

Regelbasierte High-Level-Synthese mit kommunizierenden sequenziellen Prozessen

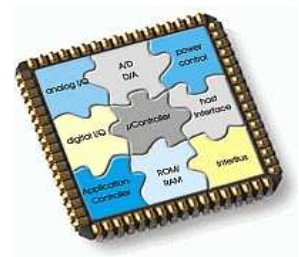
Von der imperativen algorithmischen Ebene zu RTL

Dr. Stefan Bosse
Universität Bremen

7.12.2009

Überblick

1. Motivation, Anforderungen und Ziele
2. Schaltkreisentwurf: Überblick der Syntheseverfahren für Register-Transfer-Logik
3. Kommunizierende Sequenzielle Prozesse als Programmiermodell
4. Interprozeßkommunikation
5. ConPro: imperative Programmiersprache und Synthesewerkzeug
6. Kommunizierende Sequenzielle Prozesse als Hardwaremodell
7. Regelbasiertes Syntheseverfahren
8. Scheduling & Optimierung
9. Beispiel: Parity-Berechnung



1.1 Motivation & Anforderungen

- ▶ **Steigende Entwurfskomplexität** erfordert höhere Abstraktionsebene und direkte Implementierung von Algorithmen (reaktiv wie funktional) mit einer Hochsprache in Schaltkreisen
- ▶ Entwicklung im Software-Bereich:
 - ↳ **Programmiersprache**: Direkte Maschinen-Programmierung ➤ Hochsprache
 - ↳ **Komplexität** von Daten- und Kontrollanweisungen: Niedrig ➤ Hoch
- ▶ Software-Hochsprachen bilden regelbasiert komplexe Instruktionen (z.B. Schleifen) eines Instruktionsgraphen auf eine lineare Sequenz von einfachen Maschinen-Instruktionen ab.
 - ↳ Maschinensprache \subset Software-Hochsprache
 - ↳ Automation möglich
 - ↳ *Zuerst Entwicklung von Architektur der Maschine, anschließend Entwicklung/Anwendung Hochsprache, abgestimmt auf Architektur*

1.2 Motivation & Anforderungen

- ▶ Entwicklung im Hardware-Bereich:
 - ↳ Modellierung: Boolesche Algebra ➤ Verhaltensbeschreibung (VHDL)
 - ↳ Komplexität: Sehr niedrig ➤ Mittel
- ▶ Hardware-Hochsprachen wie VHDL, aber auch SystemC, bilden Schaltkreis als Funktionsblock oder Komponenten ab (Verhalten/Struktur), und nicht den Algorithmus, den die Maschine umsetzen soll!
Problem Hardware-Modellierung $\not\subset$ Software-Modellierung
- ▶ Moderne Software-Hochsprachen haben sich als geeignet erwiesen, auch komplexe Algorithmen auf die Maschinenebene abzubilden.
 - **Automation** Software-Compiler
- ▶ **Problem** Programmiermodell vieler imperativer Hochsprachen, wie C/C++, ist auf Rechnerarchitektur mit Speichermodell und sequenzieller Ausführung durch EINE Verarbeitungseinheit ausgerichtet!
- ▶ Funktionale Hochsprachen besitzen diese Limitierung nicht explizit, sind aber unbeliebt.

1.3 Motivation & Anforderungen

- ▶ **Aufgabe der High-Level-Synthese:** Abbildung einer algorithmischen Beschreibung (des Verhaltens von Hardware) auf Register-Transfer-Ebene (RTL), die das Verhalten implementiert [Urard et al., 2008].
- ▶ Verwendung einer **traditionellen Hochsprache** wie C für den HLS-Schaltkreisentwurf bedeutet:
 1. Abbildung eines Programms auf reine nicht speicherorientierte und dezentrale RTL schränkt Programmiermodell ein: keine Zeiger, da als Speicherreferenz explizit modelliert, keine Funktionsrekursion, da kein Stack existiert usw.
 2. Oder "klassisches" Hardware-Software-Co-Design: Aus Programm wird anwendungsspezifische programmierbare Rechnerarchitektur/Mikroprozessor und Software (Maschinenprogramm) abgeleitet.
- ▶ **Problem** HLS-Schaltkreisentwurf erfordert im Gegensatz zu Software-Entwurf zusätzlichen Informationen für die HLS, da "Rechnerarchitektur" noch nicht existiert, z.B. *echt bit-skalierbare Datentypen, oder Angaben über Scheduling und Allokation*
 - ▶ **Eingrenzung Entwurfsraum**
 - ↳ nicht im Programmiermodell einer traditionellen Software-Hochsprache enthalten

1.4 Fragestellungen & Ziele

Fragestellung

- ▼ wie soll Nebenläufigkeit modelliert werden?
- ▼ wie sollen Echtzeitkriterien modelliert werden? Oder: wirklich notwendig?
- ▼ wie soll der Entwurfsraum eingegrenzt werden?
- ▼ wie soll die Parametrisierung der RTL-Architektur und des Synthesevorgangs umgesetzt werden?

Ziele

- ▼ Entwurf eines geeigneten und konsistenten Programmiermodells um
 1. Nebenläufigkeit zu modellieren,
 2. die abzuleitende RTL-Architektur parametrisieren zu können,
 3. Algorithmik mit Daten- und Kontrollanweisungen einer Hochsprache implementieren zu können,
 4. ein intuitiver Zugang auch für Software-Entwickler zu ermöglichen.
- ▼ Entwurf und Test eines geeigneten Synthese-Werkzeuges

2•1 Synthese von Digitallogikschaltungen

Beschreibungs- und Modellierungsebenen für den Schalkreisentwurf

1. Hardware-Ebene
2. Register-Transfer-Logik-Ebene
3. Algorithmische Ebene

SW

Hardware-Ebene

- Verhaltens- und Strukturbeschreibung: VHDL, Verilog...

