innenleben mit logischen Blöcken, die auch den internen Teil der in Bild 4.1 dargestellten (ehemaligen) E/A-Blöcke enthalten, bilden einen solchen Baustein. Im Übrigen liegt die Mehrschichtigkeit in den in sich geschlossenen Logikblöcken ("Baustein im Baustein") begründet, nicht etwa in der Trennung E/A-Schicht und Logikschicht.

4.3 Konkrete Optimierungsziele

Es bleiben 2 Fragen bislang unbeantwortet: Wie groß soll ein Block denn sein, und welche Verknüpfungsarchitektur soll intern gewählt werden? Die Motivation zur Segmentierung war ja bereits ausreichend dargestellt, sie zielt auf eine Beherrschbarkeit der Komplexität.

Der Preis hierfür ist die erreichbare maximale Geschwindigkeit, in Abhängigkeit von der Applikation. Dies ist anhand der Taktfrequenzen, wie sie für die erhältlichen PLDs spezifiziert sind, bei einem Vergleich SPLD/HDPLD schon nicht mehr sichtbar, denn alle HDPLDs haben die Technologiesprünge der letzten Jahre mitgemacht, die SPLDs eher nicht.

Für eine konkrete Architektur muss zunächst die Blockgröße selbst bestimmt werden. Wenige, dann aber große Blöcke, ein Ansatz eher in Richtung der SPLDs, bedeuten, dass größere Teilfunktionen in einem Block zu integrieren sind. Sie bedeuten aber zugleich, dass bei kleinen Teilfunktion ggf. viele Ressourcen ungenutzt bleiben.

Je feinkörniger die Segmentierung ausfällt, desto weniger Ressourcen werden im Allgemeinen verschwendet werden. Größere Funktionen hingegen müssen auf viele Blöcke verteilt werden, und da dies im Allgemeinen nicht nur nebenläufig, sondern seriell erfolgen muss, steigt damit auch die Laufzeit der Signale.

Die Konsequenzen aus verschiedenen Blockgrößen bezüglich der internen Verbindungen sind leicht abschätzbar, wenn man annimmt, dass ein Baustein nur eine Blockart und -größe besitzt. Ein kleiner Block mit wenigen (ca. 4–9) Eingängen und wenigen (ca. 1–2) Ausgängen bedeutet, dass für eine entsprechende Applika-

tionskapazität viele Blöcke vorhanden sein müssen. Will man nun jeden Block mit jedem anderen mit Hilfe einer einzelnen, unidirektionalen Leitung koppeln, kann man die Anzahl der Verbindungen mit $k * (k - 1)$ bei k Blöcken leicht berechnen. Dies läuft bei 1000 Blöcken schon mal auf knapp 1 000 000 (programmierbare) Verbindungen hinaus.

Bei der Konstellation kleiner Blöcke mit *beliebiger* Verbindbarkeit werden die Einsparungen, die in der Logikmatrix gelungen sind, beim Verbindungsnetzwerk wieder ausgegeben. Dies kann nicht Sinn der Architektur sein, und daher wird die Verbindungsarchitektur variiert werden. Die gängige Lösung besteht in einer Hierarchie von Verbindungstypen: Viele lokale Leitungen (zwischen benachbarten Blöcken), mäßig viele regionale (über mehrere Stationen zu etwas entfernten Blöcken) und wenige globale Leitungen (über den gesamten Baustein) ermöglichen ein universelles Routing.

Die Bausteine, die gemäß dieser Philosophie gefertigt und angeboten werden, werden Field-Programmable Gate Arrays (FPGA) genannt. Der Name stammt aus der ASIC-Technologie, wo mit Gate Array eine bestimmte Klasse von ICs (Gatter mit lokaler Verbindbarkeit) bezeichnet wurde, die nunmehr feldprogrammierbar geworden sind. Diese FPGAs stellen die Klasse der HDPLDs mit der höchsten Gatterdichte dar.

Der andere Ansatz, große Blöcke mit vielen Eingängen (ca. 16–68) und einigen Ausgängen (ca. 4–16), ermöglicht natürlich ein vollständiges Netzwerk, da nunmehr nur wenige Blöcke für die gleiche logische (Brutto-) Kapazität notwendig sind. Man hat quasi nur globale Verbindungen immer gleichen Typs, sodass einige Vereinfachungen in der Übersetzung (durch den Synthesecompiler) schon möglich sind. Leider tendieren große Blöcke dazu, im Durchschnitt schlechter ausgenutzt zu werden, denn alle Ressourcen im Baustein, deren Verbindung gelöscht wurde, sind vorhanden (im Sinne des Platzes und der Verlustleistung), aber ungenutzt (im Sinne der Nettokapazität).

Man kann ein grobes Fazit ziehen: Die Konstellation große Blöcke mit beliebiger (globaler) Verbindbarkeit ist möglich und bietet ein einheitliches, schnelles Timing. Der Nachteil liegt hierbei in der im Durchschnitt geringeren Ausnutzung der Bausteinressourcen.

Die Bausteine dieses Typs haben große Ähnlichkeit zu einer Ansammlung von SPLDs (z.B. PAL/GAL22V10) und werden daher auch Complex Programmable Logic Devices (CPLD) genannt. Die Struktur eines Blocks entspricht sozusagen in etwa einem SPLD, und oftmals wird die Blockgröße auch in der SPLD-Nomenklatur angegeben: 22 steht für die Anzahl der möglichen Eingänge einschließlich aller Ausgangsrückkopplungen, 10 für die Anzahl der möglichen Ausgänge, und der oder die Buchstaben bezeichnen grob die Ausgangsblockstruktur. V steht für variable Polaritäten sowie einem integrierten Register, mit RA werden erweiterte Möglichkeiten beim Register wie asynchroner Reset oder ggf. Wahl des Registertyps gekennzeichnet.

In Abschnitt 4.6 wird versucht, mithilfe einer relativ groben Modellierung zu Aussagen über maximale Flächennutzung bzw. minimale Laufzeiten in Abhängigkeit von der Eingangszahl und des Blocktyps zu kommen.

4.4 PLD-Klassen

Die Einführung in die einzelnen Bestandteile und die Übersicht zu den diversen Architekturansätzen haben es bereits verdeutlicht, dass man zurzeit 3 verschiedene Klassen von PLDs unterscheidet: Simple Programmable Logic Devices, Complex Programmable Logic Devices und Field-Programmable Gate Arrays. Bild 4.5 gibt einen summarischen Überblick zu den PLD-Klassen.

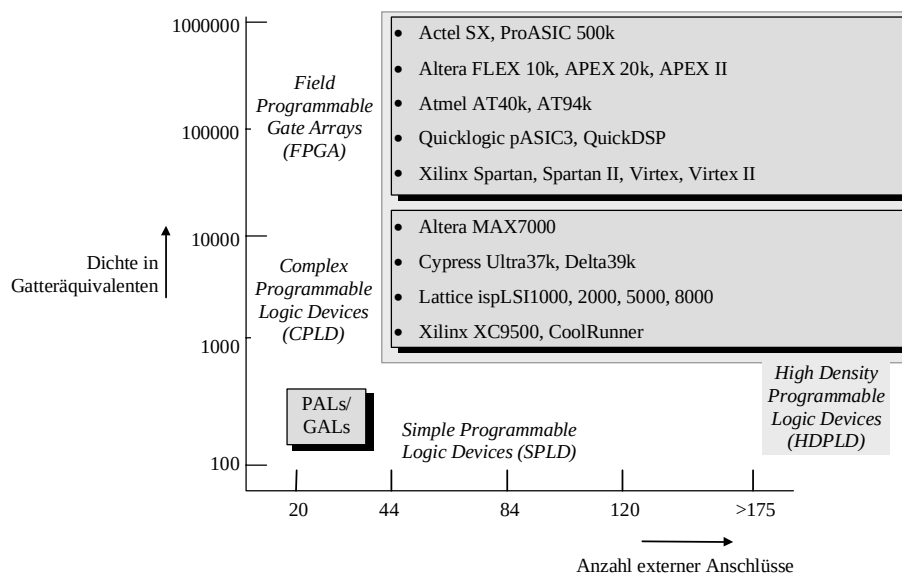


Bild 4.5 Klasseneinteilung der PLDs

Die quantitative Aussage dieser Grafik ist allerdings nicht exakt: Sowohl FPGAs als auch CPLDs reichen in Dichteregionen von 1000 bis über 1 000 000 Gatter. Obendrein muss immer wieder die Frage gestellt werden, wie die Dichte gemessen werden soll. Hier wurden einfach die Bereiche, gemessen in Anzahl der zum gleichen Design notwendigen NAND2-Gatter (logische NAND-Verknüpfung mit 2 Eingängen) angegeben, eine übliche aber etwas fragwürdige Methode. Bild 4.5 dient lediglich zum Überblick über die unterschiedlichen Baustein-Klassen.

4.5 HDPLDs in der Praxis

Bevor eine Darstellung vieler HDPLDs beginnen kann, stellt sich einem natürlich die Frage, ob es Anwendungen gibt, die bestimmte Klassen von Bausteinen oder sogar Bausteintypen selbst bevorzugen, und ob es vielleicht eine Architektur gibt, die für alle Anwendungen optimal ist. Die erste Frage muss klar mit «Ja» beantwortet werden, die zweite ebenso klar mit «Nein».

In der 1. Näherung eignet sich jeder Baustein, d.h. jede Programmiertechnologie und Architektur für jede Applikation. Anfänglich wird man den Unterschied überhaupt nicht merken, ob jetzt ein CPLD oder FPGA eingesetzt wird. Einzig die Programmiertechnologie hinterlässt vielleicht schon einige Einwirkungen, denn SRAM, EEPROM oder Antifuse wirken sich bei der Programmierungsprozedur schon aus.

Für einen ersten Einstieg eignet sich die EEPROM-Technologie in der In-System-Variante als bester Kompromiss. Wird oft umprogrammiert, also bereits im Gebiet des «Reconfigurable Computing» (→ 6) gearbeitet, ist natürlich SRAM zur Speicherung bestens geeignet, während Anti-Fuse wohl nur im professionellen Bereich anzutreffen ist.

Die Einflüsse der Architektur werden erst an Grenzen merklich. Geschwindigkeitsanforderungen oder Designkomplexität können schnell steigen, und genau dann wird die Wahl des Bausteins wichtig: CPLDs für Geschwindigkeit, FPGAs für Dichte, so lautet die vereinfachte Formel. Leider sollte man auch bedenken, dass beliebige Kombinationen zwischen Architektur (CPLD|FPGA) und Technologie (Anti-Fuse|EEPROM|SRAM) nicht möglich sind: EEPROM/CPLDs und {SRAM|Anti-Fuse}/FPGA sind gängige Kombinationen, alles andere ist (fast) nicht erhältlich.

Architektur	Anti-Fuse	EEPROM	SRAM
CPLD	-	ja, häufig in ISP	selten
FPGA	einige	selten	häufig, selten dynamisch partiell reprogrammierbar

Tabelle 4.1 Übersicht Programmiertechnologie/HDPLD-Architekturen

4.6 Ein (einfaches) PLD-Modell

In den vorangegangenen Abschnitten wurde bereits deutlich, dass die beiden Basisarchitekturen CPLD und FPGA unterschiedliche Optimierungsstrategien verfolgen: Geschwindigkeit/Deterministische Bestimmbarkeit versus Flächenausnutzung. Al-

lein diese Unterscheidung ergibt noch keine Aufklärung darüber, wo bei den beiden Architekturen dann optimale Werte erreicht werden würden.

Derartige Aussagen, die z.B. das Geschwindigkeitsverhalten und die effektiv nutzbare Fläche semi-quantitativ beinhalten, erhält man durch Modelle, und ein solches Modell soll hier vorgestellt werden. Die letztendlich gewollte Aussage ist natürlich nicht diejenige, dass die bisherige Wahl der Architektur gut ist, sondern es soll auch versucht werden, neue Wege zu finden.

4.6.1 Field-Programmable Gate Arrays

Eine Logikfunktion kann bei Implementierung in FPGAs in typischerweise zwei Richtungen optimiert werden: Siliziumfläche (Kosten) und Rechengeschwindigkeit. Hieraus lässt sich ableiten, dass man zur Modellierung zwei Fragen stellen muss:

- Jeder Block besitzt eine (bei Herstellung) festgelegte Logikkapazität (für logische Operationen auf Bitlevel). Die Frage ist, wie viele Blöcke einer gewissen Kapazität für eine allgemeine Aufgabe benötigt werden und wie sich dies mit der "Blockgröße" ändert.
- Die zweite Frage ist, wie sich die Geschwindigkeit mit der Blockgröße ändert.

Der hier verfolgte Ansatz geht davon aus, dass man zur Bewältigung einer Aufgabe eine Anzahl $k_{\text{logic},0}$ von Basislogikzellen (theoretisch mit einem Eingang, $N = 1$, was natürlich nicht realisierbar ist). Jede dieser Basislogikzellen benötigt eine Fläche $A_{\text{logic},0}$.

Die Anzahl der benötigten Logikzellen sowie die hierbei verwendete Fläche wird sich mit der Logikzellengröße, durch den wichtigsten Parameter N (Anzahl der Eingänge) beschrieben, ändern. Unter der Annahme, dass die FPGAs auf Look-Up-Tabellen basieren, wird die Fläche mit 2^N wachsend angenommen, während die Anzahl der benötigten Zellen mit $1/N$ abnimmt. Die beiden Modellierungen des Verhaltens sind semi-empirisch gewählt.

Mithilfe eines weiteren Faktors $(1 + k_{\text{internal}} \cdot N)^2$, der die Zunahme der internen Fläche, bedingt durch interne Rückkopplungen und erhöhtem Verdrahtungsaufwand, bei erhöhter Anzahl von Eingängen modellieren soll, ergibt sich die endgültige Modellierung der Fläche der Logikblöcke von FPGAs zu

$$A_{\text{logic}}(N) = A_{\text{logic},0} \cdot 2^N \cdot \frac{k_{\text{logic},0}}{N} \cdot (1 + k_{\text{internal}} \cdot N)^2 \quad (4.1)$$

Das Routing verhält sich etwas anders: Unabhängig von der Blockgröße wird angenommen, dass es zu der Anzahl der Blöcke proportional ist. Da diese Anzahl mit $1/N$ sinkt, muss zwangsläufig auch das Routing kleiner werden.

Diese Annahme dürfte nicht ganz korrekt sein. Zwar wird der Anteil des Routings, also der Verbindungskapazitäten, mit zunehmender Blockgröße sinken, weil ja wesentlich mehr intern verarbeitet wird, andererseits müssen aber mehrere Eingänge

pro Block mit Signalen versorgt werden. Für die Zwecke dieses einfachen Modells ist die Annahme jedoch zulässig und führt zu der Fläche für das Routing:

$$A_{\text{LUT},\text{routing}}(N) = A_{\text{routing},0} \cdot \frac{k_{\text{routing},0}}{N} \quad (4.2)$$

Zusammengefasst ergibt dies die Fläche für ein FPGA, das in der Lage ist, eine gewisse Rechenkapazität zu bieten, zu

$$A_{\text{LUT}}(N) = A_{\text{logic,LUT},0} \cdot 2^N \cdot \frac{k_{\text{logic},0}}{N} \cdot (1 + k_{\text{internal}} \cdot N)^2 + A_{\text{routing,LUT},0} \cdot \frac{k_{\text{routing},0}}{N} \quad (4.3)$$

Kostensensitive Applikationen werden also dadurch unterstützt, dass man diese Funktion minimiert. Quantitativ ist dies natürlich nur möglich, wenn für die Konstanten realistische Werte eingesetzt werden.

Andererseits ist die mittlere Verzögerungszeit oder – etwas besser verwendbar – der Durchsatz D von großem Interesse, auch hier wieder in Abhängigkeit von N . Die Verzögerungszeit $T_{\text{Cell},0}$ für eine Logikzelle ist dabei proportional zur Kantenlänge dieser Zelle, also zur Wurzel aus der Zellenfläche. Die Verzögerungszeit im Routing $T_{\text{routing},0}$ lässt sich hingegen dadurch abschätzen, dass man annimmt, die lokale (Nearest Neighbour) Verbindungen seien dominant. In diesem Fall bleiben die Werte unabhängig von der Blockgröße konstant, allerdings nimmt die Anzahl der Verzögerungen ab.

Der Durchsatz kann dann durch die Eingangsblockgröße N (weil die Anzahl der Blöcke mit $1/N$ abnimmt) und die Summe der beiden mittleren Delayzeiten beschrieben werden:

$$D_{\text{LUT}} = \frac{N}{T_{\text{Cell},0} \cdot 2^{N/2} + T_{\text{routing},0}} \quad (4.4)$$

Beide Gleichungen, (4.3) und (4.4), zeigen den Einfluss von N auf die reale Siliziumfläche und den Durchsatz einer LUT-basierten Architektur und berücksichtigen sowohl die Zell- als auch die Routingfläche. Bild 4.6 stellt das Ergebnis für beide Funktionen sowie das (ungewichtete) Produkt dar, basierend auf der Annahme, dass sich Zellenfläche und Routing im Verhältnis 1:4 verhalten. Diese Annahme wird durch die realen Produkte gestützt und repräsentiert das Verhältnis von $A \cdot k$, jeweils bezogen auf die Zelle und das Routing. Simulationen für ein Verhältnis 1:1 zeigen allerdings auch keine signifikanten Änderungen.

Die Darstellung in Bild 4.6 zeigt bei genauer Analyse, dass das Maximum für die Flächennutzung bereits bei $N = 2$ erreicht wird, während der Durchsatz für $N = 4 \dots 6$ maximal wird. Dies bedeutet, dass aufgrund des exponentiellen Wachstums die Flächennutzung möglichst kleine Logikblöcke bevorzugt, während bei der Geschwindigkeit durch größere Blöcke unterstützt wird. Das Verhältnis Durchsatz/Fläche wird für 3-Eingänge-1-Ausgang-Strukturen maximal, was guten Durchsatz bei moderaten Kosten bedeutet.

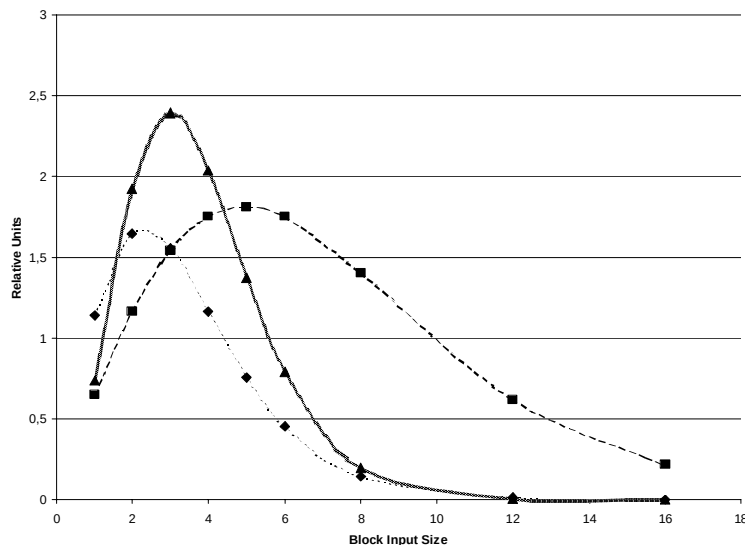


Bild 4.6 Durchsatz und Flächennutzung in LUT-basierten PLDs (FPGAs)

4.6.2 Complex Programmable Logic Devices

Die wesentliche, zumindest sichtbarste Eigenschaft der 'feingranularen' FPGAs ist die Look-Up-Table Struktur. Dies hat zur Konsequenz, dass jede Funktion mit einer maximalen Eingangszahl N realisierbar ist, andererseits aber auch ein exponentielles Wachstum der Fläche mit N . Aus diesem Grund bleibt N auf kleine Werte bis maximal 5 limitiert.

Ein anderer Fall liegt bei den 'grobgranularen' CPLDs vor, die mehrheitlich eine DNF-Struktur mit programmierbaren AND-Gattern sowie fixierten OR-Gatter (Terme) nutzt. Die Modellierung dieser Struktur erfordert andere Ansätze, weil einerseits nicht alle Funktionen darstellbar sind, andererseits aber auch das Flächenwachstum wesentlich kleiner ausfällt.

Gleichung (4.5) zeigt ein Modell für die Nutzung der Siliziumfläche innerhalb der CPLD-Struktur. In erster Näherung wächst die Anzahl der programmierbaren AND-Verbindungen (Eingänge am AND-Gatter) linear mit der Anzahl der Eingänge N , weil jeder Eingang in gepufferter sowie in invertierter Form benötigt wird.

Die Anzahl der notwendigen Zellen wird auch hier mit $1/N$ angenommen, da Zellen mit mehr Eingängen auch komplexere Funktionen aufnehmen können. Diese wachsende Parallelität bedeutet in Kombination mit dem linearen Flächenwachstum, dass in erster Näherung kein Flächeneffekt mit verändertem N erwartet wird.

$$A_{\text{PAL}}(N) = A_{\text{logic, PAL},0} \cdot k_{\text{logic},0} \cdot (1 + k_{\text{internal}} \cdot N)^2 + \frac{A_{\text{routing, PAL},0} \cdot k_{\text{routing, PAL},0}}{N} \quad (4.5)$$

In zweiter Näherung wird die Zellengröße jedoch mit $(1 + k_{\text{internal}} \cdot N)^2$ wachsen. Dieser Faktor modelliert die Block-internen Rückkopplungen, die in CPLDs mit mehr als einem Ausgang pro Block wertvoll sind und oft implementiert werden.

Die Fläche, die im Routing des Bausteins benötigt wird, bleibt – auf die Fläche eines Logikblocks bezogen – konstant. Der Wert wird von der Anzahl der Ausgänge pro Logikblock abhängen, und diese Anzahl wird in dem Modell als konstant angenommen (nur die Anzahl N der Eingänge pro Block ändert sich). Die Gesamt-routingfläche sinkt mit steigendem N , da die Zahl der Blöcke sinkt (modelliert mit $1/N$).

Gl. (4.5) modelliert die Gesamtfläche einer PAL-Struktur, unabhängig von der durchschnittlichen Nutzung (entsprechendes gilt auch für Gl. (4.3) in der Modellierung der FPGAs). Grundsätzlich gilt jedoch, dass die Wahrscheinlichkeit für eine Nichtnutzung mit der Anzahl der integrierten Ressourcen wächst – dies ist insbesondere im Bereich der CPLDs bekannt.

Der daraus entstehende Effekt lässt sich durch einen größeren Flächenbedarf modellieren. Die Verschwendung von Ressourcen betrifft insbesondere die Logikblöcke, z.B. durch konkrete Verwendung von weniger Eingängen als vorhanden. Die Flächenvergrößerung entsteht dann dadurch, dass gegenüber der rechnerischen Anzahl an Ein- und Ausgängen mehr Blöcke benötigt werden, weil die Ressourcen nicht beliebig nutzbar sind.

Dieser Effekt wird durch einen zusätzlichen Faktor $(1 + k_{\text{eff}} \cdot N)$, wobei der Faktor k_{eff} die Wahrscheinlichkeit von ungenutzten Teilen innerhalb eines Logikblocks beschreibt. Dies mündet in Gl. (4.6):

$$A_{\text{PAL,eff.}}(N) = A_{\text{logic, PAL},0} \cdot k_{\text{logic},0} \cdot (1 + k_{\text{internal}} \cdot N)^2 \cdot (1 + k_{\text{eff}} \cdot N) + \frac{A_{\text{routing, PAL},0} \cdot k_{\text{routing, PAL},0}}{N} \quad (4.6)$$

Für k_{eff} werden relativ große Werte (in Bild 4.7: $k_{\text{eff}} = 0,05$) angenommen. Dies steht im Einklang mit der Beobachtung, dass ca. 80% aller Signalfunktionen ca. 3 – 5 Termen (AND-Verknüpfungen) besitzen.

Der zweite interessante Wert betrifft den Durchsatz D . Wie bereits zuvor für die Routingfläche beschrieben wird die Verzögerungszeit pro Block in erster Näherung unabhängig von der wachsenden Eingangszahl N sein, andererseits wird die Anzahl der durchlaufenen Zellen und Routingverbindungen (bei CPLDs existiert nur eine Sorte (globales) Routing) mit etwa $1/N$ abnehmen. Dies mündet in das Durchsatzmodell

$$D_{PAL} = \frac{N}{T_{cell, PAL, 0} + T_{routing, PAL, 0}} \quad (4.7)$$

und damit linear mit N ansteigenden Durchsatz. Dieses Modell kann für große N nicht korrekt sein, es muss zwangsläufig entgegenwirkende Effekte geben. Hierzu zählen, dass

- für große Blöcke ein zusätzlicher Wachstumsanteil für die internen Rückkopplungen (siehe Gl. (4.5), $k_{internal}$) berechnet werden muss, der in einer vergrößerten Verzögerungszeit für die Zelle mündet, und dass
- für wachsende Blöcke auch der mittlere Abstand zwischen zwei Blöcken und damit die Verzögerungszeit im Routing wächst.

Beide Effekte sind proportional zu $(1 + k_{internal} \cdot N)$ einbezogen, im ersten Fall direkt einsichtig, im zweiten Fall dadurch, dass der mittlere wie maximale Abstand bei einem Kreis oder Quadrat proportional zum Umfang ist. Hieraus erhält man

$$D_{PAL} = \frac{N}{(T_{cell, PAL, 0} + T_{routing, PAL, 0}) \cdot (1 + k_{internal} \cdot N)} \quad (4.8)$$

für den Durchsatz, abhängig von der Eingangszahl N . Für große N strebt diese Funktion einem konstanten Grenzwert $1/((T_{cell, PAL, 0} + T_{Routing, PAL, 0}) \cdot k_{internal})$ entgegen. Der Unterschied in diesen beiden Modellen ist für die gegebenen Werte ($k_{internal} = 0,03$) recht gering, wie Bild 4.7 zeigt.

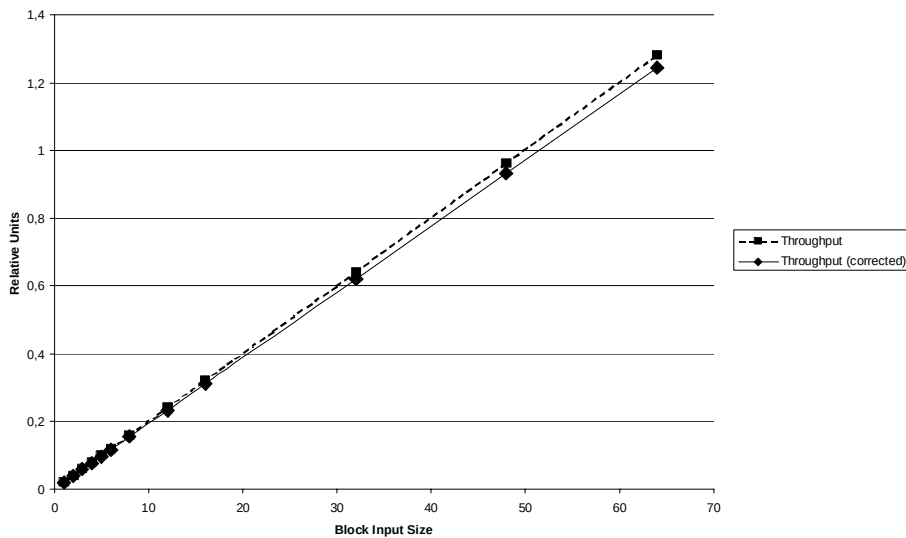


Bild 3.7 Vergleich Durchsatz CPLD mit und ohne Korrektur

Dementsprechend unterscheiden sich die Darstellungen für die Modelle (4.6) und (4.7), nun auf die Flächennutzung bezogen, ebenfalls kaum, so dass in Bild 4.8 nur die korrigierten Werte für CPLDs dargestellt sind. Die CPLD-Modelle zeigen einen relativ breiten Bereich, in dem sie optimal arbeiten können: Eingangsweiten von 8 bis 64 sind offenbar realistische Werte, die man auch im marktgängigen Angebot findet (zurzeit 18-64). Die Darstellungen für die effektive Nutzung (siehe Gl. (4.6), mit Korrekturfaktor $(1 + k_{\text{eff}} * N)$) zeigen an, dass man im Allgemeinen zu kleineren Eingangsweiten tendieren sollte – im speziellen Fall einer optimalen Abstimmung zwischen Applikation und Eingangsweite stimmt dies nicht.

Last-not-least sollte erwähnt werden, dass die tatsächliche Nutzung natürlich stark von der Fittersoftware abhängt.

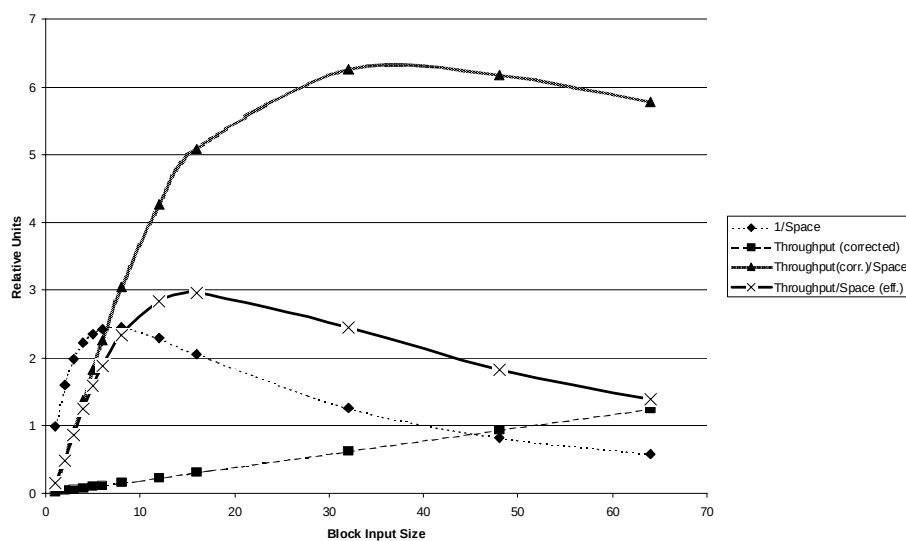


Bild 4.8 Durchsatz und Flächennutzung in PAL-basierten PLDs (CPLDs)

4.7 XC6200 als Forschungsarchitektur

Literatur: [21] www.xilinx.com

Die Firma Xilinx, seit Beginn der FPGAs (1985) federführend in der Entwicklung solcher Bausteine, führte Mitte der 90er Jahre eine Architektur am Markt ein, die mit XC6200 bezeichnet wurde. Seitens der Vermarktung war dies ein Fehlschlag, andererseits konnte man viel an dieser Architektur lernen (insbesondere auch Schwächen im Routing). Diese Architektur wird in der Forschung gerne als Referenz

renz gewählt, da sie die wohl am Besten dokumentierte industriell gefertigte Architektur darstellt.

