23 High-Level-Synthese

High-Level-Synthese befaßt sich mit Methoden zur automatischen Ableitung einer dichotomischen Hardware-Beschreibung aus einer prozeduralen Beschreibung unter Verwendung einer programmiersprache, prozeduralen Teilsprachen von Mehr-Ebenen-Hardware-Beschreibungssprachen (wie beispielsweise CVS_BK, Verilog, oder VHDL) oder einer vergleichbaren Notation. Bild 23.1 veranschaulicht die durch die High-Level-Synthese berührten Abstraktions-Ebenen. Bild 23.2 präzisiert und veranschaulicht dies durch ein Beispiel (Bubble-Sort-Algorithmus, vgl. Abschnitt 19.3). Als Problem-Formulierung dient ein Programm, ein Programm-Segment oder eine vergleichbare Quelle (Bild 23.2 a: innere Schleife des Bubble-Sort). Als Lösung entsteht eine dichotomische Hardware-Beschreibung (Bild 23.2 b, vgl. Bild 19.1, s. auch Zur PLA-Realisierung des Steuerwerk siehe Abschnitt 11.4.1 (jedoch getaktet, siehe Beispiel in [41]).

High-Level-Synthese ist im Wesentlichen ein relativ junges Forschungsgebiet [13] [21]. Daher ist auch die Terminologie noch nicht klar ausgeprägt, manchmal widersprüchlich und nicht abgestimmt mit anderen Teilgebieten der Mikroelektronik-Entwurfs-Wissenschaften. High-Level-Synthese läßt sich folgendermaßen definieren:

High-Level-Synthese transformiert eine prozedurale Problem-Beschreibung in eine funktional äquivalente dichotomische Hardware-Beschreibung

Man kann dies auch etwa wie folgt zusammenfassen:

Generierung von Hardware aus Programmen oder Programm-Teilen

Der Ursprung des Begriff "High-Level-Synthese" Systeme (HLS-Systeme) liegt etwa im Jahr 1969. In diesem Jahr wurde am IBM T.J.Watson Research Center in Yorktown Heights das Synthesesystem ALERT [20] entwickelt. Mit ALERT konnte eine in APL geschriebene Hardware-Spezifikation automatisch in eine Logikbeschreibung umgesetzt werden. Eine IBM 1800 Computer, der mit Hilfe von ALERT synthetisiert wurde, benötigte jedoch fast doppelt so viele Hardware-Elemente, wie der manuell ausgeführte Design. Dies war vor etwa 25 Jahren. Inzwischen ist die Methodik weiterentwickelt worden. Während der siebziger Jahre entstanden viele akademische Forschungsprojekte zum Thema High-Level Synthese. Die "System Architects Workbench" [50][51] ist eine bekannter Vertreter dieser Zeit. In den achtziger Jahren begann sich die Industrie für dieses Thema ernsthafter zu interessieren. Es entstanden Forschungsprojekte wie der Yorktown Silicon Compiler [4][6][11] am IBM T.J.Watson Research Center in Yorktown Heights. Ein Tutorium, schon fast eine Übersicht (survey paper) über HLS ist [40].

23.1 Motivation und Teilaufgaben

Ein wichtiges Motiv der High-Level-Synthese ist quasi die Bequemlichkeit der Problem-Formulierung durch ein kleines Programm. Programmiersprachen sind ja allgemein bekannt. Die

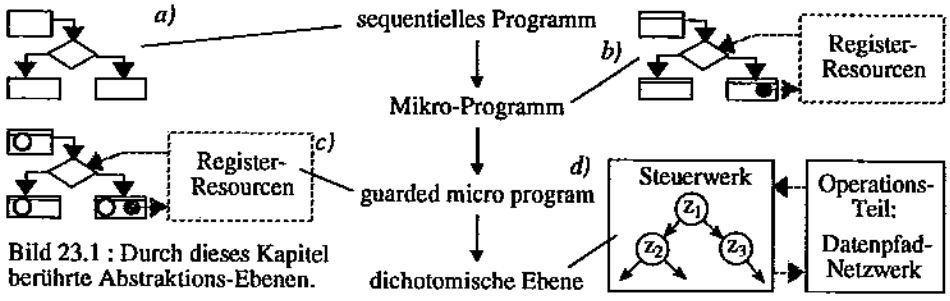


Bild 23.1 : Durch dieses Kapitel berührte Abstraktions-Ebenen.

automatische Generierung von Hardware aus einer solchen Beschreibungsform ist natürlich ein verlockendes Ziel. Es kommt nicht von ungefähr, daß Methoden der noch ganz jungen Disziplin des Hardware-Software-Ko-Design zum Teil aus der High-Level-Synthese stammen. Ein wichtiges Nebenziel muß natürlich die Optimierung sein, was beispielsweise schon frühzeitig durch die Qualität der Ergebnisse des ALERT-Projektes deutlich wurde (siehe oben). Eine Optimierung erfolgt über spezielle Kostenfunktionen, wobei heute neben dem Durchsatz wegen der mikroelektronischen Realisierung vor Allem der Flächenbedarf ein besonders wichtiges Optimierungsziel ist. Bild 23.4 zeigt die Phasen der High-Level-Synthese. Die wichtigsten Schritte sind (siehe auch Bild 23.3):

- Kompilation
- Schritt-Planung (Scheduling)
- Zuteilung (Allocation)

Zwischenformat CDFG. Kompilation dient der Erzeugung eines geeigneten Zwischenformates. Die textuelle, Kontrollfluß-orientierte Hardware-Spezifikation ist für eine maschinelle Weiterverarbeitung ungeeignet. Sie wird sie zunächst in ein Datenfluß-orientiertes Zwischenformat umgesetzt [42], welches allgemein als ***control/ data flow graph (CDFG)*** bezeichnet wird. Das *Yorktown Intermediate Format (YIF)*, sowie der *Value Trace* des System Architects Workbench sind Beispiele für solche Zwischenformate. Die algorithmische Spezifikation des Systems wird im Allgemeinen durch eine prozedurale Hardware-Beschreibungssprache repräsentiert, wie ISPS [2] oder VHDL [12]. Eine solche Spezifikation kann einen einfachen Algorithmus umfassen, wie beispielsweise zur Berechnung der Quadratwurzel, oder ein nichtrekursives Filter, als auch eine komplette Prozessor-Architektur.

Schritt-Planung (Scheduling) ist der wichtigste Schritt in der High-Level-Synthese. Man versteht darunter das Zuordnen der Operationen des CDFG zu den Zuständen des Steuerwerks. Ein Steuerwerks-Zustand wird in diesem Zusammenhang als ***Steuerschritt (engl.: scheduling step)*** bezeichnet. Ziel der Schritt-Planung ist es, entweder die Länge des Schritt-Planes (engl.: schedule) oder die Anzahl der verwendeten Hardware-Resourcen zu minimieren. Man unterscheidet daher je nach Ziel zwei Arten von Scheduling:

- **Zeitkritische Schritt-Planung** (engl.: scheduling under timing constraints):
  die Anzahl der Hardware-Resourcen wird minimiert.
- **Resource-kritische Schritt-Planung** (engl.: scheduling under resource constraints):
  die Länge des Schedules wird minimiert.

Schritt-Planung unter Vorbedingungen (scheduling under constraints). Schritt-Planung wird häufig unter Vorbedingungen (constraints) durchgeführt. Falls die Vorbedingungen zu

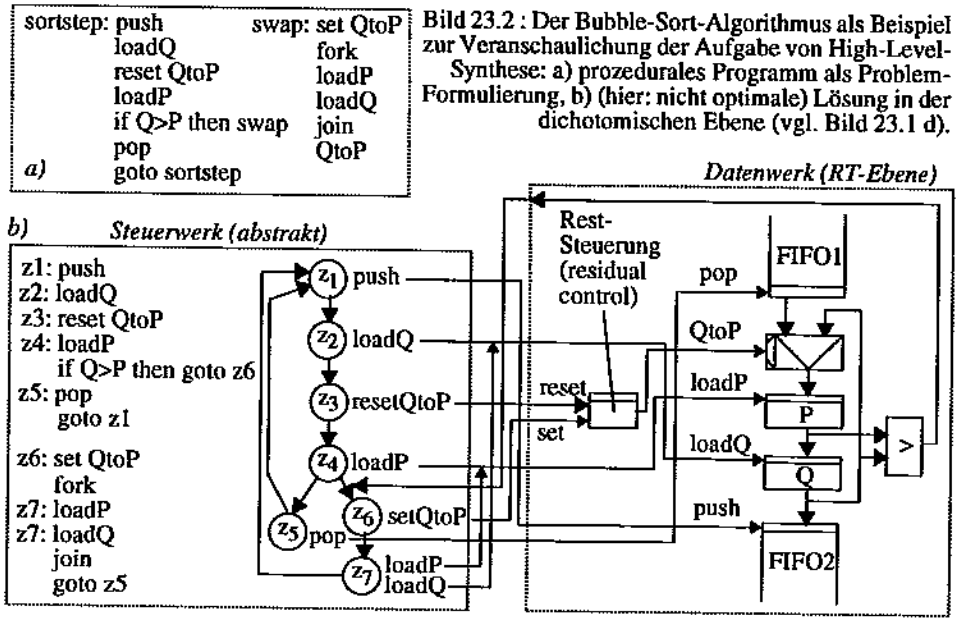


Bild 23.2 : Der Bubble-Sort-Algorithmus als Beispiel zur Veranschaulichung der Aufgabe von High-Level-Synthese: a) prozedurales Programm als Problem-Formulierung, b) (hier: nicht optimale) Lösung in der dichotomischen Ebene (vgl. Bild 23.1 d).

eng spezifiziert werden, können die Schrittplanung keine Lösung ermitteln. Scheduling under constraints bei gleichzeitiger Optimierung ist NP-vollständig [3][23]. Daher verwenden nahezu alle Schrittplanungs-Verfahren Heuristiken, die meist nur sub-optimale Lösungen ergeben. Nur bei sehr kleinen Problemstellungen ist es möglich mit, Hilfe von "Branch-and-Bound-Techniken" optimale Lösungen zu berechnen. Viele Schrittplanungs-Verfahren erwarten darüber hinaus, daß der eingegebene CDFG in einzelne Datenfluß-Graphen partitioniert ist. Jeder Datenflußgraph wird dann separat verplant. Falls die Kontrollfluß-Information bei der Schrittplanung direkt mitberücksichtigt wird, ergeben sich zwei weitere Optimierungsziele:

- optimale Schrittplanung sich gegenseitig ausschließender Operationen
- Pipeline-Schrittplanung

Zuteilung (Allokation). Unter Zuteilung bzw. Allokation (engl.: *allocation*) versteht man die Zuordnung von Operationen des verplanten CDFG zu funktionalen Einheiten (Hardware-Modulen), was natürlich die Bereitstellung von Registern, Multiplexern und Interkonnekt voraussetzt. Ein Ziel der Zuteilung ist die Minimierung dieser Hardware-Resourcen. All diese

Synthese-Phase	Aufgabe
Kompilation	Generierung einer geeigneten Zwischen-Notation
Scheduling	Operatoren an Kontroll-Schritte binden
Allocation	Operatoren und Variablen an Hardware-Resourcen binden

Bild 23.3 : Teilaufgaben der High-Level-Synthese.