Register-Transfer-Logik Ebene

- Beschreibung mit getrennten Daten- und Kontrollpfad (Δ und Γ).
- **Datenpfad**: Funktionsblöcke Π , Datenpfadselektoren Σ und Register \mathcal{R}
- **Kontrollpfad**: endlicher Zustandsautomat FSM Φ , der den Datenpfad spatial und zeitlich steuert.
- Explizites Scheduling und explizite Allokation von Ressourcen erforderlich!

HW

2•2 Synthese von Digitallogikschaltungen

Algorithmische Ebene

SW

- **Imperative Programmiersprache** Eine **Sequenz von Anweisungen** κ beschreibt das schrittweise Verhalten als Berechnung der Ausgabedaten aus Eingabedaten: $\kappa: E \rightarrow A$
- **Funktionale Programmiersprache** Eine **Menge von Ausdrücken** beschreibt das Verhalten als Relation der Ausgabedaten mit den Eingabedaten: $R \subseteq E \times A$
- Daten- und Kontrollpfad (Δ und Γ) sind transparent, werden explizit aber kombiniert durch die Anweisungssequenz beschrieben, gekapselt durch Anweisungsblöcke B .

HW

Beispiele

Robert Barbacci, 1973

Automated Exploration of the Design Space for RT-Systems

Paralleliät auf Anweisungsebene, ein Kontrollfluß, Programmiersprache

Sumit Gupta et al., 2004

SPARK: A Parallelizing Approach to the High-Level-Synthesis of Digital Circuits

C, keine Pointer, keine Funktionsrekursion, ein Kontrollfluß, RTL auf VHDL-Ebene

3•1 Programmiermodell

- Ausnutzung von Parallelität für Verbesserung von Effizienz

$$\eta = \frac{\text{Qualität}}{\text{Kosten}} \quad (1)$$

mit Qualität: minimierte Latenz oder maximierter Datendurchsatz, und Kosten: Ressourcen

- Nebenläufigkeit kann explizit durch den Entwickler und implizit durch Compiler modelliert und exploriert werden:

Explizite Parallelität

Parallelität ist Bestandteil des Programmiermodells, geeignet für grob granulierte Parallelität, gegeben durch Partitionierbarkeit des zu implementierenden Algorithmus.

- **Wissensbasierte Implementierung** von Nebenläufigkeit

Implizite Parallelität

Automatische Exploration, bevorzugt für fein granulierte Parallelität auf Datenpfadebene, i.A. durch Abrollen von Schleifen erzielbar, Nutzung von symbolischen Analysemethoden und Basisblock-Schedulern.

3•2 Programmiermodell

- Modellierung des Algorithmus mit imperativen Programmiersprachen (wie auch C) erfordert schrittweise Ausführung der vorgegebenen Befehlssequenz κ

Sequenzieller Prozeß $P = \phi(\kappa, \theta)$

Ein Prozeß ϕ ist eine Funktion von κ und Objekten θ (z.B. Speicher).

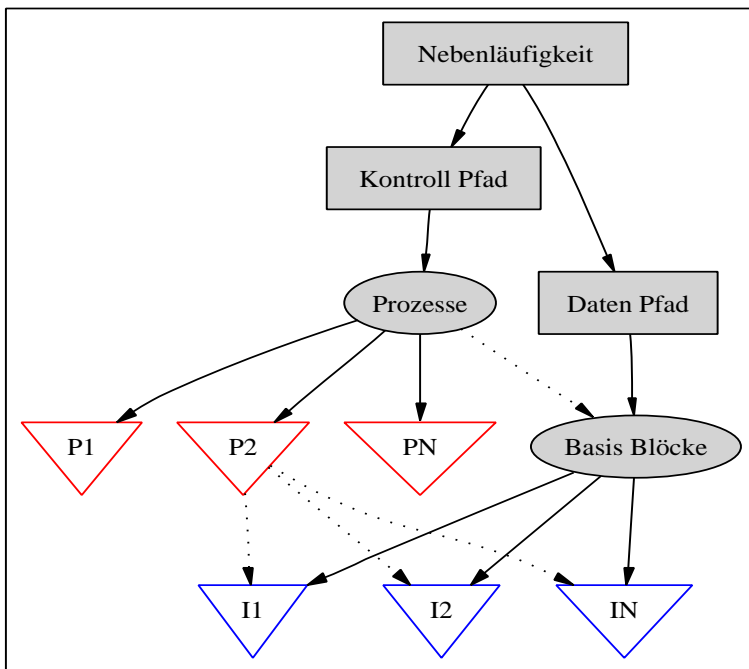
- Übergang zu **Multi-Prozeß-Modell** bekannt und akzeptiert aus Software-Programmierung: **Multi-Threading**

Eine Menge von Prozessen setzt sich aus N unabhängig ausgeführten sequenziellen Prozessen zusammen: $\Phi = \{\phi_1, \phi_2, \dots, \phi_N\}$ mit $\phi_i = \phi_i(\kappa_i, \theta_i)$

- Nebenläufigkeit auf Prozeß-Ebene erfordert **Synchronisation** \equiv Interprozeß-Kommunikation
- **Modell: Kommunizierende Sequenzielle Prozesse** (CSP, [C. Hoare, 1985])
- Neben lokalen Objekten θ gibt es eine Menge globaler, geteilter Objekte Θ .