Die XC6200-Architektur ist hierarchisch strukturiert: Eine Matrix von 4×4 einfachen Logikzellen wird zu einem Block mit entsprechenden Routingressourcen zusammengefasst, 4×4 davon zu einem 16×16 -Cell-Block. Die verfügbaren Bausteine setzen sich dann aus 3×3 (XC6209), 4×4 (XC6216), 6×6 (XC6236) bzw. 8×8 (XC6264) solcher 16×16 -Cell-Blocks zusammen. Eine Beschreibung der Bausteine erfolgt dementsprechend in der Reihenfolge der Logikzelle, der Routingstrukturen an der Logikzelle und im 4×4 -Block und der Baustein-übergreifenden Routingstruktur.

Den Abschluss dieses Abschnitts bilden dann Abschätzungen des Transistoraufwands. Hier soll gezeigt werden, in welcher Größenordnung Logik- und Routingstrukturen jeweils in Bezug auf den Transistoraufwand liegen.

4.7.1 Verwendete Multiplexerstrukturen

Innerhalb der Mikroarchitektur des XC6200 (wie auch in anderen FPGAs werden vielfach Multiplexerstrukturen genutzt, deren Transistorbudget für den Aufwand insgesamt verantwortlich ist. Aus diesem Grund werden diese Strukturen hier kurz dargestellt.

Bild 4.9 zeigt einen 4:1-Multiplexer, als Look-Up Table nutzbar. In diesem Fall sind die Steuereingänge (E1/E0) die Eingänge der LUT, während die Dateneingänge mit Speicherzellen für den Inhalt der LUT versehen sind. Auf diese Weise lässt sich eine beliebige Boolesche Funktion der Eingänge E1/E0 speichern.

Der jeweils adressierte Inhalt einer Zelle wird durch das Decodiernetz mit Hilfe von Pass-Transistoren (NMOS) selektiert. Diese Selektierung bedeutet zugleich, dass hier viele Transistoren verwendet werden und die Laufzeit (kritischer Pfad) bestimmt wird.

Die Aufwand an Transistoren ist entsprechend der LUT-Funktionalität hoch. Im allgemeinen Fall eines 2^k :1-Multiplexers, der einer LUT mit k Eingängen entspricht, gilt, dass die Decodierung ($2^{k+1}-2$) Transistoren zuzüglich $2 * k$ Transistoren für die Invertierung enthält (ein Inverter in CMOS-Technologie benötigt ein Transistorpaar) und zur Speicherung ($2^k * m$) Transistoren, m als Anzahl der Transistoren pro Speicherzelle (SRAM: $m = 5$) benötigt werden.

Bild 4.10 zeigt eine andere Struktur, einen dynamischen 2:1-Multiplexer. Hier sind ein Steuereingang und zwei Dateneingänge so verschaltet, dass Laufzeit-variable einer der beiden Eingänge (mit wechselnden Werten) ausgewählt wird. Das Transistorbudget ist für einen 2^k :1-Multiplexer ($2^{k+1}+2*k-2$) Transistoren.

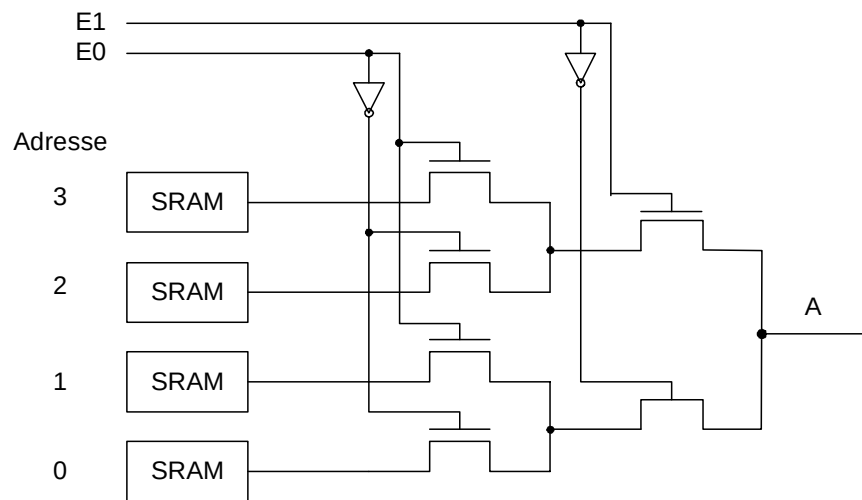


Bild 4.9 4:1-Multiplexer als Look-Up Table

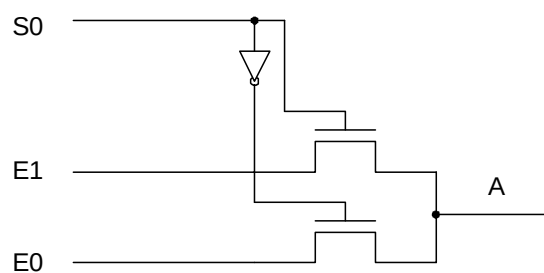


Bild 4.10 Dynamischer 2:1-Multiplexer

Eng damit verwandt ist ein serieller Multiplexer im Routing (Bild 4.11). Hier werden die Informationen Laufzeit-statisch in Speicherzellen hinterlegt, die dann nur einen Weg aktiv schalten.

Eine Eingangsinformation an E_x muss in diesem Fall $\log_2(\text{Eingangsbreite})$ Transistoren durchlaufen (kritischer Pfad). Das Transistorbudget beträgt für einen $2^k:1$ -Multiplexer $(2^{k+1} + 2 * k - 2)$ Transistoren zuzüglich $(k * m)$ für die Speicherung der Informationen.

Der Multiplexer im Routing kann auch in einer parallelen Variante implementiert werden. Bild 4.12 zeigt den Aufbau: Die Decodierung, welche Leitung (Laufzeit-statisch) aktiv durchgeschaltet wird, erfolgt zur Compilezeit und führt zu mehr Konfigurationsbits, von denen nur eines aktiv (= '1') sein darf.

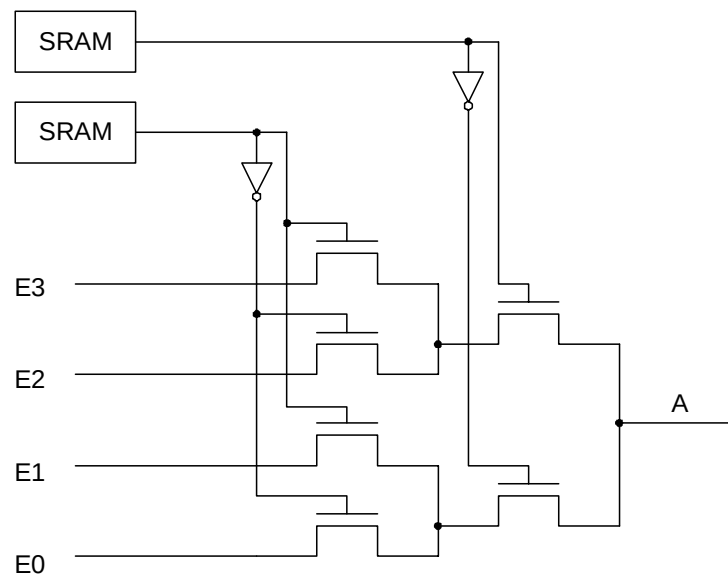


Bild 4.11 4:1-Multiplexer für Routing, serielle Variante

Der Transistoraufwand kann natürlich wesentlich größer in diesem Fall sein, verglichen mit dem seriellen Multiplexer. Bei einem 2^k :1-Multiplexer werden 2^k Transistoren für die Auswahl zuzüglich $(2^k * m)$ Transistoren für die Speicherung der Informationen benötigt.

Der Vorteil dieser Variante besteht natürlich darin, dass der kritische Pfad innerhalb des Multiplexers nur einen Transistor lang ist, unabhängig von der Eingangsbreite, während beim seriellen Multiplexer der Pfadlänge mit $\log_2(\text{Eingangszahl})$ wächst.

Für kleine Eingangsbreiten, wie sie z.B. in den Logikzellen von FPGAs, ist der Unterschied zwischen serieller und paralleler Variante eher gering, je nach Technologie m kann die parallele Variante sogar günstiger sein. Für den Einsatz im Routing sind die Eingangsbreiten meist größer, so dass ein Kompromiss zwischen Flächenbedarf und Geschwindigkeit getroffen werden muss.

Tabelle 4.2 fasst den Bedarf an Transistoren zusammen.

Für die Technologiekonstante m gelten folgende Werte: SRAM: $m = 5$, EEPROM/EPROM: $m = 0$ (weil dieser Speicher in den Pass-Transistor integriert sein kann).

In den weiteren Schätzungen zum Transistoraufwand wird die SRAM-Technologie zur Speicherung angenommen, ferner serielle Multiplexer anstelle von parallelen Varianten. Die Transistorzahlen bleiben natürlich Schätzungen mit einer gewissen Ungenauigkeit, sie sollen der Darstellung der Größenordnungen dienen.

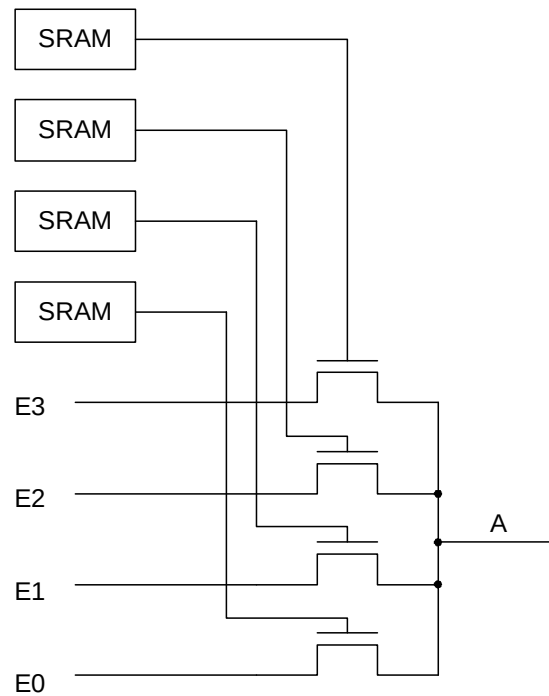


Bild 4.12 4:1-Multiplexer im Routing, parallele Variante

Typ	Anzahl Signaleingänge	Anzahl Pass-Transistoren	Anzahl Speichertransistoren
Look-Up Table (k)	k	$2^{k+1} + 2 * k - 2$	$2^k * m$
Dynamischer Multiplexer ($2^k:1$)	2^k	$2^{k+1} + 2 * k - 2$	-
Sequenzieller Multiplexer ($2^k:1$)	2^k	$2^{k+1} + 2 * k - 2$	$k * m$
Paralleler Multiplexer ($2^k:1$)	2^k	2^k	$2^k * m$

Tabelle 4.2 Übersicht zu Multiplexern und Transistoraufwand

4.7.2 XC6200 Zellenstruktur

Bild 4.13 zeigt die Zellenstruktur der XC6200-Familie, auf die reine Logik einschließlich der Datenspeicherung beschränkt. Die Zelle stellt eine 3-Eingangs-zu-1-Ausgang-Struktur (3E1A) dar, wobei allerdings nicht alle logischen Funktionen implementierbar sind.

Eine Besonderheit liegt in dem Register Protection. Hier kann die Zelle so geschaltet werden, dass der zuletzt gespeicherte Wert im D-Flipflop immer erhalten bleibt. Da dieser Wert beim Laden der Konfiguration ebenfalls gesetzt werden kann (als Startwert für den Betrieb), erhält man über die Register Protection eine bequeme Möglichkeit zur Speicherung von Konstanten.

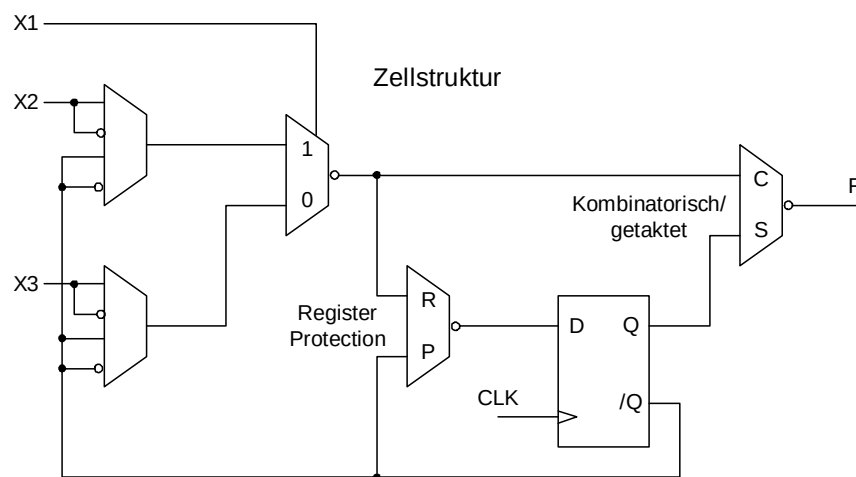


Bild 4.13 Logikzellenstruktur XC6200-Familie (Xilinx)

Das Transistorbudget einer solchen Zelle lässt sich wie folgt ermitteln:

- Die Multiplexer bei den Eingängen X2 und X3 sind vom Typ 4:1-Multiplexer ($k = 2$). Bei angenommener sequenzieller Bauweise beträgt die Anzahl der Pass-Transistoren jeweils 10, zuzüglich 4 für die beiden Eingangsinvertierungen. Für die Speicherung sind $4 * m$ Transistoren notwendig.
- Der von X1 gesteuerte dynamische Multiplexer benötigt insgesamt 4 Transistoren, von der Ausgangsinvertierung wird abgesehen.
- Register Protection und die Auswahl sequenziell/kombinatorisch stellen beide jeweils einen 2:1-Multiplexer dar. Der Aufwand beträgt jeweils 4 Transistoren zuzüglich m Transistoren für die Konfigurationsspeicherung, die Ausgangsinvertierung bleibt unberücksichtigt.

- Das Taktflanken-gesteuerte Register benötigt ca. 10 Transistoren (für ein Taktzustands-gesteuertes Flipflop wären 5 Transistoren notwendig).

Summarisch lässt damit der Transistoraufwand pro XC6200-Logikzelle mit 50 Transistoren und $6 * m$ Speichertransistoren beschreiben. Für SRAM ($m = 5$) sind dies zusammen 80 Transistoren.

Eine variierte Struktur der Logikzelle ist in Bild 4.14 dargestellt: Anstelle der verschiedenen Multiplexer wurde eine Look-Up Table eingefügt, die einen vollständigen Designraum aller Funktionen mit 3 Eingängen und einer Rückkopplung ermöglicht.

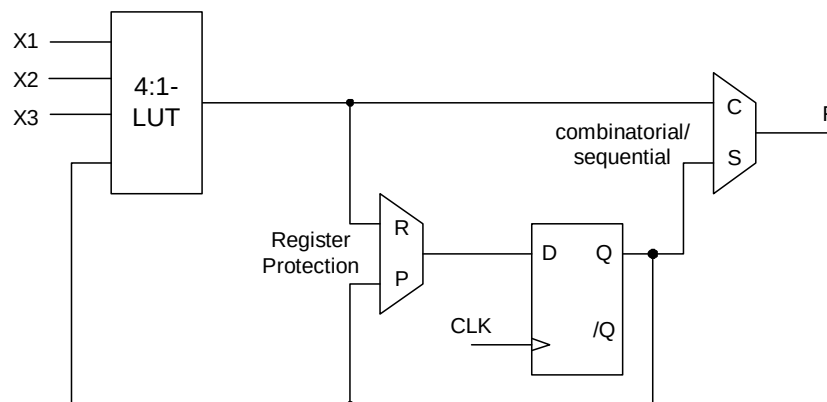


Bild 4.14 Modifizierte XC6200-Logikzellenstruktur

Die Look-Up Table benötigt 38 Pass-Transistoren sowie $16 * m$ Speichertransistoren, dafür entfallen 32 Pass-Transistoren und $4 * m$ Speichertransistoren. Für SRAM-Technologie ergibt sich hieraus ein Gesamtbedarf von 146 Transistoren pro Logikzelle.

4.7.3 Lokale Routingstrukturen in der Zelle

Bild 4.15 zeigt das Routing, das in dem elementaren Logikblock, auch als Basiszelle bezeichnet, vorhanden ist. Dieses Routing dient der Auswahl der Eingänge X1 bis X3 sowie dem Ausgangsrouting.

- Das Eingangsrouting wählt für jeden Eingang gesondert die Datenquelle aus. Möglich sind die 4 unmittelbaren Nachbarn (N, S, E, W) sowie die Routingleitungen N4, S4, E4 und W4, die blockübergreifend (4*4-Block) Signale routen können.

Jeder der Eingangsmultiplexer wählt damit aus 8 Quellen auf, so dass der Transistorbedarf (ohne Ausgangsinvertierung) 20 Pass-Transistoren sowie $3 * m$

Speichertransistoren beträgt. Für SRAM-Technologie ergibt dies 105 Transistoren für alle Eingangsmultiplexer.

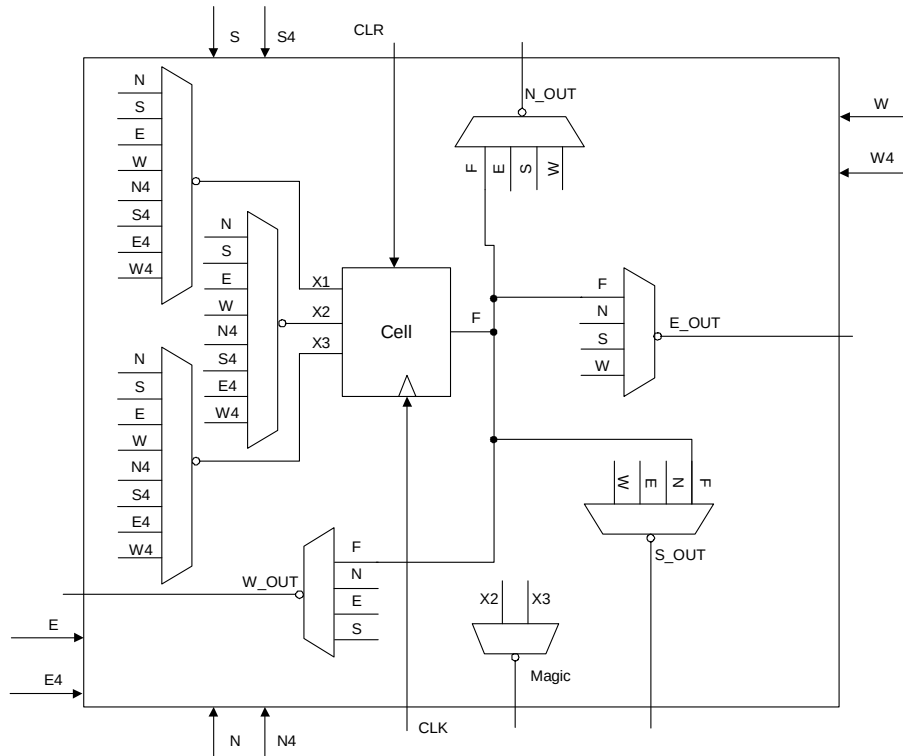


Bild 4.15 XC6200 Basiszellen-Routing

- Die Ausgangsmultiplexer bestimmen das lokale Routing, indem jeweils das Signale zu den unmittelbaren Nachbarn festgelegt wird. Sie besitzen jeweils 10 Pass-Transistoren und $2 \cdot m$ Speichertransistoren, zusammen 80 Transistoren bei SRAM-Technologie.
- Mit Hilfe der Magic-Line kann pro Zelle einer der Eingänge X2 oder X3 in das block- und bausteinübergreifende Routing überführt werden. Die genauen Details werden im nächsten Abschnitt gezeigt, der Aufwand pro Zelle beträgt 4 Pass- und m Speichertransistoren.

Insgesamt beträgt das Transistorbudget für das Zell-basierte Routing 104 Pass-Transistoren und $18 \cdot m$ Speichertransistoren, insgesamt also 194 Transistoren (bei SRAM). Dies bedeutet, dass bereits beim lokalen Routing ein erheblicher Mehraufwand im Routing festzustellen ist.

4.7.4 XC6200 Routingstruktur

Entsprechend dem hierarchischen Aufbau der XC6200-Architektur sind die Routingressourcen oberhalb der Basiszelle in verschiedenen Hierarchieebenen aufgebaut:

- 4*4 Basiszellen sind zu einem Block mit eigenem Routing sowie Übergängen zwischen globalen und lokalen und zwischen den Blockgrenzen ausgestattet. Auf dieser Ebene können auch Taktsignale zugeordnet werden.
- 4*4 Blöcken bzw. 16*16 Basiszellen sind zu einer weiteren Hierarchiestufe zusammengefasst. Auf dieser Blockebene existieren wiederum eigene Routingressourcen, hier werden die Clear-Signale für die Register verteilt.
- Eine Anzahl dieser 16*16-Blöcke (beim XC6216: 4*4) sind dann zum Baustein mit I/O-Blöcken und einem globalen Routing zusammengefasst.

4.7.4.1 4*4-Blöcke im XC6200

4*4 Basiszellen sind zu einem Block zusammengefasst (siehe auch Bild 4.15). Während die Zellen untereinander nur durch 'Nearest-Neighbour'-Verbindungen miteinander verbunden sind, sind im Blockrouting auch regionale und vor allem globale Leitungen vorgesehen. Bild 4.15 zeigt zunächst, dass je 2 Magic Lines – pro Basiszelle sind es 2 in verschiedene Richtungen – pro Switchelement im Blockrouting enden.

In jedem Switchelement sind zwei Multiplexer vorhanden: Ein 10:1-Multiplexer bestimmt die Leitung X4 (X = N, S, E, W), die blockübergreifend 4 Zellen erreicht, ein weiterer 8:1-Multiplexer bestimmt die lokalen Eingänge der nächstliegenden Basiszelle (der ja ein Nachbar fehlt). Am Beispiel des SSwitch soll dies erläutert werden:

- 1 SOut aus dem lokal anliegenden Block, 1 NIn aus dem lokal anliegenden, zum benachbarten 4*4-Block gehörenden Block.
- 2 Magic Lines (MS, MSA) aus dem gleichen 4*4-Block, eine Magic Line (MN, ohne 'A') aus dem Nachbarblock.
- Die S4-Leitung aus dem Nachbarblock (4*4).
- Die S16-Leitung aus dem aktuellen 16*16-Block, die N16-Leitung aus dem benachbarten 16*16-Block.
- 2 Chip-globale Leitungen, NCL und SCL (CL = Chip Length)

Diese Routingressourcen sind für die S4-, E4- und W4-Switches identisch, bei N4 kommen noch weitere hinzu. Zudem besitzt der N4-Switch eine Kaskade von 2 4:1-Multiplexern, um den spezifischen Takt für den 4*4-Block auszuwählen.

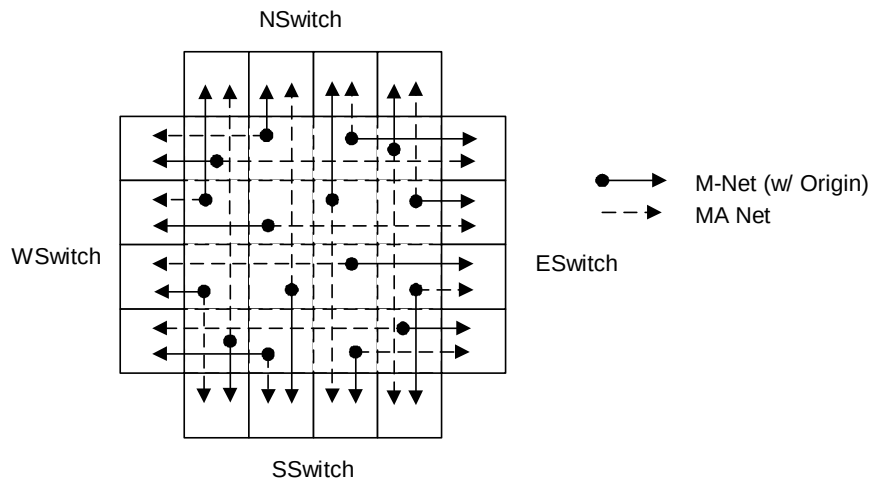


Bild 4.16 XC6200 Routing an den Blockgrenzen

Der zweite Multiplexer (8:1) nimmt die Funktionen wahr, die in dem Block-basierten Routing von der Basiszelle selbst bestimmt werden. Hier werden zusätzlich die lokalen Ausgänge generiert, die im benachbarten 4*4-Block geroutet werden. Dieser Multiplexer bleibt in der nachfolgenden Schätzung unberücksichtigt.

Die Transistorschätzung geht pro 4*4-Block von 16 16:1-Multiplexern und 8 4:1-Multiplexern (zur Taktgenerierung) aus. Hieraus folgen 38 Pass-Transistoren, sowie $4*m$ Speichertransistoren pro 16:1-Multiplexer und $10 + 2*m$ Transistoren für die 4:1 Transistoren.

4.7.4.2 16*16-Blöcke

Für einen 16*16-Block stehen weitere Routingressourcen zur Verfügung. Je Richtung existieren pro 4*4 Block 4 Leitungen (FastLaneTM), die 16 Basiszellen überstreichen und bereits beim 4*4-Blockrouting erwähnt wurden. Diese Leitungen werden durch Multiplexer am Rand des Blocks getrieben, Quellen können hier die Chip-globalen Leitungen und lokale Daten sein.

Insgesamt ergibt dies pro 16*16-Block 64 Multiplexer á 8:1 mit einem Transistoraufwand von $20 + 3*m$ Transistoren. Hinzu kommen 16 8:1-Multiplexer zur Auswahl der Clear-Signals, das 16*16-Block-lokal ist. Diese Multiplexer haben ebenfalls einen Aufwand von $20 + 3*m$ Transistoren.

4.7.4.3 Baustein-globales Routing

Pro 16*16-Block existieren wiederum pro Richtung 16 Chip-weite Leitungen, deren Quelldaten aus den I/O-Puffern der Bausteine stammen. Während in diesen I/O-Puffern auch lokale Multiplexer als Ersatz für die fehlenden Arrayelemente zu finden sind (und daher nicht mitgezählt werden, kommt jeweils ein neuer 8:1-Multiplexer hinzu, der die Chip-weite Leitung treibt. Der Aufwand hierfür beträgt $20 + 3 \cdot m$ Transistoren.

4.7.5 Aufwand zur Programmierung

Die Programmierung erfolgt über 8-bit breite Schreibzugriffe. Hierfür müssen 4096 Logikblöcke (12 bit) und in jedem 4 verschiedene Bereiche beschrieben werden, insgesamt also eine 14 bit Adressierung.

Diese Adressierung benötigt somit viele Datenleitungen und einen Selektierungsmechanismus an jeder Einheit (bzw. zentral), der aus den 14 bit Adressen in einem zweistufigen Verfahren den Zugriff selektiert. Jede 12-bit-Selektierung benötigt ein AND-Gatter (mit 12 Eingängen, ggf. invertiert), so dass der Aufwand pro Zelle 24 Transistoren und 4 weitere (für die interne Selektierung) beträgt.

Dies wird als Aufwand für die Programmierbarkeit geschätzt. Die Leitungsführungen bleiben unberücksichtigt, stellen aber einen erheblichen Flächenfaktor dar.

4.7.6 Transistorschätzungen

Tabelle 4.3 fasst die Transistorschätzungen in einem XC6216 zusammen. Diese Schätzungen beinhalten keinerlei Treibertransistoren, wie sie typischerweise zur Verstärkung und zur Unterdrückung von Skewzeiten (= Versatz der Zeiten an unterschiedlichen Orten im Chip) bei globalen Signalen eingesetzt werden. Das Ziel dieser Schätzungen liegt darin, einen ungefähren Überblick über die tatsächlichen Verhältnisse zu geben.

Das Verhältnis von Transistoren im Logikblock zu Transistoren im Routing liegt bei ca. 1:3 bis 1:4. Dies wird immer wieder bestätigt: Das programmierbare Routing nimmt sehr viel Kapazitäten in Anspruch.

Weiterhin gibt der Hersteller Xilinx an, dass in einem XC6216 16000 – 24000 Systemgatter integriert (programmiert) werden können. Diese Zahl ist so zu interpretieren, dass ein Chip, bestehend aus einem Meer von Systemgatter (Sea of Gates) (ein NAND-Gatter mit 2 Eingängen), als ASIC verwendet wird und die gleiche Schaltung ergibt. Je nach Schätzvarianten sind die Systemgatterangaben passend oder zu hoch (theoretisches Maximum).

20000 Systemgatter lassen sich in reiner CMOS-Technologie durch 80000 Transistoren darstellen. Dies ergibt dann das Verhältnis von 1:20 (1:23) zwischen notwendigen und tatsächlichen Transistoren in einem XC6216.

	Basiszelle Logik	Basiszelle Routing	4*4-Block- routing	16*16-Block- routing	Baustein- routing	Gesamt (XC6216)	Gesamt (S6)
Pass-Transistoren pro Einheit (durch- schnittlich)	50 (56)	104	43	24	20	836608	861184
Speichertransisto- ren pro Einheit (durchschnittlich)	6*m (18*m)	18*m	5*m	3,75*m	3*m	123392*m	172544*m
Anzahl Einheiten im XC6216	4096	4096	4096	1024	256		
Programmierung	114688					114688	114688
Transistoren im XC6216 (SRAM)	327680 (598016)	794624	278528	43776	8960	1568256	1838592
Proz. Anteil	20,9 (32,5)	50,7 (43,2)	17,8 (15,1)	2,8 (2,4)	0,6 (0,5)	Prog.: 7,3 (6,2)	

Tabelle 4.3 Zusammenfassung Ressourcen im XC6216

4.8 Reale Architekturen: ispLSI5000 und Virtex II

Zwei derzeit lieferbare und zu der neuesten Generation gehörende Bausteinfamilien sollen im Folgenden kurz vorgestellt werden: ispLSI5000 der Fa. Lattice Semiconductor [18] und Virtex II der Fa. Xilinx [21].

Die ispLSI5000-Familie stellt im Rahmen der CPLD-Architekturen eine Implementierung mit sehr vielen Eingängen pro Logikblock dar. Dies ist nach den Ausführungen in Abschnitt 4.6 durchaus wirtschaftlich, allerdings zurzeit ungewöhnlich. Die Virtex-II-Familie von Xilinx gilt als die Familie mit den größten FPGAs.

Zuvor muss noch ein Timingmodell eingeführt werden, mit dem die zeitlichen Verhältnisse in den Bausteinen beschrieben werden können. Hierzu wird gemäß Bild 4.17 eine CPLD-Architektur zugrunde gelegt. Die darin aufgeführten Zeiten (#1–#7) haben folgende Bedeutung:

- #1 und #2 stehen für Logic Cell Delay (t_{LCD}) bzw. Additional Logic Delay (t_{ALD}). Diese Zeiten beschreiben die Durchlaufzeit einer Eingangsänderung bis zum Zeitpunkt der zugehörigen Ausgangsänderung. Sie werden unabhängig von der Änderungsrichtung und etwaiger interner Logik angegeben.