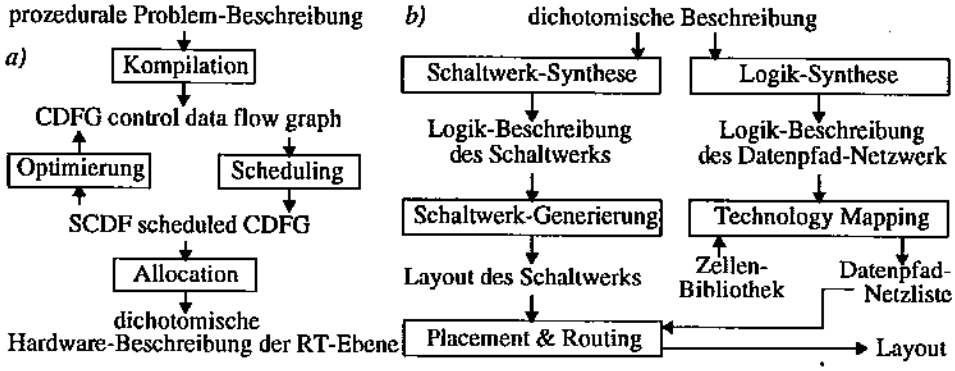


Bild 23.4 : Phasen und Einbettung der High-Level-Synthese: a) Phasen, b) Schritte danach.

Adjazenz-Liste Datenstruktur für Graphen
Adjazenz-Matrix Datenstruktur für Graphen
ALAP as late as possible
ALERT bekanntestes frühes High-Level-Synthese-System (IBM, 1969 [20])
allocation Zuteilung, Allokation
Allokation s. unter Zuteilung
allocated (eine FU ist) zugeteilt
ASAP as soon as possible
CDFG control / data flow graph
CMU Carnegie-Mellon-University
constraint (einschränkende) Vorgaben
DAG directed acyclic graph - ungerichteter azyklischer Graph
DFG Datenfluß-Graph
DG distribution graph
Dichotomie Paarung abstraktes Steuerwerk / in der RT-Ebene dargestelltes Datenwerk
dichotomische Ebene s. Dichotomie
FDS force-directed scheduling
force-directed scheduling Kraft-gesteuerte Schrittplanung (s. Abschnitt 23.3.2.5)
FU functional unit, Funktions-Einheit
HLS High-Level-Synthese
ISPS Hardware-Sprache der CMU
Kontrolle *(control)* Synonym f. Steuerung
Kontroll-Fluß Steuerungs-Fluß
Mobilität (einer Operation): Anzahl k der Steuerschritte, in denen die Operation ausgeführt werden kann unter Einhaltung von Präzedenzen und Vorbedingungen
NP-vollständig höchste Komplexitätsklasse: Rechenzeit exponentiell z. Problemgröße
RT Register-Transfer
SCDF scheduled CDFG
schedule (Steuer-)Schritt-Plan
scheduled (ein Operator ist) eingeteilt
Scheduling s. unter Schritt-Planung
scheduling (Steuer-)Schritt-Planung
Schritt-Planung Zuordnung v. Steuerschritten (control steps) zu Operatoren eines DDG
single assignment code Programm, bei welchem Variablen nur ein einziges Mal ein Wert zugewiesen wird
variable disambiguation Konversion eines Programmes in *Single-Assignment-Kode*
YIF Yorktown Intermediate Format
Zuteilung (engl.: *allocation*) Zuordnung von FUs (Hardware-Resourcen) zu Operatoren

Bild 23.5 : Glossar und Verzeichnis der Abkürzungen.

Minimierungen führen auf das Problem der Suche von vollständigen Subgraphen (Maximal Clique Covering), wiederum ein klassisches NP-vollständiges Problem.

Verzahnung. Schrittplanung und Allokation sind stark miteinander verzahnt. Eine optimale Synthese würde eigentlich eine gleichzeitige Schrittplanung und Allokation erfordern. Jede der beiden Aufgabe ist jedoch schon in sich so komplex, daß nahezu alle HLS-Systems die beiden Aufgaben voneinander trennen. Einige wenige führen die Allokation schon vor dem Schrittplanung durch [7]. Nur bei sehr starker Vereinfachung der Problemstellung und kleinen Eingaben gibt es Lösungsansätze für gleichzeitige Schrittplanung und Allokation [24].

Weitere Verarbeitung (nicht Gegenstand dieses Kapitels). Die Register-Transfer-Beschreibung, die durch die High-Level-Synthese erzeugt wird, kann durch bekannte Verfahren der Logiksynthese, Steuerwerk-Synthese und automatischem Placement&Routing in Layout umgesetzt werden (Bild 23.4 b). Man spricht dabei von *register transfer level synthesis (Synthese der RT-Ebene)*. Diese spaltet sich in Logiksynthese und Steuerwerk-Synthese. Die Logiksynthese wird durch Mehr-Ebenen- (multi-level) Synthesewerkzeuge wie MIS ausgeführt [5]. MIS führt auch eine Technologie-Abbildung (technology mapping) durch. Am wichtigsten bei der Steuerwerk-Synthese ist die optimale Kodierung der Steuerwerk-Zustände. Das Programm MUSTANG [16][18] ist eines der bekanntesten Softwarepakete zur Lösung diese Aufgabe. Das Steuerwerk wird oft als PLA-Lösung realisiert (vgl. Abschnitt 11.4.1). Ein automatisches Placement und Routing generiert schließlich das Layout. Kompilation und Schrittplanung, sowie deren Ein-Ausgabeformate werden in den folgenden Kapiteln behandelt.

23.2 Kompilation & Optimierung

Eine der ersten prozeduralen Hardware-Beschreibungssprachen war ISPS [2]. ISPS wird als Eingabesprache im *System Architects Workbench* verwendet. Die meisten neueren Systeme verwenden VHDL [12]. Eine prozeduralen Hardware-Beschreibungssprachen ist sehr ähnlich zu einer höheren Programmiersprache. Daher wurden auch Programmiersprachen als Eingabesprachen für die High-Level-Synthese verwendet, wie ADA [26] oder C [17]. Dazu muß die Programmiersprache lediglich um die folgenden Mechanismen erweitert werden:

- Deklaration von Schaltungs-Ein- und Ausgängen
- Deklaration von Bitbreiten für Variablen
- Spezifikation von Timing Constraints für Operationen

Wie schon zuvor erwähnt, wird eine algorithmische Hardware-Spezifikation vor dem Scheduling in ein datenflußorientiertes Zwischenformat genannt Control/Data-Flow-Graph umgesetzt. Zunächst soll der Begriff Datenflußgraph genauer erläutert werden.

Ein Datenflußgraph ist ein azyklischer gerichteter Graph $DFG(O, D)$ bestehend aus n Operationen $O = \{o_i \mid 1 \le i \le n\}$ und einer Relation $D = \{(o_i, o_k) \mid 1 \le i \le n, 1 \le k \le n\}$, welche die Datenabhängigkeiten zwischen Operationen spezifiziert. $(o_i, o_k) \in D$ bedeutet,

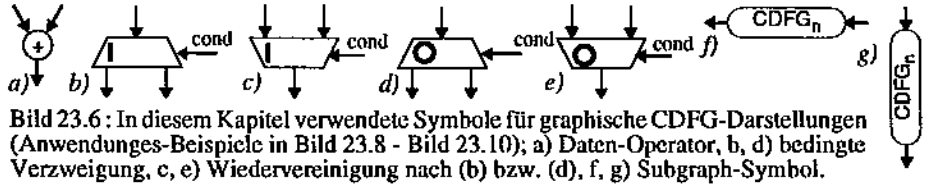


Bild 23.6 : In diesem Kapitel verwendete Symbole für graphische CDFG-Darstellungen (Anwendunges-Beispiele in Bild 23.8 - Bild 23.10); a) Daten-Operator, b, d) bedingte Verzweigung, c, e) Wiedervereinigung nach (b) bzw. (d), f, g) Subgraph-Symbol.

daß Operation o_i beendet sein muß, bevor Operation o_k beginnt. Gerichtete Graphen werden im Allgemeinen über Adjazenzlisten oder Adjazenzmatrizen dargestellt. Diese beiden Arten der Darstellung können jedoch nur kommutative Operationen behandeln, darüberhinaus benötigen Adjazenzmatrizen zu viel Speicherplatz. Daher wird in der High-Level Synthese eine Graph-Darstellung verwendet, ähnlich der Netzlisten-Darstellung von digitalen Schaltungen.

Operatoren werden als Operator-Knoten modelliert, die Eingangs- und Ausgangs-Ports besitzen. Alle mit einem Ausgangs-Ports verbundenen Eingangs-Ports werden in einer Liste zusammengefaßt. Diese Kombination von Ausgangs-Port- und Eingangs-Port-Liste wird als Datenfluß "f" bezeichnet. Schaltungseingänge und Konstanten werden als Datenflüsse ohne Ausgangs-Port repräsentiert. Datenflüssen mit leerer Eingangs-Port-Liste repräsentieren Schaltungsausgänge.

```
input: a
output: c

b := (a + 2) * 3;
b := b + 1;
c := b + a;
```

a)

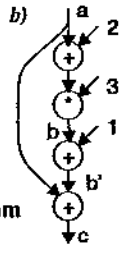


Bild 23.7 : Programm (a) und DFG (b).

Single-Assignment-Form. Reine Datenflußgraphen reichen zur Darstellung von algorithmischen Spezifikationen ohne prozedurale Kontrollfluß-Konstrukte wie if-Anweisungen oder Schleifen (Bild 23.7). Jede Variable wird einem Datenfluß zugeordnet. Falls eine Variable mehrfach auf der linken Seite einer Zuweisung auftaucht, wird für jede Zuweisung ein eigener Datenfluß generiert (b'). Dies entspricht der Konversion eines Programmes in *Single Assignment Code* und wird auch variable *disambiguation* genannt. Komplexe Ausdrücke werden in Operatorbäume zerlegt, welcher aus unären, binären oder ternären Operatoren bestehen. Der ternäre Operator entspricht dem Select Operator der Programmiersprache C (condition ? true_value : false_value) und wird durch einen Multiplexer implementiert. Er ist nicht zu verwechseln mit dem Select-Control-Operator, der im folgenden Anschnitt behandelt wird.

```
if (i<2) then
  i := i+1
else
  i := i-3;
```

a)

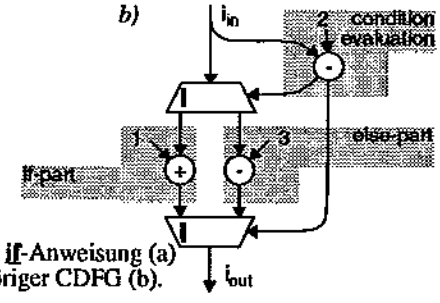


Bild 23.8 : if-Anweisung (a) und zugehöriger CDFG (b).

Control / Data Flow Graph (CDFG). Zur Darstellung von if-Anweisungen und Schleifen muß der Datenflußgraph um spezielle Kontroll-Operatoren erweitert werden. Der so erweiterte Datenflußgraph dann ein Control/Data Flow Graph (CDFG).

Für jeden Zweig einer IF-Anweisung werden zwei getrennte CDFG Subgraphen erzeugt, in Bild 23.9 bezeichnet mit $CDFG_{true}$ und $CDFG_{false}$. Für jede Variable die innerhalb der IF-Anweisung vorkommt wird ein Select Branch und Select Merge Control-Knoten Paar erzeugt. Abhängig von der Auswertung der Bedingung in der if-Anweisung wird der Datenfluß f_{in} entweder mit f_{true_in} oder f_{false_in} fortgesetzt (Symbole: s. a. Bild 23.6).