3•3 Programmiermodell

Unterschiedliche Ebenen der Nebenläufigkeit auf 1. Kontrollpfadebene, 2. Datenpfadebene



I

- Instruktion
 - ↳ Datenoperation I_i^Δ
 - ↳ Kontrolloperation I_i^Γ

P

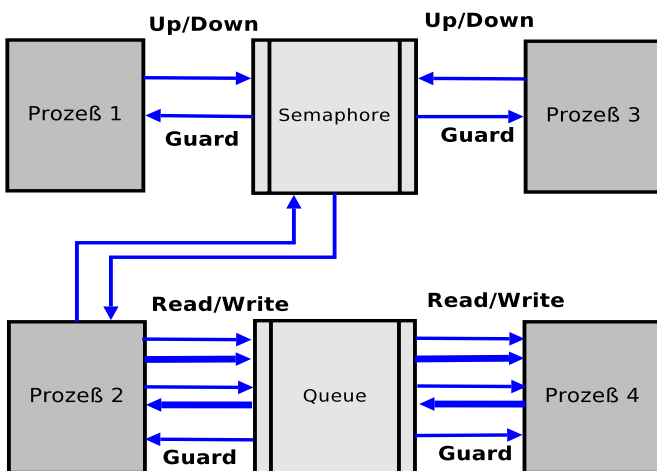
- Prozeß
 - ↳ Schrittweise Ausführung einer Menge von Instruktionen $\kappa = \{I_1, I_2, \dots, I_N\}$
 - ↳ Parallele Ausführung der Prozeßmenge $\Phi = \{P_1, P_2, \dots, P_N\}$

Basis Block

- Bindung von Datenpfadoperationen $\kappa' = \{I_n^\Delta, I_{n+1}^\Delta, \dots, I_{n+m}^\Delta\}$

3•4 Programmiermodell

Das Multi-Prozeß-Modell mit Anfrage-basierter Synchronisation



Geteilte Objekte

- Zugriff auf geteilte Objekte erfolgt anfragebasiert mit Handshake $REQ \Leftrightarrow ACK$

Guard

- Ein Guard schützt ein Objekt.
- Der Guard wird mit einem Scheduler S implementiert.
- Nur ein Prozeß erhält den Guard \equiv Anfrage ausgeführt

Prozeß

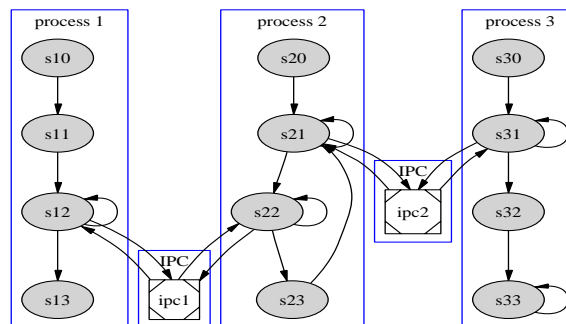
- Ein Prozeß mit aktiver Anfrage wird solange blockiert bis die Ressource vom Scheduler zugeteilt wird
- Zugriffszeit $\tau \geq \tau_0$

4•1 Interprozeß-Kommunikation

- Nebenläufigkeit auf Kontrollpfadebene erfordert Synchronisation, da Prozesse der Menge Φ unabhängig zueinander ausgeführt werden.
- Geteilte Ressourcen Θ müssen geschützt und konkurrierende Zugriffe aufgelöst und/oder serialisiert werden.
- Synchronisation ist hier Interprozeß-Kommunikation (IPC), die eine Relation/Korrelation der Kontrollpfade von einzelnen Prozessen herstellt:

$$l_i : \phi_n \Leftrightarrow \phi_m \quad (2)$$

mit l : IPC-Objekt aus der Menge \mathfrak{I}



4•2 Interprozeß-Kommunikation

Primitiven der Interprozeßkommunikation (Abstrakte Datentypen Objekte) \mathfrak{I}

Mutex

Mutualer Ausschluß für atomaren Zugriff auf Ressourcen, Operationen auf Objekt: **{lock, unlock}**, Bedingung: LOCK= \emptyset

Semaphore

Geschützter Zähler ϕ mit $\phi \geq 0$ für Produzent-Abnehmer-Anwendung, Operationen auf Objekt: **{down, up}**, Bedingung: $\phi > 0$

Event

Ereignis Synchronisation S, Operationen auf Objekt: **{await, wakeup}**, Bedingung: $\uparrow S$

Barriere

Gruppenkommunikation, Operationen auf Objekt: **{await}**, Bedingung: $N_{\text{await}}=N_{\text{group}}$

Timer

Zeitliche Ereignis Synchronisation T, Operationen auf Objekt: **{await, start, stop}**, Bedingung: $\uparrow T$

Queue

Synchronisierter Datenaustausch zwischen Prozessen, Operationen auf Objekt: **{read, write}**, Bedingungen: $\#avail > 0$, $\#free > 0$.

5•1 Programmiersprache ConPro

Eigenschaften

- ▶ Strukturierte **imperative Programmiersprache** mit komplexen **Kontrollstrukturen**:
 1. Bedingte Verzweigung (if-then-else und match-with)
 2. Schleifen (bedingt while-do, unbedingt always-do, Zählschleife for-do)
 3. Ausnahme-Signale mit Signal-Behandlung (exceptions)
- ▶ Syntax i.A. in Klartext (keine Hyroglyphen), intuitiv
- ▶ **Kern-Datentypen**: {integer, logic, bool, char} ↪ **echt bit-skalierbar**
- ▶ Nutzerdefinierte Datentypen: **Produkt- und Summentypen**:
 1. Ein- und mehrdimensionale Felder (array)
 2. Heterogene Strukturen (structure), Bit-Strukturen, Komponenten-Interface
 3. Symbolische Listen (enumeration)
- ▶ **Kern-Objekttypen**: {Register, Variable (RAM), Signal, Queue, Channel}
 - ↪ Lokale (exklusiver Zugriff) und globale Sichtbarkeit (geteilter Zugriff)
 - ↪ Register: CREW-Verhalten, Variablen: EREW-Verhalten
 - ↪ Signale: Hardware-Schnittstelle
- ▶ Alle Objekte können parametrisiert werden (Implementierung, Verhalten)
- ▶ **Abstrakte Datentyp Objekte** (ADTO) mit methodenbasierten Aufruf