Die Verzögerung für zusätzliche Logik (t_{ALD}) beschreibt in CPLDs den Fall, dass zu der in einem Block vorhandenen Logik weitere Kapazitäten notwendig sind, diese dann hinzugefügt werden (aus einem vorhandenen Pool), jedoch etwas längere Laufzeiten besitzen. Hier wird die Differenz zu #1 angegeben.

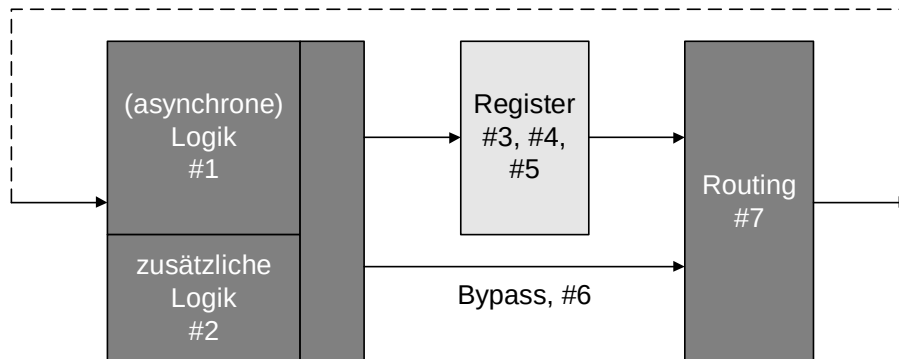


Bild 4.17 Timingmodell für CPLDs

- #3–#5 stehen für Setup- und Hold-Time sowie Register Clock-to-Output Delay (t_{SU} , t_H und t_{RCOD}) bei Speicherung in einem Register. Wird also ein Speicherelement z.B. für eine Zustandsmaschine benötigt, dann sind in dem im PLD korrespondierenden Register diese Zeiten einzuhalten.

Die Setup-Time ist dabei diejenige Zeit, die ein Eingangssignal vor der Taktflanke konstant anliegen muss, um den Speichervorgang korrekt ablaufen zu lassen. Mit Hold-Time wird die entsprechende Zeit *nach* der Taktflanke (oder dem Taktereignis) bezeichnet, in dem das Signal noch konstant bleiben muss, um den Speichervorgang nicht zu stören. Die Register Clock-to-Output Delay Time schließlich gibt die Verzögerung an.

- #6 bezeichnet die Bypasszeit (t_{BPD}). Diese Zeit ist in Zeitkalkulationen einzusetzen, wenn kein Register, sondern ein asynchrones Schaltnetz beschrieben wird.
- #7 schließlich beschreibt die Verzögerungszeiten in der Routingmatrix (t_{RMD}). Zeiten stimmt hier, denn dies ist abhängig von der Anzahl der Eingänge, die angeschlossen werden, und so werden gewöhnlich mehrere Zeiten genannt.

Für FPGAs wird dieses Timingmodell ebenfalls übernommen, wobei die Routingressourcen nunmehr aus verschiedenen Verbindungsarten zusammengesetzt sind, die beim jeweiligen FPGA-Typ erläutert werden. Fast immer sind 3 verschiedene Klassen enthalten: *Lokale* Verbindungen koppeln räumlich benachbarte Logikzellen, *regionale* überstreichen größere Abstände, und *chipglobale* Verbindungen sind meistens 1- und 2-dimensional vorhanden und dienen der Verteilung häufig genutzter Signale, z.B. Takte.

4.8.1 Lattice ispLSI5000

4.8.1.1 Architektur

Wie schon erwähnt ist die ispLSI5000-Familie [18] recht neu und wartet auch mit einer Vielzahl von Details im Aufbau auf. Bild 4.18 zeigt die logische Struktur des Generic Logic Blocks (GLB) dieser Familie.

Die summarische Blockkapazität mit 68 Eingängen und 64 Ausgängen (68E/64A) zeigt, dass diese GLBs die größten innerhalb der marktgängigen CPLDs darstellen. Die Eingänge werden zu 160 Termen für die Logik verknüpft und jeweils 5 Terme per OR zusammengefasst, und zwar 2fach. Für jede Makrozelle kann nun gewählt werden, ob 5 Terme ausreichen (Bypass), oder ob bis zu 35 Terme mit Hilfe des Product Term Sharing Arrays (PTSA) zusammengeschlossen werden müssen. Hierdurch wird eine große Flexibilisierung der inneren Kapazitäten erreicht.

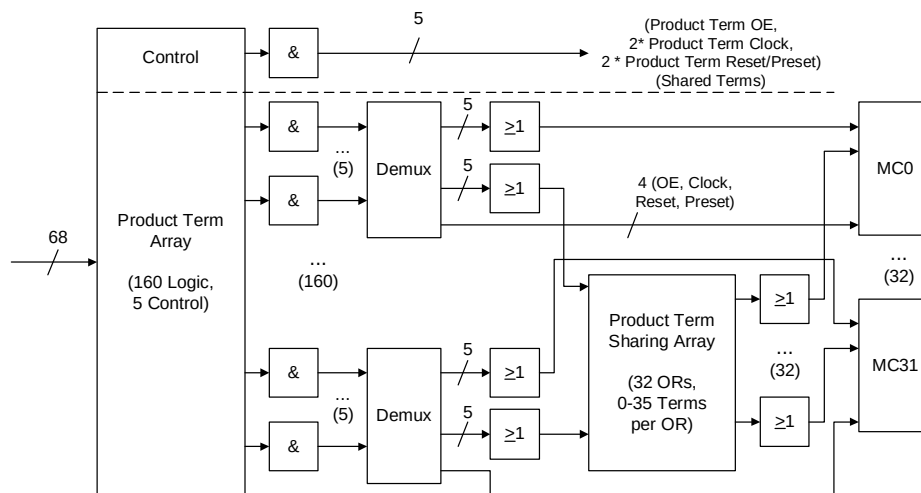


Bild 3.18 Generic Logic Block (GLB) der ispLSI5000-Familie

Die Kontrollsignale für die Makrozellen umfassen blockweite wie lokale Signale, auch hier wieder eine flexible Auswahl unter den Möglichkeiten. Last not least sind die Makrozellen konfigurierbar, und zwar durch eine Invertierung (XOR-Bit), im Registertyp (D-, T-Flipflop, Latch) und in der Auswahl der Steuersignale. Pro Makrozelle sind 2 voneinander unabhängige Signale möglich, je 1 kombinatorisches und 1 getaktetes Signal; diese stammen einmal aus dem Bypass und dem PTSA. Die volle Flexibilität ist damit nur für 32 Ausgangssignale gegeben, dennoch werden einige der Zellen doppelt genutzt werden können.

Lattice bietet sehr viele Timinginformationen zu der ispLSI5000-Familie an. Die in Tabelle 4.4 zusammengefassten Zahlen stellen dabei nur einen Bruchteil des vom Hersteller zur Verfügung stehenden Timingmodells dar, außerdem sind die jeweils maximalen Werte z.B. für die Zeit #2 eingetragen, während bei #1 diejenigen für den High-Speed-Mode ausgewählt wurden (es existiert auch ein Low-Power-Mode). Lattice bietet im Übrigen einen Timing Analyzer (ispTA) an, dessen Benutzung dringend zu empfehlen ist.

Zeit	ispLSI5000-125	ispLSI5000-70	Bemerkung
#1 (t_{LCD})	4,0 ns	8,3 ns	High-Speed Mode Maximalwert
#2 (t_{ALD})	2,0 ns	4,0 ns	
#3 (t_{SU})	1,0 ns	1,7 ns	
#4 (t_{H})	2,5 ns	5,3 ns	
#5 (t_{RCOD})	1,0 ns	1,0 ns	
#6 (t_{BPD})	0,0 ns	0,0 ns	
#7 (t_{RMD})	1,0 ns	1,2 ns	
Frequenz	118 MHz	61 MHz	

Tabelle 4.4 Zeiten im Timingmodell ispLSI5000

4.8.1.2 Elektrische Eigenschaften

Die ispLSI5000-Familie von Lattice ist für den Betrieb mit 3,3 V ausgelegt. Dem entsprechend enthält Tabelle 4.5 auch nur 1 Zeile, weitere elektrische Varianten existieren zurzeit nicht.

Teilfamilie	Betriebsspannung	Anzahl I/O	Unterstützte Anschlussnormen
ispLSI5000V	3,3 V $\pm$ 0,3 V	192–288	TTL, LVTTTL, LVCMOS2

Tabelle 4.5 Elektrische Eigenschaften Lattice-ispLSI5000-Familie, Teil 1

Die Verlustleistung in den ispLSI5000-Bausteinen wurde aus den Datenblättern durch Anwendung der Verlustleistungsmodelle bestimmt. Tabelle 4.6 fasst die elektrischen Eigenschaften der Lattice ispLSI5000-Familie zusammen. Dabei fällt eine relativ hohe DC-Verlustleistung auf, die ihren Grund in der großen Matrix hat. Hier bleibt festzuhalten, dass Lattice diese Familie speziell für Applikationen mit sehr viel Logik vorgesehen hat, und hier zeigen diese Bausteine ihre Vorteile.

Device/Modus	$P_{\text{INT, DC}}$ [mW]	Steigung [mW/MHz]	Anzahl Counter	$P/f^* \text{ cnt}$ [mW/MHz]	f_{MAX} [MHz]
ispLSI5384VA/High Speed	1188	5,5	24	0,23	125
ispLSI5384VA/Low Power	528	5,5	24	0,23	125

Tabelle 4.6 Elektrische Eigenschaften Lattice-ispLSI5000-Familie, 2. Teil

Die Programmspeicherzellen der ispLSI5000-Familie in EEPROM-Technologie ausgelegt. Boundary-Scan-Testinterface und Programmierinterface sind wie üblich dabei integriert und entsprechen den Vorgaben von IEEE 1149.1.

Familie	Speicherart (Programmierung)	In-System- Programmierung	Partielle dynamische Programmierung	Testinterface
ispLSI5000	EEPROM	IEEE 1149.1	–	IEEE 1149.1

Tabelle 4.7 Programmier- und Testinterface Lattice ispLSI5000

4.8.1.3 Varianten

Lattice bezeichnet die ispLSI5000-Familie mit SuperWide – zu Recht. Die ispLSI-5000-Familie hat tatsächlich die größten Blöcke 1. Ordnung (68E/64A) aller marktüblichen PLDs. Diese CPLDs werden daher sicher dann eingesetzt, wenn diese Blockgröße den entscheidenden Vorteil bringt – und diese Applikationen existieren.

Die Familie selbst ist in 3 Varianten lieferbar. Tabelle 4.8 fasst die Eigenschaften zusammen.

	ispLSI5256VA	ispLSI5384VA	ispLSI5512VA
Kapazität	12 000	18 000	24 000
Macrocells	256	384	512
User-IO	192	288	288
Speed Grades	-125, -100, -70	-125, -100, -70	-110, -100, -70
Temp. Grades	-C, -I (nur -70 Speed)	-C, -I (nur -70 Speed)	-C, -I (nur -70 Speed)

Tabelle 4.8 ispLSI5000-Familie von Lattice

Die Angaben zur Geschwindigkeit entsprechen den maximalen Betriebsfrequenzen z.B. für einen Binärzähler. Als Gehäusevarianten sind 208-poliges PQFP und BGA mit 208, 272 und 388 Pins vorhanden. Alle 3 Bausteine existieren in Pin-kompatiblen Varianten (PQFP208 und BGA272), sodass ein Upgrade im Design möglich ist.

4.8.2 Xilinx Virtex II

4.8.2.1 Architektur

Die Configurable Logic Blocks (CLBs) der Virtex-II-Familie [21] von Xilinx, sind gegenüber der Virtex-Architektur weiterentwickelt. Als Basiselement dient eine 4:1-Look-Up-Table-Struktur, diese ist aber in einige Kontrollstrukturen eingebettet. Die CLBs zeigen durch die Zusammenfassung von 4 Slices eine 38E/8A-Struktur. Zum Aufbau (Bild 4.19) im Einzelnen:

Jeder Configurable Logic Block enthält 4 Slices, jeder Slice wiederum 2 4:1-Look-Up-Table mit umgebender Kontrolllogik, und Bild 6.18 zeigt eine LUT, also $\frac{1}{2}$ Slice. Die Look-Up-Table sind als Logikfunktion, als 16-Bit-Speicher (distributed SelectRAM) oder als Shift-Register konfigurierbar. Als Logikfunktion betrieben entspricht dies der üblichen Anwendungsweise, wobei durch die Carry-Logik zusätzliche arithmetische Kapazität hinzukommt. Innerhalb eines Slice kann ein vollständiger 2-Bit-Addierer mit Übertrag für die nächste Stufe integriert werden. 4:1-Multiplexer passen in einen Slice, 8:1 in 2 Slices und 32:1-Multiplexer in 8 Slices, also 2 Configurable Logic Blocks.

Als Speicher konfiguriert nimmt jeder Slice 16 Bit auf. Hier ist auch ein ROM-Modus einstellbar, der also die in der Konfiguration gespeicherten Daten im Betrieb unverändert lässt. In 1 CLB sind 64 Bit als Single-Ported- und 32 Bit als Dual-Ported-RAM konfigurierbar; der duale Zugriff bedeutet dabei, dass ein asynchroner Read und synchroner Write einen Port belegen und ein zweiter Port für asynchrones Lesen zur Verfügung steht. Synchrone Lesevorgänge können durch das integrierte Register einfach implementiert werden. Für die Adressleitungen stehen zusätzliche Inputs an den LUTs bereit (siehe Bild 4.18).

Die Konfiguration als Shift-Register nutzt ebenfalls die Speicherkapazitäten des LUT, um ein 16-Bit-Shift-Register zu implementieren. Auch hier werden die Schreibvorgänge synchronisiert, während das Lesen (jedes Bit kann durch Adressleitungen ausgelesen werden) asynchron ist oder via Register synchronisiert werden kann. Die beiden Register pro Slice teilen sich im Übrigen die Takt- und Kontrollsignale und sind als flankensensitive Flipflops oder als zustandsgesteuerte Latches konfigurierbar.

Die Virtex-II-Architektur zeigt die verstärkte Tendenz, mehr als nur Logik und Verbindungen im Baustein anzubieten. Zusätzlich zu den konfigurierbaren Zellen sind auch statisches RAM (SelectRAM) und 18*18-Bit-Multiplizierer in den Virtex-II-ICs zu finden, und zwar bis zu 192 pro Baustein. Diese Multiplizierer

sind für Algorithmen der Signalverarbeitung notwendig und wesentlich schneller und energie- sowie flächeneffizienter als Implementierungen in Logikzellen.

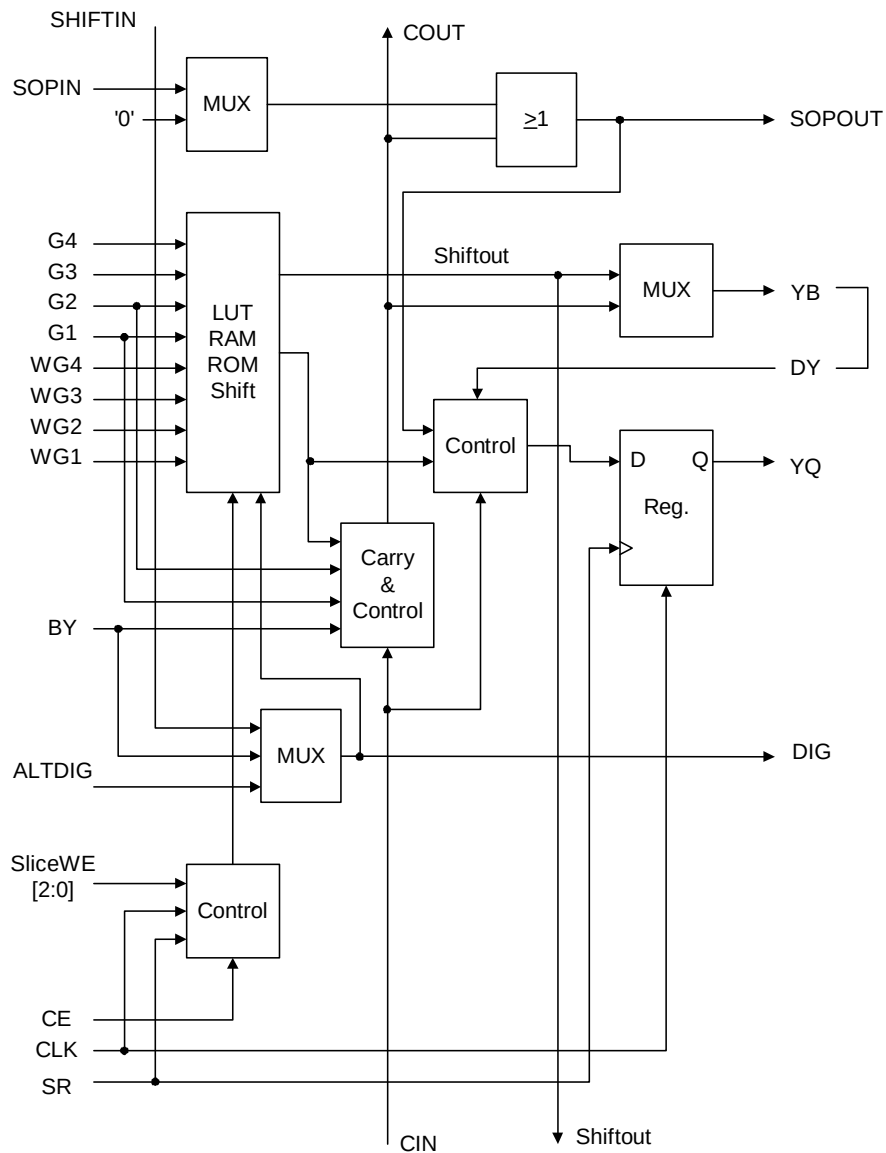


Bild 4.19 Xilinx-Virtex-II-Slice (pro CLB existieren 4 Slices)

Die Virtex-Familie von Xilinx bietet als FPGA wieder mehrere Verbindungsarten an; die Notwendigkeit hierzu wird noch durch die Zusammensetzung der kleinsten logischen Blöcke (1 CLB besteht aus 4 Slices zu je 2 Look-Up-Tabellen) verstärkt. In diesen Bausteinen sind zu finden:

- Lokale Verbindungen innerhalb der CLBs, um Rückkopplungen unter den LUTs zu ermöglichen: Pro CLB sind 8 Fast Connects verfügbar.
- Lokale Verbindungen zwischen horizontal, vertikal und diagonal benachbarten CLBs: Pro CLB existieren 16 Verbindungen zu den Nachbarn.
- Regionale Verbindungen (1): Für jede CLB existieren horizontale und vertikale Verbindungen zum nächsten und übernächsten Nachbarn (Double Lines, pro Zeile/Spalte 40).
- Regionale Verbindungen (2): Zur Verbindungen mit den Blöcken, die 3 oder 6 CLBs entfernt sind, dienen die Hex Lines (pro Zeile/Spalte je 120).
- Regionale Verbindungen (3) (Dedicated Routing): Jede CLB hat Zugriff auf 4 horizontale, tristatefähige Busleitungen (partitionierbar in Teile, je 4 CLBs überspannend) und 2 vertikale Carry-Signale.
- Globale Verbindungen (1): Je 24 horizontale und vertikale Long Lines pro CLB überspannen den gesamten Baustein und sind bidirektional angelegt.
- Globale Verbindungen (2): Je Quadrant eines IC sind 8 globale Taktverteilernetze implementiert. Diese Verteilernetze können von 16 Clockinputs getrieben werden. 1 Digital-Clock-Manager (DCM) kann die Eingangsfrequenzen maximal vervierfachen und durch 16 teilen.

Es ist sicher deutlich, dass Xilinx hier sehr viel Aufwand in die Möglichkeiten zum Signalrouting gesteckt hat. Da dies auch bedeutet, dass diverse Zeiten für die unterschiedlichen Wege anzugeben wären, hat Xilinx hierauf verzichtet (siehe Tabelle 4.9). Die dort angegebenen Frequenzen stammen aus den Beschränkungen für den internen Takt, nicht aus einer Berechnung der Signallaufzeiten. Man beachte außerdem die negative Hold-Zeit in Tabelle 4.9.

Zeit	Virtex II (-5)	Virtex II (-4)	Bemerkung
#1 (t_{LCD})	0,39 ns	0,44ns	4-Input
#2 (t_{ALD})	0,24 ns	0,28 ns	5-Input (add. zu #1)
#3 (t_{SU})	0,33 ns	0,37 ns	
#4 (t_H)	-0,08 ns	-0,09 ns	Negative Zeit!
#5 (t_{RCOD})	0,5 ns	0,57 ns	
#6 (t_{BPD})	0,0 ns	0,0 ns	
#7 (t_{RMD})	—	—	(zur Zeit keine Ang.)
Frequenz	750 MHz	650 MHz	Taktbeschränkung

Tabelle 4.9 Timingmodell Xilinx-Virtex-II-Familie

Für die genaue Berechnung der Signallaufzeiten vor dem Download in den Baustein stellt Xilinx Simulationsdaten zur Verfügung, die anhand des aktuellen Designs generiert werden.

4.8.3 Elektrische Eigenschaften

Die Virtex II-Familie von Xilinx ist in 0,15- μm -Technologie hergestellt und wird mit 1,5 V betrieben. Zur Unterstützung der Vielzahl der I/O-Standards ist eine separate Spannungsversorgung notwendig. Dies ergibt folgendes Bild für die elektrischen Eigenschaften:

Familie	Betriebsspannung	Anzahl I/O	Unterstützte Anschlussnormen
Virtex XC2V	1,5 V $\pm$ 0,75 V	88–1108	LVTTL, LVCMOS2, LVCMOS18, LVCMOS15, PCI-X (133 MHz) und PCI 3,3 (33, 66 MHz), GTL, GTL+, HSTL, SSTL3, SSTL2, AGP, LVDS, LVPECL, LDT

Tabelle 4.10 Elektrische Eigenschaften Virtex-II-Familie, Teil 1

Die Werte für Verlustleistungen dürften etwa im Bereich der für 1,5-V-Systeme üblichen Werte liegen und die für die Virtex-Familien noch unterschreiten. Zur Berechnung bzw. Abschätzung stand der Power Estimator [21] leider noch nicht zur Verfügung, sodass auf die Angaben zu den dynamischen Verlustleistungen verzichtet werden musste.

Device	$P_{\text{INT, DC}}$ [mW]	Steigung [mW/MHz]	Anzahl Counter	$P/f^* \text{ cnt}$ [mW/MHz]	f_{MAX} [MHz]
XC2V40	113		64		750
XC2V6000	300		3840		750

Tabelle 4.11 Elektrische Eigenschaften Virtex-II-Familie, Teil 2

Die XC2V-Bausteine von Xilinx arbeiten alle auf Basis von SRAM-Zellen. Das Programmierinterface wartet im Vergleich zur Virtex-Familie mit Besonderheiten auf. So ist die partielle Rekonfiguration möglich (über IEEE 1149.1 bzw. IEEE 1532 als neue Norm und bestimmte Xilinx-spezifische Modi). Hierzu können die zur Konfiguration notwendigen Pins, die normalerweise eine Doppelfunktion erfüllen, im Programmiermodus verweilen. Weiterhin können die Daten nach dem DES-Verfahren geschützt werden, ein DES-Decryptor ist auf dem Chip integriert.

Familie	Speicherart (Programmierung)	In-System- Programmierung	Partielle dynamische Programmierung	Testinterface
Virtex II	SRAM	seriell, parallel (8 Bit), jeweils aktiv/passiv, IEEE 1149.1 (IEEE 1532), mit DES- Decryptor	ja	IEEE 1149.1

Tabelle 4.12 Programmier- und Testinterface Xilinx-Virtex-II-Familie

4.8.4 Varianten

Xilinx hat in den Virtex-II-Derivaten sehr viel SRAM integriert. Hierzu wurden 18 KBit-Blöcke gewählt, die als 512*36 Bit, 1024*18 Bit oder 2048*9 Bit genutzt werden können. Der Vorteil dieser Konfigurationen besteht darin, pro 8 Bit ein Paritätsbit speichern zu können.

Die gleichen Blöcke sind auch als 16384*1 Bit, 8192*2 Bit oder 4096*4 Bit, dann ohne Paritätsbit nutzbar. Weiterhin kann ein Single- oder Dual-Port Zugriff konfiguriert werden. Bei Dual-Ported-Memory können die Zugriffports durchaus verschiedene Daten- und Adressweiten besitzen.

Tabelle 4.13 führt alle lieferbaren und angekündigten Virtex-II-Varianten auf. In dieser Tabelle wird die Kapazität dabei in Systemgattern, den Datenblättern entsprechend angegeben, sodass bei Vergleich wiederum die gebotene Vorsicht angewendet werden sollte!

	XC2V40	XC2V80	XC2V250	XC2V500
Systemgatter	40k	80k	250k	500k
Configurable Logic Blocks	64	128	384	768
Block RAM [Bit]	73 728	147 456	442 368	589 824
Distributed RAM	8192	16 384	49 152	98 304
Anzahl DCM	4	4	8	8
Anzahl Mult.	4	8	24	32
User I/O	88	120	200	264
Speed Grades	-4, -5, -6	-4, -5, -6	-4, -5, -6	-4, -5, -6
Temp. Grades	-C, -I	-C, -I	-C, -I	-C, -I

Tabelle 4.13a Virtex-II-Familie von Xilinx (Teil 1)

	XC2V1000	XC2V1500	XC2V2000	XC2V3000
Systemgatter	1M	1.5M	2M	3M
Configurable Logic Blocks	1280	1920	2688	3584
Block RAM [Bit]	737 280	884 736	1 032 192	1 769 472
Distributed RAM	163 840	245 760	344 064	458 752
Anzahl DCM	8	8	8	12
Anzahl Mult.	40	48	56	96
User I/O	432	528	624	720
Speed Grades	-4, -5, -6	-4, -5, -6	-4, -5, -6	-4, -5, -6
Temp. Grades	-C, -I	-C, -I	-C, -I	-C, -I

Tabelle 4.13b Virtex-II-Familie von Xilinx (Teil 2)

	XC2V4000	XC2V6000	XC2V8000	XC2V10000
Kapazität	4M	6M	8M	10M
Configurable Logic Blocks	5760	8448	11 648	15 360
Block RAM [Bit]	2 211 840	2 654 208	3 096 576	3 538 944
Distributed RAM	737 280	1 081 344	1 490 944	1 966 080
Anzahl DCM	12	12	12	12
Anzahl Mult.	120	144	168	192
User I/O	912	1104	1108	1108
Speed Grades	-4, -5, -6	-4, -5, -6	-4, -5, -6	-4, -5, -6
Temp. Grades	-C, -I	-C, -I	-C, -I	-C, -I

Tabelle 4.13c Virtex-II-Familie von Xilinx (Teil 3)

In den Tabellen wurden zusätzlich zu den CLBs auch die Anzahl der Multiplizierer sowie die Digital-Clock-Manager angegeben. Diese zeigen zusätzliche Kapazitäten, insbesondere für arithmetische Funktionen und Taktgenerierung an.

Die Gehäuseformen, in denen Xilinx die Virtex-E-Familie anbietet, sind dem Umfang entsprechend vielfältig: Ball Chip-Scale CS 144-polig, Fine-Pitch BGA

mit 256, 456 und 676 Anschlüssen, BGA mit 575 und 728 Anschlüssen sowie Flip-Chip BGA und Fine-Pitch BGA mit 896, 957, 1152 und 1517 Pins sind vertreten.

5 Hardware/Software-Interface und Software im Configurable Computing

Bei Bausteinen aus dem Gebiet des Configurable Computing handelt es sich um *programmierbare Bausteine* – dementsprechend muss es auch ein Hardware/Software-Interface sowie Übersetzungsalgorithmen für dieses Gebiet geben. Dies ist Thema dieses Kapitels.

Dieses Thema hat natürlich viele Facetten. Zunächst soll der gesamte Softwareprozess im Überblick dargestellt werden (→ 5.1). Danach folgt eine kurze Einführung in die Designphilosophie für Programme (im Configurable Computing, → 5.2).

Abschnitt 5.3 gibt dann einen Einblick in die Übersetzungsalgorithmen zur Abbildung der Logikverknüpfungen auf die realen Bausteine.

5.1 Softwareprozess im Configurable Computing

Der Softwareprozess – hierunter sollen alle Vorgänge von der Sourcecodeeingabe über Simulationen bzw. Verifikationen bis hin zur endgültigen Übersetzung verstanden werden – ist für Bausteine im Configurable Computing (immer noch) nahezu identisch mit demjenigen für Hardwarebausteine (ASICs etc.). Der Grund hierfür ist naheliegend, haben sich die PLDs doch aus den ASICs bzw. Standardbausteinen heraus entwickelt. Zudem ist das Ausführungsparadigma – maximale Parallelität in den kombinatorischen Teilen, Synchronisierung der Rechnungen durch speichernde Elemente (Flipflops), zusammengefasst Finite-State-Machine-Verhalten – für Hardware und programmierbare Hardware identisch.

Dies hat jedoch weitreichende Konsequenzen: Ein(e) Softwareentwickler(in), der/ die PLDs aufgrund der hervorragenden Laufzeiteigenschaften in Applikationen einsetzen wollen, muss sich mit dem Hardwaredesignprozess und den Hardwarebeschreibungssprachen auseinandersetzen. Diese Hürde verhindert viele Einsatzfälle. Weiterhin muss man feststellen, dass die Eigenschaft der Programmierbarkeit nahezu nicht ausgenutzt wird: Die Systeme gelten im Betrieb als unveränderlich.

Bild 5.1 stellt den Softwareprozess für PLDs (und natürlich auch für Hardware) dar. Als erstes fällt die Rolle der Simulation auf: Während bei Mikrocontrollerbasierten Applikationen die Simulation eigentlich eher beiläufig auftritt, der Applikationstest jedoch als echter Test im endgültigen System abläuft, wird bei der Hardwareentwicklung gefordert, jeden Schritt intensiv zu simulieren. Dies stammt aus der Entwicklung für ASICs und Full-Custom-ICs, bei denen besonders wichtig ist, dass das erste Design funktionsfähig ist (*'First Shot to Silicon'*). Es hat sich bei PLDs erhalten, weil die Zeit (und nicht nur die logische Richtigkeit) bei Hardware-

programmen eine gewichtige Rolle spielt und das Auffinden von Zeitfehlern durch Simulation wesentlich einfacher ist.