Die f_{true_in} Datenflüsse aller zusammengehöriger Select Branch Control-Knoten bilden Eingänge für den CDFG-Subgraph $CDFG_{true}$. Jene Ports des Select Branch Control-Knoten, die mit $CDFG_{true}$ verbunden sind werden durch einen senkrechten Strich markiert. Die Datenflüsse f_{false_in} und f_{false_out} sind Ein- und Ausgänge für den CDFG-Subgraphen $CDFG_{false}$. Bild 23.8 zeigt ein Beispiel für die Darstellung einer IF-Anweisung als CDFG.

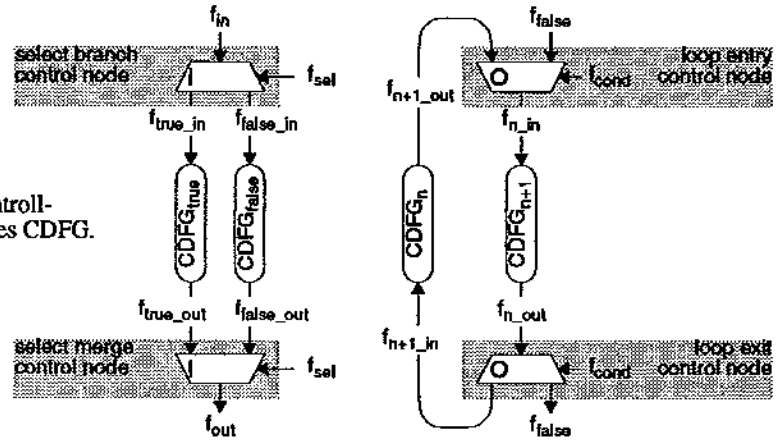


Bild 23.9 : Kontroll-Operatoren eines CDFG.

Disjunkte Subgraphen. Bei Iterationen werden ebenfalls zwei disjunkte Subgraphen erzeugt, $CDFG_{n+1}$ und $CDFG_n$ (Bild 23.9). Die Indizes geben einen Hinweis darauf, wie oft der entsprechende Subgraph ausgeführt wird. Ein Loop-Entry und Loop-Exit Knotenpaar wird für jede in der Schleife verwendete Variable erzeugt. Beim ersten Eintritt in die Schleife wird $CDFG_{n+1}$ ausgeführt, unabhängig davon, welchen werden f_{cond} besitzt.

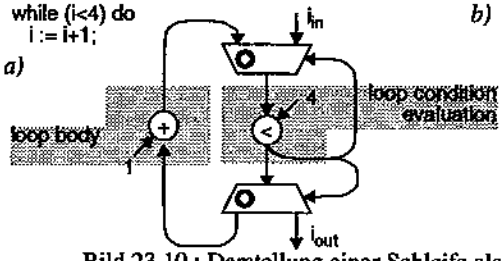


Bild 23.10 : Darstellung einer Schleife als CDFG: a) Fragment des Algorithmus, b) CDFG.

Solange dann f_{cond} den Wert "true" besitzt, werden $CDFG_n$ und $CDFG_{n+1}$ nacheinander ausgeführt, so lange bis f_{cond} "false" ist. Danach ist $CDFG_n$ n mal ausgeführt worden und $CDFG_{n+1}$ n+1 mal. Bild 23.10 zeigt ein Beispiel für die Umsetzung einer Schleife in einen CDFG.

Design Data Structure (DDS). Die hier beschriebene CDFG Darstellung ist vollständig Datenfluß-orientiert, dadurch daß Pseudo-Knoten zur Darstellung von Prozeduralen Konstrukten in einen Datenflußgraphen eingefügt werden. Diese Art der Darstellung entspricht der "Design Data Structure" (DDS) des ADAM Systems [33] oder dem "Yorktown Intermediate Format" (YIF) [11]. Der "System Architects Workbench" von der Carnegie Mellon Universität verwendet ein mehr Kontrollfluß-orientiertes Zwischenformat, den "Value Trace" (VT) [39]. Einige Systeme verwenden auch Parse-Trees [17], obwohl diese Darstellung für eine Datenflußanalyse erhöhten Rechenaufwand erfordert. Die Bezeichnung CDFG für das Zwischenformat wurde von dem HAL System der Universität von Ottawa [44] übernommen.

Optimierungen. Nach der Übersetzung in den CDFG werden noch verschiedene Optimierungen ausgeführt [47]. Es handelt sich dabei um Optimierungen, ähnlich wie sie von Compilern für höhere Programmiersprachen durchgeführt werden wie die Propagierung von Konstanten und die Elimination von "dead code" und gemeinsamer Unter-Ausdrücke. Darüber hinaus wer-

den noch Hardware-Spezifische Optimierungen ausgeführt, wie etwa die Ersetzung von Multiplikationen mit 2^i durch Schift-Operationen.

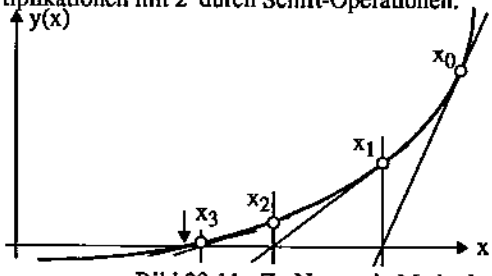


Bild 23.11 : Zu Newton's Methode.

Ein einfaches Beispiel [40] zur Veranschaulichung der Synthese-Phasen (Bild 23.12) ist Teil eines Programm zur Berechnung der Quadratwurzel von X nach Newton's Methode.(Repetitorium:) Die Methode sucht den Punkt y(x)=0 einer Funktion y(x) durch sukzessive Ableitung von Tangenten ab einem guten Anfangspunkt x_0 (siehe Bild 23.11). Von jedem Punkt x_i wird ein besserer Punkt x_{i+1} abgeleitet gemäß folgender Gleichung:

$$x_{i+1} = x_i - \frac{y(x_i)}{y'(x_i)} \quad \text{wobei:} \quad y'(x_i) \neq 0 \tag{23.1}$$

Zur Ableitung der Quadratwurzel $\sqrt[2]{a}$ verwenden wir die Form:

$$y(x) = x^2 - a \quad \text{wobei aus } y(x)=0 \text{ folgt:} \quad x^2 = a \quad \text{or:} \quad x = \sqrt[2]{a} \tag{23.2}$$

In der Praxis sind nur wenige Iterationen nötig (4 für unser Beispiel). Eine Minimax polynomische Approximation ersten Grades (nämlich eine Gerade) für das Intervall [1/16, 1] ergibt den Anfangswert.

```
Y := 0.222222 + 0.888889 * X;
I := 0;
DO UNTIL  I > 3  LOOP
Y := 0.5 * ( Y + X / Y );
I := I + 1;
ENDDO;                                  a)
```

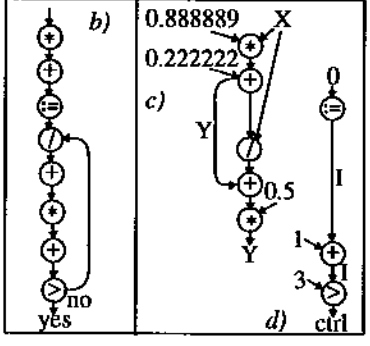


Bild 23.12 : Quadratwurzel sqrt: a) Spezifikation (high-level specification), b-d) interne Darstellungsform: b) Kontroll-Graph (KG) , c-d) Datenfluß-Graph (DFG).

Die graphische Darstellung des Algorithmus (Bild 23.12) zeigt Datenfluß-Graph (c-d) und Kontroll-Graph (b) getrennt zwecks besserer Lesbarkeit. Der ***Kontroll-Graph (KG)*** wird direkt von der explizit durch das Programm (Bild a) gegebenen Sequenz abgeleitet, wie Compiler arithmetische Ausdrücke erfassen würde.

Der ***Datenfluß-Graph (DFG)*** zeigt die wesentliche Ordnung der Operationen, impliziert durch die Datenabhängigkeiten in der Spezifikation. Beispielsweise in Bild 23.12 hängt die Addition oben im Diagramm von der Verfügbarkeit der Ergebnisse der Multiplikation ab.

Deshalb muß die Multiplikation zuerst ausgeführt werden. Andererseits gibt es keine Datenabhängigkeit zwischen der Operation I + 1 innerhalb der Schleife und irgendeiner Operation innerhalb der Kette, die Y berechnet, obwohl diese durch den gezeigten Kontroll-Graphen geordnet sind.

Parallelität. Deshalb kann I + 1 sowohl parallel zu diesen Operationen ausgeführt werden, als auch vor oder nach diesen. Der DFG kann auch dazu benutzt werden, Abhängigkeiten zu entfernen gegenüber der Art und Weise wie die internen Variablen in der

Spezifikation verwendet wurden, weil jeder Wert, generiert durch eine Operation und verbraucht durch eine andere Operation, eindeutig durch eine Kante des Graphen dargestellt ist. Diese ***Präzisierung der Variablen (variable disambiguation)***, ähnlich dem was Compiler während der globalen Datenfluß-Analyse tun, ist wichtig zur Neuordnung der Operationen und Vereinfachung der Datenpfade.

Optimierung. Einige optimierende Transformationen, wie die Vereinfachung von Ausdrücken, können bereits während dieser Phase durchgeführt werden. In Bild 23.12, beispielsweise, kann 0.22 + 0.88 * X als 0.22 * (1 + 4 * X) geschrieben werden, was einfacher zu implementieren ist, da die Multiplikation durch einen Schift ersetzt werden kann. Wenn man die richtigen Drähte aufgreift, brauch man nur eine Multiplikation und ein Inkrement anstelle einer Multiplikation und einer Addition (mehr Details in Bild 23.13).

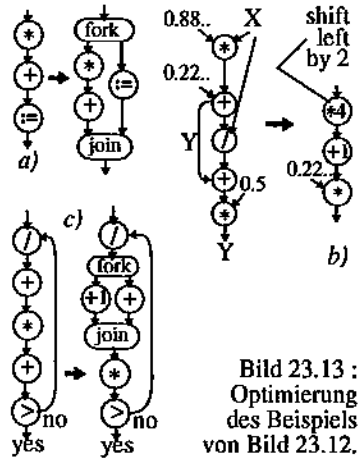


Bild 23.13 : Optimierung des Beispiels von Bild 23.12.

Eine weitere lokale Transformation ist das sogenannte *Schleifen-Aufrollen (loop unrolling).* Auch mehr Hardware-spezifische lokale Transformation können angewandt werden. In unserem Beispiel,

- Das Schleifenende-Kriterium kann zu I = 0 geändert werden mit einer Zwei-Bit-Variablen für I.
- Die Multiplikation mal 0.5 kann durch einen Rechts-Schift um eins ersetzt werden, durch Wegwerfen des LSB.
- Die Addition von 1 zu I kann durch eine Inkrement-Operation ersetzt werden.

Bild 23.14 a zeigt die interne Darstellung nach diesen Optimierungen. Schleifen-Aufrollung ist auch in diesem Fall möglich, da die Anzahl der Iterationen kein und konstant ist(Bild 23.14 b).