5•2 Programmiersprache ConPro

Eigenschaften

- ▶ Externe Modellierung von ADTOs mit dem **External-Module-Interface (EMI)**
 - ↪ Modellierung des ADTO auf Hardware-Beschreibungsebene (modifiziertes VHDL mit Skripterweiterung), Spezifikation der Schnittstelle VHDL ⇔ ConPro Prozeß
- ▶ Grob granulierte Nebenläufigkeit mit konkurrierend ausgeführten **sequenziellen Prozessen** auf **Kontrollpfadebene**
- ▶ Fein granulierte Nebenläufigkeit auf **Prozeß- und Datenpfadebene** mit **gebundenen Blöcken**
- ▶ **Interprozeßkommunikation** mit ADTOs:
 - {Mutex, Semaphore, Event, Barriere, Timer, Queue, Channel}
- ▶ Prozeß ≡ ADTO, **Prozeßkontrolle**: methodenbasiert {start, stop, call}
Prozeß-Arrays können implementiert werden ("Vektorrechner")
- ▶ Funktionen
 1. Inline-Makro ➤ Instruktionssequenz-Substituierung
 2. Geteilter Funktionsblock ➤ Implementierung mit Prozeßmodell und Zugriffs-Scheduler
- ▶ **Ausdrücke**: arithmetisch, boolean, relational
- ▶ **Zugriff** auf global geteilte Objekte wird automatische mit **Scheduler** geschützt und serialisiert, verschiedene Scheduling-Strategien stehen zur Verfügung (static, FIFO).

5.3 Programmiersprache ConPro

Prozesse

```
process p1:
begin
...
end;
array pa:
  process[4] of
begin
...
end;
```

Blöcke

```
begin
  I1;I2;...
end with expr="shared";
begin
  I1;I2;...
end with bind;
⇔
I1,I2,...;
```

Objekt- und Typendefinition

```
reg x,y: int[8];
queue q: char with depth=8;
type st : {s1:int[4];d:logic;};
reg s: st;
array a1: reg[12] of logic[8];
array a2: var[100] of st;
type bt : {b1: 0; b2: 1 to 3;};
reg bb: bt;
object t1: timer;
sig sh: logic;
type states : { ST;REQ;END; };
reg state,next_state: states;
```

Objekte ADTO

```
t1.init ();
t1.time (10 microsec);
t1.start ();
...
t1.await ();
...
p1.start ();
pa.[2].call ();
```

Ausdrücke

```
a1.[i]←1;
b←x < y and x <> 0;
state←REQ,x←y;
x ← (y*2-x)-x*2;
a1.[0]←a1.[1] land 0xF0;
bb.b1←1;
```

Kontrollstrukturen

```
if a < 10 then x ← x+1
else x ← 0;

for i = 1 to 100 do
  x ← a*2-i;

wait for s=1;

match state with
begin
  when ST: s.s1←1;
  when REQ: s.s1←2;
  when others: raise Err;
end;
```

Funktionen

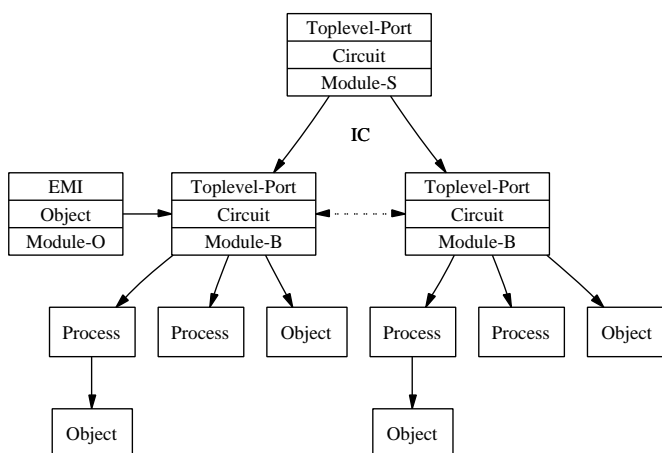
```
function f(x:int[5])
  return (a:bool,b:int[8]):
begin
  a← x=0; b ← x*12;
end;
...
{q1,r2} ← f(10);
```

Exceptions

```
exception RX_err;
try
begin
  ... raise RX_err
end with
begin
  when RX_err: ...
end;
```

6.1 Hardwaremodell

Design-Hierarchie und Relation zu Modulen aus dem Softwaremodell



Modul-S

Strukturelles Modul ermöglicht Verhaltensmodule (B) in einem System-On-Chip zu vereinigen.

Modul-B

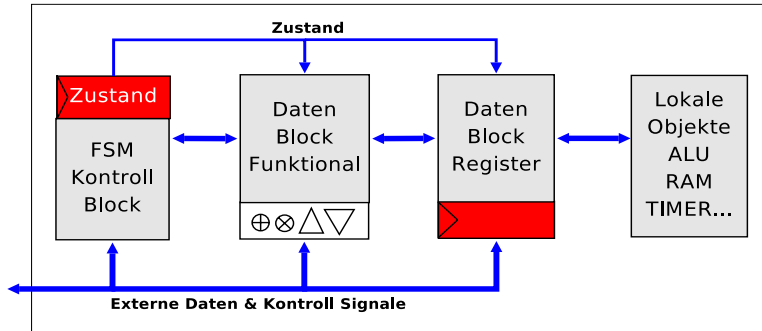
Verhaltensmodul beschreibt Prozesse, globale und lokale Objekte. Ein Toplevel-Port definiert Schnittstelle zur Außenwelt.