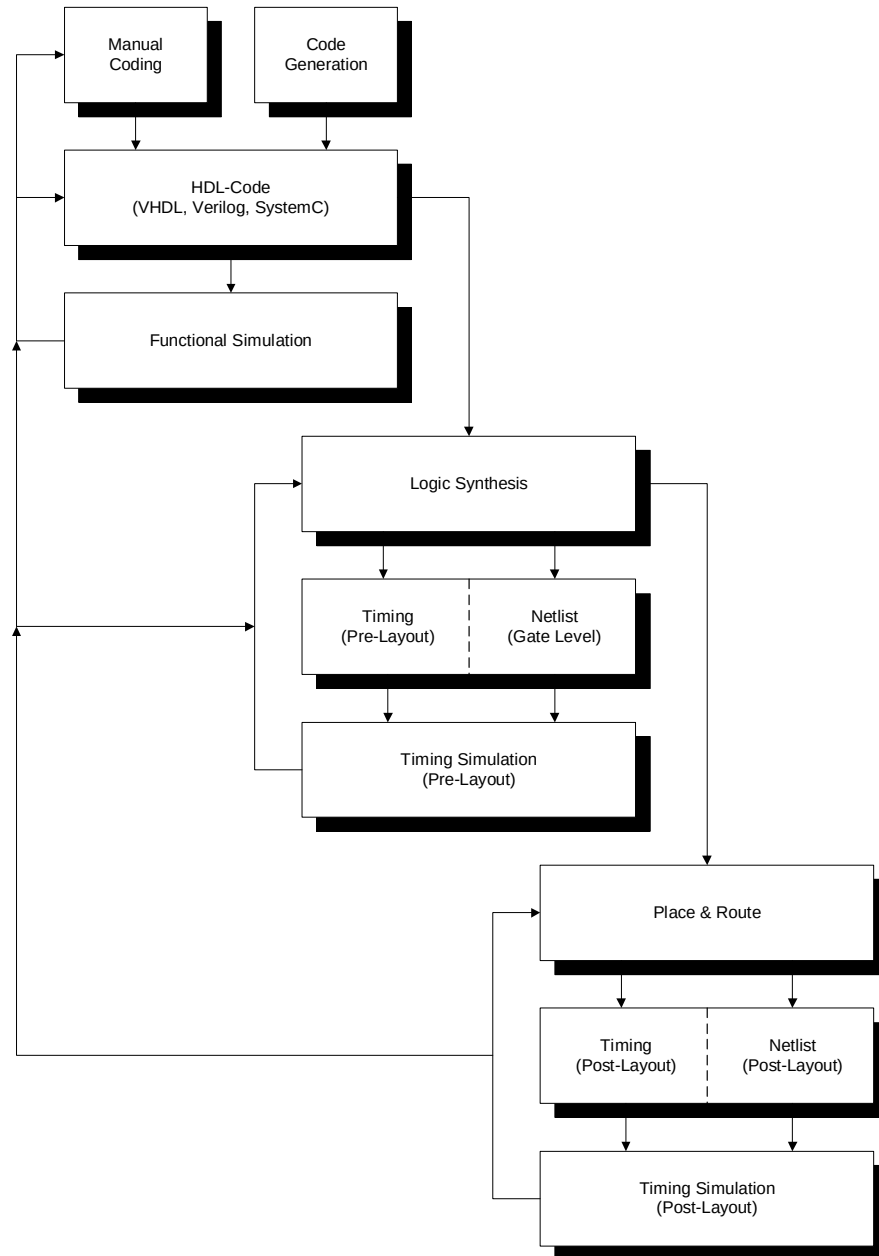


Bild 5.1 HDL-Design- und Simulationsfluss

Zugleich ist dem Bild 5.1 zu entnehmen, dass der gesamte Designfluss in 3 scheinbar wasserfallartig gekoppelte Teilprozesse gegliedert ist: Teil 1 besteht aus der Erstellung des Designs in einer Hochsprache (HDL, Hardware Description Language), Teil 2 in der Zielbaustein-unabhängigen Übersetzung, Teil 3 dann in der Abbildung auf die reale Hardware. Diese Kette entspricht tatsächlich einem Wasserfall, wenn es 'nur' um die logischen Verknüpfungen geht.

Im Fall von Zeitvorgaben, die im HDL-Sourcecode zwar vorkommen (können), zurzeit jedoch nur für Simulationszwecke verwendbar sind, entspricht die Kette nicht dem Wasserfallmodell. Die Zeitvorgaben aus dem Sourcecode werden nicht oder nur indirekt (per Optimierung) berücksichtigt, der *formale* Parameter Zeit ist nicht *funktional*. Ergibt nun die Post-Layout-Simulation Zeitfehler, d.h. Überschreitungen der vorgesehenen Laufzeiten, so wird dadurch ggf. eine komplett neue Beschreibung erzwungen (*Back Annotation*).

Mit anderen Worten: Da der Parameter Zeit in Hardwaredesigns und im Configurable Computing eine wesentlich andere (wichtigere) Rolle als im Softwaredesign spielt, sollte er durch den Designer berücksichtigt werden. Dies ist aber direkt unmöglich, und die indirekte Kopplung ist sehr mittelbar und erfahrungsbasiert.

5.2 Design von Programmen im Configurable Computing

5.2.1 Programmiermodell in PLDs

Bild 5.2 zeigt das Grundmodell für programmierbare Logikbausteine.

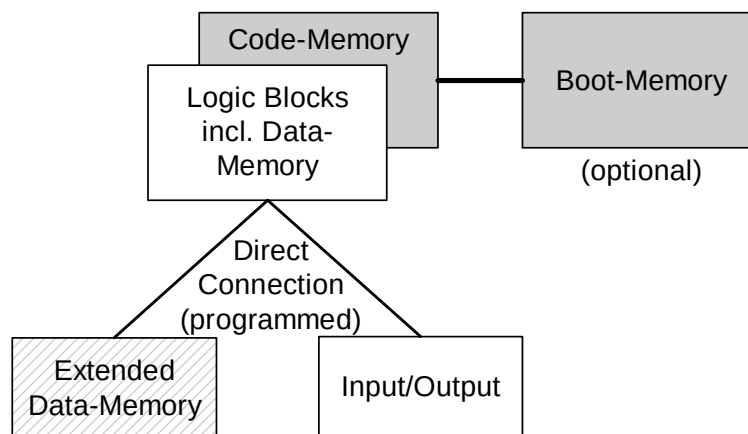


Bild 5.2 Grundmodell für Programmable Logic Devices

Dieses Modell zeigt die verschiedenen Speicherarten und ihre Nutzung, daneben die Rechelemente (Logikblöcke) und die In-/Output-Bereiche. Augenscheinlich sind Ähnlichkeiten zum Von-Neumann-Rechnermodell durchaus zu sehen, wobei auf die unterschiedlichen Berechnungsmethoden (in Space/in Time) hingewiesen (→ 2).

Maximal vier Speicherarten, unterschieden durch die Nutzung, stehen in PLDs zur Verfügung: Code-Speicher, genutzt als Konfigurationsspeicher, optionaler Boot-Speicher, der das Programm bei flüchtigem Konfigurationsspeicher enthält, (normaler) Datenspeicher, der eigentlich keine Daten, sondern Zustände speichert, sowie optional der erweiterte Datenspeicher.

Die Unterscheidung zwischen Zustandsspeicher und erweitertem Datenspeicher erscheint zunächst nicht wichtig; indes kann der Zustandsspeicher sowohl Daten- als auch Sequenzinformationen für Zustandsmaschinen aufnehmen, der Speicher ist integraler Bestandteil der verarbeitenden Logik und mit dem Registerfile einer RISC-Maschine vergleichbar.

Der erweiterte Datenspeicher ist zurzeit nur bei einigen (großen) PLDs zu finden und dient meist der Aufnahme von Datenmengen (z.B. Tabellen für Transformationen, aber auch dynamische Daten). Dieser Speicher ist mit dem Datenspeicher des Von-Neumann-Modells vergleichbar.

Zudem gibt es noch einen weiteren Unterschied im Layout der Bausteine: Während der erweiterte Datenspeicher konzentriert angeordnet werden kann, ist der Zustandsspeicher auf dem Baustein verteilt (und damit auch "locker" angeordnet).

Code- (Konfigurations-) und Datenspeicher sind strikt voneinander getrennt. Während auf Datenspeicher durchaus sequenzielle Zugriffe möglich sind (und über den Zustandsspeicher gerade Sequenzen definiert werden), ist der Codespeicher ("locker") verteilt, und die Informationen sind ständig parallel verfügbar.

Bild 5.3 zeigt nun das hardwarenahe Programmiermodell: Eine Menge von Logikblöcken schaffen die Möglichkeit, innerhalb der Ressourcen jede Funktion zu etablieren. Hierzu sind drei Konfigurationen notwendig: Die Logik, also die Funktionalität, muss in die Logikblöcke abgebildet werden (1), die Verbindungen zwischen diesen Blöcken (das Routing, (2)) müssen geschaffen werden, und die Verbindungen zum bzw. vom I/O müssen geschaltet werden (ebenfalls Routing, (3)).

Damit sind schon die wesentlichen Aufgaben einer hardwarenahen Programmierung sowie der Übersetzungsprogramme, die die Abbildung von höheren Sprachen auf die Hardware vornehmen, beschrieben: Abbildung auf die Logik, Platzierung der Logikblöcke im Baustein (Placement) und Verdrahtung (Routing). Eine derartige manuelle Programmierung ist aktuell nur noch selten anzutreffen, sie wurde vor einigen Jahren für PLDs in grafischer Form angeboten. Ein Beispiel für die textuelle, maschinennahe Programmierung ist die Sprache NML, Native Mapping Language, der Firma PACT für die XPP-Systeme (eXtreme Processing Platform), die in den Bereich der rekonfigurierbaren Systeme gehören.

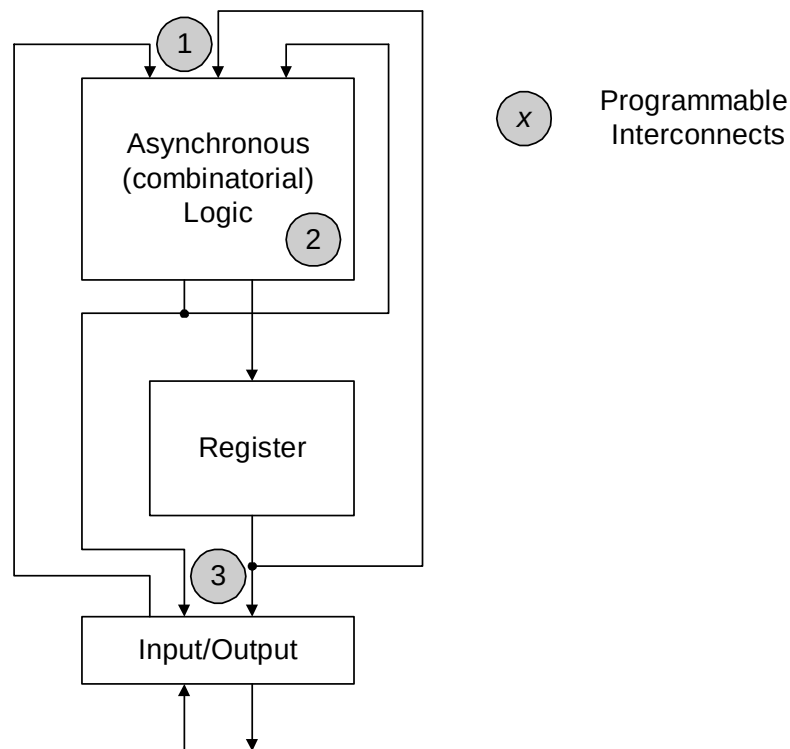


Bild 5.3 Hardwarenahes Programmiermodell

Ein einfacher Boolescher Assembler, der die logischen Gleichungen darstellt, ist damit im Sinne des Routing nicht maschinennah (nur im Fall der SPLDs könnte dies gelten). Eine solche Assemblersprache, die zwar die logischen Verknüpfungen, nicht aber die Lage im Baustein und die Art der Verbindungen darstellt, ist mit der Netzliste in Benutzung (z.B. zur Darstellung der Technologie-unabhängigen Übersetzung, → Bild 5.1).

Der Ausschnitt in Bild 5.4 stellt zwei Teile der Netzliste dar: Eine Verdrahtungsanweisung (NET) verbindet einfache Signale, während eine Logikblockanweisung (SYM) Ein- und Ausgänge eines Blocks sowie die Logikzuweisung in Form einer Tabelle (ähnlich zum OpenPLA-Format, einem allgemein bekannten Format zur Angabe logischer Zusammenhänge in Tabellenform) angibt. Die logische Funktion lautet:

$$G0_D = G1_PIN * G2_PIN * /G3_PIN + G1_PIN * /G2_PIN * G3_PIN + /G1_PIN * G2_PIN * G3_PIN + /G1_PIN * /G2_PIN * /G3_PIN$$

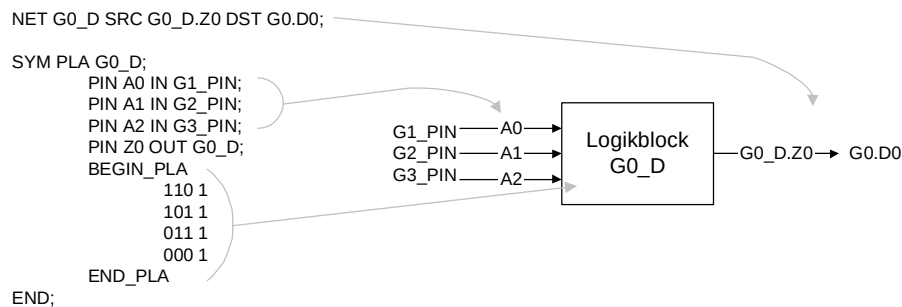


Bild 5.4: Ausschnitt aus einer Netzliste (Lattice Advanced Format LAF)

5.2.2 Prinzipien der HDL-Programmierung

Die Beschreibung des Algorithmus für einen Baustein aus dem Bereich des Configurable Computing in einer Hochsprache (HDL) muss eine wesentliche Schwierigkeit überwinden: Programmierbare Hardware arbeitet inhärent parallel, d.h., der Parallelitätsgedanke muss in der Codierung sichtbar sein. Andererseits sind die üblichen Algorithmen eher sequenziell codiert (was auch eher der menschlicher Denkweise entspricht und für Software ständig trainiert wird). Diese Kluft muss überbrückt werden.

Was heißt hier eigentlich sequenziell? Der wesentliche Punkt ist in diesem Fall, dass in einer Anweisung neu berechnete Daten den nachfolgenden Anweisungen sofort zur Verfügung stehen, umgekehrt jedoch nicht. Mit anderen Worten: Das Wesentliche ist das Zuweisungskonzept für die Daten.

Die Lösung sieht für die HDL-Entwicklung dann so aus (die Nomenklatur ist entsprechend VHDL, Very High-Speed Integrated Circuits Hardware Description Language gewählt):

- Die typischen Variablen heißen hier Signale, und eine Signalwertzuweisung sollte nur an einer einzigen Stelle erfolgen (mit Ausnahmen). Die einzige Stelle kann innerhalb eines IF- oder CASE-Konstrukts natürlich auch mehrfach sein, sie muss es sogar, weil in jedem Zweig die Signale gesetzt werden müssen.
- Die parallele Algorithmierung erfolgt in (parallel zueinander ausführenden) Prozessen. Diesen Prozessen wird im Allgemeinen eine Sensitivitätsliste mitgegeben, d.h. eine Liste von Signalen, auf deren Änderungen hin im Prozess eine neue Berechnung durchgeführt werden soll.

In der Simulation lässt sich dies so abbilden, dass die Berechnung eines Prozesses jedes Mal gestartet wird, wenn eine Änderung an mindestens einem der Signale in der Sensitivitätsliste vorliegt (Anmerkung: Die Signale führen dazu auch einen Zeitbegriff mit sich).

In der Synthese bedeutet dies, dass der Bereich, der diesen Prozess später repräsentiert, mindestens die Signale aus der Sensitivitätsliste als Eingänge führt.

- Innerhalb der Prozesse wird der Algorithmus sequenziell codiert, wobei für die Signale eine besondere Regelung gilt: Eine Signalzuweisung erfolgt erst am Ende eines Prozesses. Auf diese Weise kann die sequenzielle Denkweise beibehalten werden, zugleich wird dem Hardwareaspekt genügt.

Für Daten, die bereits während der Berechnung verändert werden müssen, existiert eine andere Objektklasse: *Variable*. Ob diese Daten synthetisiert werden oder nicht, hängt vom Problem ab.

- Prozesse, die von den gleichen Signalen (oder zumindest von zeitgleich verändernden Signalen) in der Simulation berechnet werden, durchlaufen den Code, die Signalzuweisung erfolgt erst am Ende *aller* Prozesse zu diesem Zeitpunkt. Ggf. erfolgt dann eine erneute Berechnung der Prozesse (zum gleichen realen Zeitpunkt mit einem virtuellen Versatz um ein δ von 0), falls wiederum Änderungen in der Sensitivitätsliste festgestellt werden.

Dies bedeutet für das Softwaredesign einer Hardware, dass als erster Schritt eine Partitionierung in Teile gefunden werden muss, die in Prozessen beschrieben werden sollen. Diese Partitionierung wird im Designprozess meist ständig verfeinert, um synthesesfähige Beschreibungen zu erhalten.

5.3 Übersetzungsalgorithmen

5.3.1 Logiksynthese

Eng gekoppelt mit dem Begriff des Configurable Computing steht der Begriff der Logiksynthese, also des Zusammensetzens der Hardware (bzw. der Konfiguration) gemäß den Beschreibungsweisen, die in einer der möglichen Hochsprachen erstellt wurden. Dieser Synthese geht der Ruf der Sprachverwirrung voraus, da sich Synthese auf verschiedene Level von Beschreibungsweisen bezieht.

Zunächst muss festgestellt werden, dass es ein automatisches Synthesesystem auf Systemebene zurzeit (2004) nicht gibt, weder im kommerziellen noch im forschenden Bereich. Unter diesem Tool würde man die Schaffung eines elektronischen Systems mit Hard- und Software aus einer höheren Beschreibung heraus verstehen. Synthese bezieht sich daher mit verschiedenen Level auf verschiedene Beschreibungsweisen, die alle mehr oder weniger interaktiv in die elektronischen Systeme überführt werden.

Bei dieser Aufgabe durchläuft die Synthese eine Reihe von Aufgaben, die im folgenden erläutert werden; Bild 5.5 gibt das Y-Diagramm wieder, das eine Einordnung der Synthese nach Entwurfssichten und -ebenen darstellt.

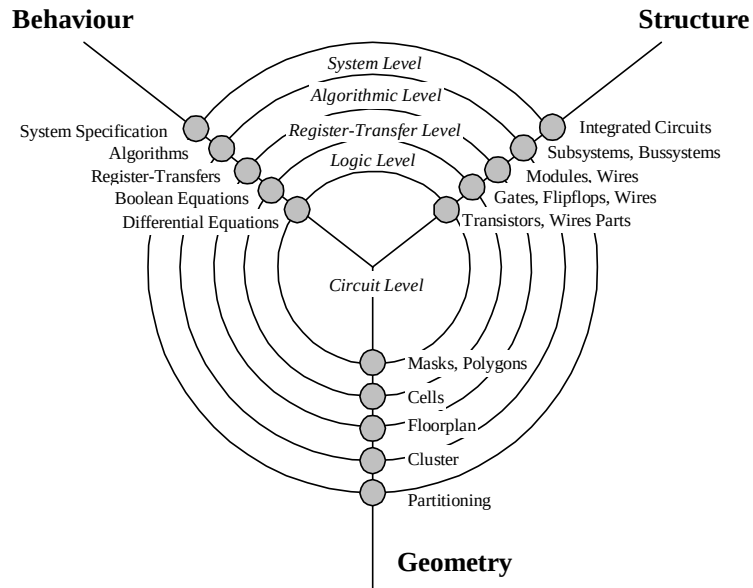


Bild 5.5 Y-Diagramm nach Gajski und Walker

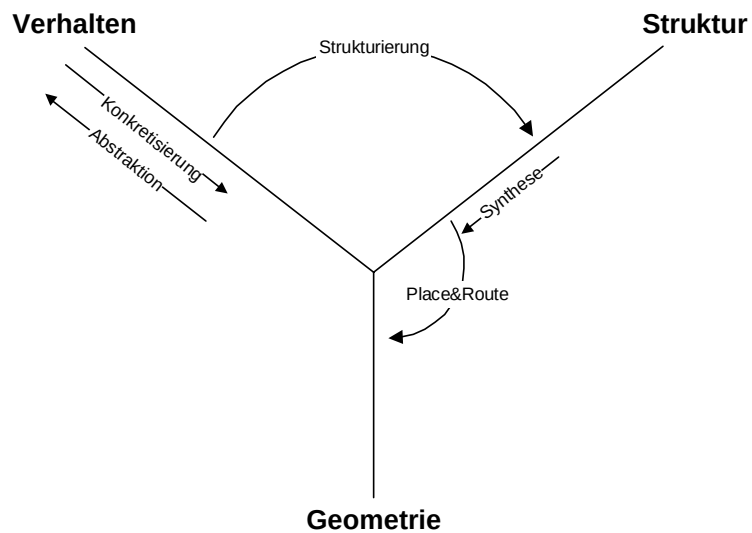


Bild 5.6 Aufgaben im Syntheseprozess

Bild 5.6 zeigt die wesentlichen Aufgaben bei dem gesamten Syntheseprozess. Hierbei sind Nutzer- und Compileraktionen zusammengefasst. Die Verhaltenssicht beschreibt ein Design (auch einen Teil) in seinem Verhalten als Ganzes, während die strukturelle Sicht das Design durch Zusammenfügen aus kleineren Teilen annimmt.

Bild 5.7 stellt die Gesamtaufgaben in einer absteigenden Hierarchie dar und ordnet dies den Syntheseselevel zu, die allgemeinüblich dazu definiert werden. Wichtig ist die Bemerkung, dass die Synthese ein Quasi-Kontinuum von Aufgaben darstellt!

<u>Synthesis Level</u>	<u>Synthesis Task</u>	<u>Responsibility</u>
<i>System</i>	Partitioning	User
	Pipelining	User
	Scheduling	User
Algorithm	Retiming	User
	Register Allocation	User/Compiler
	Resource Allocation	User/Compiler
Register Transfer Level	Register Inferencing	Compiler
	State Machine Synthesis	Compiler
	Multi-Level Optimisation	Compiler
	Two-Level Optimisation	Compiler
Logic	Redundancy Removing	Compiler
	Technology Mapping`	Compiler
	Technology Translation	Compiler
<i>Physical</i>	Physical Synthesis	Compiler (only ASIC)

Bild 5.7 Aufgaben innerhalb der Synthese

Gliederung (Partitioning)

Ein Design wird in kleine Abschnitte aufgeteilt, die später separat als Module, ICs oder Leiterplatten realisiert werden. Kriterien sind hierfür das Verhalten des Designs (funktionelle Gliederung), die Schaltung selbst (strukturelle Gliederung) oder eine Kombination aus beiden. Die Gliederung wird häufig unter Gesichtspunkten wie Verbesserung der Lokalität der Signale, des Signallaufzeitverhaltens, der Kosten, der Chipfläche, zumeist weniger unter physikalischen, mehr unter funktionalen Aspekten.

Pipelining

Das Pipelining gliedert den Verarbeitungsfluss eines Algorithmus in eine Anzahl sequentieller Abschnitte, die parallel zueinander ausgeführt werden. Dies geschieht innerhalb der einzelnen Teile, wobei das funktionelle Pipelining die gemeinsame Nutzung der Ressourcen im Vordergrund sieht, das generalisierte Pipelining hingegen die Durchsatzoptimierung durch Weglängenausgleich.

Scheduling

Das Scheduling oder die Ablaufplanung ordnet die Operationen in einer verhaltensbezogenen Beschreibung einer Sequenz von Betriebszuständen (z.B. Tasks, States) zu. Dabei kann in dieser Designphase die Reihenfolge von Einzeloperationen geändert werden, die Daten- und Prioritätsbeziehungen sowie die allgemeine Verhaltensintegrität bleiben erhalten. Beispielsweise können hierbei Datenflüsse neu sortiert werden, um zu einer schnelleren Bearbeitung zu kommen, während das Ergebnis selbstverständlich unbeeinflusst bleibt.

Retiming

Retiming ist ein spezieller Begriff aus der sequenziellen Optimierung, bei dem Logikteile über Registergrenzen hinweg verschoben werden. Dieses Retiming dient insbesondere dem Hauptzweck, die Länge der Signalwege zwischen ähnlich getakteten Registern zwecks optimiertem Gesamtdurchsatz anzugleichen.

Registerzuordnung

Die Registerzuordnung bzw. Registervergabe selektiert in einem Taktzyklus Register zur Speicherung von Daten, die in einem späteren Taktzyklus wiederum benötigt werden. Um den im VHDL-Konzept beispielsweise vorhandenen Variablen nicht jeweils einen Speicherplatz zuzuordnen, wird versucht, nur den Zuweisungen eines Wertes an eine Variable den benötigten Speicherplatz zuzuordnen und somit nur aktuelle Werte zu speichern.

Dieser Ansatz ähnelt in starkem Maße dem Compilerbau bzw. der Theorie, über Kettenrechnungen hinweg mit einer minimalen Anzahl von Registern (= Inhalt von Registervariablen) auszukommen. Um dieses Problem optimal im Sinne von minimal zu lösen, ist natürlich die Speicherung von Werten einer Variablen zu verschiedenen Zeitpunkten in verschiedenen Registern gestattet.

Im Übrigen stellen die Registerzuordnung und die Ressourcenzuordnung die überlappende Tätigkeiten zwischen Synthese gemäß algorithmischem und Register-Transfer-Level dar.

Ressourcenzuordnung

Mit Ressourcenzuordnung wird der Vorgang bezeichnet, mit dem Komponenten wie Addierer, ALU etc. ausgewählt werden, um die Operationen gemäß der verhaltensbezogenen Beschreibung zu implementieren. Dieser Vorgang ist

ausgesprochen wichtig und wird mit Hilfe der vorhandenen Bibliotheksfunktionen durchgeführt. Dabei müssen Entscheidungen getroffen werden, die eventuelle Mehrfachnutzung von Ressourcen (bei nicht-gleichzeitiger Ausführung) ermöglichen.

Register Inferencing

Register Inferencing stellt fest, welche Werte über Taktzyklen hinweg gespeichert werden müssen, ggf. unter welchen Bedingungen. Diese Feststellung ist z.B. für Wait-Bedingungen gemäß VHDL-Beschreibung sehr wichtig und führt zur Einfügung von zusätzlichen Registern mitsamt Steuerleitungen.

State Machine Synthese

Mit der Synthese von Zustandsschaltungen beginnt zugleich die Synthese durch Logikbeschreibung. Ein Zustandsdiagramm bzw. Zustandstabelle wird in die binäre Codierung der symbolischen Zustände einer Finite State Machine (FSM) umgewandelt. Diese Codierungen bestimmen die Anzahl der Register in der resultierenden Umgebung sowie die Funktionen des umgebenden Logiknetzwerks.

An dieser Stelle werden 2 Arten von Optimierungen durchgeführt: Minimierung der Anzahl der Zustände durch Verschmelzen gleichwertiger Zustände sowie Optimierung des Logiknetzwerks für die Zustandsübergänge durch Bestimmung der Menge der Zustandskodierungen.

Optimierung von mehreren Logikebenen

Die kombinatorische Logik lässt sich durch eine Netzliste von Gattern beschreiben, die ihrerseits aus bekannten Funktionen wie NAND, NOR etc. oder aus individuell formulierten logischen Funktionen (DNF, KNF) bestehen. Die Optimierung dieser Funktionen kann z.B. in Richtung Flächenbedarfreduzierung durch Mehrfachnutzung verlaufen, oder es wird zu Laufzeitminimierung hin optimiert, z.B. durch individuelle Funktionen, bei steigendem Flächenbedarf.

Optimierung zweier Logikebenen

Für die bekannte AND-OR-Struktur (Sum-Of-Products, DNF), die in SPLDs und CPLDs sowie in vollständiger Form in FPGAs Einzug gefunden hat, existieren eine Reihe von Minimierungstools (auf lokaler und globaler Basis), die zu (nahezu) minimalen Ergebnisse, Fläche und Laufzeit betreffend, kommen (Beispiel: Berkeley's Espresso).

Entfernen von Redundanzen

Redundanzen im Synthese-Bereich liegen dann vor, wenn ein Gatter ohne Funktionsänderung aus dem Logikplan entfernt werden kann. Dies vergeudet Chipfläche, ggf. Verlustleistung und Schaltzeit, so dass eine Optimierung im Sinne von Entfernung notwendig ist.

Technologie-Zuordnung

Die Technologie-Zuordnung wandelt ein Technologie-unabhängiges Boolesches Netzwerk (z.B. in Form der Netzliste, → Bild 5.4) in eine spezifische Netzliste eines Halbleiterherstellers um (im Beispiel Bild 5.4: Vom LAF in LIF, Lattice Internal Format). Dies geschieht mit dem Ziel der optimalen Anordnung und Verbindung der Elemente und Bibliotheksfunktionen.

Technologie-Umsetzung

bezeichnet eine Umsetzung von einer Implementierung in eine andere, z.B. durch Ähnlichkeitszuordnung oder durch Wandlung in eine Technologie-unabhängige Netzliste mit anschließender Rückwandlung.

Physikalische Synthese

Die physikalische Synthese schließlich liefert das Design auf Maskenebene zur ASIC-Herstellung. Hierzu gehören Floorplaning, Platzierung und Verdrahtung auf dem Chip, so dass technologiespezifische Layout-Generatoren zum Einsatz kommen.

Damit ist die Erwähnung der CAE-Umgebung und der Arbeitsschritte von der Beschreibung zur Synthese beendet. Als Fazit bleibt, dass diese Arbeiten in starkem Maße interaktiv sein müssen. Ebenso sind Architekturrückwirkungen bei PLDs (anders als bei ASICs, die z.T. sehr gleichförmig strukturiert sind) nicht zu vermeiden, so dass die Logikoptimierung von vorgegebenen Strukturen ausgehen muss und kaum zu allgemeinen Beschreibungen einer Booleschen Netzliste kommen kann, die für alle feldprogrammierbaren Bausteine gleichzeitig optimal sein kann.

5.3.2 Algorithmen zum Mapping

Dieser Abschnitt befasst sich mit folgendem Problemkreis: Gegeben sei eine Beschreibung des Problems; gesucht ist dazu die optimale Lösung, die einerseits das Problem löst, also eine Funktion bietet, die allen Anforderungen entspricht, andererseits in einen gegebenen Baustein mit seinen internen Ressourcen und Strukturen passt.

Diese Lösung lässt sich nicht in einem Durchgang finden; aus diesem Grund werden Algorithmen hierzu in den folgenden beiden Abschnitten besprochen, wobei auf diesen Abschnitt die Beschreibungen der Logik in kanonischen Normalformen und deren Minimierung fallen.

Die Beschreibungen in diesem Abschnitt beschränken sich auf Schaltnetze; dies kann für ein Mapping und Routing für reale PLDs ohne Beschränkung der Allgemeinheit durchgeführt werden, da alle PLDs eine Körnigkeit aufweisen, die kombinatorische von sequenziellen Elementen voneinander trennt. Die Schaltnetze,

die im Folgenden besprochen werden, verbinden in diesem Sinn die (eventuellen) Speicherglieder miteinander.