In unserem Beispiel verwendet ein nicht optimierter Geradeaus-Schrittplan nur eine einzige FU und ein Ein-Port-Speicher (Bild 23.15 a). Jede Operation benötigt einen eigenen Steuerschritt, sodaß die Berechnung 3 + 4 * 5 = 23 Kontroll-Schritte dauert. Werden statt Speicher Einzel-Register für Y und I und ein Register T benutzt, dann benötigen die Operationen I: = 0 und Schift keinen eigenen Kontroll-Schritt, und die Berechnung erfordert nur 2 + 4 * 4 = 18 Schritte (Bild 23.15 b). 545

Register I hat seine eigene Flächen-effiziente Reset-Hardware die parallel zu Kontroll-Schritt

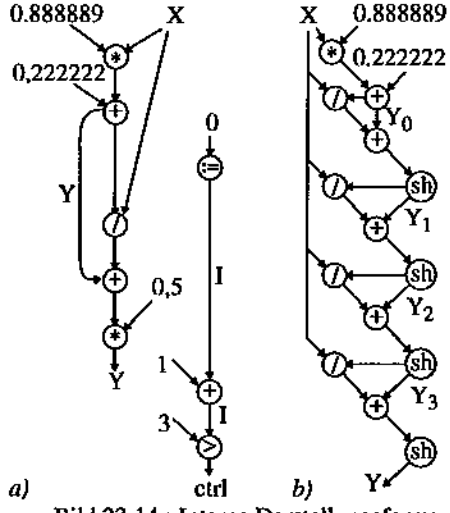


Bild 23.14 : Interne Darstellungsform; a) nach etwas Optimierung, b) nach dem Aufrollen der Schleife (loop unrolling).

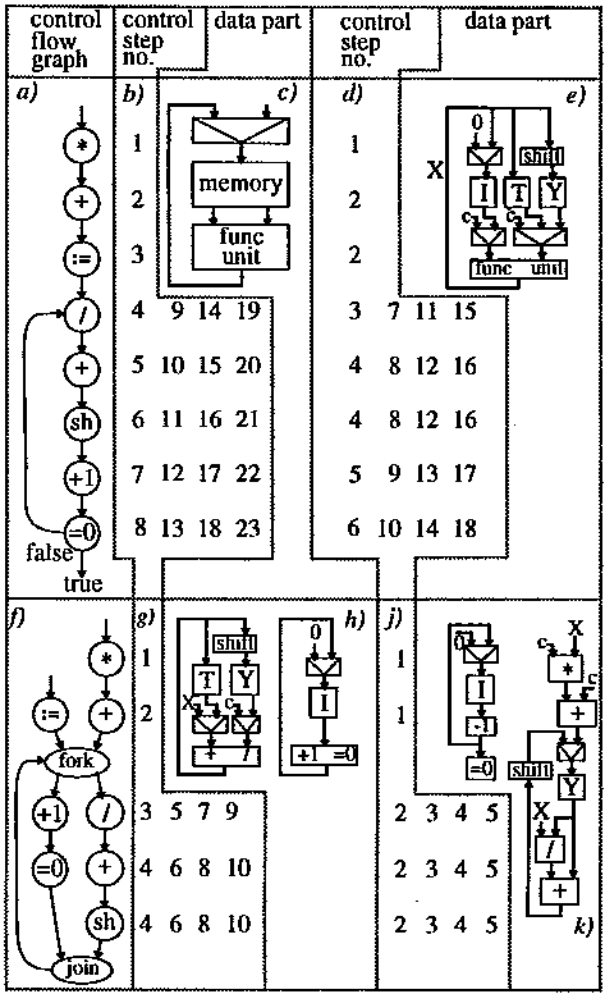


Bild 23.15 : Einige mögliche Steuerschritt-Pläne und darauf aufbauende Datenpfad-Zuteilungen (Allokationen).

Nr. (1 oder) 2 laufen kann. Die Schift-Operation erfordert nur eine Verdrahtungs-Änderung am Register-Eingang, sodaß neben Addition und "register write" kein extra Kontroll-Schritt benötigt wird.

Zur weiteren Beschleunigung der Berechnung, können die Operationen im Kontroll-Graph mit maximaler Parallelität eingeteilt werden durch Verdichtung der Kontroll-Schritte, indem nur die wesentlichen

Abhängigkeiten beachtet werden, die durch den DFG und die Schleifen-Grenzen gegeben sind (s. Bild 23.15 c für unser Beispiel). Man beachte je zwei Dummy-Knoten zur Markierung der Schleifen-Grenzen. Da die Schift-Operation frei ist, können bei 2 FUs die Operationen nun in 2 + 4 * 2 = 10 Kontroll-Schritten eingeteilt werden. 546

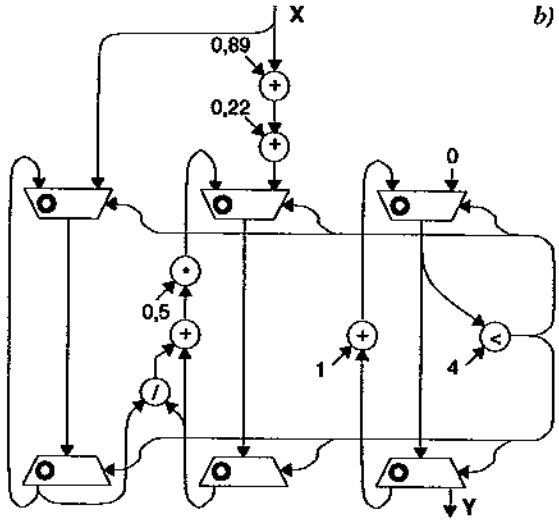
Chaining. Die Zahl der Kontroll-Schritte kann weiter reduziert werden durch Ausführung geordneter Operationen im gleichen Kontroll-Schritt, (was oft "chaining" genannt wird). In unserem Beispiel kann die Initialisierung in einem Kontroll-Schritt ausgeführt werden, und dann die Iteration in einem anderen Kontroll-Schritt, sodaß die Berechnung nur 5 Schritte erfordert. (Bild 23.15 d).

Noch keine Pipelines. Bisherige Betrachtungen gingen von nicht-überlappenden Operationen aus, d. h., die Operationsfolge hat keine Pipeline-Anordnung. Die bisher in diesem Kapitel beschriebenen Techniken können jedoch erweitert werden für den Umgang mit Pipeline-Strukturen. Einige Literaturhinweise werden in Abschnitt 23.4 referenziert.

Zuteilung (Allokation). In der Allokation ist ein Ziel die **Minimierung** des Hardware-aufwandes. Eine Gesamt-Minimierung ist meist zu kompliziert, sodaß häufig Subsysteme

```
Y := 0,22 + 0,89 * X;
i := 0;
while i<4 do
  Y := 0,5 * (Y + X/Y);
  i := i + 1;
endwhile
```

a)



b)

Bild 23.16 : Quadratwurzel-Berechnung und zugehöriger CDFG: a) Algorithmus, b) CDFG.

separat minimiert werden, obwohl dies im Allgemeinen zu suboptimalen Ergebnissen führt. Diese Gruppierung der Operationen wird oft beeinflußt durch die Möglichkeiten der FUs. Wenn beispielsweise keine FU vorhanden ist, die sowohl addieren als auch multiplizieren kann, so müssen Additionen und Multiplikationen in separaten Gruppen gehalten werden. Diese Allokation von FUs für die Schrittpläne in Bild 23.15 a, b, und c, ist in diesem Sinne minimal. Für die Allokation in Bild 23.15 d ist dies nicht der Fall. Da die beiden Addierer niemals gleichzeitig aktiv sind, müßte ein einziger Addierer genügen; allerdings werden dann zusätzliche Multiplexer benötigt.

Steuerwerk-Synthese. Sind Schrittplan und Datenpfade gewählt, wird dies durch Synthese eines entsprechenden Steuerwerkes realisiert, wobei folgende zwei Ansätze möglich sind.

Wird eine fest **verdrahtete Steuerung** (engl.: ***hardwired control***) gewählt, so entspricht ein Steuerschritt dem Zustand eines endlichen Automaten (engl.: *finite state machine* oder *FSM*). Wenn Ein- und Ausgänge der FSM (also die Schnittstelle zum Datenteil) bestimmt worden sind (während der allocation), kann die FSM mit bekannten Methoden synthetisiert werden, incl. Zustands-Kodierung und Optimierung der Schaltnetze [16], [18]. Wurde jedoch die **mikroprogrammierte Steuerung** (engl.: ***microcoded control***) gewählt, so entspricht ein Steuerschritt einem Mikroprogramm-Schritt.

Ein komplexeres Beispiel. Bild 23.10 zeigt ein komplexeres Beispiel für einen CDFG. Das Programmfragment in Bild 23.10 berechnet die Quadratwurzel $y = \sqrt{x}$, nach dem Newton-Verfahren. Die Anzahl der benötigten Iterationen ist sehr klein (im Beispiel: 4). Eine lineare Approximation der Wurzelfunktion durch die Punkte $y_1 = 1/16, x_1 = 1/4$ und $y_2 = 1, x_2 = 1$ liefert den Startwert für das Newton-Verfahren .

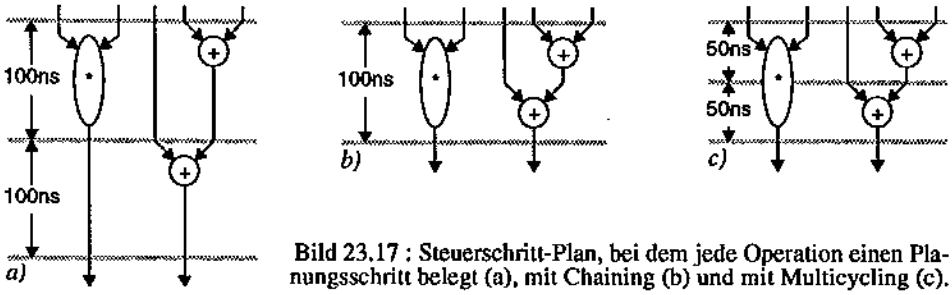


Bild 23.17 : Steuerschritt-Plan, bei dem jede Operation einen Planungsschritt belegt (a), mit Chaining (b) und mit Multicycling (c).

23.3 Schrittplanung (Scheduling)

Resultat der High-Level-Synthese ist eine Register-Transfer-Beschreibung (Bild 23.4). Dabei wird von einem festen Repertoire von Funktionalen Einheiten ausgegangen. Aufgabe des Scheduling ist die Zuordnung von CDFG-Operationen zu Zuständen eines Steuerwerks. Die ist die Wahl der Taktperiode des Steuerwerks ist von entscheidender Bedeutung. Wählt man die Taktzeit des Steuerwerks so, daß sie der maximalen Verzögerungszeit der zur Verfügung stehenden Funktionalen Einheiten entspricht, kann eine sogenannte Kontrollschritt-Planung (control step scheduling) ausgeführt werden. Pro Taktschritt führt jede funktionale Einheit genau eine Operation des CDFG aus.

Schrittplanungs-Verfahren. Oftmals ist die Ausführungszeit der funktionalen Einheiten sehr unterschiedlich. Dadurch kommt es zu einer schlechten Auslastung von funktionalen Einheiten mit kurzen Ausführungszeiten. Man wählt dann eine Taktperiode, die dem Mittel der Verzögerungszeiten aller Funktionalen Einheiten entspricht, und gleicht abweichende Ausführungs-zeiten durch "Chaining" und "Multicycling" aus (Bild 23.17). Schrittplanungs-Verfahren mit Chaining und Multicycling werden im Allgemeinen immer aus Kontrollschritt-Planungs-Verfahren (control step scheduling) abgeleitet, daher ist die nachfolgende Betrachtung der Schrittplanungs-Verfahren auf Kontrollschritt-Planung beschränkt.