Prozeß

Prozeß besteht aus endlichen Zustandsautomaten, Funktionsblöcke und lokalen Objekten (Register).

6•2 Hardwaremodell

Implementierung eines Prozesses



FSM

Endlicher Zustandsautomat, Moore-Typ, takt-synchron

Kontrollpfad Γ

Abbildung der Prozeß-Instruktionen auf Zustände:

$$\Gamma: \kappa \rightarrow \Sigma = [\sigma]$$

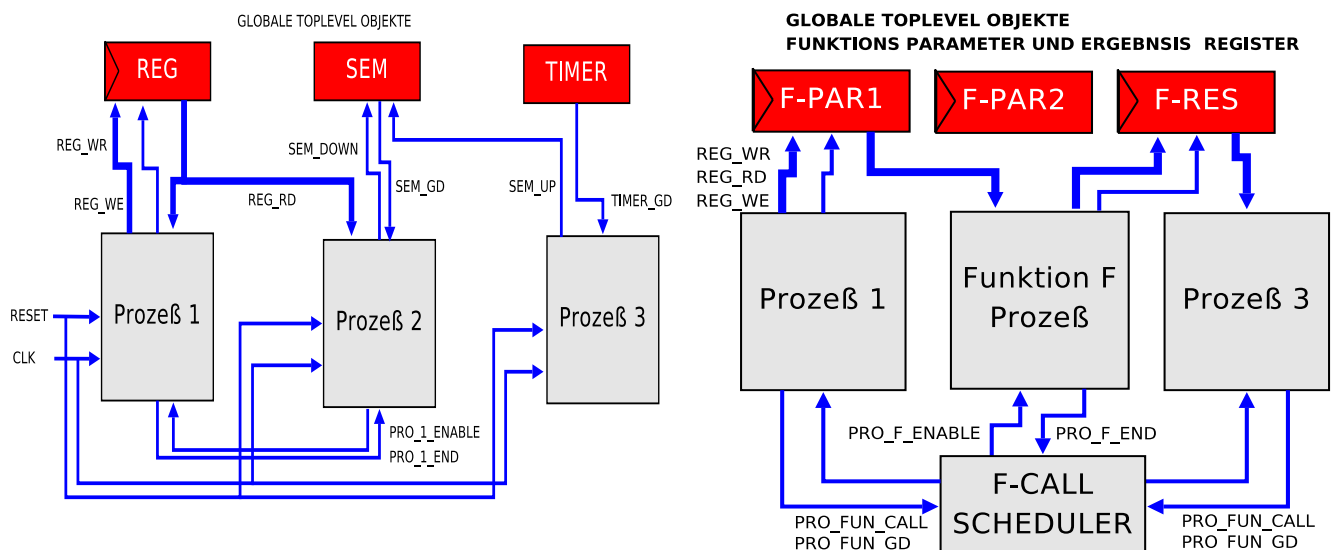
Datenpfad Δ

Abbildung der Ausdrücke auf kombinatorische Blöcke:

$$\Delta: \varepsilon \rightarrow F(\sigma)$$

6•3 Hardwaremodell

- Direkte Abbildung von Objekten (Registern, IPC, Prozesse usw.) auf Hardware-Blöcke
- Links: Prozeßblock-Schnittstelle und System-Einbettung mittels Hardware-Signalen
- Rechts: Geteilter Funktionsblock mit Prozeßblock sowie Parameterregistern



7.1 Regelbasiertes Syntheseverfahren

Eigenschaften

- ▶ Traditioneller Compiler-Flow mit Zwischenrepräsentation (IR) des Syntax-Graphen
- ▶ Zwischenrepräsentation mit **Mikro-Instruktionen** (μ Code) bedeutet Transformation eines Graphens Υ in eine lineare Liste []:
 - ▶ vereinfachte Optimierung
 - ▶ direkte Abbildung auf RTL mit Zustandsautomaten möglich
 - ▶ nur wenige symbolische Befehle:
{**MOVE, EXPR, JUMP, FALSEJUMP, FUN, BIND, NOP**}
- ▶ Menge von Regeln χ bestimmt Abbildung (Synthese) der Prozeß-Instruktionen κ :

Zwischenrepräsentation μ Code

$$\chi_{\kappa\mu}: \Upsilon(\kappa) \rightarrow [\mu]$$

Scheduling und Allokation

Register-Transfer-Logik

Abbildung der linearen Liste von Mikro-Instruktionen $[\mu]$ auf Zustandsliste $\Sigma = [\sigma]$ mit $\sigma_n: \sigma_n \rightarrow \{\sigma_i\} | E$, und Datenpfad $\Delta = [\delta]$ mit $\delta_n: E \rightarrow A | \sigma_n$