5.3.2.1 Die Beschreibungsfindung für kanonische Normalformen

Wie bereits dargestellt wurde ($\rightarrow$ 2.1.1), lassen sich beliebige Schaltnetze, also Funktionen von Schaltvariablen, durch die kanonischen Normalformen KDNF und KKNF beschreiben, indem die zu '1' führenden Eingangskombination (bei der KDNF, zu '0' bei der KKNF) als Konjunktion (Disjunktion) beschrieben werden und alle entstehenden Terme anschließend durch Disjunktion (Konjunktion) miteinander verbunden werden. Diese Beschreibungsform gilt natürlich für Schaltnetze mit einem Ausgang, für mehrere Ausgänge werden entsprechend mehrere Normalformen geschaffen.

Die konsequente Anwendung von Tautologien (Äquivalenzen der Booleschen Algebra) kann aus der entstandenen Normalform eine minimierte Form geschaffen werden, die sich durch die minimierte Anzahl der Disjunktionen (Konjunktionen) auszeichnet; der Aufwand hierfür kann allerdings beträchtlich sein.

Eine gut geeignete Möglichkeit, die Lösungsformulierung für eine Ausgangsvariable gleich in minimierter Form zu finden, stellen die sogenannten KV-Diagramme (Karnaugh-Veitch):

Definition 4.1:

Ein **Karnaugh-Veitch-Diagramm** ist die graphische Darstellung einer Wertetabelle bzw. Schaltfunktion, die in zweidimensionaler Matrixform angeordnet ist. Jedem Punkt der Matrix ist genau eine der möglichen Kombinationen der Schaltvariablen zugeordnet. Zu diesem Zweck werden bei n Variablen der einen Dimension $n/2$ (n gerade) bzw. $(n+1)/2$ (n ungerade) zugeordnet, der anderen Dimension $n/2$ bzw. $(n-1)/2$. Die Gesamtanzahl der Matrixfelder beträgt bei n Schaltvariablen 2^n , so dass auf jede Dimension $2^{n/2}$ (für n gerade) Zeilen oder Spalten entfallen. Die Anordnung der Felder wird so vorgenommen, dass sich beim Übergang in horizontaler oder vertikaler Richtung nur der Wert einer Variablen ändert; dies gilt auch für die Ränder links/rechts und oben/unten, das KV-Diagramm kann als Zylinder in jeder Richtung gedacht werden.

Die Anwendung der KV-Diagramme liegt darin, dass die KDNF direkt aus ihnen heraus formuliert werden kann: Jede '1' als Eintrag im Diagramm wird als Konjunktion der Eingangsvariablen aufgefasst, die Disjunktion über alle Konjunktionen ergibt die Schaltfunktion.

Bild 5.8 zeigt ein KV-Diagramm als Beispiel für die Umsetzung von hexadezimalen Zahlen, als Binärzahl mit 4 Stellen (B0 .. B3) dargestellt in BCD-Darstellung (BCD0 .. BCD4):

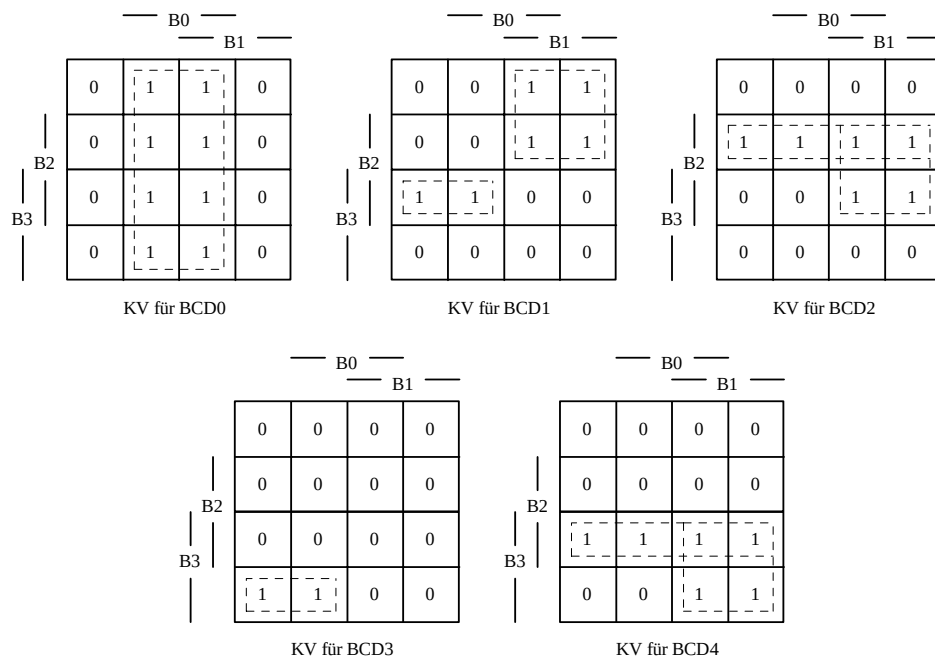


Bild 5.9 KV-Diagramme mit Zusammenfassung

5.3.2.2 Die Minimierung nach Quine-McCluskey

Die KV-Diagramme werden für Werte $n > 6$ sehr unübersichtlich, da dann 128 und mehr Matrixeinträge auftreten. Zwar lassen sich dann noch maschinelle Auswerteverfahren anwenden, jedoch gibt es hier besser implementierbare Algorithmen, von denen einer, benannt nach den Autoren Quine und McCluskey, gezeigt werden soll.

Das Quine-McCluskey-Verfahren geht zunächst von Schaltfunktionen in KDNF aus. Für die Schreibweise werde vereinbart, dass die Minterme der Schaltfunktion nicht durch die negierten bzw. nichtnegierten Variablen, sondern durch ihre **Binäräquivalente** dargestellt werden:

- 1 steht für eine nicht negierte Variable,
- 0 steht für eine negierte Variable
- - steht für eine nicht auftretende Variable

Die Schaltfunktion wird durch eine Funktionstabelle gegeben, in der die Reihenfolge der Funktionswerte so angeordnet ist, dass die Binäräquivalente der Terme aufsteigenden Dualzahlen entsprechen. Die Terme selbst werden zusätzlich mit Dezimalzahlen bezeichnet.

Bild 5.10 zeigt die Funktionstabelle für den Dual-BCD-Codierer, Codierung für BCD1:

Dez.	B3	B2	B1	B0	BCD1
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	1
7	0	1	1	1	1
8	1	0	0	0	0
9	1	0	0	1	0
10	1	0	1	0	0
11	1	0	1	1	0
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	0
15	1	1	1	1	0

Bild 5.10 Funktionstabelle für BCD1

Im ersten Schritt werden die Minterme nach gewichteten Gruppen in eine zweite Tabelle eingetragen, wobei das Gewicht einer Gruppe durch die Anzahl der Einsen in den Binäräquivalenten bestimmt wird.

Im zweiten Schritt werden die Minterme benachbarter Gruppen (!) zu einem um eine Variable kürzeren Term zusammengefasst, sofern dies möglich ist; zu diesem Zweck dürfen sich die Minterme nur in einer Stelle um 0 und 1 unterscheiden, das zugrundeliegende Logikgesetz lautet dazu

$$(x * y) + (x * /y) = x$$

Die Durchführung des zweiten Schritts erfolgt ggf. iterativ (wie in Bild 5.11 dargestellt), bis keine Vereinfachungen mehr möglich sind. Zu diesem Zweck beginnt man systematisch mit dem ersten Tabelleneintrag und vergleicht mit der nachfolgenden Gruppe. Verläuft der Vergleich erfolgreich, dann wird der neue Eintrag in die nächste Tabelle durchgeführt, gleichzeitig alle beteiligten Minterme gekennzeichnet (in Bild 5.11 mit !), die aber noch weiterhin verwendbar sind. Treten in der nachfolgenden Tabelle identische Einträge mehrfach auf (im Beispiel 2,3;6,7 und 2,6;3,7), dann wird einer gestrichen. Nicht gekennzeichnete Einträge heißen *Primimplikanten*.

Dez.	B3	B2	B1	B0	M	Gruppe
2	0	0	1	0	!	1
3	0	0	1	1	!	2
6	0	1	1	0	!	
12	1	1	0	0	!	
7	0	1	1	1	!	3
13	1	1	0	1	!	

Dez.	B3	B2	B1	B0	M	Gruppe
2,3	0	0	1	-	!	1
2,6	0	-	1	0	!	2
3,7	0	-	1	1	!	
6,7	0	1	1	-	!	
12,13	1	1	0	-	!	

Dez.	B3	B2	B1	B0	M	Gruppe
2,3;6,7	0	-	1	-		1
2,6;3,7	0	-	1	-		

Bild 5.11 Iterative Durchführung des 1. Schritts nach Quine-McCluskey

Aus obigem Beispiel wird die Schaltfunktion in der Indexdarstellung bzw. in der Booleschen Form dann

$$\text{BCD1} = (2, 3, 6, 7) + (12, 13)$$

$$\text{BCD1} = /B3 * B1 + B3 * B2 * /B1$$

lauten. Dies ist, wie ein kurzer Vergleich mit dem KV-Diagramm ergibt, bereits die minimale Form, im Allgemeinen jedoch nicht.

	2	3	6	7	12	13
2,3;6,7	*	*	*	*		
12,13					*	*

Bild 5.12 Primimplikantentafel

Die dritte Schritt besteht in der Ausfüllung neuer Tafeln: Die **Primtermtabellen** bzw. **Primimplikantentafeln**. Diese Tabellen sind zweidimensional aufgebaute Darstellungen der Zusammensetzung eines Primimplikanten, mit der Zielsetzung, diejenigen Primimplikanten zu finden, so dass alle Minterme darin enthalten sind oder überdeckt werden. Die verbleibenden Primimplikanten werden dann **wesentliche Primimplikanten** genannt.

Zu diesem Zweck werden in der horizontalen Richtung alle Minterme, in der vertikalen Richtung alle Primimplikanten aufgetragen. Jeder Kreuzungspunkt, also die Zugehörigkeit eines Minterms zu einem Primimplikanten, erhält ein Kreuz.

Jetzt werden sogenannte **Kernimplikanten** gesucht; Kernimplikanten sind dadurch gekennzeichnet, dass nur sie ein Kreuz in einer Mintermspalte enthalten, wodurch sie in jedem Fall zur minimierten Form gehören (im Beispiel (2,3,6,7)). Minterme, die bei einem Kernimplikant zusätzlich auftreten, werden mit den entsprechenden Mintermen der anderen Primimplikanten durch Striche verbunden (hier: 3 und 7). Wird ein Primimplikant vollständig durch einen Kernimplikanten abgedeckt, so ist er nicht wesentlich und kann gestrichen werden (oben: (3,7)). Aus den verbleibenden Primimplikanten wird dann eine minimale Überdeckung aller Minterme ausgewählt.

Für das obige Beispiel ergeben sich dann die in Bild 5.12 mit '*' gekennzeichneten wesentlichen Primimplikanten

$$\begin{aligned} \text{BCD1} &= (2, 3, 6, 7) + (12, 13) \\ \text{BCD1} &= /B3 * B1 + B3 * B2 * /B1. \end{aligned}$$

5.3.2.3 Weitere Minimierungen

Prinzipiell lässt sich eine KDNF bzw. KKNF in eine Logik der Reihenfolge NOT-AND-XOR überführen, die quasi durch Ersatz des (inklusive) OR durch das Exclusive-OR entsteht. Eine solche Logik wird als Reed-Muller-Logik bezeichnet. Sie ergibt meist keine Verbesserung, im Einzelfall (prominentes Beispiel: Binärzähler) jedoch große Verbesserungen. Die technische Bedeutung liegt darin, dass die zugrundeliegende Hardware in PLDs gerade diese Form unterstützt.

Das Vorhandensein einer "optimalen" Implementierung mittels Reed-Muller-Logik ist nicht unbedingt einfach zu erkennen, wie die KV-Diagramme aus Bild 5.13 zeigen. Die hier zugehörigen Gleichungen lauten:

$$\begin{aligned} B0 &:= B0 \text{ XOR } 1; \\ B1 &:= B1 \text{ XOR } B0; \\ B2 &:= B2 \text{ XOR } (B1 * B0); \\ B3 &:= B3 \text{ XOR } (B2 * B1 * B0); \end{aligned}$$

Der einfachste Fall tritt auf, wenn es sich um eine "reine" Implementierung der Reed-Muller-Logik wie im Fall der Gleichung für B1 (und auch B0) handelt. Hierbei tritt ein "Schachbrettmuster" auf, das im Übrigen bei anderer Verteilung der Variablen in Bild 5.13 wirklich ein Schachbrettmuster ergeben hätte).

Weiterhin lassen sich KDNF und KKNF in ein Netzwerk von NAND-Gattern, auch mit nur zwei Inputs, transformieren, wobei die Transformation ebenfalls nicht unbedingt vereinfachend wirken muss. Der Vorteil dieses Verfahrens liegt in der Reduktion auf einen Gattertyp, der sich in Sea-Of-Gates-ASICs befindet, so dass das Technologie-unabhängige Optimieren mit dem -abhängigen Verfahren in Übereinstimmung gerät.

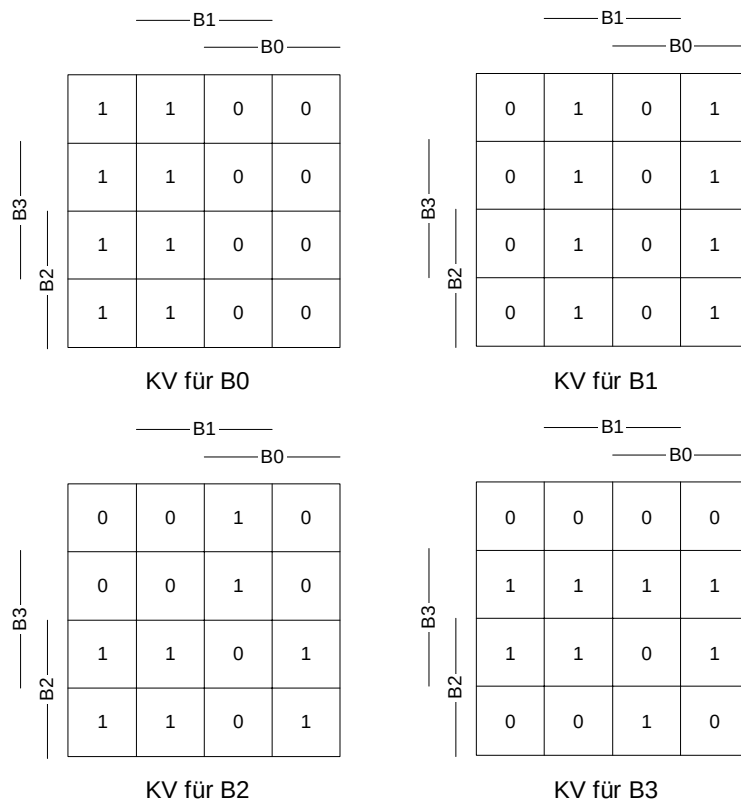


Bild 5.13 KV-Diagramme für 4-Bit-Binärzähler

Abgesehen von diesem Sonderfall bleiben aber weitere Verfahren, die Technologie-unabhängig sind, auf das Erkennen von Redundanzen bzw. gemeinsamen Teilen beschränkt. Die Technologie-abhängigen Verfahren, im nächsten Unterabschnitt zumindest teilweise erwähnt, berücksichtigen die jeweilige Architektur und schaffen so das optimale Logiknetzwerk.

5.3.3 Mappingalgorithmen

Der Schritt, der auf die Logikoptimierung und die Eliminierung von Redundanzen folgt, ist naturgemäß technologieabhängig: Die zugrundeliegende Architektur des Zielbausteins wird benutzt, um die optimierte Logikfunktion aufzunehmen. Da aber, wie in Kapitel 3 bereits gezeigt wurde, die Architekturen z.T. erheblich voneinander abweichen, bleibt nur ein technologieabhängiges Vorgehen.

Damit stellt sich aber eine andere Frage, die im Rahmen dieses Curriculums nur angesprochen, nicht aber beantwortet werden kann: **Wie wird der Baustein, der die**

Schaltung trägt, letztlich ausgewählt? Die Antwort liegt fast immer in der Erfahrung des Designingenieurs sowie anderen, subjektiven Kriterien. Eine maschinell gangbare Methode wäre natürlich ein mehrfaches Mapping; jenes scheidet weniger aus Zeitgründen aus, vielmehr sind die Beschreibungssprachen für optimales Mapping herstellerabhängig, Hersteller-unabhängige Tools bieten nach wie vor nur suboptimale Ergebnisse.

Im Folgenden sollen für drei Zielsysteme, die Sea-Of-Gates-Struktur mit NAND-Gattern, häufig bei ASICs anzutreffen, die Look-Up-Table-Struktur z.B. in Xilinx XCs und die Multiplexerstruktur wie z.B. bei Act1-, Act2- und Act3-ICs, Algorithmen zum Auffinden von Mappings dargestellt werden.

5.3.3.1 Library-basierte Mappingtechnologie

Zunächst wird davon ausgegangen, dass das Original-Netzwerk in eine kanonische, 2-Input-NAND-Gatter (im Folgenden mit NAND2 bezeichnet) basierte Zerlegung transformiert wurde. Dieses Boolesche Netzwerk, engl. Direct Acyclic Graph (DAG) genannt, enthält keine Zyklen mehr. Dies kann in jedem Fall geschehen, da die NANDs eine logische Basis bilden. Der Weg hierhin kann beispielsweise über KDNF oder KKNF führen, die AND-, OR- und NOT-Gatter lassen sich einfach durch NANDs ersetzen.

Diese Zerlegung besitzt den Vorteil, dass dafür Bibliothekselemente existieren müssen, sofern ein NAND2-Element existiert (wovon man grundsätzlich ausgehen kann). Andererseits existieren im Allgemeinen viele Zerlegungen in eine derartige Form, wobei keine Aussagen oder Annahmen über den optimalen Zustand gemacht werden.

Die vorhandene Zerlegung wird in einen 'Wald von Bäumen' zerlegt; während im allgemeinen mehrere Outputs vorhanden sein werden, wird nunmehr jedes Teilnetz so gewählt, dass er nur noch einen einzigen Output hat (Single-Output-DAG). Die Zerlegung kann ebenfalls soweit gehen, dass mit Ausnahme der Inputs kein Knoten des Baums mehr als einen Ausgang hat; durch die Duplizierung der Eingänge kann man daraus einen Binärbaum erhalten, indem für jeden Fanout des Eingangsknoten eine eigene Instanz gebildet wird.

Die entstandenen Binärbäume werden nun einzeln auf die vorhandene Hardware abgebildet, indem Teile oder der komplette Baum mit Bibliothekselementen verglichen werden. Grundlage zur Auswahl der besten Abbildung bildet eine Kostenfunktion, die mit Bibliothekselementen verbunden ist. Diese Kostenfunktion kann sowohl die Anzahl der genutzten Logikgatter wie auch ggf. die Schaltgeschwindigkeiten umfassen.

Prinzipiell werden alle möglichen Abbildungen gesucht und deren 'Kosten' berechnet. Diese Methode ist zwar zeitaufwendig, führt aber zu optimalen Ergebnissen, sofern die Bibliothek entsprechend gestaltet wurde. Obendrein stellt die Bibliotheksmethode einen großen Vorteil dar, da ASIC-Hersteller ihre Bibliotheken für

die Bausteinfamilien anbieten können, Hersteller-unabhängige Anbieter jedoch die Logik- und Mapping-Programme.

Folgendes Beispiel, wiederum für die Binär-BCD-Codierung, Bit BCD1 durchgeführt, möge die Ausführungen untermauern:

Bild 5.14 zeigt zunächst eine der möglichen NAND2-Zerlegungen, die z.B. aus der Umsetzung der KDNF erhalten werden kann:

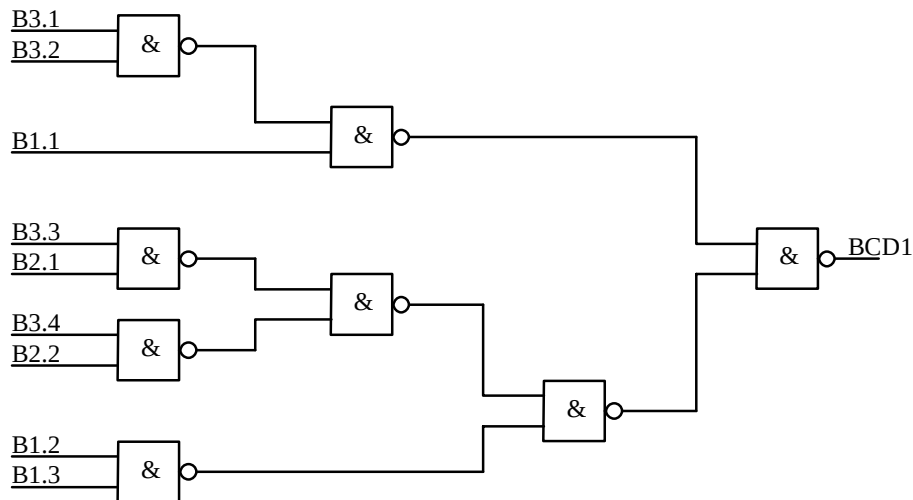


Bild 5.14 Binär-BCD-Codierer in NAND2-Form

Die einzelnen Instanzen der Eingänge sind mit einer fortlaufenden Subnummer versehen, um eine entsprechende automatische Bearbeitung zu ermöglichen. Ferner sollte beachtet werden, dass eine Minimierung bereits durchgeführt werden konnte; bei der Umsetzung der Disjunktion in NAND2-Gatter wurde die Invertierung bereits in der vorigen Stufe durchgeführt, dies konnte berücksichtigt werden.

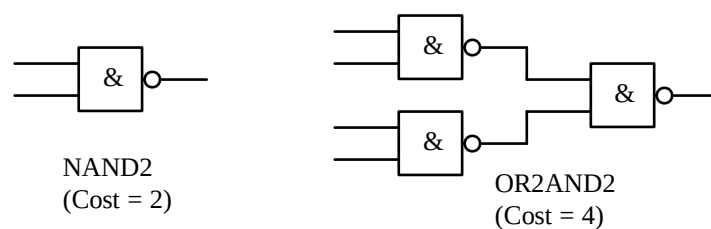


Bild 5.15 Modellbibliothek mit 2 Elementen

Bild 5.15 zeigt einige denkbare Bibliothekselemente mit einer angenommenen Kostenfunktion, die auf der Zählung der Eingänge beruht. Derartige Bibliotheks-

funktionen werden vom IC-Hersteller angeboten. Die darin vorhandenen Elemente entstammen beispielsweise der Tatsache, dass der Baustein über andere Logikelemente verfügt, z.B. 4-Input-NANDs (NAND4):

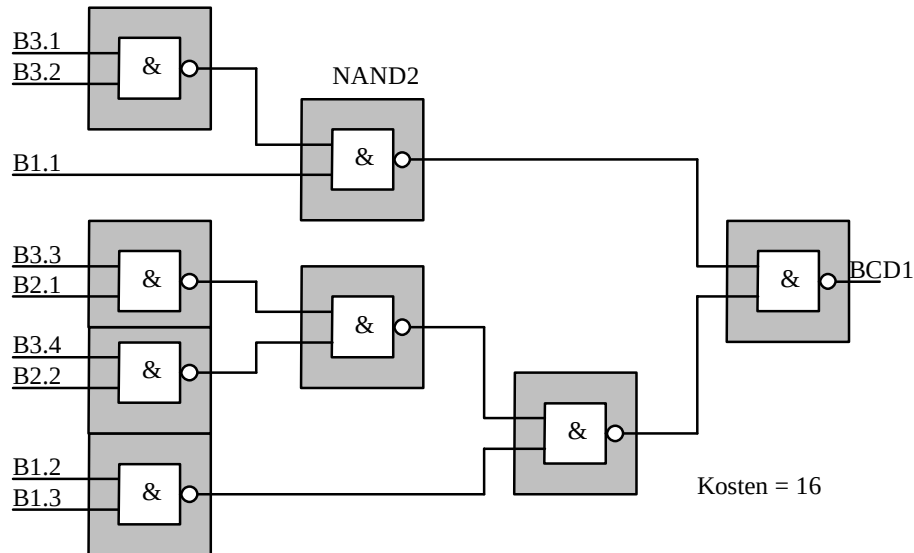


Bild 5.16 Mapping mit NAND2-Elementen für die BCD1-Funktion

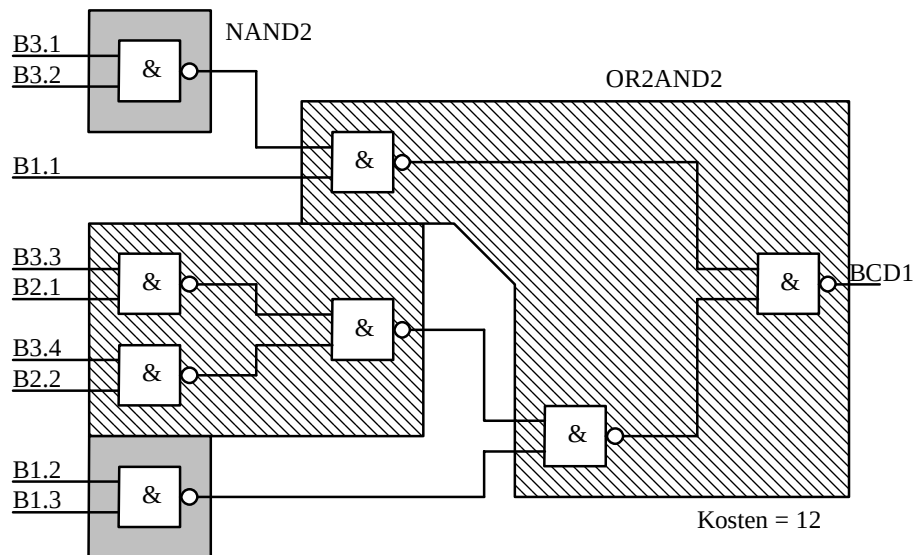


Bild 5.17 Mapping mit NAND2- und OR2AND2-Elementen für die BCD1-Funktion

Das in der Bibliothek aufgefunden Element OR2AND2 hat die Eigenschaft, genau dann am Ausgang 0 zu präsentieren, wenn an mindestens einem Eingang beider Teilzweige eine 0 anliegt; m.a.W.: Die ODER-Verknüpfung zweier UND-Gatter mit je zwei Eingängen.

Diese Bibliothek kann z.B. die in Bild 5.16 und 5.17 dargestellten Mappings mit den dazugehörigen Kosten erzeugen. Die Kostenfunktion, die sich nur auf die Anzahl der Eingänge bezieht, ist naturgemäß auch anderweitig definierbar, z.B. für die Geschwindigkeit der Signale.

5.3.3.2 Look-Up-Table-basierte Mappingtechnologie

Logikzellen, die auf dem Inhalt von sogenannten Look-Up-Tabellen (LUT) basieren, sind u.a. in XC4000 der Firma Xilinx zu finden. Diese Tabellen stehen im Allgemeinen in RAM-Zellen und definieren das Verhalten der Zelle auf beliebige Weise.

Für eine k -Input, 1-Output-LUT ergeben sich damit 2^k verschiedene Kombinationen, von denen allerdings etliche durch Permutation sowie durch Eingangs- und Ausgangsinvertierung ineinander überführbar sind, so dass ein Bibliothekssystem ggf. nur einen Bruchteil beinhalten muss. Ein Programm, das die Übereinstimmung eines Bibliothekselements mit einem Teilnetz überprüfen soll, benötigt hingegen alle Varianten und muss dementsprechend die Bibliothek wieder expandieren.

Im Folgenden wird ein Algorithmus näher betrachtet; dieser wurde für k -Input, 1-Output-LUTs entwickelt und optimiert. Die Xilinx-Bausteine beinhalten üblicherweise 4 Inputs mit 1 Output, gelegentlich auch 5 Inputs mit 2 Outputs. Letztere benötigen dementsprechend spezielle Mapping-Programme mit anderen Prioritäten als die hier vorgestellten Algorithmen. Prinzipiell lässt sich das Vorgestellte aber übertragen.

Die Strategie eines LUT-Mapping-Algorithmus kann zwei Richtungen verfolgen:

- Die Anzahl der LUTs kann minimiert werden (**geringster Platzbedarf**)
- Die Anzahl der LUT-Ebenen kann minimiert werden (**größte Geschwindigkeit**)

Beide Strategien schließen sich im Normalfall gegenseitig aus. **Der Chortle-crf-Algorithmus**, der nun betrachtet wird, verfolgt die Strategie des Minimierens der LUT-Anzahl.

Ausgangspunkt für den Algorithmus ist wiederum eine Zerlegung des Netzwerks in Bäume, jeder Baum wird wieder für sich betrachtet. Die Darstellung der Booleschen Funktionen im Baum selbst erfolgt mittels AND-, OR- und NOT-Gatter. Bild 5.18 gibt ein Beispiel.

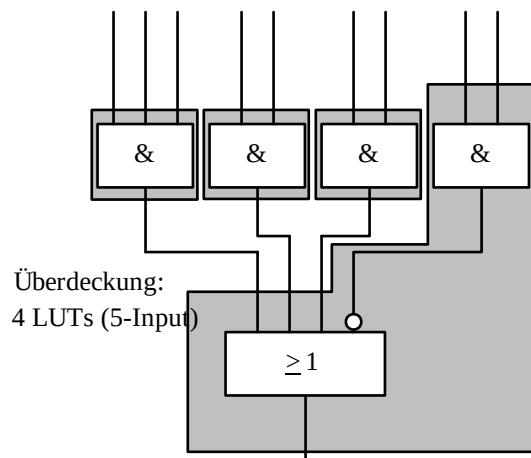


Bild 5.18 Boolesches Netzwerk mit LUT-Überdeckung

Die Überdeckung durch Look-Up-Tables wurde unter der Annahme erreicht, dass 5 Inputs bei 1 Output pro LUT zur Verfügung stehen. Diese Überdeckung entstand spontan, sie erfüllt alle Bedingungen und soll nun minimiert werden.