In der Einleitung wurde nach Scheduling Ziel in zwei Arten von Scheduling klassifiziert:

- Zeitkritische Schrittplanung (scheduling under timing constraints): die Länge des Schrittplans wird minimiert.
- Resource-kritische Schrittplanung (scheduling under resource constraints): Hardware-Resourcen werden minimiert.

Schrittplanung unter Nebenbedingungen (scheduling under constraints) ist ein NP-vollständiges Problem, d.h.jeder korrekte Algorithmus zur Lösung dieses Problems benötigt exponentielle Rechenzeit. Ein Sonderfall entsteht, wenn bei der Resource-kritischen Schrittplanung von einer unbegrenzten Menge von zur Verfügung stehender funktionaler Einheiten ausgegangen wird. In diesem Fall kann ein kürzestes Schedule mit quadratischem Rechenaufwand ermittelt werden. Verfahren hierfür sind das **"As Soon As Possible" (ASAP)**-Scheduling und **"As Late As Possible" (ALAP)**-Scheduling. Beide Verfahren sind die Grundlage für heuristische Methoden beim Scheduling under Constraints.

Eingabe für das Scheduling ist der Control/Data Flow Graph CDFG(O, D), mit n Operationen $O = \{o_i \mid 1 \le i \le n\}$.

Die Datenabhängigkeiten werden durch $D = \{ (o_i, o_k) \mid 1 \le i \le n, 1 \le k \le n, i \ne k \}$ beschrieben. Eine Datenabhängigkeit $(o_i, o_k) \in D$ wird zwecks besserer Lesbarkeit nachfolgend durch $o_i \rightarrow o_k$ dargestellt. Operation o_i heißt direkter Vorgänger von o_k. Operation o_k heißt direkter Nachfolger von o_i. Die Funktionen $\mathrm{DSucc}(o_i)$ und $\mathrm{DPred}(o_i)$ liefern die direkten Vorgänger und Nachfolger einer der Operation o_i:

$$\mathrm{DSucc}(o_i) = \{ o_k \mid o_i \rightarrow o_k \in D \} \tag{23.3}$$

$$\mathrm{DPred}(o_i) = \{ o_k \mid o_k \rightarrow o_i \in D \} \tag{23.4}$$

Die Menge aller Vorgängeroperationen $\mathrm{Succ}(o_i)$ beschreibt alle Operationen für die es einen Pfad $o_j \rightarrow \ldots \rightarrow o_k$ gibt. Die Menge aller Vorgänger-Operationen $\mathrm{Pred}(o_i)$ beschreibt alle Operationen für die es einen Pfad $o_k \rightarrow \ldots \rightarrow o_j$ gibt.

$$\mathrm{Succ}(o_i) = \{ o_k \mid \exists o_j \rightarrow \ldots \rightarrow o_k \} \tag{23.5}$$

$$\mathrm{Pred}(o_i) = \{ o_k \mid \exists o_k \rightarrow \ldots \rightarrow o_j \} \tag{23.6}$$

23.3.1 Schrittplanungs-Algorithmen

Es gibt zweielementare Klassen von Schrittplanungs-(Scheduling-) Algorithmen. Es gibt ***transformationelle Algorithmen*** und ***iterativ-konstruktive Algorithmen***. Ein transformationeller Algorithmus beginnt mit einem Default-Schrittplan (default schedule), gewöhnlich entweder maximal seriell oder maximal parallel, und leitet andere Schritt-Pläne daraus durch Transformationen ab. Die fundamentalen Transformationen erfolgen durch Wandlung von Serien von Operationen oder Blöcken von Operationen in eine parallele Anordnung und umgekehrt (Bild 23.18). 549

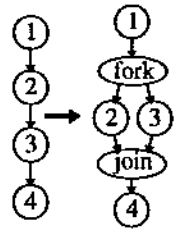


Bild 23.18 : Elementare Schrittplan-Transformationen.

Transformationelle Algorithmen

Transformationelle Methoden der Optimierung sind schon ab Seite 469 eingeführt worden, wenngleich nicht direkt unter dieser Überschrift (siehe auch Bild 23.13, Bild 23.14 und Bild 23.15, sowie Bild 23.18.

Iterativ-konstruktive Algorithmen

Die andere Klasse von Algorithmen, die iterativ-konstruktiven Algorithmen, bauen einen Schrittplan (schedule) auf durch sukzessives Hinzufügen einzelner Operationen. Dies ist ein Unterschied darin, wie die nächste einzuteilende Operation gewählt wird, und wie bestimmt wird, wohin jede Operation eingeteilt wird.

23.3.2 ASAP and ALAP Schrittplanung

Der ASAP-Schrittplanungs-Algorithmus wird wie folgt definiert: Für einen gegebenen Datenflußgraph $\mathrm{CDFG}(O, D)$, werden sukzessive Operationsmengen $\mathrm{ASAP}_j, j = 1, \ldots, \mathrm{SMAX}$ berechnet, für die gilt: $o_i \in \mathrm{ASAP}_j$ genau dann wenn j der frühestmögliche Zeitpunkt ist, and dem Operation o_i ausgeführt werden kann. $o_i \in \mathrm{ASAP}_j$ bedeutet, daß Operation o_i in Scheduling Step j "gescheduled" wird.

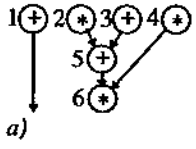
Zunächst werden alle Operationen, die keine direkten Vorgänger haben in den ersten Scheduling Step gesetzt. Schrittweise werden alle direkten Nachfolgermengen berechnet, so lange bis keine Nachfolger mehr existieren. Die Variable SMAX liefert die Länge des Schedules. Die Funktion S liefert die Nummer des Scheduling Steps dem Operation o_i durch das ASAP-Scheduling zugeordnet wurde:

$$S(o_i) = j \Leftrightarrow o_i \in \text{ASAP}_j. \tag{23.7}$$

Beginnt man das Scheduling mit allen Operationen, die keine direkten Vorgänger haben und setzt alle direkten Vorgänger in Scheduling Step j-1 dann erhält man die ASAP Scheduling Prozedur.

23.3.2.1 Algorithmus: ASAP-Schrittplanung

Die einfachste Art der Schrittplanung, die ASAP-Schrittplanung (as soon as possible (ASAP) scheduling), arbeitet lokal in der Wahl sowohl der nächsten einzuteilenden Operation als auch, wo diese plaziert wird. Es wird davon ausgegangen, daß die Anzahl der FUs bereits festliegt. Operationen werden erst topologisch sortiert; d. h., wenn Operation x_i darauf beschränkt (constrained) ist, der Operation x_j zu folgen wegen erforderlichem Datenfluß oder einer Kontroll-Relation, dann folgt x_j der Operation x_j in der topologischen Reihenfolge. Operationen werden einzeln genommen in dieser Reihenfolge und jede wird an den frühestmöglichen Steuerschritt gesetzt, im Rahmen ihrer Abhängigkeit von anderen Operationen und den Schranken der Resourcen-Verwendung. Bild 23.19 zeigt einen DFG und dessen ASAP-Schrittplan. sso



b)

step no.	*	+
1	2	1
2	4	3
3		5
4	6	

Bild 23.19 : Ein ASAP-Steuerschritt-Plan (b) zu: a) DDG (Datenabhängigkeitsgraph).

Ein Problem mit diesem Algorithmus besteht darin, daß Operationen in kritischen Pfaden keine Priorität gegeben wird, sodaß bei begrenzten Resourcen weniger kritischen Operationen knappe Resourcen zuerst zugeteilt werden und daher kritische Operationen blockiert werden. Dies wird gezeigt in Bild 23.19, wo Operation 1 für früher eingeteilt ist als Operation 3, die sich im kritischen Pfad befindet, sodaß 3 für später eingeteilt ist als nötig, Dies erzwingt einen Schedule, der länger ist als das Optimum.

$\text{ASAP}_1 := \emptyset;$

for all o_i with $\text{DPred}(o_i) = \emptyset$

  $\text{ASAP}_1 := \text{ASAP}_1 \cup \{o_i\}$

$j := 1;$

while $\text{ASAP}_j \neq \emptyset$

  $\text{ASAP}_{j+1} := \emptyset;$

  for all $o_i \in \text{ASAP}_j$

for all $o_k \in \mathrm{DSucc}(o_i)$ with all $\mathrm{DPred}(o_k) \subseteq \bigcup_{r=1}^{j} \mathrm{ASAP}_r$

$\mathrm{ASAP}_{j+1} := \mathrm{ASAP}_{j+1} \cup \{o_k\}$;

j := j+1;

endwhile;

SMAX := j-1;

23.3.2.2 Algorithmus: ALAP-Schrittplanung

Der ALAP-Schrittplanungs-Algorithmus wird wie folgt definiert: Für einen Datenflußgraph CDFG(O, D), werden sukzessive Operationsmengen $\mathrm{ALAP}_j, j = 1, \ldots, \mathrm{SMAX}$ berechnet, für die gilt: $o_i \in \mathrm{ALAP}_j$ genau dann wenn j der spätestmögliche Zeitpunkt ist, and dem Operation o_i ausgeführt werden kann. $o_i \in \mathrm{ALAP}_j$ bedeutet, daß Operation o_i in Planungs-Schritt (Scheduling Step) j "gescheduled" wird. Es folgt der Algorithmus:

$\mathrm{ALAP}_{\mathrm{SMAX}} := \emptyset$;

for all o_i with $\mathrm{Succ}(o_i) = \emptyset$

$\mathrm{ALAP}_{\mathrm{SMAX}} := \mathrm{ALAP}_{\mathrm{SMAX}} \cup \{o_i\}$

j := SMAX;

while $\mathrm{ALAP}_j \neq \emptyset$

$\mathrm{ALAP}_{j-1} := \emptyset$;

for all $o_i \in \mathrm{ALAP}_j$

for all $o_k \in \mathrm{DPred}(o_i)$ with all $\mathrm{DSucc}(o_k) \subseteq \bigcup_{r=j}^{\mathrm{SMAX}} \mathrm{ASAP}_r$

$\mathrm{ALAP}_{j-1} := \mathrm{ALAP}_{j-1} \cup \{o_k\}$;

j := j-1;

endwhile;

Die Funktion L liefert die Nummer des Steuerschritts (scheduling step) dem Operation o_i durch die ALAP-Schrittplanung zugeordnet wurde:

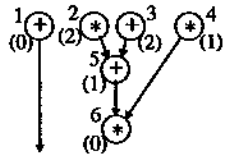
$$L(o_i) = j \Leftrightarrow o_i \in \mathrm{ALAP}_j \tag{23.8}$$

23.3.2.3 Listen-Schrittplanung

Bei Listen-Schrittplanung (engl.: List Scheduling [15]) handelt es sich um eine Heuristisches Verfahren für Resource-kritische Schrittplanung. Listen-Schrittplanung verwaltet in einer Liste ***"fertige Operationen" (ready operations)***. Das sind Operationen deren sämtliche Vorgänger bereits eingeteilt wurden. Jeder zu belegende Planungsschritt wird aus der Liste der fertigen Operationen gefüllt, bis alle zur Verfügung stehenden funktionalen Einheiten belegt sind. Alle nicht mehr in den Kontrollschritt passenden Operationen werden in den nächsten Steuerschritt (scheduling step) verschoben. Wenn die Liste leer ist wird sie neu berechnet und das List

Scheduling wird mit beim nächsten Planungsschritt fortgesetzt, so lange bis alle Operationen eingeteilt sind. Die Liste wird dabei nach einem Prioritäts-Kriterium sortiert. Dies ist im allgemeinen die Mobilität der Operationen (siehe "Force Directed Scheduling"). Operationen mit geringster Mobilität werden zuerst eingeteilt.