$$\chi_{\mu\Sigma\Delta}: [\mu] \rightarrow [\sigma, \delta]$$

Zeitschritt-Zuordnung

- ▶ Weitere Regelsätze χ_T definieren Transformationen

7.2 Regelbasiertes Syntheseverfahren

Beispiel

▼ ConPro

```
for i = 1 to 10 do
  y ← y * (i + 1), x ← x+1;
```

▼ Anwendung der Regeln: $\chi_{loop-def}, \chi_{bounded-block}, \chi_{assign}, \chi_{expr-flat} \rightarrow \mu$ Code

```
    i1_for_loop:
i1_for_loop_cond:  move (LOOP_i_0,1) with ET=I[5]
                   bind (2)
                   expr ($immed.[1],10,>=,LOOP_i_0) with ET=I[5]
                   falsejump ($immed.[1],i1_for_loop_end)
    i2_bind_to_3:
                   bind (5)
                   expr ($immed.[2],LOOP_i_0,+,1) with ET=I[8]
                   expr (y,y,*, $immed.[2]) with ET=I[8]
                   nop
                   expr (x,x,+,1) with ET=I[8]
                   nop
    i2_bind_to_3_end:
                   nop
    i1_for_loop_incr:
                   bind (3)
                   expr (LOOP_i_0,LOOP_i_0,+,1) with ET=I[5]
                   nop
                   jump (i1_for_loop_cond)
    i1_for_loop_end:
```