Zu diesem Zweck wird versucht, die Wurzel-LUT zu teilen, um mehr als nur eine der oberen LUTs in diese miteinbeziehen zu können. Das Ergebnis sieht dann so aus:

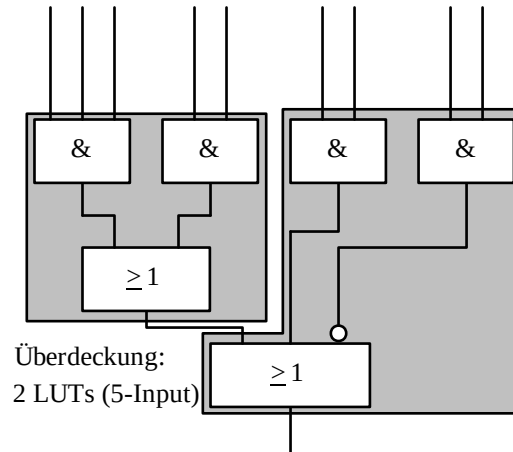


Bild 5.19 Boolesches Netzwerk mit optimierter LUT-Überdeckung

Der Algorithmus für diesen Zweck geht von der Wurzel-LUT (Root-LUT) aus und versucht, diese unter Berücksichtigung zweier Ziele (in der Gewichtung) aufzuteilen:

1. Die Anzahl der benutzten LUTs soll minimal werden.
2. Die Anzahl der Eingänge in der neuen Wurzel-LUT soll minimal werden.

Das zweite Ziel bestimmt implizit, dass das darüber liegende Teilnetz optimal implementiert werden muss.

Der Algorithmus konstruiert zunächst aus einer LUT eine verbleibende Rest-LUT mit einer geringeren Zahl an Eingängen sowie einer 'Second-Level'-LUT, die den Rest aufnimmt. Die neue Root-LUT wird derart konzipiert, dass sie direkt an die Eingänge des Netzwerks reicht (bis zur Zahl k , hier immer zu 5 angenommen), andererseits eine minimale Anzahl von Eingängen zu LUTs aufweist, die nicht der obersten Reihe (Blättern, Leaf) angehören. Diejenige Aufteilung, die die geringste Anzahl an LUTs aufweist, gilt als optimal; bei gleicher Anzahl wird anhand der minimalen Anzahl der Eingänge der Rest-Root-LUT entschieden.

Anschließend wird mit dem verbleibenden Restnetz entsprechend verfahren. Das Problem wurde als äquivalent zum Bin-Packing-Algorithmus erkannt, bei dem die minimale Anzahl von Abdeckungen gesucht wird, um einen Satz von Größen komplett abzudecken, wobei die Summe der Größen in jedem Teil eine bestimmte Konstante nicht überschreitet.

Anschließend werden – bei kompletten Netzwerken mit mehreren Ausgängen – weitere Optimierungen durchgeführt. Hierzu gehören

- Zusammenfassung rekonvergenter Teile (gemeinsame Eingänge sowie Ausgang) zu einer LUT
- Duplizierung von Logikteilen zur Integration in LUTs

5.3.3.3 Mappingalgorithmen für Multiplexerstrukturen

Die in Abschnitt 2.1.2 dargestellte Multiplexerstruktur wird beispielsweise in der Act2-Familie von Actel genutzt. Für die kombinatorischen Elemente ist diese Multiplexerfunktion in Bild 5.20 dargestellt. Diese Funktion lässt sich als Binärbaum (**BDD, binary decision diagram**) darstellen, wobei – bedingt durch die Logikverknüpfung der auswählenden Variablen – mehrere Darstellungen dieses Binärbaums möglich sind. Ein BDD ist ein direkter azyklischer Graph mit einer Wurzel und 2 Endknoten (0 und 1). Gerade diese Bedingung eines Wurzelknotens bewirkt, dass es mehrere BDDs geben kann, abhängig von der Wahl dieses Knotens.

Aus diesem Grund wird eine etwas vereinfachte Darstellung gewählt:

$$z = /a * (/b * c + b * d) + a * (/e * f + e * g)$$

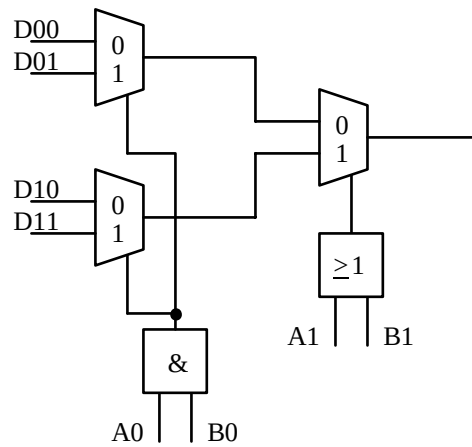


Bild 5.20 Multiplexerstruktur der kombinatorischen Logikmodule Act2

Bild 5.21 stellt diese vereinfachte Multiplexerstruktur und den dazugehörigen BDD dar:

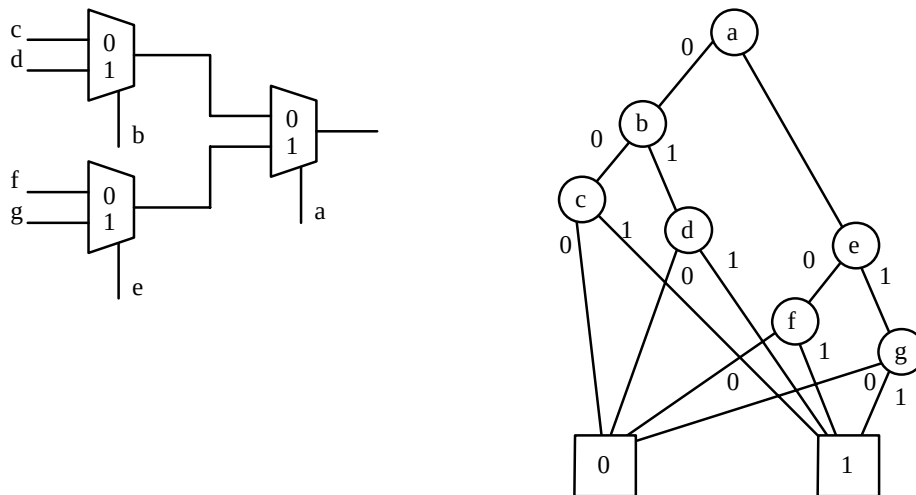


Bild 5.21 Vereinfachte Multiplexerstruktur und Binary Decision Diagram

Die Mappingalgorithmen versuchen nun, durch Isomorphie-Untersuchungen der gewünschten Funktion, ebenfalls als BDD dargestellt, mit dem BDD des Logikmoduls bzw. Teilen davon diese in ein Logikmodul unterbringen. Für die obige vereinfachte Multiplexerstruktur kann beispielsweise die Äquivalenzfunktion zweier Schaltvariablen $(x * y + /x * /y)$ durch

$a = x$, $b = y$, $e = y$, $c = 1$, $d = 0$, $f = 0$ und $g = 1$ erreicht werden. Wird die Isomorphie gefunden, so gilt die Funktion als in ein Logikmodul gemappt, ansonsten werden zwei oder mehrere Module miteinander verbunden.

6 Spezielle Themen des Configurable Computing

Nachdem nun die klassischen Bausteine aus dem Bereich *Configurable Computing* dargestellt worden sind, soll im abschließenden Kapitel die Aufmerksamkeit auf spezielle Themen gerichtet werden. Das spezielle Thema schlechthin heißt dabei *Reconfigurable Computing*.

Um dieses Thema zumindest in Ansätzen umfassen darzustellen, wird im ersten Abschnitt versucht, die verschiedenen Hardwareansätze im Reconfigurable Computing zu klassifizieren. Diese Klassifizierung ist bislang nicht allgemeingültig, so dass hier noch zukünftige Änderungen zu erwarten sind.

Im zweiten Teil wird dann Compilertechnologie behandelt. Hierbei geht es darum, Ansätze zu beschreiben, die aus einer Softwarebeschreibung ausführbaren Code erzeugen. Die Schwierigkeit ist dabei, dass weder sequenzielle Programme noch Konfigurationen für Reconfigurable Computing geeignet sind: Es muss eine Folge von Konfigurationen erzeugt werden, und dies beinhaltet auch den Wechselzeitpunkt. Dies stellt ein neues Schedulingproblem dar.

6.1 Klassifizierung der Hardware

Abgesehen von den programmierbaren Logikbausteinen (PLDs), deren intrinsische Datenbreite das Bit ist und die als komplette Bausteine erhältlich sind, sind weitere Ansätze zu Architekturen, die in das Gebiet des Configurable Computing fallen, sichtbar. Eine gewisse Klassifizierung war bereits in Kapitel 2, insbesondere Abschnitt 2.4 gegeben: Diese bezog sich im Wesentlichen auf die intrinsische Wortbreite und die damit verknüpften Operationen (arithmetisch, logisch).

Hier wird versucht, eine Gesamtklassifizierung zu geben. Dabei sollte berücksichtigt werden, dass dies zurzeit in fließender Diskussion ist, so dass hier nicht das 'Endergebnis' bekannt gegeben werden kann.

Bild 6.1 zeigt eine aktuelle Einteilung der Architekturen im Bereich des Configurable Computing. Die wesentliche Einteilung in Fine-Grain- (Bit als intrinsische Verarbeitungsbreite) und Coarse-Grain-Architekturen (Wort als intrinsische Verarbeitungsbreite) ist anerkannt, wobei im Allgemeinen keine Aussagen zu der optimalen Anwendbarkeit gemacht werden können (in speziellen Anwendungsgebieten allerdings doch, z.B. in der digitalen Signalverarbeitung).

Die programmierbaren Logikbausteine sind die bekannteste Form der CC-Devices, die übrigen müssen kurz erläutert werden:

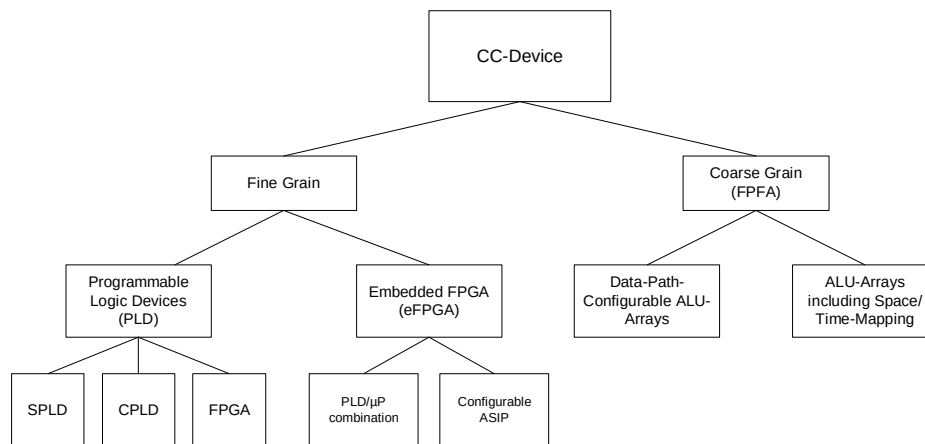


Bild 6.1 Klassifizierung der Architekturen im Bereich Configurable Computing

Der Bereich der *embedded FPGAs* wird derzeit als einer von zwei Ansätzen für die Zukunft angesehen. Hiermit werden FPGA- (bzw. PLD-)ähnliche Strukturen bezeichnet, die in Bausteinen als konzentrierter Kernel oder verteilt vorhanden sind. Die Bausteine selbst beinhalten aber noch andere Strukturen wie beispielsweise Mikroprozessoren bzw. Mikrocontroller. Die Form der Kopplung mit dem Prozessorteil unterscheidet dann auch die beiden Implementierungen:

μP/PLD-Mischung

Hiermit wird ein Nebeneinander von Mikroprozessor (der klassischen Bauart) und eFPGA-Struktur mit entsprechender Kopplung bezeichnet. Im Designprozess bedeutet dies, dass eine Partitionierung (im Sinn des Hardware/Software Co-Design) in Aufgaben für den Mikroprozessor und den eFPGA-Teil erfolgen muss, bevor beide Teile getrennt beschrieben werden.

Im eFPGA-Teil können dann typische Hardwareteile wie Schnittstellenbedienung, Interrupt Service etc. implementiert werden, oder man nutzt diesen Teil zur Beschleunigung von Rechnungen (Configurable Hardware Accelerator).

Der wesentliche Vorteil dieses Ansatzes liegt darin, dass er aus Sicht der Hardware sehr schnell aufzubauen ist. Problematisch ist hier die Entwicklung, die in getrennten Wegen erfolgen muss. Beispiel für μP/PLD-Kombinationen ist die AT94k-Serie von Atmel: Hier ist ein 8-bit-Mikroprozessor (AVR) mit einem FPGA (ca. 40000 Systemgatterfunktionen) kombiniert.

Configurable ASIP

ASIP bedeutet *Application-Specific Instruction Set Processor*, ist also die Bezeichnung für Prozessoren, deren Befehlssatz an die Applikation angepasst wurde. Hier bietet es sich an, neben einem allgemeingültigen Befehlssatz den spezifischen

Teil konfigurierbar zu gestalten, so dass eine Hardwareplattform an viele Applikationen angepasst werden kann.

Bei diesem Ansatz ist die eFPGA-Struktur dementsprechend eng mit dem Mikroprozessorkern verbunden, insbesondere mit dem Registersatz des Prozessorkerns. Diese Strukturen lassen sich auch nicht allgemein verwenden, sondern nur zur Implementierung von anwendungsspezifischen Instruktionen.

Diesem Ansatz wird zurzeit große Chancen im Bereich des High-Performance-Computing bei geringem Energieaufwand eingeräumt. Die Entwicklung des anwendungsspezifischen Instruktionssatzes und der Applikation kann zumindest theoretisch gemeinsam erfolgen, entsprechende Tools befinden sich in der Forschung. Beispiel für diese Klasse sind die Xtensa-Prozessoren der Firma Tensilica ('Extensible Processors') und die S5-Prozessoren der Firma Stretch [32], die auf diesem Prozessormodell aufbauen und es um einen konfigurierbaren Datenpfad ISEF (Stretch Instruction Set Extension Fabric) erweitern..

Data-Path-configurable ALU-Arrays:

Dieser Ansatz gehört in die Klasse der Coarse-Grain Architekturen, da hier die intrinsische Verarbeitungsbreite im Wort (16 bis 32 bit) liegt. Die Struktur besteht meist aus zweidimensionalen Arrays von ALUs und Registern [33, Abschnitt 10.3]. Ein Algorithmus wird auf dieses Array verteilt, der Datenstrom bestimmt die Verarbeitungsfrequenz innerhalb einer Konfiguration.

Dieser Ansatz wird insbesondere bei Algorithmen im Bereich der digitalen Signalverarbeitung favorisiert, da hier große Datenmengen von meist begrenzten Algorithmen (Filter etc.) bearbeitet werden. Eine automatische Generierung der Konfiguration und insbesondere eines Konfigurationsflusses aus einer Softwarehochsprache heraus (z.B. C) befindet sich derzeit in der Forschung bzw. Entwicklung. Beispiel für diese Klasse ist die XPP-Architektur [33, Abschnitt 10.3] der Fa. PACT, München.

ALU-Arrays mit Space-Time-Mapping

Eine besondere Klasse von Data-Path-konfigurierbaren ALU-Arrays wird noch von der Allgemeinheit unterschieden: Die ALU-Arrays mit Möglichkeit zum Space-Time-Mapping [33, Abschnitt 10.2]. Hiermit werden Hardwarearchitekturen beschrieben, die eine Ausführung eines Binärcodes in der Sequenz einer Instruktionsfolge oder durch Abbildung auf ein ausführendes Array ermöglichen. Dies bedeutet den dynamischen Wechsel in der Ausführungsdimension und stellt eine hohe Herausforderung dar.

Die damit verbundenen Chancen liegen darin, dass man auch den Instruktionsfluss des zugehörigen Mikroprozessormodells zur Konfiguration nutzen kann, da dieser ja auch in der Fläche ausführbar ist. Somit wird die Compilertechnologie erheblich erleichtert: Nur die optimierte Compilierung für das Mikroprozessormodell ist notwendig, um zugleich Konfigurationen zu schaffen.

Architekturen dieses Typs befinden sich zurzeit nur in der Forschung, kommerzielle Produkte sind bislang nicht angekündigt.

6.2 XPP-Architektur (Fa. PACT)

Um eine Grundlage für die folgende Diskussion zur Compilertechnologie zu besitzen, wird hier kurz die von PACT [30] vorgestellte Architektur, mit XPP bezeichnet (*eXtreme Processing Platform*). Diese Plattform ist als Coprozessor konzipiert, mit Hilfe eines RISC-Prozessorkerns, der auch auf dem Chip integriert ist, wird hieraus ein vollständiger Prozessor mit enormen Rechenkapazitäten.

Die XPP-Architektur weist 4 Ebenen der Hierarchie auf, die der Hardware-Objekte, der Processing Array Elements (PAE), der Processing Array Cluster (PAC) und die der Devices selbst. Die Hardware-Objekte beinhalten elementare Operationen wie Verknüpfungen, Speicher oder Routing. Aus Sicht der Konfiguration sind dies die Ziele, mit denen die Laufzeit-Struktur des Bausteins festgelegt werden.

Der/die Softwareentwickler/in sieht hingegen die Processing Array Elements, die aus Objekten zusammengesetzt sind. Er/sie kann ein PAE für den Algorithmus nutzen, teilweise sogar die Objekte darin. Der Algorithmus wird aber nicht nur auf ein PAE, sondern auf den Cluster (PAC) oder sogar auf das gesamte Device abgebildet. Diese nächsthöheren Ebenen dienen damit u.a. der Chipverwaltung und der Skalierbarkeit, weniger der Programmierung.

6.2.1 Hardware-Objekte

Auf unterster Ebene existieren 9 verschiedene Hardware-Objekte, das ALU-Object, RAM-Object, Forward-Register-Object (FREG), Backward-Register-Object (BREG), das Programmable Gate Array Object (PGA), Data Channels, Event Channels, Network Switch Object und die I/O-Objects. Das ALU-Object (siehe Bild 6.2) besteht im Innern aus einem rekonfigurierbaren Zustandsautomaten, bei dem in den Eingangsregistern Daten und Events gespeichert werden. Die Art der Verarbeitung ist dann abhängig von der Konfiguration, denn die ALU wird auf die üblichen Verknüpfungen wie Addition, Multiplikation etc. eingestellt. Nicht nur das Ergebnis, sondern auch Events für nachfolgende Einheiten sind im Ausgangsregister speicherbar.

Die Eingangsdaten und -events stammen aus dem horizontalen Routing, die Ausgänge werden wieder in dieses Routing geführt. Je Richtung sind hierfür 6 Leitungssysteme (32 Bit für Daten) vorgesehen. Die ALU kann dabei maximal 3 Eingangsbusse zu 2 Ausgängen verarbeiten, beispielsweise in Form der MAC-Operation, die $32 * 32$ Bit multipliziert und zu einem 32 Bit Wert addiert, die Ausgangsbreite beträgt dann 64 Bit.

Forward- und Backward-Register Object dienen dem vertikalen Routing im Rahmen einer höheren Einheit, dem Processing Array Cluster (PAC, siehe Bild

Das RAM-Object enthält lokales SRAM für Look-Up Table innerhalb von Algorithmen und zur Speicherung von Ergebnissen. Die Speichertiefe einer derartigen RAM-Zelle ist naturgemäß begrenzt, aktuell werden pro Zelle 256 Worte (à 32 Bit) gespeichert. Dies bietet für viele Algorithmen gerade aus der Signalverarbeitung (denken Sie an Filter- und Transformationsalgorithmen) genügend Raum, um Koeffizienten und andere Konstanten nahe an den Rechenwerken zu lagern.

Reicht der Speicherplatz nicht aus, bietet sich die Möglichkeiten, über die I/O-Objekte externes SRAM mit nahezu beliebiger Größe anzuschließen. Diese I/O-Objekte sind den PACs zugeordnet, in der aktuellen Konfiguration der XPU128-ES hat jedes 2 Ports à 32 Bit. Ein derartiges I/O-Object kann in einem speziellen RAM-Mode betrieben werden, sodass Adress- Steuer- und Datenleitungen spezifisch zugewiesen sind.

Interessant an den aufgezählten Hardware-Objekten ist, dass auch FPGA-ähnliche Strukturen in der XPP-Architektur existieren. Diese PGAs waren in den ursprünglichen Darstellungen noch nicht vorhanden, sind nunmehr jedoch vorgesehen, da sich Kontrollstrukturen, insbesondere auf Bitebene, besser so implementieren lassen.

Das letzte der aufgezählten Hardwareobjekte ist das Network Switch Object (Switch-Object). Zwei Arten von Kommunikation sind in der XPP-Architektur zu finden: Daten und Ereignisse. Beide werden über horizontale Signalleitungen, die stückweise in Teileinheiten der XPP-Architektur vorhanden sind, verteilt. Zur Konfiguration eines Netzwerks zwecks Kommunikation sind dann die Switch-Objects notwendig, die ihrerseits die Leitungstücke miteinander verbinden bzw. voneinander trennen. Man kann es also so formulieren: Das horizontale Routing ist segmentierbar, das vertikale Routing über FREG und BREG einrichtbar.

6.2.2 Processing Array Elements und Processing Array Cluster

Bild 6.3 zeigt die nächste Hierarchieebene in der XPP-Architektur, das Processing Array Element (PAE). Dieses Element ist aus den Hardwareobjekten sowie Signalleitungen zusammengesetzt, und es existieren zwei wichtige Typen: ALU-PAE und RAM-PAE. Beide PAE-Typen sind nahezu identisch aufgebaut, der Unterschied liegt in dem RAM- bzw. ALU-Object.

Die PAEs besitzen drei wesentliche Kommunikationssysteme: Datenleitungen in der generischen Breite (32 Bit), Eventleitungen mit 1 Bit Breite und den Konfigurationsbus. Die PAEs stellen damit das operative Herzstück der XPP-Architektur dar, denn hier wird die Operation, das lokale Speichern und das Datenrouting konfiguriert, und die Events steuern den Datenfluss innerhalb des gesamten Rechners.

Diese Eventsteuerung ist ausgesprochen wichtig, denn die Gültigkeit der Operationen ist von der Verfügbarkeit der Daten abhängig. Die Events steuern den

Datenfluss, indem z.B. ALU-Operationen gestattet oder unterdrückt werden. Hierzu sind die Eventleitungen RESET, STEP, STOP, GO und RELOAD an jeder ALU definiert, die entsprechende Operationen auslösen, während die ALU selbst die Events U, V und W (abhängig von der definierten Operation) erzeugt. Die generierten Events können den klassischen Flags entsprechen, sie können auch (im Rahmen) selbst definiert werden.

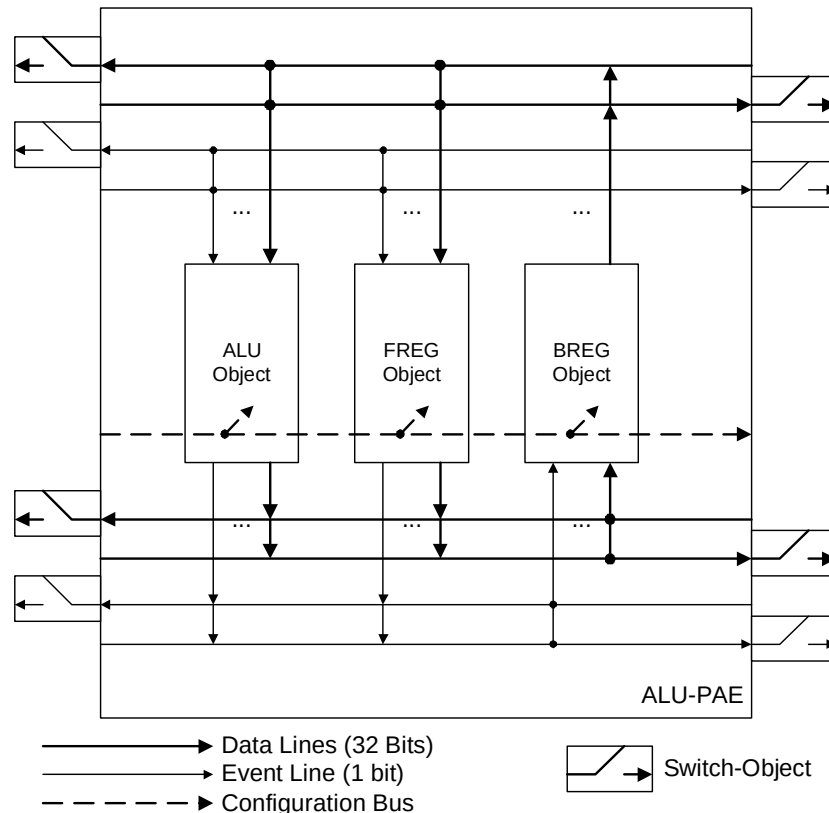


Bild 6.3: Processing Array Element (ALU-Typ) mit Switch Objects

Einige von diesen PAEs sind zusammen mit I/O-Objekten, Connect-PAEs (CON-PAE) zur Verbindung mit anderen Clustern und diversen Switch-Objekten zu einem Processing Array Cluster (PAC, siehe Bild 6.4) zusammengefasst. Beim Referenzchip XPU128-ES sind dies 64 ALU-PAEs und 16 RAM-PAEs (aber kein PGA-PAE), und zu jedem PAC gehören 4 I/O-Objekte mit je 2 Ports á 32 Bit. Bei anderen Devices kann die Zusammensetzung eines PACs durchaus variieren. Jeder PAC ist mit einem Configuration Manager (CM) gekoppelt, dessen Aufgabe in der dynamische (Re-)Konfiguration besteht.

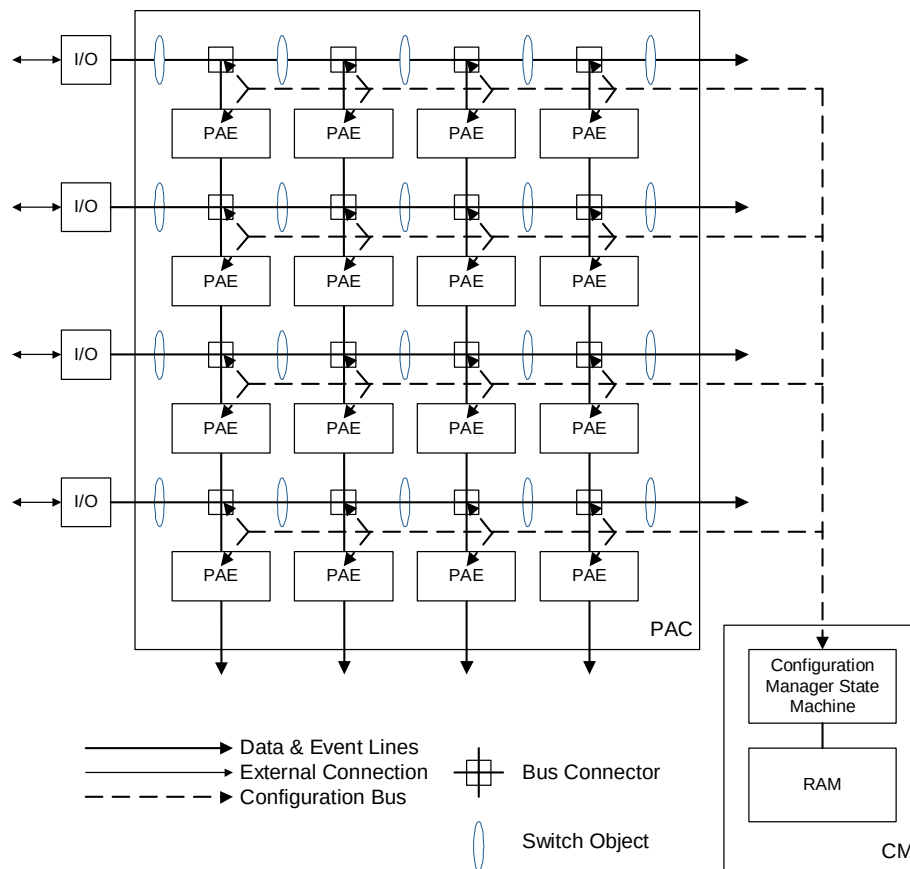


Bild 6.4: Aufbau Processing Array Cluster (PAC) der XPP-Architektur (ohne CON-PAE)

In einem XPP-Device sind dann mehrere PACs integriert, in der aktuellen XPU128-ES sind es 2 PACs, bei der XPU192 4 (kleinere PACs). Die CMs selbst werden hierbei mit einer supervising CM (SCM) verbunden, um den Programm- und Konfigurationslauf im gesamten XPP-Device steuern zu können. Es besteht auch die Möglichkeit, mehrere XPP-Chips über die I/O-Objekte miteinander zu koppeln. In diesem Fall muss eine externe SCM definiert werden, damit auch die Kopplung wie ein ausführendes Device erscheint.

Wichtig ist, dass die PACs das aktuelle Ziel der Programmierung in einer Hochsprache darstellen: C-Programme zielen auf ein PAC einschließlich des externen RAM.

6.3 Programmiermodell im Vergleich zu CC (4.2)

Die wohl wichtigste Komponente für das Design von Applikationen mithilfe der neuen Bausteine ist das Programmiermodell, das Designern zur Verfügung gestellt wird. Hierbei ist alleine der Begriff *Programmiermodell* weit definierbar: Mikroprozessorarchitekten verstehen hierunter etwas anderes als Softwaredesigner. Aus diesem Grund sei hier eine Definition [34] angegeben, die keinen Anspruch auf Vollständigkeit erhebt, sondern lediglich für dieses Skript Gültigkeit erhält:

Definition 6.1:

Ein (high-level) *Programmiermodell* beschreibt die Systemplattform zum Ablauf von Software auf einer Ebene, die von Details wie Bussysteme, CPU-Modell, low-level Peripherie etc. abhebt. Zur Beschreibung gehören u.a. die Definition von Softwarekomponenten zum Erreichen von Aktionen auf der Systemplattform, das Interface zwischen diesen Softwarekomponenten und das zeitliche Verhalten der Systemplattform.

Für eine detaillierte Behandlung dieses stark im Fluss befindlichen Gebiets müssen nun die verschiedenen Deviceformen unterschieden werden.

6.3.1 Programmiermodell für hybride Bausteine (µP/PLD)

Ursprünglich sah man den PLD-Anteil innerhalb von µP/PLD-Kombinationen als einen von der CPU unabhängigen Beschleuniger für bestimmte Aufgaben an [34]. Beide Komponenten können dann beliebig über I/O-Queues miteinander verbunden werden, was letztlich eine lose Kopplung bedeutet.

Das Schreiben einer Applikation bedeutet demnach, Teile zu identifizieren, die in den jeweiligem Hardwareteil zum Ablauf kommen: Die 'klassische' Partitionierungsaufgabe. Die Identifizierung der Teile kann z.B. anhand des zeitlichen Verhaltens geschehen.

Dieses Programmiermodell ist auch aktuell gültig, da es besonders einfach ist und für viele Anwendungen auch ausreicht. Leider nicht für alle, denn wenn es um mehr als das pure Zusammenspiel von unabhängigen Teilen geht – etwa um die Beschleunigung einer Applikation – funktioniert dieser Ansatz nicht mehr.

Aus diesem Grund wird das System – softwareseitig – als ein System von Threads mit einer entsprechenden Kommunikation aufgefasst [34]. Die Kommunikation soll in diesem Fall auch synchronisieren können, was schnell zum Message Passing bzw. Shared-Memory-Modell führt: Beide Mechanismen haben Vor- und Nachteile (z.B. sichere Entkopplung versus schnelle Datenübertragung bei großen Mengen), sind aber im Wesentlichen gleichberechtigt.