Listen-Schrittplanung bewältigt dieses Problem durch ein globaleres Kriteriums für die Auswahl der nächsten einzuteilenden Operation. Für jeden einzuteilenden Steuerschritt werden die Operationen die bei diesem Steuerschritt zur Einteilung verfügbar sind (also diejenigen, deren Vorläufer schon eingeteilt sind), in einer Liste geführt, geordnet nach einer Prioritäts-Funktion. Alle Operationen aus der Liste werden der Reihe nach genommen und eingeteilt wenn die benötigten Resourcen in diesem Schritt noch frei sind; andernfalls wird dies aufgeschoben bis zum nächsten Schritt. Wenn keine Operationen mehr eingeteilt werden können, geht der Algorithmus zum nächsten Steuerschritt, die verfügbaren Operationen werden ermittelt und geordnet, und die Prozedur wird wiederholt. Dies setzt sich fort bis alle Operationen eingeteilt sind.



Schrittplan

Schritt Nr.	*	+
1	2	3
2	4	3
3	6	1

Bild 23.20 : Listen-Schrittplan zu Bild 23.19.

Bild 23.20 zeigt den Listen-Schrittplan zum Graph in Bild 23.19. Hier ist die Priorität, in Klammern angegeben an jedem Knoten, ist die Länge des Pfades von der Operation bis zum Ende des Blocks. Da Operation 3 eine höhere Priorität hat als Operation 1, wird sie zuerst eingeteilt, was in diesem Fall einen optimalen Schrittplan ergibt.

23.3.2.4 Formale Definition zeitkritischer Schrittplanung:

Zeitkritische Schrittplanung (time constrained scheduling) ist die Suche nach der Minimalmenge von Hardware-Resourcen für eine gegebene Zeitschranke. Die Zeitschranke muß hierbei vernünftig gewählt damit überhaupt ein Schedule gefunden werden kann. Üblicherweise verwendet man den die untere Grenze für die Schedule Länge als Zeit-Constraint. Dies ist genau der Wert der Variable "SMAX", die beim ASAP-Schrittplanung berechnet wird. Man könnte dies auch als "Minimum Time Constrained Scheduling" bezeichnen. Dieses Problem wird formell als lineares Optimierungs-Problem über ganzzahligen Variablen definiert (englisch: Integer Linear Programming (ILP)) [32].

Die frühestmöglichen Zeitpunkte zur Ausführung einer Operation, berechnet durch ASAP und ALAP-Schrittplanung werden mit $S(o_i)$ und $L(o_i)$ bezeichnet (siehe (23.7) und (23.8)).

Es wird von einer festen Menge von z Operationstypen ausgegangen $T = \{t_k | \ 1 \le k \le z\}$. Der Typ einer Operation gibt an was für eine Operation es sich handelt, also Multiplikation, Addition, Subtraktion, etc. Die Funktion $\mathrm{Ty}(o_i) = t_k$ liefert den Typ der Operation o_i. Alle Ein- und Ausgänge von Operation haben gleiche Bitbreite.

Es gibt ein Repertoire von m funktionalen Einheiten $F = \{f_k | \ 1 \le k \le m\}$. Jede Funktionale Einheit f_k hat eine Kostenmaß zugeordnet $\mathrm{Cost}(f_k)$, welches angibt wieviel Chip-Fläche die Funktionale Einheit benötigt.

Die Funktion $FU(f_k) = \{t_{k_1}, ..., t_{k_x} \mid t_{k_i} \in T\}$ liefert die Menge der Operationstypen f_k, die von der Funktionalen Einheit f_k ausgeführt werden können. Im Allgemeinen sind funktionale Einheiten ALUs, die mehr als einen Operationstyp ausführen können.

M_{f_k} sind ganzzahlige Variablen, mit denen gezählt wird wieviele funktionale Einheiten vom f_k aktuell verwendet werden.

$x_{i,j}$ sind 0/1-Variablen die den Operationen o_i zugeordnet sind. Variable $x_{i,j}$ wird auf 1 gesetzt wenn Operation o_i in Scheduling Step j gescheduled wird; ansonsten gilt $x_{i,j} = 0$.

Die zeitkritische Schrittplanung wird nun formuliert als Minimierung von:

$$\sum_{k=1}^{m} (\text{Cost}(f_k) \cdot M_{f_k}) \tag{23.9}$$

Unter folgenden Bedingungen:

$$\sum_{o_i \in FU(f_k)} x_{i,j} - M_{f_k} \leq 0 \text{ für } 1 \leq j \leq s,\ 1 \leq k \leq m \tag{23.10}$$

$$\sum_{j=S(o_i)}^{L(o_i)} x_{i,j} = 1 \text{ für } 1 \leq i \leq n \tag{23.11}$$

$$\left(\sum_{j=S(o_i)}^{L(o_i)} (j \cdot x_{i,j}) - \sum_{j=S(o_i)}^{L(o_i)} (j \cdot x_{k,j}) \right) \leq -1 \text{ für alle } o_i \rightarrow o_j \tag{23.12}$$

Die Zielfunktion (23.9) gibt an, daß die Gesamtkosten (Fläche) aller benötigten funktionalen Einheiten minimiert werden soll. Bedingung (23.10) sichert, daß nicht mehr als M_{f_k} funktionale Einheiten pro Steuerschritt verwendet werden. Bedingung (23.11) sichert daß jede Operation o_i in genau einen Steuerschritt j gescheduled wird. Man Beachte, daß der Bereich in den die Operation o_i eingeteilt werden kann durch $S(o_i)$ und $L(o_i)$ eingeschränkt ist. Bedingung (23.12) sichert, daß keine Datenabhängigkeiten verletzt werden.

Das so formulierte Optimierungs-Problem kann durch Branch-&-Bound-Techniken gelöst werden. Zu diesem Zweck existieren kommerzielle Softwarepakete, wie das Programm LINDO [37]. Man beachte, daß solche Verfahren unter Umständen exorbitanten Rechenaufwand erfordern. Daher verwenden die meisten Systeme heuristische Schrittplanungs-Verfahren. Das bekannteste dieser Verfahren ist Force-Directed Scheduling.

23.3.2.5 Force-Directed Scheduling

Force-directed scheduling (FDS, deutsch: Kraft-bestimmte Schritt-Planung) operiert global in der Art und Weise der Auswahl der nächsten einzuteilenden Operation und des Steuerschritt in welchen diese plaziert wird. Bei FDS ist der Leit-Faktor für die Entscheidung welche Operation als nächste eingeteilt wird, und, wo diese eingeteilt wird, eine sogenannte Kraft (force), die auf diese Operation ausgeübt wird. Die Kraft zwischen einer Operation und einem bestimmten Steuerschritt ist proportional die Anzahl der Operationen des gleichen Typ, die zum gleichen Steuerschritt passen würden. Deshalb tendiert Schrittplanung mit dem Ziel der Kraft-Minimierung dazu, die Inanspruchnahme von FUs auszugleichen. Dabei wird ein Schrittplan

erzeugt, der die Anzahl der Resourcen minimiert, die nötig sind um einer gegebenen Zeit-Beschränkung (time constraint) zu entsprechen.

Distribution Graph (DG). Zur Berechnung der Kraft für einen DFG, wird zuerst ein Verteilungsgraph (engl.: *distribution graph*, oder *DG*) aufgestellt für jeden Satz von Operationen die sich in eine FU teilen könnten. Der DG zeigt für jeden Steuerschritt, ***wie schwer beladen dieser Schritt ist***, wobei von Gleichwahrscheinlichkeit aller möglichen Schritt-pläne ausgegangen wird.

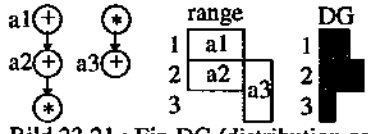


Bild 23.21 : Ein DG (distribution graph)

Der DG wird ermittelt durch Auffindung des frühsten und des spätesten Steuerschrittes, zu welchem die Operation ausgeführt werden könnte, im Rahmen der gegebenen Zeitschranken und Präzedenzen zwischen den Operatoren.

Dies wird ***Mobilität*** k genannt (engl.: ***mobility***) für diese Operation. Wenn eine Operation in einem von k Steuerschritten ausgeführt werden könnte, dann wird 1/k an die entsprechenden Steuerschritte im Graphen geschrieben. Bild 23.21 zeigt beispielsweise einen DFG, den Bereich der Schritte für jede Operation, und den entsprechenden DG für die Addition, wobei eine Zeitschranke von 3 Steuerschritten angenommen wird. Addition a1 muß in Schritt 1 eingeteilt werden, was für diesen Schritt 1 ergibt. Ähnlich ergibt Addition a2 eine 1 für Schritt 2. Addition a3 könnte entweder in Schritt 2 oder in Schritt 3 eingeteilt werden, was für jeden dieser Schritte 1/2 ergibt. Für jede mögliche Zuordnung einer Operation x zu einem Steuerschritt i, wird die Kraft F(i) berechnet mit: [552]

$$F(i) = \text{"SUM(j)"}\ DG(j) * x(i,j) \quad \text{oder,} \quad F_i = \sum_j DG_j \cdot x_{ij} \ , \qquad (23.13)$$

wobei DG(j) der Wert des DG bei Steuerschritt j ist und x(i,j) die Änderung der Wahrscheinlichkeit von x in Steuerschritt j wenn x in Steuerschritt i eingeteilt wird. In Bild 23.21 beispielsweise ergibt sich die Kraft involviert beim Zuordnen von a3 zu Schritt 2 mit 1 1/2 * 1/2 + 1/2 * (-1/2). Der erste Term ergibt sich aus der Änderung durch *Hinzufügen von a3 zu Schritt 2*, da seine Wahrscheinlichkeit sich von 1/2 zu 1 ändert, während der zweite Term vom Entfernen einer möglichen Zuordnung zu Schritt 3 kommt, was einen Wechsel der Wahrscheinlichkeit von 1/2 zu 0 verursacht. Diese Kraft gibt an, daß a3 nicht an Schritt 2 gehen sollte, da dieser schon schwer beladen ist. Andere Kräfte werden addiert, welche den Effekt der Einteilung einer Operation auf ihre Vorläufer und Nachfolger reflektieren, denn wenn eine Operation einmal eingeteilt ist, hat sie die Mobilität ihrer Nachbarn geändert.

Wenn alle Kräfte ermittelt sind, wird das Operations-/Steuerschritt-Paar mit der größten negativen Kraft (oder am wenigsten positiven Kraft) eingeteilt. In obigem Beispiel würde a3 zuerst in Schritt 3 eingeteilt. Der DG und die Kräfte werden aktualisiert und die Prozedur wird wiederholt.

Das FDS-Verfahren (Force-Directed Scheduling [45]) ist in seiner ursprünglichen Version ein Verfahren für *minimal zeitkritische Schrittplanung* (minimum time-constrained scheduling). Die Mechanismen von Force-Directed Scheduling sind durch Erweiterungen [46] auch für Resource-kritische Schrittplanung (resource-constrained scheduling) und Pipeline-Schritt-planung (pipelined scheduling) anwendbar.