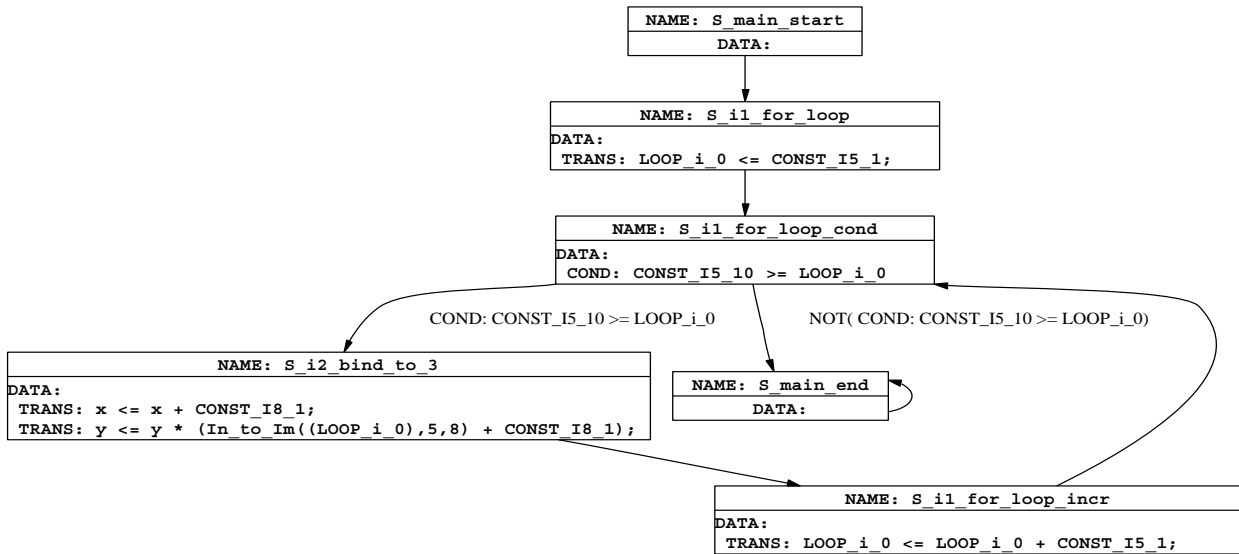
with
Parametrisierung von
Ausdrücken

bind
Bindung von
Instruktionen in
einen logischen
Zeitschritt

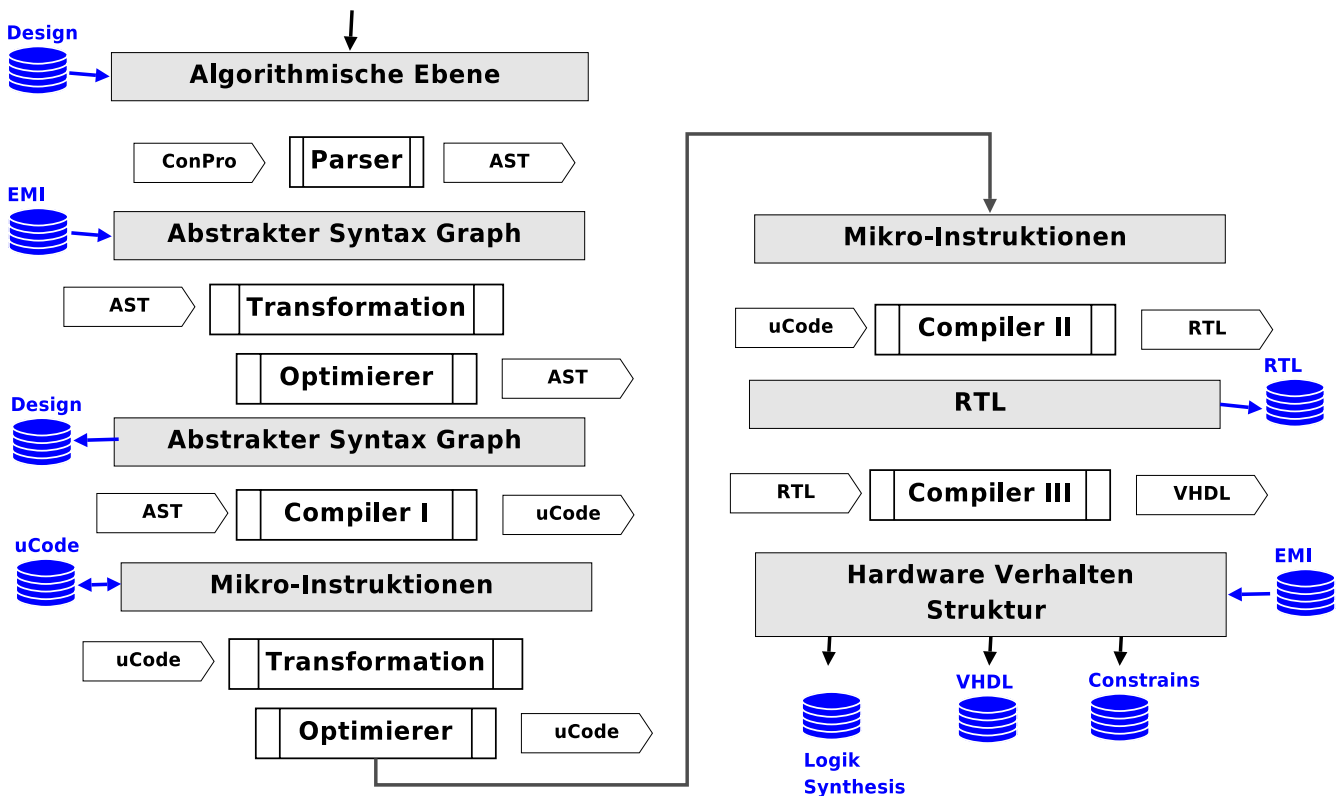
7.3 Regelbasiertes Syntheseverfahren

Beispiel (Fortsetzung)

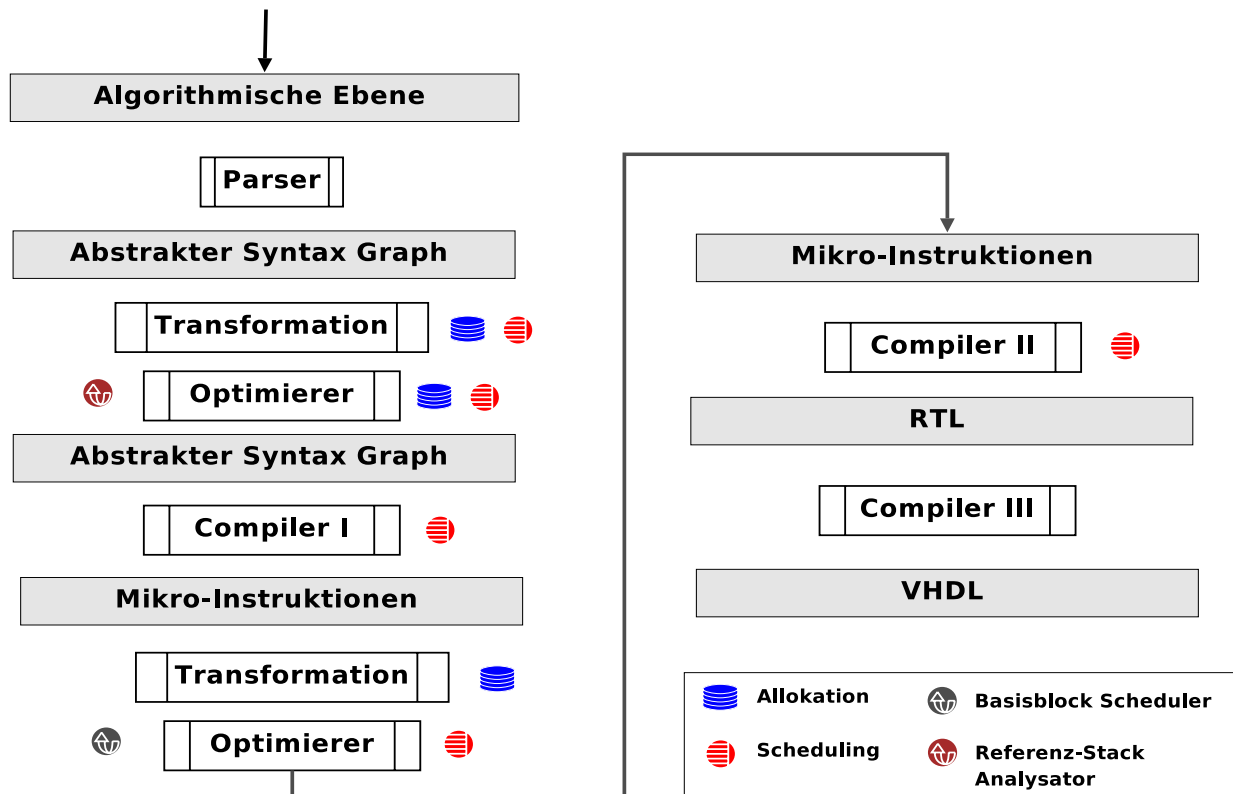
▼ RTL



7.4 Regelbasiertes Syntheseverfahren



7.5 Regelbasiertes Syntheseverfahren



8.1 Scheduling & Allokation

Regelbasierte Synthese

Syntheseregeln χ bestimmen wesentlich zeitliches Verhalten und Allokation (Temporäre Ausdrücke!)

Optimierung

Datenpfadanalyse: Konstantenfaltung reduziert Ausdrücke, Deadcode-Optimierer reduziert nicht genutzte Objekte und Instruktionen

Referenz-Stack Optimierer und Scheduler

Symbolische Analyse von Ausdrücken und Speicher-Objekten (Register und Variablen).