Einhergehend mit dem Design der Applikation als Multithreaded-System müssen Randbedingungen für die einzelnen Threads festgehalten werden, die eine Ent-

scheidungsgrundlage für die Abbildung liefern (können und müssen). Die Randbedingungen sind meist zeitlicher Natur, energetische und andere spielen aktuell keine praktische Rolle, was sich aber ändern könnte.

Bei diesem Modell bleiben dennoch einige herausfordernde Probleme:

- Um eine optimale Abbildung zu ermöglichen, muss die Beschreibung der Threads in einer Abbildungs-unabhängigen Sprache erfolgen; zudem benötigt diese Sprache eine Eingabemöglichkeit für Randbedingungen, möglichst sogar in funktionaler Form (d.h. durch ein Programm in echte Aktionen umsetzbar).
- Die Abbildung muss möglichst maschinell erfolgen, und sie soll – wenn möglich – zu einem Ergebnis führen.

Der zweite Punkt ist vergleichsweise unkritisch. Die Abbildungsstrategie ist zu meist so, dass zu Beginn alles im Prozessor abläuft. In einem iterativen Prozess wird solange auf den PLD-Teil abgebildet, bis alle Randbedingungen der Aufgabenstellung (z.B. Echtzeitfähigkeit, Performance) erfüllt sind. Dieses Verfahren, aus dem Gebiet des Hardware/Software Co-Design stammend, benötigt natürlich möglichst genaue Schätzungen über den Ressourcenverbrauch für Threads (Zeit auf CPU, Fläche auf PLD) und für die Interthread-Kommunikation.

Der erste Punkt ist prinzipieller und wesentlich kritischer: Eine derartige Sprache ist als Produktionssprache nicht vorhanden. Modellierungssprachen wie UML erfüllen dies zwar (in etwa), können aber nicht direkt (zumindest nicht effizient) in Code umgesetzt werden.

Die Zeit als funktionaler Parameter ist selten implementiert (z.B. in einer Weiterentwicklung von C zur Bestimmung von Worst-Case Execution Times, WCET, [35]), und die wenigen Implementierungen sind sehr singulär.

6.3.2 Programmiermodell für Array-basierte Bausteine

Array-basierte Bausteine, auch als FPFA (Field-Programmable Functional Arrays) bezeichnet, besitzen ein Programmiermodell, das dem der PLDs entspricht: Die komplette Nebenläufigkeit aller Aktionen, die Kommunikationsfähigkeit über das gesamte Device (zumindest im Prinzip) und die Ausführung einer Aktion in einem Takt, verbunden mit einer gegebenen Synchronisation, ermöglichen ein solches Modell, das wesentlich Basis-orientierter ist als das für μ P/PLD-Kombinationen.

Der wesentliche Unterschied zu PLD-Modellen ist natürlich leicht zu erkennen: Die intrinsische Datenbreite ist bei PLDs 1, bei FPFAs jedoch > 1 (z.B. 32).

Aus diesem Basismodell dann höhere Modelle, die zur Bearbeitung von Threads geeignet sind und andere Kommunikationsmechanismen beinhalten, zu entwickeln, ist vergleichsweise einfach und wird an dieser Stelle nicht weiter diskutiert. Schwieriger sind Fragen der dynamischen Zuordnung von Ressourcen bzw. zum Space-Time-Mapping: Kann man von der tatsächlich zur Verfügung stehenden Anzahl von Array-Elementen (bei angenommener Identität aller) im Programmiermodell abstrahieren?

Die positive Beantwortung dieser Frage führt tatsächlich zum Space-Time-Mapping. Wenn die Ausführung eines Programms nur von der Anwesenheit eines einzigen Basiselements abhängt, andererseits die Ausführung in mehreren Basiselementen die Veränderung von Ausführungsrandbedingungen (hier: Zeit), nicht aber eine grundsätzliche Änderung beinhaltet, dann ist dies ein großer Schritt zur erweiterten Skalierbarkeit der Bausteine. Solche Systeme sind aktuell bestenfalls in Forschungssystemen vorhanden (→ [33, Abschnitt 10.2]).

Das Programmiermodell für FPFAs wird zurzeit so behandelt, dass die Anzahl der Array-Elemente zur Modellierungs- oder Compilierungszeit bekannt ist.

6.4 Compilertechnologie

Am Beispiel der Array-Architektur XPP (→ 6.2) soll erläutert werden, was derzeit im Rahmen eines C-Compilers an Möglichkeiten zur Programmerzeugung bestehen. Zunächst muss hierzu der Übersetzungsprozess diskutiert werden, schließlich ist die XPP-Architektur als Co-Prozessor ausgelegt. Hieraus ergeben sich Sourcecodeeinschränkungen für den XPP-Teil (→ 6.4.2).

Die eigentliche Übersetzungsarbeit hat mehrere Eckpunkte: Natürlich lassen sich die einzelnen Statements sehr schön auf die Array-Elemente abbilden, die (Zeit-)sequenzielle Folge ist dann an der konfigurierten Verdrahtung zu sehen. Allerdings ist es das erklärte Ziel dieser Architektur, möglichst viele Operationen parallel zueinander auszuführen

6.4.1 Übersetzungsprozess

Der typische Übersetzungsprozess ist in [38] beschrieben und in Bild 6.5 dargestellt. In der *Preprocessing-Phase* werden Architektur-unabhängige Schritte durchgeführt, insbesondere aber FOR-Loops, die vom Softwareentwickler als solche gekennzeichnet sind, aufgelöst (loop unrolling, → [33, Abschnitt 6.2.2]). Weiterhin wird per Datenabhängigkeitsanalyse versucht, die inneren FOR-Loops für eine überlappende Ausführung zu identifizieren (→ 6.4.3).

Damit steht bereits ein Kandidat von Programmkonstrukten fest, der für eine Übersetzung in das XPP-Array wichtig ist: FOR-Loops. Hiermit sind speziell solche Schleifen gemeint, deren Ausführungsanzahl bereits zur Compilezeit bekannt sind. Schleifen mit Laufzeit-bestimmter Schleifenanzahl werden meist als WHILE- bzw. DO-WHILE-Schleife bezeichnet.

Der Vorgang der temporalen Partitionierung (TempPart) ist der wohl wichtigste und neueste im Bereich des rekonfigurierbaren Rechnens. Ausgehend davon, dass das Programm nicht komplett in ein PAC (→ 6.2.2) passt, wird in dieser Phase versucht, Teilkonfigurationen zu bilden. Hierdurch entsteht echtes *Reconfigurable Computing*.

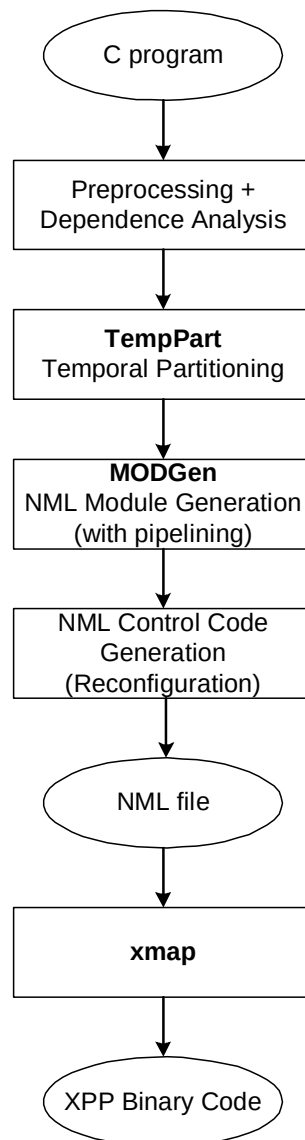


Bild 6.5 XPP-VC Übersetzungsfluss

Im Anschluss an diese Vorgänge sind die Generierung der Module in der Sprache *Native Mapping Language* (die Assemblersprache der XPP-Architekturen, hier werden den mit Koordinaten benannten PAEs Aktionen zugewiesen) und der Rekonfigurationsanweisungen.

6.4.2 Sourcecodeeinschränkungen

Die wesentlichen Sourcecodeeinschränkungen, die zur Übersetzungsfähigkeit in die XPP-Architektur vorgegeben sind, betreffen die folgenden Punkte:

- Datentypen: Floating-Point, `struct/union` und Pointer.
- Programmcode: Rekursive Funktionen, Systemaufrufe, irregulärer Kontrollfluss

Die Einschränkungen bezüglich der Datentypen sind dabei verständlich: Floating-Point-Operationen werden in der Hardware (den PAEs) nicht unterstützt und müssten somit vergleichsweise umständlich emuliert werden. Der Ausschluss der Floating-Point-Typen ist daher nur solange begründet, wie keine PAEs mit entsprechenden Operationen vorhanden sind.

Zusammengesetzte Strukturen (`struct`, `union`) werden aus Vereinfachungsgründen nicht eingesetzt, ebenso wie Pointer.

Funktionsaufrufe, insbesondere rekursive Funktionen sind ein besonderes Problem. Prinzipiell ist es möglich, eine nicht-rekursive Funktion in den Code einzubetten. Der Ausschluss von nicht-rekursiven Funktionen ist also nicht prinzipieller Natur, sondern eher aus pragmatischen Gründen. Zugelassen sind hingegen bestimmte XPP-Funktionsaufrufe, z.B., um auf externes RAM zuzugreifen (`XPP_getstream`, `XPP_putstream`...).

Rekursive Funktionen hingegen müssen ausgeschlossen werden, weil die XPP-Architektur keinen Stack besitzt. Die Rekursivität ist deshalb so gut bei Von-Neumann-Maschinen einzusetzen, weil diese fast immer einen Daten- und Rücksprungstack besitzen. Solange dieser Stack nicht überläuft, kann eine Funktion sich selbst wiederum aufrufen. Das Fehlen dieser Funktionalität in der XPP-Architektur könnte zwar durch eine spezielle Konstruktion (Software-Stack) aufgehoben werden, hierauf wird aus Aufwandsgründen jedoch verzichtet. Insbesondere wären die Laufzeiteffekte sehr negativ.

Als irregulärer Kontrollfluss werden Statements wie `break` (!), `continue`, `goto` und `label` bezeichnet. In dieser Liste stellt `break` eine Besonderheit dar: Der Ausschluss dieses Softwareelements macht `switch/case`-Anweisungen sinnlos, folgerichtig werden auch diese ausgeschlossen. Letzterer Ausschluss wäre nicht notwendig, jedoch wird die Sourcecodeanalyse aufwendiger beim Verbleib des `break`, weil nunmehr der Kontext geprüft werden muss.

6.4.3 Pipeline Vectorization

Durch ein mit *Pipeline Vectorization* bezeichneten Verfahren [36] wird die parallele Ausführung mehrerer Loop-Durchläufe ineinander verschachtelt, so dass möglichst wenig Speicherzugriffe mit möglichst hoher paralleler Ausführung der Rechenoperationen gekoppelt werden.

Das Verfahren umfasst die folgenden Schritte:

- Kandidatenselektion

- Schleifennormalisierung
- Abhängigkeitsanalyse
- Auflösung von Array-Abhängigkeiten
- Generierung des Datenflussgraphen
- Einführung von Feedbacks
- Pipelining und Timing

Es sieht also insgesamt recht aufwendig auf, ist aber andererseits "straight forward" und lässt sich am Besten an zwei Beispielen erläutern, die [36] entnommen sind. Das erste Beispiel implementiert ein Flanken- bzw. Kantendetektionsprogramm, das zweite die Erzeugung von Fibonacci-Zahlen.

Beispiel 6.1: Kantendetektion

Das folgende Programm umfasst eine Kantendetektion, die in einem zweidimensionalen Array `p1[HLEN][VLEN]` Kanten bestimmt und in einem zweiten Array `p2[HLEN][VLEN]` speichert:

```
char p1[HLEN][VLEN], p2[HLEN][VLEN];
...
/* Vertical and horizontal edge detection */
for( v = 1; v < VLEN-2; v++ )
{
    for( h = 1; h < HLEN-2; h++ )
    {
        int v_edge = (p1[v-1][h+1] - p1[v-1][h-1]) +
                     2*(p1[v][h+1] - p1[v][h-1]) +
                     (p1[v+1][h+1] - p1[v+1][h-1]);
        int h_edge = (p1[v+1][h-1] - p1[v-1][h-1]) +
                     2*(p1[v+1][h] - p1[v-1][h]) +
                     (p1[v+1][h+1] - p1[v-1][h+1]);
        int tmp = abs(v_edge) + abs(h_edge);
        if( tmp > 255 )
            tmp = 255;
        p2[v][h] = (char) tmp;
    }
}
```

Beispiel 6.2: Erzeugung der Fibonacci-Zahlen

Das zweite Programm erzeugt die ersten N Fibonacci-Zahlen und speichert sie als Integerwerte in einem Array `x[N]`.

```
int x[N], i;
...
```

```

x[0] = 0;
x[1] = 1;
for( i = 2; i < N; i++ )
{
    x[i] = x[i-1] + x[i-2];
}

```

Kandidatenselektion

Im Beispiel 6.1 gibt es zwei FOR-Loops mit konstanten (Compilezeit-bestimmten) Grenzen: Eine äußere und eine innere. Als notwendige Kriterien für einen geeigneten Kandidaten zur Pipeline-Vektorisierung gelten: Innerste Schleife, FOR-Loop (oder auch WHILE-Loop mit Indexvariablen) mit konstanten Grenzen.

Diese Kriterien sind von der inneren Schleife im Beispiel 6.1. und von der einzigen Schleife in 6.2 erfüllt.

Schleifennormalisierung

Eine Schleifennormalisierung soll alle Abhängigkeiten auf eine einzige Variable zurückführen. Dies bedeutet, dass weitere Variablen auf diese abgebildet werden müssen. Diese Indexvariable soll mit dem Index 0 beginnen und eine Schrittweite von 1 haben.

Weiterhin ist gefordert, dass alle Abhängigkeiten von dieser Indexvariablen linear sein sollen, also bei der Indexvariablen I in der Form $S \cdot I + C$. S gilt dabei als die Zugriffsweite (*access' stride*).

Lässt sich eine Forderung nicht erfüllen, gilt die Schleife nicht mehr als Vektorisierungskandidat.

Beispiel 6.1 (Fortsetzung)

Die Normalisierung führt zu folgendem Code für die innere Schleife:

```

for( h = 0; h < HLEN-3; h++ )
{
    int v_edge = (p1[v-1][h+2] - p1[v-1][h]) +
                 2*(p1[v][h+2] - p1[v][h]) +
                 (p1[v+1][h+2] - p1[v+1][h]);
    int h_edge = (p1[v+1][h] - p1[v-1][h]) +
                 2*(p1[v+1][h+1] - p1[v-1][h+1]) +
                 (p1[v+1][h+2] - p1[v-1][h+2]);
}

```

Beispiel 6.2 (Fortsetzung)

Die normalisierte Generierung der Fibonacci-Zahlen lautet:

```

int x[N], i;
...

```

```
x[0] = 0;
x[1] = 1;
for( i = 0; i < N-2; i++ )
{
    x[i+2] = x[i+1] + x[i];
}
```

Abhängigkeitsanalyse

Es existieren drei grundsätzliche Typen von Datenabhängigkeiten in einem Programm:

- *True* oder *Flow Dependences*: Der Wert einer Variablen ist in einer Zuweisung definiert und wird lesend in einer nachfolgenden Anweisung benötigt (Read-After-Write, RAW).
- *Anti-Dependences*: Eine Variable wird in einer Anweisung gelesen, und in einer nachfolgenden Anweisung wird der Variablen ein neuer Wert zugewiesen (Write-After-Read, WAR).
- *Output Dependences*: Zwei Wertzuweisungen in unterschiedlichen Anweisungen erfolgen an eine Variable (Write-After-Write, WAW).

Diese drei Abhängigkeiten entsprechen denjenigen aus der superskalaren Welt ([33, Abschnitt 5.2]. Während in der Softwareanalyse jedoch häufig die internen Prozessorregister analysiert werden, existieren hier diese ausgezeichneten Register nicht: Für FPFAs (Field-Programmable Functional Arrays) werden Variablen in Speicherarrays gelagert (skalare Variable können entweder aufgelöst werden, einen temporären Charakter haben und dann in einem internen Register gespeichert werden, oder müssen ebenfalls im Hauptspeicher stehen).

Für das hier beschriebene Verfahren ist es weiterhin interessant, ob die Abhängigkeit Schleifen-unabhängig oder Schleifen-getragen ist. Im ersten Fall stehen die beiden Anweisungen innerhalb einer Schleife, im zweiten Fall über mindestens eine Schleifeniteration hinweg. Als Abhängigkeitsdistanz (Dependence Distance) bezeichnet man die Anzahl der Iterationen zwischen diesen Statements.

Falls nun das Rückschreiben auf das Speicherarray immer in-order verläuft (so lautet die Grundannahme beim XPP), sind ausschließlich die True Dependencies von Interesse. Für diese wird geprüft, ob die analysierte Distanz (0: Loop-independent) immer auftritt, diese werden als regulär bezeichnet.

Die Synthese wird dann unter Beachtung der Abhängigkeiten durchgeführt. Abhängigkeiten führen dann zu weniger Parallelität, weil spätere Schleifendurchläufe nicht mit berücksichtigt werden können.

Beispiel 6.2 (Fortsetzung)

Die Abhängigkeitsdistanz für die Generierung der Fibonacci-Zahlen ist 1 (und 2).

Auflösung von Array-Abhängigkeiten

Grundsätzlich ist es nicht notwendig, die Array-Abhängigkeiten weiter aufzulösen, allerdings ist das Handling für skalare Variable einfacher (und in dem in [36] dargestellten Algorithmus essentiell). Um die Schleifen-getragenen Abhängigkeiten auflösen zu können, werden für jede gelesene Arrayvariable eine skalare substituiert.

Beispiel 6.2 (Fortsetzung)

Für die Generierung der Fibonacci-Zahlen müssen zwei neue temporäre Variable new0 und new1 eingeführt werden:

```
int x[N], i;
...
x[0] = 0;
x[1] = 1;
int new0 = x[0];
int new1 = x[1];
for( i = 0; i < N-2; i++ )
{
    x[i+2] = new0 + new1;
    new0 = new1;
    new1 = x[i+2];
}
```

Generierung des Datenflussgraphen

Aus dem nunmehr transformierten Sourcecode wird im nächsten Schritt ein kombinatorischer, azyklischer Datenflussgraph erstellt. Hierzu werden die internen Datenabhängigkeiten analysiert, für jede Operation ein eigener Operator ausgewählt und mit den Quellen und Senken verbunden.

Das gewählte Verfahren ist also eine einfache, direkte Übersetzung des Schleifenkörpers. Im Schleifenkörper werden keinerlei Ressourcen gemeinsam genutzt, es wird also mit maximaler Parallelität gearbeitet. Zudem – und das ist das Ziel dieses Verfahrens – ermöglicht dies im Anschluss ein Pipelining über mehrere Schleifendurchläufe.

Da Zyklen nicht erlaubt sind (es wird auch immer nur die innerste Schleife übersetzt!), kann nur linearer Code vorkommen. Verzweigungen werden parallel zueinander übersetzt, und ein Multiplexer wählt das jeweilige Ergebnis aus. Diese Parallelitäten können ggf. sogar wieder auf die gleiche Hardware abgebildet werden, wenn beispielsweise zwei Pfade sich gegenseitig ausschließen (if ..., else ...), aber auf ähnliche Verknüpfungen abgebildet werden. Der folgende Code zeigt ein derartiges Beispiel, indem eine bedingte Wertzuweisung, die in einem Zweig die Addition, im anderen die Subtraktion zweier Zahlen beinhaltet, auf die gleiche

Hardware abgebildet wird. Addition und Subtraktion sind so eng miteinander verwandt, dass die gleiche Hardware mit einem geringen Aufwand an jeweiliger Umschaltung verwendet werden kann.

```

if( b < 0 )
    c = a + b;
else
    c = a - b;

```

Für ALU-Arrays gilt, dass verschiedene Zweige, sich gegenseitig ausschließend, grundsätzlich auf die gleiche ALU abgebildet werden können, wenn die ALU-Funktion durch Laufzeit-Bedingungen umschaltbar ist (Event-Funktion bei XPP).

Obwohl dies sehr einfach und direkt aussieht, bergen die bedingten Verzweigungen ein Problem in sich. Angenommen, in verschiedenen Zweigen werden nur einem Teil der Variablen neue Werte zugewiesen, was passiert dann mit den nicht-zugewiesenen Variablen? Diese behalten nach allgemeinem Syntaxverständnis (in Software) ihren Wert, und die muss im Datenfluss berücksichtigt werden. Leider führt dies zu Schleifen im Schleifenkörper, was gesondert behandelt werden muss.

Die Speicherzugriffe sollten ferner minimiert werden. Dies kann dadurch geschehen, dass Werte, einmal aus dem Speicher gelesen, solange behalten werden, bis sie nicht mehr benötigt werden. Das Beispiel 6.1 zeigt hierzu eine Implementierung.

Beispiel 6.1 (Fortsetzung)

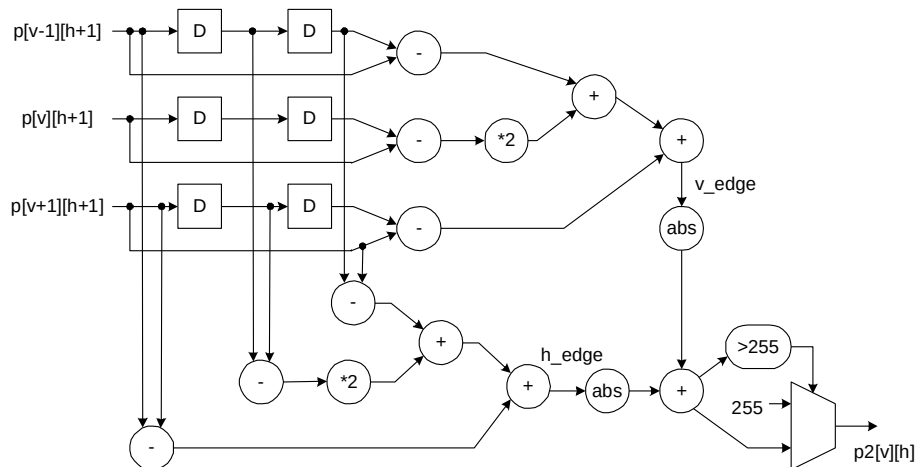


Bild 6.6 Datenflussgraph Flankendetektion

Die Schleife wird mit h als Schleifenparameter durchlaufen, so dass die sequenzielle Speicherung in den D-Flipflops nacheinander die Array-Werte für $p[\dots][h]$

1] ... $p[\dots][h+1]$ liefern. Der übrige Datenflussgraph ist direkt dem Code entnommen, die bedingte Wertzuweisung eines Integerwerts an eine Variable vom Typ Character ist mithilfe des erwähnten Multiplexers realisiert.

Beispiel 6.2 (Fortsetzung):

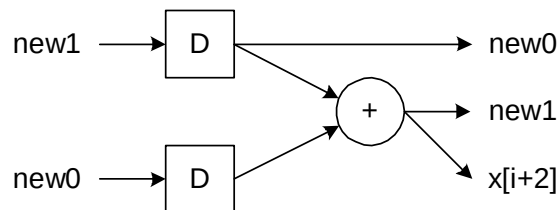


Bild 6.7 Erste (inkomplette) Version Datenflussgraph Fibonaccizahlengenerierung

Bild 6.7 zeigt die direkte Übersetzung des Beispiels zur Fibonaccizahlengenerierung. Das Problem mit diesem Beispiel ist, dass nun Schleifen-abhängige RAW-Hazards auftreten. Dies bedeutet, dass zwar die Erzeugungsreihenfolge erhalten bleibt, andererseits jedoch die erzeugten Werte rückgekoppelt werden müssen. In diesem Fall liegt eine reguläre Abhängigkeit vor, d.h., die Schleifendistanz ist konstant (= 1).

Einführung von Feedbacks

Um die Schleifen-abhängigen Datenabhängigkeiten korrekt umsetzen zu können, müssen wie im Beispiel 6.2 erwähnt Daten rückgekoppelt werden. Hierbei sind der Initialisierungsfall und der Betriebsfall für Schleifen noch zu unterscheiden: Solange die Abhängigkeitsdistanz noch nicht durchlaufen ist, gibt es auch keine korrekten Werte zur Rückkopplung.

Die Strategie ist sehr einfach: Eingangsseitig wird bei den rückgekoppelten Variablen, die also in einem Schleifendurchlauf gelesen und geschrieben werden, zwischen der Initialisierung und der Rückkopplung unterschieden, etwa durch eine Hilfsvariable (siehe Beispiel 6.2, nächste Fortsetzung). Die Rückkopplung erfolgt auf ein Speicherregister zu Beginn der Schleifenhardware, so dass der Durchlauf mit den alten Werten beginnt und die neuen erzeugt.

Beispiel 6.2 (Fortsetzung):

In dem Beispiel zur Fibonaccizahlengenerierung werden zwei Hilfsvariablen, `new0` und `new1` genutzt. Diese Variablen sind miteinander verkettet, d.h., `new1` (alt) wird zu `new0` (neu). Die neu eingeführte Variable `scalar_in` wird im Schleifenkopf beispielsweise auf '1' initialisiert und im Schleifenkörper auf '0' gelegt, so dass die Unterscheidung beider Fälle möglich ist.

Dies ergibt dann folgendes Bild für den Datenflussgraphen:

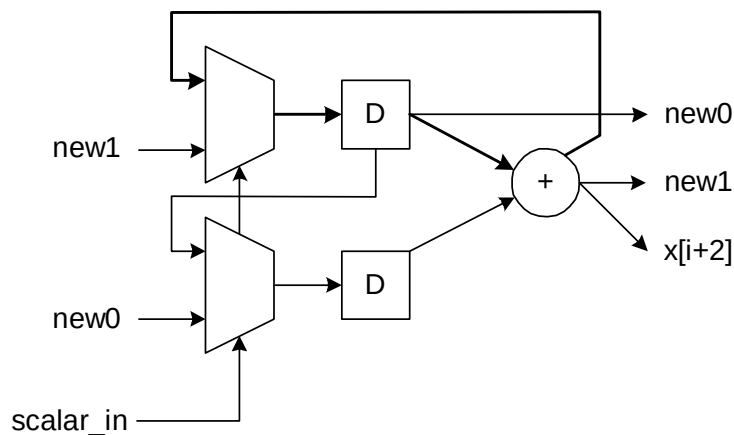


Bild 6.8 Komplettierte Version Fibonaccizahlengenerierung

Pipelining und Timing

Die jetzt vorhandenen Datenflussgraphen zielen auf die Berechnung eines Ergebnisses (oder eines Vektors von Ergebnissen) innerhalb eines Takts, sieht man einmal von der Ladephase ab, falls mehrere Arrayelemente hierfür notwendig sind (siehe Beispiel 6.1). Dieses Form der Berechnung kann sehr ineffizient sein, wenn sich die Laufzeiten der einzelnen Operatoren addieren (bei erzwungener Sequenzialität).

Die Methode zur Reduzierung der Laufzeit heißt *Pipelining*: Eine Aktion, hier die komplette Berechnung des Schleifenkörpers, wird in mehrere Teilaktionen mit Zwischenspeicherung der Ergebnisse zerlegt. Die Teilaktionen werden sukzessive durchlaufen und die Ergebnisse taktgesteuert gespeichert, wobei eine durchlaufene Pipelinestufe im Anschluss (nächster Takt) wiederum den nächsten Teilwert berechnen kann.

Pipelining erhöht zugleich die Latenzzeit, d.h. die Zeit, die der erste und der letzte Wert zur Fertigstellung benötigen (Filling/Flushing der Pipeline). Dies wird meist nicht als Problem angesehen, weil diese Zeit meist vernachlässigbar klein gegenüber der restlichen Laufzeit ist, stellt aber ein Problem bei Schleifen mit nur wenigen Durchläufen dar.

Pipelining muss für jeden Zweig die gleiche Anzahl der Stufen erzeugen. Die Berechnung kann mit sehr vielen Pipelinestufen versehen werden, mit dem Effekt, dass die Taktzeit sehr klein wird. In der Praxis wird es jedoch Begrenzungen geben: Zum einen ist der oftmals vorhandene Speicherzugriff meist begrenzend, zum anderen können Rückkopplungen wie im Beispiel 6.2, die nicht weiter unterteilt werden können (dies würde das Verhalten ändern!), nur in einer

Pipelinstufe realisiert werden. Wenn die Rückkopplungszeit $t_{feedback}$ (Zeit zur Berechnung eines Werts und zur Rückkopplung) und die Anzahl der Taktzyklen, die für einen Speicherzugriff notwendig sind, bekannt sind, gilt für die Anzahl N_{PC} an Taktzeiten T_C eines gegebenen Takts folgende Gleichung:

$$N_{PC} = \max\left(\left\lceil \frac{T_{feedback}}{T_C} \right\rceil, N_{mem}\right) \quad (6.1)$$

Die Zeit, die ein Wert zur Berechnung benötigt, ist dann $N_{PC} * T_C$, bei Vernachlässigung der Füll- und Leerzeiten wird dann eine Schleife mit n Iterationen in der Zeit $n * N_{PC} * T_C$ bearbeitet.

Beispiel 6.1 (Fortsetzung):

In dem Beispiel 6.1 werden zwei neue Pipelinestufen eingeführt. Dies ergibt dann folgenden Datenflussgraphen:

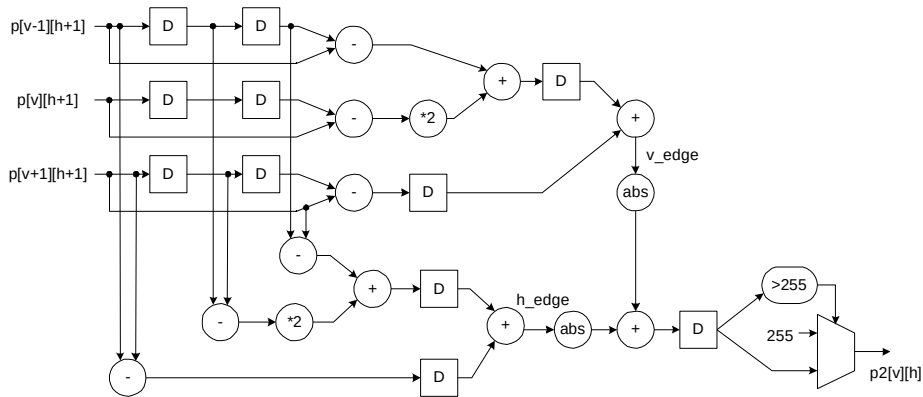


Bild 6.9 Pipelineversion des Datenflussgraphen für Flankendetektion

Für ALU-Array-basierte Berechnung (FPFA, XPP) wird für das Pipelining meist ein vereinfachter Ansatz gewählt: Das Ergebnis jeder Berechnung wird einfach gespeichert. Dies ergäbe im Beispiel 6.1 insgesamt (mit Speicherzugriff) 9 Stufen. Wichtig ist hierbei, dass Zweige mit kürzerem Weg verzögert werden müssen, um die Korrektheit der Ergebnisse sicherzustellen.