Das FDS-Verfahren arbeitet auf der Basis von Wahrscheinlichkeitsverteilungen für die Einteilung der Operation zu Steuerschritten. Die Wahrscheinlichkeit für die Zuordnung einer Operation in einen Planungs-Schritt ergibt sich aus der Mobilität der Operation, welche durch ASAP- und ALAP-Schrittplanung bestimmt wird. Eine Gleichverteilung aller Operationen nach ihren Mobilitäten beschreibt der sogenannte DG (distribution graph).

Durch das Einteilen einer Operation o_i für Zeitschritt j wird die potentielle Gleichverteilung aller Operationen nach dem DG gestört. Das Ausmaß dieser Störung wird durch die Stärke (Force) der Einteilung von o_i zu Zeitschritt j ausgedrückt. Das FDS-Verfahren wählt die Einteilung mit der geringsten Kraft, und berechnet danach den DG neu. Dieser Prozeß wiederholt sich bis alle Operationen eingeteilt sind.

Definitionen:

$T = \{t_k \mid 1 \le k \le z\}$ "Menge der Operationstypen" (23.14)

$\mathrm{Ty}(o_i) = t_k$ "Typ einer Operation o_i" (23.15)

$S(o_i)$ "Planungs-Schritt von o_i bei ASAP-Schrittplanung" (23.16)

$L(o_i)$ "Planungs-Schritt von o_i bei ALAP-Schrittplanung" (23.17)

$\mathrm{Range}(o_i) = [S(o_i), L(o_i)]$ "Schrittplanungs-Intervall von o_i" (23.18)

$\mathrm{Mob}(o_i) = L(o_i) - S(o_i) + 1$ "Mobilität der Operation o_i" (23.19)

$\mathrm{Prob}(o_i, j) = 1/\mathrm{Mob}(o_i)$ falls $j \in \mathrm{Range}(o_i)$ sonst 0. (23.20)

die Wahrscheinlichkeit, daß Operation o_i Zeitschritt j zugeordnet wird.

$x(o_i, j, k) = 1 + \mathrm{Prob}(o_i, j)$ für $j = k$ und $x(o_i, j, k) = 0 - \mathrm{Prob}(o_i, j)$ für $j \neq k$ (23.21)

die Abweichung von der Gleichverteilung falls Operation o_i für Zeitschritt j eingeteilt wird". Für $j, k \in \mathrm{Range}(o_i)$ $x(o_i, j, k)$ berechnet sich x als:

$$x(o_i, j, j) = 1 - \frac{1}{\mathrm{Mob}(o_i)} \text{ und} \tag{23.22}$$

$$x(o_i, j, k) = -\frac{1}{\mathrm{Mob}(o_i)} \text{ für } j \neq k \tag{23.23}$$

Für jeden Operationstyp t_k und jeden Zeitschritt j wird der DG berechnet:

$$\mathrm{DG}(t_k, j) = \sum_{\mathrm{Type}(o_i) = t_k} \mathrm{Prob}(o_i, j) \tag{23.24}$$

Die Funktion FO gibt die Kraft (FOrce) an, die in Zeitschritt k resultiert, wenn Operation o_i für Planungs-Schritt j eingeteilt wird.

$$\mathrm{FO}(o_i, j, k) = \mathrm{DG}(\mathrm{Ty}(o_i), j) \cdot x(o_i, j, k) \tag{23.25}$$

Die Funktion TFo (Total Force) berechnet die Summe aller Kräfte.

$$\mathrm{TFo}(o_i, j) = \sum_{k = S(o_i)}^{L(o_i)} \mathrm{Fo}(o_i, j, k) \tag{23.26}$$

23.3.2.6 Allgemeine Berechnung der Kräfte

Wird eine Operation o_i für einen Zeitschritt (scheduling step) j eingeteilt (scheduled), so ergeben sich für die von ihr abhängigen Operation Einschränkungen des Schrittplanungs-Bereiches (scheduling range). Diese Einschränkungen gegenüber (23.18) erzeugen wiederum neue Kräfte, die sich zu (23.26) dazu addieren. Um die Kräfte solcher Einschränkungen berechnen zu können, muß zunächst eine allgemeinere Methode für die Berechnung der Gesamtkraft (total force) gefunden werden. Man erhält eine solche Methode indem man die Formel (23.26) für die Gesamtkraft umschreibt. Zunächst wird der Term $\mathrm{Fo}(o_i, j, k)$ aus der Summe in (23.26) isoliert:

$$\mathrm{TFo}(o_i, j) = \mathrm{Fo}(o_i, j, j) + \sum_{\substack{k = S(o_i) \\ (j \neq k)}}^{L(o_i)} \mathrm{Fo}(o_i, j, j) \tag{23.27}$$

Danach ersetzt man $\mathrm{Fo}(o_i, j, j)$ durch die Definition (23.25) $\mathrm{DG}(\mathrm{Type}(o_i), j) \cdot x(o_i, j, j)$:

$$\mathrm{Fo}(o_i, j) = \mathrm{DG}(\mathrm{Ty}(o_i), j) \cdot x(o_i, j, j) + \sum_{\substack{k = S(o_i) \\ j \neq k}}^{L(o_i)} \mathrm{DG}(\mathrm{Ty}(o_i), j) \cdot x(o_i, j, k) \tag{23.28}$$

Nun wird Gleichung (23.22) benutzt um $x(o_i, j, j)$ und $x(o_i, j, k)$ zu ersetzen:

$$\mathrm{Fo}(o_i, j) = \mathrm{DG}(\mathrm{Ty}(o_i), j) \cdot \left(1 - \frac{1}{\mathrm{Mob}(o_i)}\right) + \sum_{\substack{k = S(o_i) \\ j \neq k}}^{L(o_i)} \mathrm{DG}(\mathrm{Ty}(o_i), k) \cdot -\frac{1}{\mathrm{Mob}(o_i)} \tag{23.29}$$

$$\mathrm{Fo}(o_i, j) = \mathrm{DG}(\mathrm{Ty}(o_i), j) - \frac{\mathrm{DG}(\mathrm{Ty}(o_i), k)}{\mathrm{Mob}(o_i)} - \sum_{\substack{k = S(o_i) \\ j \neq k}}^{L(o_i)} \frac{\mathrm{DG}(\mathrm{Ty}(o_i), k)}{\mathrm{Mob}(o_i)} \tag{23.30}$$

Dies führt zu alternativen Formel für die Berechnung der Total Force:

$$\mathrm{Tfo}(o_i, j) = \mathrm{DG}(\mathrm{Ty}(o_i), j) - \sum_{k = S(o_i)}^{L(o_i)} \frac{\mathrm{DG}(\mathrm{Ty}(o_i), k)}{\mathrm{Mob}(o_i)} \tag{23.31}$$

Basierend auf (23.31) wird die Berechnung der allgemeinen Kraft (GFo = General Force) definiert, für den Fall daß eine Operation o_i nur noch in einem Intervall $[a,b]$ eingeteilt werden kann. Gleichung (23.31) wird dann zum Sonderfall für (23.32) ($\mathrm{Gfo}(o_i, j, j) = \mathrm{Tfo}(o_i, j)$). Außerdem gilt: $\mathrm{Tfo}(o_i, S(o_i), L(o_i)) = 0$.

Definition der allgemeinem Kraft (general force):

$$\mathrm{Gfo}(o_i, a, b) = \sum_{k=a}^{b} \frac{\mathrm{DG}(\mathrm{Ty}(o_i), j)}{b-a+1} - \sum_{k=S(o_i)}^{L(o_i)} \frac{\mathrm{DG}(\mathrm{Ty}(o_i), k)}{L(o_i)-S(o_i)+1} \qquad (23.32)$$

$$S(o_i) \le a \le b \le L(o_i)$$

Mit Gleichung (23.32) können nun auch diejenigen Kräfte berechnet werden, die sich durch eingeschränkte Schrittplanungs-Bereiche ergeben. Durch das Zuordnen einer Operation o_i zu einem Planungs-Schritt (scheduling step) j werden im Allgemeinen die Schrittplanungs-Bereiche der Folgeoperationen $\mathrm{Succ}(o_i)$ und der Vorgängeroperationen $\mathrm{Pred}(o_i)$ eingeschränkt. Die Kräfte (forces), die durch diese Einschränkungen der Schrittplanungs-Bereiche entstehen, werden mit $\mathrm{SuccFo}(o_i, j)$ (Successor Force, Nachfolger-Kraft) und $\mathrm{PredFo}(o_i, j)$ (Predecessor Force, Vorläufer-Kraft) bezeichnet und durch (23.32) berechnet.

23.3.2.7 Algorithmus: Force-Directed Scheduling

Die folgende Spezifikation faßt den Algorithmus für Force-directed Scheduling (Kraft-orientierte Schrittplanung) zusammen.

```
ASAP_scheduling;
ALAP_scheduling;
not_scheduled := {o_1, ..., o_n};
while not_scheduled ≠ ∅ do
  minForce := ∞;
  repeat
    "schedule operation o_i ∈ not_scheduled with minimum
     Tfo(o_i, j) + SuccFo(o_i, j) + PredFo(o_i, j) into time step j";
    not_scheduled := not_scheduled ∪ {o_i};
  until "all operations are scheduled"
```

Der hier zusammengefaßte Algorithmus für Force-directed Scheduling (Kraft-orientierte Schrittplanung) wurde in den Abschnitten 23.3.2.5 und 23.3.2.6 erläutert.

23.4 Pipeline-Synthese

Bisherige Betrachtungen gingen von nicht-überlappenden Operationen aus, d. h., die Operationsfolge hat keine Pipeline-Anordnung. Die bisher in diesem Kapitel beschriebenen Techniken können jedoch erweitert werden für den Umgang mit Pipeline-Strukturen [1] [14] [19] [35] [36] [37] [43]. Mangels Raum können diese Techniken hier jedoch nicht behandelt werden.

23.5 Hardware/Software-Ko-Design

Hardware/Software Ko-Design ist ein neuer Zweig der Informatik [8] [9] [30] [31] [34] [48] [49] [52] [53], dem eine bedeutende Zukunft vorhergesagt werden kann. Wie High-Level-

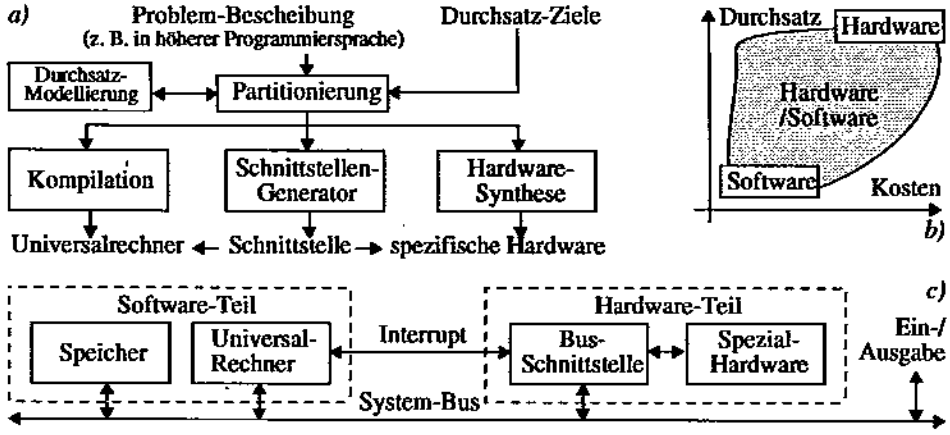


Bild 23.22 : Hardware/Software-Ko-Design; a) Verfahren, b) Entwurfsraum, c) eine Architektur.