Ziel: Reduktion von 1. Zuweisungen, 2. Speicher-Objekten, 3. Ausdrücken
Ebene: Abstrakter Syntax Graph (AST)

Basisblock Optimierer und Scheduler

Nebenläufigkeits- und Datenabhängigkeitsanalyse

Ziel: Reduktion von Zeitschritten, Minimierung der Latenz

Ebene: Mikro-Instruktionen

Expression Scheduler

Einhaltung von zeitlichen Randbedingungen in Ausdrücken

Ebene: Mikro-Instruktionen

8•2 Scheduling & Allokation

Regelbasierte Synthese

Syntheseregeln χ können parametrisch selektiert werden:

1. bei der/für die Objektdefinition
 - Zugriffs-Scheduler
 - Implementierungsparameter und architekturen
2. und auf Prozeß-Block-Ebene, beliebig geschachtelt und fein granuliert:
 - Kontrollstrukturen wie Schleifen: **loop-unroll**
 - Nebenläufigkeit von Datenanweisungen: **bind**
 - Ausdrucksmodelle: **flat, shared, binary**
 - Allokation von temporären Registern: **shared, exclusive**
 - Scheduler- und Optimierungsstrategien: **refstack, basicblock, expression**

```
reg x: int[7] with scheduler="fifo";
process p1:
begin
  begin
    for i = 1 to 4 do begin x ← x+i; end with unroll;
  end with expr="flat";
  for i = 2 to 4 do begin x ← x*i; end with unroll;
end with scheduler="basicblock" and
      expr="shared";
```

8•2 Scheduling & Allokation

Referenz-Stack Optimierer und Scheduler

Symbolische Analyse von Ausdrücken und Speicher-Objekten (Register und Variablen).

- Für jedes Speicherobjekt Θ gibt es einen Ausdrucks-Stack $T(\Theta)$:
 $T(\Theta)=[\varepsilon_N, \varepsilon_{N-1}, \dots, \varepsilon_0]$
- Jede Wertzuweisungen mit einem Ausdruck ε_i für Objekt Θ wird dem Stack (Liste) hinzugefügt. Das Topelement ist immer der aktuelle Wert.
- Es entsteht eine Spur für alle Objekte, die verändert werden.
- Die Wertzuweisungen werden zunächst zurückgehalten.
- Durch Rückwärtssubstitution und Konstantenfaltung wird nur noch der letzte Ergebniswert einmalig dem Objekt zugewiesen.
► Scheduler mit ALAP-Verhalten

```
reg x,y,z: int[16];
x ← 10;          T(x)=[10]
y ← 11;          T(y)=[11]
x ← x+1;         T(x)=[T(x,0)+1;10]
z ← x+y-1;       T(z)=[T(x,1)+T(y,0)-1]
⇒ x=T(x,0)+1=10+1=11, y=11,
  z=T(x,1)+T(y,0)-1=T(x,0)+1+11-1=10+1+11-1=21;
```

8•3 Scheduling & Allokation

Basisblock Optimierer und Scheduler

Nebenläufigkeits- und Datenabhängigkeitsanalyse.

- Basisblöcke sind Bereiche im Programmflußgraphen, die nur einen Kontrollzugang am Kopf und nur einen Kontrollausgang am Ende haben, und keine weiteren Seiteneingänge.
- Ein Basisblock der nur aus Datenanweisungen besteht (Major-Block), wird in elementare Minor-Blöcke zerlegt, die wenigstens eine Anweisung enthalten (bei einem gebundenen Block der ganze Block)
- Es werden Datenabhängigkeitsgraphen für jeden Major-Block erstellt.
- Nicht abhängige Anweisungen werden durch einen Scheduler mit ASAP-Verhalten in einen gebundenen Block (ein Zeitschritt) zusammengefaßt.

Expression Scheduler

Einhaltung von zeitlichen Randbedingungen in Ausdrücken

- Es werden Ausdrucks-Laufzeitfunktionen $T(F,D)$ definiert, die einen normierten Laufzeitwert im Bereich $0..1$ liefern, und von der Funktion F und der Datenwortbreite D abhängen.
- Komplexe Ausdrücke werden in temporäre Subaudrücke zerlegt, die kumulativ maximal eine Laufzeit 1 besitzen (Inferenz von temporären Registern!).

9•1 Beispiel: Parity-Berechnung

```
open Core;
open Process;
open Mutex;

const WIDTH: value := 64;
reg d: logic[WIDTH];
reg p: logic;
export p,d;

function parity (x: logic[WIDTH])
    return (p: logic):
begin
    reg pl: logic;
    reg xl: logic[WIDTH];
    xl ← x;
    pl ← 0;
    for i = 0 to WIDTH-1 do
        begin
            pl ← pl lxor xl[i];
        end with PARAMS;
    p ← pl;
end with PARAMP;

process main:
begin
    d ← 0x12345670;
    p ← parity(d);
end;
```

- Design-Implementierung mit verschiedenen Schleifenparametern (PARAMS) und Prozeß-Parametern (PARAMP).
- Design wurde auf Standard-Zellen-Bibliothek SXLIB mit Leonardo Spectrum synthetisiert.
- Nachfolgende Tabelle zeigt Ergebnisse. TU: von ConPro berechnete Zeitschritte für Funktion parity

Parameter	Time, Gates, Register, Path Delay
default	196 TU, 972, 82, 5.7 ns
basicblock	130 TU, 937, 79, 5.3 ns
unroll	67 TU, 1525, 138, 4.1 ns
refstack+unroll	3 TU, 879, 69, 3.4 ns
basicblock+unroll	65 TU, 1538, 137, 4.1 ns
refstack+basicblock+unroll	2 TU, 853, 66, 3.4 ns

Fragen ?

```
reg x: logic[8];  
process p1:  
begin  
  ev.await ();  
  y ← f(x);  
end;  
process p2:  
begin  
  for i =  
    1 to 10 do  
    x ← x+1;  
    ev.wakeup ();  
end;
```



```
library IEEE;  
use IEEE....  
port(  
  signal PORT_x_RE:  
    out std_logic;  
  signal PORT_x_RD:  
    in std_logic_vector(7 downto 0);
```