6.4.4 Loop Transformationen

Loop Unrolling ist ein Begriff aus der Compilertechnologie insbesondere für superskalare Prozessoren [33]. Dies wird auch – stärker spezifizierend – als Software Unrolling bezeichnet. Die wesentlichen Schritte bestehen darin, dass zwei (oder mehrere) Schleifen aneinandergelegt, ein (Compiletime) Register Renaming durchgeführt und die Instruktionen dann neu sortiert (instruction scheduling) werden, um

z.B. überflüssige Speicherzugriffe zu verhindern. Der positive Laufzeiteffekt tritt durch die Minderung der Speicherzugriffe und die potenzielle Erhöhung der Instruktionsparallelität ein. Gleiche Effekte können auch durch *Loop Merging* erzielt werden.

Während die Zielrichtung in Prozessor-basierten Systemen fast ausschließlich die Vergrößerung der Schleifen ist, liegt bei FPFA- bzw. FPGA-basierten Systemen eine Notwendigkeit zur Anpassung der Schleifen vor. Der beschriebene Algorithmus vektorisiert immer die innerste Schleife, so dass genau zwei Probleme auftreten: Der Schleifenkörper ist zu groß, um in die Hardware zu passen, oder zu klein, um einen Performancegewinn (aufgrund des Hardware-Overheads) zu bringen.

Loop Unrolling und Loop Merging helfen im Fall einer zu kleinen Schleife. Dies zeigt sich z.B. im Fall eines variierten Beispiels 6.1 zur Kantenberechnung.

Beispiel 6.1 (Fortsetzung)

Das Beispiel 6.1 hätte auch in folgender Weise definiert sein können:

```
int x[N], i, k;
...
for( v = 1; v < VLEN-2; v++ )
{
    for( h = 0; h < HLEN-3; h++ )
    {
        int v_edge = 0;
        for( k = 0; k < 3; k++ )
        {
            v_edge += p1[v+k-1][h+2] - p1[v+k-1][h];
        }
        v_edge += p1[v][h+2] - p1[v][h];
        int h_edge = 0;
        for( k = 0; k < 3; k++ )
        {
            h_edge += p1[v+1][h+k] - p1[v-1][h+k];
        }
        h_edge += p1[v+1][h+1] - p1[v-1][h+1];
    }
}
```

Die inneren Schleifen sind jetzt diejenigen mit der Indexvariablen k , die also die jeweiligen Kantenwerte berechnen. Der hierbei zu erzielende Parallelismus ist natürlich geringer, verglichen mit dem ursprünglichen Beispiel.

Hier hilft das Loop Unrolling, das in diesem Fall die beiden inneren Schleifen komplett auflöst und wiederum zum ursprünglichen Sourcecode zurückführt.

Loop Tiling adressiert den umgekehrten Fall, der entsteht, wenn eine komplett entrollte Schleife zu groß für die aufnehmende Hardware ist. Der Algorithmus

versucht in diesem Fall, nur einen Teil der Schleifendurchläufe zu entrollen und komplett in Hardware zu mappen, während ein Teil der Schleifenstruktur erhalten bleibt.

```

for( i = 0; i < m; i++ )
{
    PRE( i );
    for( j = 0; j < N; j++ )
        F( i, j );
    POST( i );
}

```

Bild 6.10 Schleifenstruktur

Für eine "gute" Schleifenzerlegung kann eine Schätzung gegeben werden. Angenommen, die Schleife hat den allgemeinen Charakter wie in Bild 6.10 dargestellt, und es existieren Schätzungen für den Hardwareaufwand für $PRE(i)$ ($area_{PRE}$), $POST(i)$ ($area_{POST}$) und $F(i, j)$ ($area_F$). Bei bekannten Hardwareressourcen ($area_{HW}$) kann dann t_{SIZE} angegeben werden:

$$t_{SIZE} = (area_{HW} - area_{PRE} - area_{POST}) / area_F \quad (6.2)$$

```

for( i = 0; i < m; i++ )
{
    PRE( i );
    for( jt = 0; jt < (n-1)/tsize+1; jt++ )
        for( j = 0; j < min(tsize, n-jt*tsize); j++ )
            F( i, j + jt*tsize );
    POST( i );
}

```

Bild 6.11 Loop Tiling in Anwendung auf Code aus Bild 6.10

Hieraus lässt sich eine neue Version der Schleifenkonstruktion aus Bild 6.10 angeben (Bild 6.11). Leider ist dies nicht die Lösung, denn die innere Schleife besitzt keinen konstanten oberen Wert. M.a.W.: Jeder Schleifendurchlauf (in jt -

Schleife) hätte einen anderen Schleifenkörper (denn die innere Schleife soll ja gerade vektorisiert werden).

Die Lösung besteht darin, das Minimum $\min(tsize, n-jt*tsize)$ durch $tsize$ zu ersetzen ($tsize$ ist eine Konstante, so dass das Minimum niemals größer sein kann) und die Korrektheit der Rechnungen ($n-jt*tsize$ könnte ja kleiner als $tsize$ sein) durch zusätzliche Bedingung

$$j < n - jt \times tsize \quad (6.3)$$

$$jt \neq (n-1)/tsize \vee j < (n-1)\%tsize - 1 \quad (6.4)$$

zu garantieren.

6.4.5 Temporale Partitionierung

Der letzte Punkt der hier behandelten, aktuellen Compilertechnologie zur Übersetzung in ALU-Arrays betrifft die temporale Partitionierung (siehe auch Bild 6.5). Dies ist deswegen so interessant, weil sich zum ersten Mal der automatische Weg in ein **Reconfigurable Computing** aufzeigt.

Temporale Partitionierung wird zurzeit dann angewendet, wenn die Zielhardware (bei XPP: ein PAC) zu klein ist, um den kompletten Teil aufzunehmen. Bild 6.12 zeigt den Ablauf beim vektorisierenden XPP-Compiler. Zunächst wird für das C-Programm, bei dem nun schon die oben erwähnten Sourcecode-Transformationen durchgeführt sind, durch den SUIF-Compiler [39] ein Abstract Syntax Tree (AST) aufgestellt. Hierin befinden sich Knoten, vergleichbar mit Basis-Blöcken für Mikroprozessoren.

Für jeden dieser AST-Knoten wird geschätzt, wie viele Ressourcen er im PAC benötigt, und benachbarte Knoten werden solange hinzugemischt, bis der Cluster voll ist. Oberstes Prinzip ist dabei, dass Partitionen mit einem Eintrittspunkt, ggf. aber mehreren Austrittspunkten zusammengestellt werden.

Die Schätzungen werden im Nachgang auf Validität verifiziert. Bei Überschreitung der PAC-Kapazität wird eine Neuübersetzung bzw. -zusammensetzung initiiert. Die Folge der entstehenden Konfigurationen wird dann durch den Configuration Manager (CM) der XPP-Architektur behandelt.

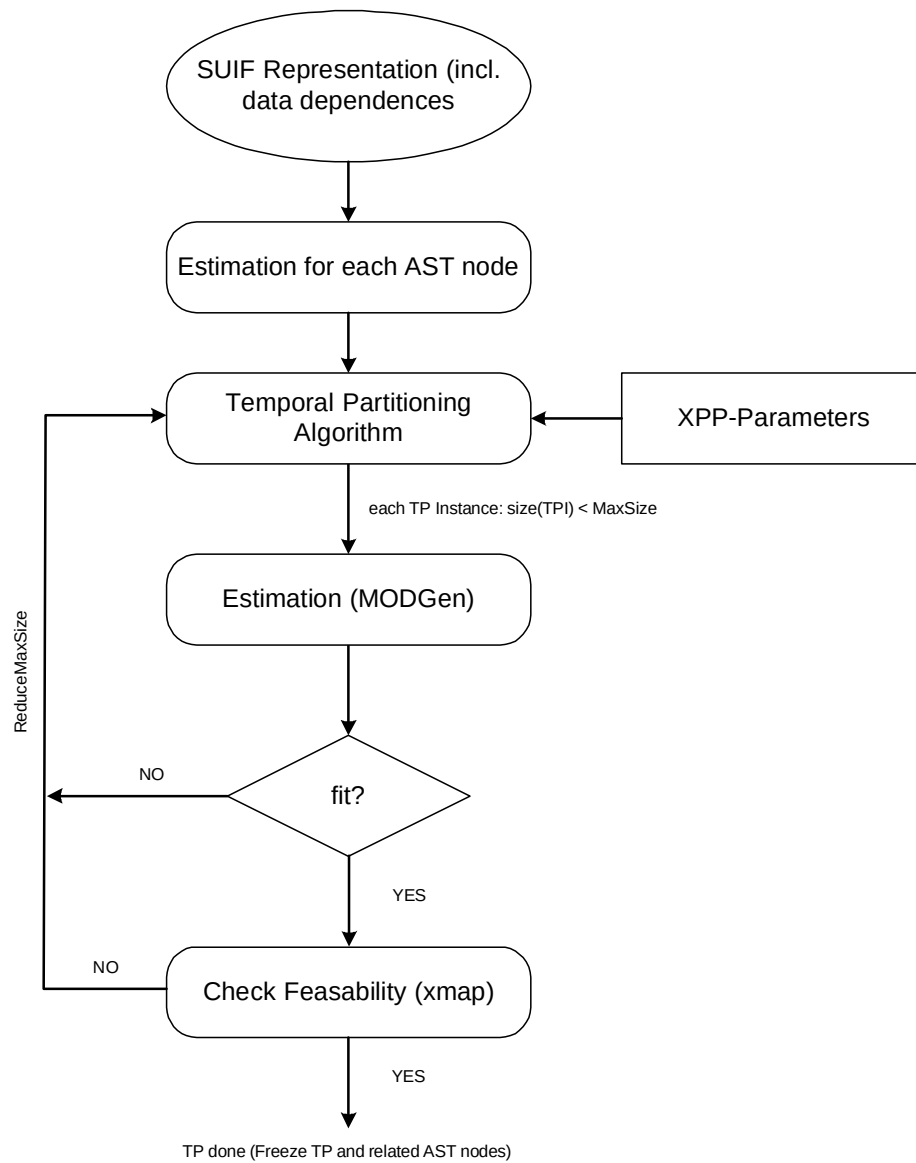


Bild 6.12 Ablauf temporale Partitionierung im XPP-C-Compiler

Glossar

Anti-Fuse

Programmspeichertechnologie für PLDs: Eine Verbindung wird beim Programmieren durch einen Überstrom erzeugt (Metallleitung durch isolierendes Silizium), unprogrammiert existiert keine Leitung. Die Programmierung ist nicht reversibel.

Boolesche Algebra

Algebra für Schalt- oder Boolesche Variablen, die ihrerseits nur 2 Werte (0 oder 1) annehmen können.

CPLD

Complex Programmable Logic Device, Bezeichnung für HDPLD-Klasse mit wenigen großen Blöcken (meist PAL-Struktur). Kennzeichen ist eine einzige Sorte von programmierbaren Verbindungen zwischen den Blöcken mit immer identischem Zeitverhalten, unabhängig von der Lage der Blöcke.

DNF

Disjunktive Normalform: Jede Logikfunktion wird als ODER-Verknüpfung von UND-Verknüpfungen von Booleschen Variablen und deren Negation dargestellt.

EEPROM

Electrically Erasable Programmable Read-Only Memory: Aus der Speichertechnologie für Rechner stammend wird hier die Programmierung durch Ladungen auf einem isolierten Transistorgate vorgenommen. Die Ladungen sind löschtbar.

FPFA

Field-Programmable Functional Array, Bezeichnung für Bausteine aus dem Bereich des Configurable Computing, die aus Arrays von arithmetischen und logischen Verknüpfungen auf Wortbasis bestehen. Die Datenverbindungen in diesem Array sind konfigurierbar.

FPGA

Field-Programmable Gate Array, Bezeichnung für HDPLD-Klasse mit vielen kleinen Blöcken (LUT- oder multiplexerbasiert). Kennzeichen ist das Vorhandensein verschiedener Arten von programmierbaren Verbindungen zwischen den Blöcken mit unterschiedlichem Zeitverhalten.

FPL(D)

Field Programmable Logic (Device): Allgemeine Bezeichnung für Feldprogrammierbare Bausteine. Das 'Feld' bezeichnet dabei den Ort der Programmierung (außerhalb der Fabrik).

FSM

Finite State Machine, technisch benutzt Bezeichnung für den Deterministischen Endlichen Automaten (DEA bzw. DFA), meist basierend auf einem synchronen Design.

Fuse

Eigentlich Fusable Link: Metallverbindung im Baustein, die beim Programmieren zerstört wird. Wurde in PLDs durch Anti-Fuse abgelöst.

GAL

Generic Array Logic (Handelsmarke), Speicherung der programmierbaren Funktion in Form von Disjunktiven Normalformen (DNF) mit programmierbaren UND-Eingängen. Die Architektur ist der PAL-Architektur äquivalent.

Gatteräquivalent

Bezeichnung für die (etwa) zu einem NAND-Gatter mit 2 Eingängen (NAND2) äquivalente Verknüpfung. Wird meist in der Form benutzt, dass die Anzahl der NAND2-Gatter angegeben wird, die zur Realisierung eines Teils (oder des Bausteins) notwendig wären.

HDPLD

High-Density PLD: Allgemeine Bezeichnung für FPLDs mit hoher Dichte, geeignet zur Aufnahme ganzer Systeme.

LSI

Large Scale Integration, einige 10000 Gatteräquivalente pro Chip

LUT

Look-Up-Table, tabellenartige Speicherstruktur mit Eingängen als Adressleitungen und Ausgängen für die Tabelleninhalte. Diese Struktur entspricht dem Aufbau von EPROM-Bausteinen. Im Vergleich zu der PAL-Struktur sind hier die AND-Eingänge festverdrahtet, die ORs jedoch programmierbar.

MSI

Medium Scale Integration, einige 1000 Gatteräquivalente pro Chip

Multiplexer

1-aus- 2^k -Auswahlsystem: Mit Hilfe von k Steuereingängen wird ein Dateneingang von 2^k möglichen ausgewählt, dessen Logikpegel am einzigen Ausgang des Multiplexers erscheint.

PAL

Programmable Array Logic (Handelsmarke), Speicherung der programmierbaren Funktion in Form von Disjunktiven Normalformen (DNF) mit programmierbaren AND-Eingängen, aber festen OR-Verbindungen.

PLD

Programmable Logic Device: Allgemeine Bezeichnung für (feld-)programmierbare Logikbausteine.

Routing

Begriff aus der Übersetzung von textuellen Designbeschreibungen in die Schaltung: Nachdem die Schaltungsteile platziert worden sind, müssen die notwendigen Verbindungen (Route) geschaltet werden.

Schaltalgebra

Algebra für Schalt- oder Boolesche Variablen, die ihrerseits nur 2 Werte (0 oder 1) annehmen können.

SPLD

Simple Programmable Logic Device, Bezeichnung für FPLD-Klasse geringer Dichte mit nur einem Block innerhalb des Bausteins.

SRAM

Static Random Access Read/Write Memory: Speichertechnologie aus dem Prozessorbereich, wobei die Information durch 1 von 2 erlaubten Zuständen in einem Flipflop gespeichert wird.

SSI

Small Scale Integration, Integration bis zu einigen 100 Gatteräquivalente auf einem Chip

VLSI

Very Large Scale Integration, >10000 Gatteräquivalente pro Chip

Literatur

- [1] AUER, A., KIMMELMANN, R.: *Schaltungstest mit Boundary Scan*. Heidelberg: Hüthig Verlag, 1996.
- [2] BORGMAYER, J.: *Grundlagen der Digitaltechnik*. München: Carl Hanser Verlag, 1997.
- [3] DE MICHELI, G.: *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994
- [4] DEHON, A., WAWRZYNEK, J.: Reconfigurable Computing: What, Why and Implications for Design Automation. *Design Automation Conference DAC99*, San Francisco, 1999.
- [5] MÄRTIN, C.: *Rechnerarchitekturen*. München: Carl Hanser Verlag, 2000.
- [6] PERNARDS, P.: *Digitaltechnik II, Einführung in Schaltwerke*. Heidelberg: Hüthig Verlag, 1995
- [7] REICHARDT, J., SCHWARZ, B.: *VHDL-Synthese*. München: Oldenbourg Verlag, 2000
- [8] SIEMERS, C.: Designtipp Taktteiler. *ElektronikPraxis*, Heft 1, 2000, S. 36–38.
- [9] SIEMERS, C. (HRSG.): *Hardwaremodellierung*. München: Carl Hanser Verlag, 2000.
- [10] SIEMERS, C.: *Prozessorbau*. – Carl Hanser Verlag München Wien, 1999.
- [11] SIEMERS, C.: Rechenfabrik: Ansätze für extrem parallele Prozessoren. *c't* H. 15, 2001, S. 170–179.
- [12] SIKORA, A.: *Programmierbare Logikbauelemente*. München: Carl Hanser Verlag, 2001.
- [13] www.actel.com
- [14] www.altera.com
- [15] www.atmel.com
- [16] www.arm.com
- [17] www.cypress.com
- [18] www.latticesemi.com
- [19] www.mips.com
- [20] www.quicklogic.com
- [21] www.xilinx.com
- [22] BROWN, STEPHEN D., FRANCIS, ROBERT J., ROSE, JONATHAN, VRANESIC, ZVONKO G.: *Field Programmable Gate Arrays*. Kluwer Academic Publishers, Boston 1992.
- [23] HEUSINGER; RONGE; STOCK: *PLDs und FPGAs in der Praxis*. Franzis-Verlag München, 1994.

- [24] SIEMERS, C.: *Logikbausteine: feldprogrammierbare High-Tech-Komponenten*. Vogel Verlag Würzburg, 2001.
- [25] SIEMERS, C., SIKORA, A. (HRSG.), *Taschenbuch Digitaltechnik*. Fachbuchverlag Leipzig im Carl Hanser Verlag, München Wien, 2003.
- [26] DE MAN, H., *Designing Nano-Scale Systems for the Ambient Intelligent World*. it – Information Technology 45(6), S.310-317 (2003).
- [27] MÜLLER-SCHLOER, C. ET.AL, *Organische Computersysteme*. PARS Mitteilungen Nr. 20, ISSN 0177-0454, November 2003.
- [28] MARTIN, A.J., NYSTRÖM, M., WONG, C.G., *Three Generations of Asynchronous Microprocessors*. IEEE Design&Test 20(6), pp. 9-17 (2003)
- [29] <http://tech-www.informatik.uni-hamburg.de/lehre/ss2002/pc-technologie/docs/microprocessor-summary.pdf>.
- [30] <http://www.pactcorp.com>
- [31] J. SPARSØ, S. FURBER (EDS.) "Principles of Asynchronous Circuit Design". Kluwer Academic Publishers, The Netherlands.
- [32] <http://www.stretchinc.com>
- [33] SIEMERS, C.: *Rechnerarchitektur I/II*. Vorlesungsskript Institut für Informatik der TU Clausthal, Clausthal-Zellerfeld, und Fachhochschule Nordhausen, Nordhausen, 2004.
- [34] ANDREWS, D; NIEHAUS, D.; ASHENDEN, P.: *Programming Models for Hybrid CPU/FPGA Chips*. IEEE Computer 37(1) pp. 118-120 (2004)
- [35] <http://www.wcet.at/>
- [36] WEINHARDT, M., LUK, W., *Pipeline Vectorization*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Feb. 2001, pp. 234-248.
- [37] CARDOSO, JOÃO M.P., *A Novel Algorithm Combining Temporal Partitioning and Sharing of Functional Units*. Proceedings IEEE 9th Symposium on Field-Programmable Custom Computing Machines, Rohnert Park, CA, USA, April 2001.
- [38] CARDOSO, JOÃO M.P., WEINHARDT, MARKUS, *XPP-VC: A C Compiler with Temporal Partitioning for the PACT-XPP Architecture*. Field-Programmable Logic and Applications FPL 2002.
- [39] SUIF compiler system: <http://suif.stanford.edu>

Sachwortverzeichnis

- Abhängigkeitsanalyse 117
- Abhängigkeitsdistanz 117
- ALU-Array
 - Data-Path configurable 104
 - Space-Time-Mapping 104
- Ambient Intelligence Devices 3
- AmI *Siehe* Ambient Intelligent Devices
- AND-Funktion 9
- Antifuse 19
- Application-Specific Instruction Set Processor *Siehe* ASIP
- Application-Specific Integrated Circuit 1
- Architekturen
 - Coarse-Grain 102
 - Fine-Grain 102
- arithmetische Operation 28
- ASIC 1
- ASIP 103
- Aussagenlogik 9
- Automat
 - Mealy 31
 - Medwedjew 31
 - Moore 31
- Back Annotation 77
- BDD 99
- BDSL 24
- Binary-Decision-Diagram *Siehe* BDD
- Boolesche Algebra 9
 - logische Basis 9
 - Wahrheitstabelle 10
- Boot 21
- Boot-Speicher 78
- Boundary Scan Definition Language 24
- Boundary Scan Test 23, 24
 - TAP 25
- C
 - Datenflussgraph 118
 - for 112
 - Pipeline Vectorization 114
 - while 112
- Charakteristische Zeiten 7
- Clock 29
- Coarse-Grain-Architekturen 102
- Codespeicher 78
- Codierungsmodell 29
- Complex Programmable Logic Devices 21, 44
- Computing in Space 9
- Computing in Time 9
- Configurable Computing 1, 7
 - Klassifizierung 28, 102
 - Konfigurationsspeicher 15, 18
 - Operationsprinzip 8
 - Programmierbarkeit 27
 - Softwareprozess 75
 - Speichertechnologie 22
 - Zeitbedingungen 77
- CPLD 21, 44, 49
 - Durchsatz 50
 - Logikzelle 50
 - Routing 50
 - Timingmodell 63
- Data-Path-configurable ALU-Arrays 104
- Datenflussgraph 118
 - Timing 121
- Datenspeicher 78
- DEA 30
- Designfluss 76
- Deterministic Finite Automaton 30
- deterministischer endlicher Automat 30
- DFA 30
- Digitale Signalverarbeitung 69
- Disjunktion 9
- disjunktive Normalform 9, 11
- DNF 9, 11
- DNF-Struktur 49
- DSP 69

- Effizienz 4
- eFPGA 103
- ASIP 104
- Designprozess 103
- Mischung mit μ P 103
- Embedded FPGA 103
- Energieeffizienz 4
- Entwurfsebenen 81
- Entwurfsprozess
- Back Annotation 77
- Entwurfsebenen 81
- Entwurfssichten 81
- Logikoptimierung 85
- Netzliste 79
- parallele Algorithmierung 80
- Parallelität 80
- Partitionierung 81, 83
- Physikalische Synthese 86
- Pipelining 84
- Register Inferencing 85
- Registerzuordnung 84
- Ressourcenzuordnung 84
- Retiming 84
- Scheduling 84
- Sensitivitätsliste 81
- Simulation 80
- State Machine Synthese 85
- Strukturelle Sicht 83
- Synthese 81
- Technologie-unabhängige
- Übersetzung 79
- Technologie-Zuordnung 86
- Verhaltenssicht 83
- Y-Diagramm 81
- Entwurfssichten 81
- EPROM 21
- Exclusive-OR 10, 13
- eXtreme Processing Platform ... *Siehe* XPP
- Fehlererkennung 36
- Field-Programmable Functional Array *Siehe* FPFA
- Field-Programmable Gate Array... 44
- Field-Programmable Logic Devices *Siehe* PLD
- Fine-Grain-Architekturen 102
- Finite State Machine 75, 85 *Siehe* FSM
- Flipflop 16
- charakteristische Gleichung 18
- Clock-to-Output-Time 64
- D-Flipflop 18
- Hold-Time 64
- JK-Flipflop 18
- RS-Flipflop 18
- Setup-Time 64
- T-Flipflop 18
- Floating Gate 21
- FPFA 2, 5, 30, 111, 117
- XPP 5
- FPGA 44, 47
- Durchsatz 48
- Flächennutzung 48
- Logikzelle 47
- Modellbildung 47
- Routing 47
- XC6200 53
- FPLD *Siehe* PLD
- FSM 31
- Next State Decoder 31
- Output Decoder 31
- Zustandscodierung 33, 34
- GAL 12
- GAL22V10 37, 38
- OLMC 40, 41
- Gatteräquivalente 45
- Generic Array Logic 12
- Hardware Description Language *Siehe* HDL
- Hazard
- Abhängigkeitsdistanz 117
- Anti-Dependence 117
- Output Dependence 117
- RAW 117
- Schleifen-getragen 117
- Schleifen-unabhängig 117

True Dependence	117	LUT	14, 37, 43, 53
WAR.....	117	Mapping	86
WAW.....	117	Algorithmen	93
HDL.....	77, 80	BDD	99
Back Annotation	77	Chortly-crf-Algorithmus.....	97
HDPLD.....	38, 42	Library-basiert.....	94
Determinismus	46	Look-Up-Table-basiert.....	97
Flächennutzung.....	46	Multiplexer-basiert	99
Geschwindigkeit	46	Mikroprozessor.....	8
Timingmodell	63	Minimierung	
Herstellungstechnologie.....	71	Quine-McCluskey	89
High-Density Programmable Logic		Multiplexer	53
Devices	38, 42	Dynamische Multiplexer	53
IEEE 1149.1.....	23, 24, 26	LUT	53
Instruktion.....	8	paralleler Multiplexer im Routing	
In-System-Programmable	23		55
ISP	23	Serieller Multiplexer im Routing	
ispLSI5000.....	65		54
JTAG	24	Negation	9
Karnaugh-Veitch-Diagramm	<i>Siehe</i>	Netzliste.....	79
KV-Diagramm		NML	113
KDNF	87	Normalformen	
KKNF	87	disjunktiv	9
Konfigurationsspeicher	15, 18, 78	Operation	
Speichertechnologie.....	19	arithmetisch	28
Konjunktion	9	logisch	28
Vollkonjunktion	10	Optimierung	
KV-Diagramm	87	Logik-	85
Minimierung	88	OR-Funktion.....	9
Latch	16	Organic Computing	3
Lattice		PAL	2, 12, 37, 43
ispLSI5000.....	65	PAL22V10	37, 38
Leistungseffizienz.....	4	OLMC	40, 41
Logikoptimierung	85	Paralleler Multiplexer im Routing.	55
Minimierung	89	partielle dynamische	
logische Basis	9	Rekonfigurierbarkeit	24
logische Gatter	45	partielle Rekonfiguration.....	71
logische Operation	28	Partitionierung	81, 83, 103, 110
Look-Up-Table	14	Pass-Transistoren.....	19
Loop Merging	123	Physikalische Synthese.....	86
Loop Tiling	124	Pipeline Vectorization	114
Loop Transformationen.....	122	Abhängigkeitsanalyse.....	117
Loop Unrolling	112, 122	Access Stride	116

- Feedback 120
- Timing 121
- Pipelining 84, 118, 121
- PLD 1, 7
 - Betriebssystem 24
 - Boot 21
 - Boot-Speicher 78
 - Codespeicher 78
 - CPLD 44
 - Datenspeicher 78
 - FPGA 44
 - Geschichte 2
 - HDPLD 38
 - In-System-Programmable 23
 - Klassen 45
 - Konfigurationsspeicher 78
 - Mapping 86
 - maschinennahe Programmierung 78
 - Modell 77
 - partielle dynamische
 - Rekonfigurierbarkeit 24
 - Programmiermodell 78
 - Routing 86
 - SPLD 37
 - Zustandsspeicher 78
- PLL 70
- Programmable Array Logic 2, 12
- Programmable Logic Devices...*Siehe* PLD
- Programmierbarkeit 27
- Programmiermodell 78, 110
 - µP/PLD-Kombination 110
 - FPFA 111
 - Message Passing 110
 - Multithreading 110
 - Shared Memory 110
- Prozesse
 - Sensitivitätsliste 81
- Quine-McCluskey
 - Minterme 90
 - Primimplikanten 90
- Quine-McCLuskey 89
- Kernimplikanten 92
- Rapid Prototyping 21
- RAW-Hazard 117
- Reconfigurable Computing ..6, 7, 46, 102, 112
 - Operationsprinzip 8
 - Pipeline Vectorization 114
 - Temporale Partitionierung 125
- Rekonfiguration
 - partiell 71
- Reprogrammierbarkeit 2, 27
- Retiming 84
- Routing 19, 55, 86
 - Timingmodell 64
- Schaltalgebra 9
 - logische Basis 9
 - Wahrheitstabelle 10
- Schaltnetz 16, 29
- Schaltwerk 16, 29
 - einfach 31
 - Next State Decoder 31
 - Output Decoder 31
 - Zustandscodierung 33, 34
 - Zustandsgraph 34
 - zustandsorientiert 31
- Schaltwerktafel 32, 34
- Scheduling 84
- Sea-of-Gates 62
- Sensitivitätsliste 81
- Serieller Multiplexer im Routing... 54
- Simple Programmable Logic Device 37, 43
- Simulation 76
- Space-Time-Mapping 104, 111
- Speichertechnologie 19, 22
 - Antifuse 19
 - EPROM 21
 - SRAM 21
- SPLD 37, 43
- SRAM 21
- statisches RAM 21
- Sum-of-Products *Siehe* DNF
- Synthese 81

Syntheseverfahren		lokales Routing.....	58
Schaltwerk	32	LUT-basiert	58
Systemgatter	62	Magic Line	59
Takt	29	Programmierung	62
Taktflankensteuerung.....	18	Register Protection	57
Taktzustandssteuerung.....	16	Routing.....	60
Technologie-Zuordnung	86	Systemgatter	62
Test Access Point.....	25	Transistorschätzung.....	62
Timingmodell	63	Xilinx	
Transistor		Virtex II	68
MOS.....	21	XC2V	68
Verlustleistung.....	9	XC6200	53
VHDL	80	XOR-Funktion.....	10, 13
Virtex II	68	XPP.....	5, 105
Vollkonjunktion	10	Configuration Manager	108
Von-Neumann-Paradigma	7	Events	108
Codespeicher.....	15	for-loops	112
Operationsprinzip	8	Hardware-Objekte	105
Wahrheitstabelle	10	Native Mapping Language	113
WAR-Hazard.....	117	NML	113
WAW-Hazard	117	Pipeline Vectorization	114
WCET.....	111	Processing Array Cluster.....	108
Worst-Case Execution Times	111	Processing Array Element.....	107
XC2V	68	Sourcecodeeinschränkungen	112, 114
XC6200.....	53	Y-Diagramm.....	81
16*16-Block	61	Zustandscodierung.....	33, 35
4*4-Block	60	Zustandsgraph	34
Ausgangsrouting.....	59	Zustandsminimierung.....	33, 34
Basiszelle	57	Zustandsspeicher	78
Eingangsrouting.....	58		
Globales Routing	62		