Synthese geht auch Hardware/Software Ko-Design von Problem-Formulierungen hoher Abstraktions-Ebenen aus. Ziel dieses neuen Forschungsgebiets ist die Entwicklung von Verfahren zur automatischen Partitionierung eines Datenverarbeitungsproblems in einen Hardware-Teil und einen Software-Teil sowie die notwendige Hardware/Software-Schnittstelle (Bild 23.22 a). Ein Optimierungsziel ist dabei die Erfüllung der Durchsatz-Anforderungen (Bild 23.22 a) zu minimalen Kosten (Bild 23.22 b). Bild 23.22 c zeigt eine mögliche Basis-Architektur für Hardware/Software Ko-Design [28] [29] [52]. Durch Anwendung konfigurierbarer Hardware (Anwender-programmierbare Hardware [10] [27] u. a., s. a. Abschnitte 3.3 und 3.4) sind auch auf der Hardware-Seite sehr kurze Debugging-Zyklen erreichbar. Wie High-Level-Synthese ist auch Hardware/Software Ko-Design ein wichtiger Faktor der Wettbewerbsfähigkeit durch Möglichkeiten zur Verminderung der Personalkosten bzw. Verbesserung der Designer-Produktivität bei der Produkt-Entwicklung (siehe auch Kasten "Lean Engineering" auf Seite 61).

23.6 Literatur

[1] A. Ast: Ein Generator für ALU-Pipeline-Architekturen; Dissertation, Universität Kaiserslautern, (in Vorbereitung für 1994)

[2] M.R. Barbacci: Instruction Set Processor Specification (ISPS): The Notation and its Applications; IEEE Transaction on Computers, vol. C-30, no. 1, pp. 24-40, Jan. 1981.

[3] D. Bernstein, D. Rodeh, I. Gertner: On the Complexity of Scheduling Problems for Parallel/Pipelined Machines; IEEE-Tr. on Computers, C-38,9, p. 1308-1313, Sept. 1989.

[4] R. Brayton, N. Brenner, C. Chen, G. De Micheli, C. Mullen, R. Otten: The Yorktown Silicon Compiler; ISCAS '85, Kyoto, Japan, June 1985.

[5] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, A. Wang: MIS: A Multiple-Level Logic Optimization System; IEEE Trans. on CAD, CAD-6/no. 6, pp.1062-1081, 1987.

[6] R. K. Brayton, R. Camposano, G. De Micheli, R. Otten, J. van Eijndhoven: The Yorktown Silicon Compiler; in (ed.: D. Gajski) Silicon Compilation, Addison Wesley, 1988.

[7] F. Brewer, D. Gajski: Chippe: A System for Constraint Driven Behavioral Synthesis; IEEE Transactions on CAD, vol. 9, no. 7, pp. 681-695, July 1990.

[8] K. Buchenrieder: Codesign and Concurrent Engineering; Computer, Jan. 1993

[9] K. Buchenrieder, C. Veith: CODES: A Practical Con-current Design Environment; hand-out from ACM / IEEE 1st Int'l Workshop on Hard-ware-Software Co-Design, Estes Park, Colo, 1992

[10] A. Buell, K. L. Pocek: Proc. IEEE Workshop on FPGAs for Custom Computing Machines, April 5-7, 1993, Napa, CA; IEEE Computer Society Press, Los Alamitos, 1993

[11] R. Camposano: Structural Synthesis in the Yorktown Silicon Compiler; VLSI 87, VLSI Design of Digital Systems, North-Holland, pp. 61-72, 1988.

[12] R. Camposano, R. M. Tabet: Design Representation for the Synthesis of Behavioural VHDL Models; Proc. 9th Int'l Conf. on CHDL, North Holland, June 1989.

[13] R. Camposano, W. Wolf (editors): High Level Synthesis; Kluwer, Boston, 1990

[14] C. Chen, M. Moricz: Data Path Scheduling for Two-Level Pipelining; 28th Design Automation Conference, pp. 603-606, 1991.

[15] S. Davidson, D. Landskov, B. Shriver, P Mallet: Some Experiments in Local Microcode Compaction for Horizontal Machines; IEEE Tr. C-30,7, p. 460-477, July 1981.

[16] G. De Micheli, R.K. Brayton, A. Sangiovanni-Vincentelli: Optimal State Assignment for Finite State Machines; Trans. on CAD, vol. 4, no. 3, pp. 269-285, July 1985.

[17] G. De Micheli, D.C. Ku: Hercules: A System for High Level Synthesis; Proc. 25th Design Automation Conference, New York, ACM/IEEE, pp. 483-488, June 1988.

[18] S. Devadas, H. Ma, A.R. Newton, A. Sangiovanni-Vincentelli: MUSTANG: State Assignment for Finite State Machines Targeting Multi-Level Logic Implementations; Transactions on CAD, vol. 7, no. 12, pp. 1290-1300, Dec. 1988.

[19] B. Fjellborg: Pipeline Extraction for VLSI Data Path Synthesis; Dissertation no. 273, Dept. of computer Science, Linköping University, Linköping, Schweden

[20] T. D. Friedman, S.C. Yang: Methods used in an Automatic Logic Design Generator (ALERT); IEEE Transactions on Computers, Vol. C-18, pp. 593-614, 1969.

[21] D. Gajski, N. Dutt, A. Wu, S. Lin: High-Level Synthesis - Introduction to Chip and System Design; Kluwer, 1992

[22] M.R. Garey, D. S. Johnson: Computers and Intractability - A Guide to the Theory of NP-Completeness; W. H. Freeman and Company, San Francisco, 1979

[23] M.R. Garey, D. S. Johnson: Complexity Results for Multiprocessor Scheduling under Resource Constraints; SIAM J. computing, vol. 4, no. 4, pp. 397-411, Dec. 1975.

[24] C. Gebotys, M. Elmasry: Simultaneous Scheduling and Allocation for Cost Constrained Optimal Architectural Synthesis; 28th Design Automation Conference, 1991.

[25] C. Gebotys: Synthesizing Embedded Speed-Optimized Architectures; IEEE Journal of Solid-State Circuits, J-SSC 28,3 (March 1993)

[26] E.F. Girczyc, J.P. Knight: An ADA to Standard Cell Compiler based on Graph Grammars and Scheduling; Proc. ICCD'84, pp. 726-731, 1984.

[27] H. Grünbacher, R. Hartenstein: FPGAs - Architectures and Tools for Rapid prototyping; Springer-Verlag, Heidelberg, 1993

[28] R. Gupta, C. Coelho, D. De Micheli: Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components; Proc. DAC 1992, IEEE 1992

[29] R. Gupta, C. Coelho, D. De Micheli: Program Implementation Schemes for Hard-ware-Software Systems; hand-out from ACM / IEEE 1st Int'l Workshop on Hard-ware-Software Co-Design, Estes Park, Colo, 1992

[30] R. Gupta, G. De Micheli: Hardware-Software Cosynthesis for Digital Systems; IEEE De-

sign & Test of Computers, **10**,3, Sept 1993

[31] R. Hartenstein: Hardware/Software Co-Design; Proc. 3rd Int'l Workshop on Field-programmable Logic and Applications, Oxford, UK, Sept. 1993

[32] C.T. Hwang, J.H. Lee, Y.C. Hsu: A Formal Approach to the Scheduling Problem in High Level Synthesis; IEEE Transactions on CAD, vol. 10, no. 4, Apr. 1991.

[33] R. Jain, K. Küçükçakar, M. J. Mlinar, A. C. Parker: Experience with the ADAM Synthesis System; Proc. 26th Design Automation Conf., Las Vegas, pp. 56-61, 1989.Applicationsw; IEEE Design & Test of Computers, **10**,3, Sept 1993

[34] A. Kalavade, E. A. Lee: A Hardware-Software Codesign Methodology for DSP; IEEE Design & Test of Computers, **10**,3, Sept 1993

[35] P. M. Kogge: The Architecture of Pipelined Computers; Hemisphere Publ. Corp., 1981.

[36] K. Kuchinski et al.: Parallelism Exploration for Automatic Design of VLSI Systems; Proc. 1st. Nordic Workshop on Parallel Algorithms and Architectures in Vision and Image Processing, Linköping, Sweden 1990

[37] M. Lam: Software Pipelining: An Effective Scheduling Technique for VLIW Machines; Proc. SIGPLAN'88, Conference on Programming Design and Implementation, Atlanta, Georgia, June 1988.

[38] Lindo Systems Inc.: Linear Interactive and Discrete Optimizer for Linear, Integer and Quadratic Programming Problems.

[39] M.C. McFarland: The VT: A Database for Automated Digital Design; Report DRC-01-4-80, Design Research Center, Carnegie-Mellon University, Dec. 1978.

[40] M.C. McFarland, A. C. Parker, R. Camposano: The High-Level Synthesis of Digital Systems; Proc. IEEE 78.2, pp. 301-318, (Febr. 1990)

[41] C. Mead, L. Conway: Introduction to VLSI Systems; Addison-Wesley, 1980

[42] A. Orailoglu, D. Gajski: Flow Graph Representation; 23rd Design Aut. Conf., 1986.

[43] N. Park, A.C. Parker: Sehwa: A Software Package for Synthesis of Pipelines from Behavioural Specifications; IEEE Trans. on CAD, vol. 7, no. 3, pp. 356-370, Mar. 1988

[44] P.G. Paulin, J.P. Knight, E.F. Girczyc: HAL: A Multi Paradigm Approach to Automatic Data Path Synthesis; Proc. 23rd Design Automation Conf., pp. 263-270, June 1986.

[45] P.G. Paulin, J.P. Knight: Scheduling and Binding Algorithms for High Level Synthesis; Proc. 26th Design Automation Conference, Las Vegas, pp. 1-6, June 1989.

[46] P.G. Paulin, J.P. Knight: Force Directed Scheduling for the Behavioral Synthesis of ASIC's; IEEE Transactions on CAD, vol. 8, no. 6, pp. 661-679, June 1989.

[47] W. Rosenstiel: Optimizations in High-Level Synthesis; Microprocessing and Microprogramming, pp. 347-352, 1986.

[48] A. Sedlmeier, K. Buchenrieder: Handout 2nd Int'l workshop on Hardware / Software Codesign (Codes/CASHE'93), Siemens-AG, München 1993

[49] A. Smailagic, D. P. Siewiorek: A Case Study in Embedded-System Design: The Vu Man 2 Wearable Computer; IEEE Design & Test of Computers, **10**,3, Sept 1993

[50] D.E. Thomas, E.M. Dirkes, R.A. Walker, J.V. Rajan, J.A. Nestor, R.L. Blackburn: The System Architects Workbench; Proc. 25th DAC, pp. 337-343, June 1988.

[51] D.E. Thomas, E.D. Lagnese, R.A. Walker, J.A. Nestor, J.V. Rajan, R.L. Blackburn: Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench; Kluwer Academic Publishers, 1990.

[52] D.E. Thomas, J.K. Adams, H. Schmit: A Model and Methodology for Hardware-Software Codesign; IEEE Design & Test of Computers, **10**,3, Sept 1993

[53] W. Wolf: Hardware-Software Codesign; IEEE Design & Test of Computers, **10**,3, Sept 1993