High-Level-RTL-Synthese mit einem Multi-Prozess-Modell

Von der imperativen algorithmischen Ebene zu der Register-Transfer Digitallogik-Ebene

Dr. Stefan Bosse
Universität Bremen

30.6.2010

Inhaltsverzeichnis

Überblick und Einführung von Syntheseverfahren für Register-Transfer Systeme 1
- Überblick und Einführung von Syntheseverfahren für Register-Transfer Systeme 1
- Synthese von Digitallogikschaltungen 2
- Synthese von Digitallogikschaltungen :: RTL (I) 3
- Synthese von Digitallogikschaltungen :: RTL (II) 4
- Algorithmische Ebene und Synthese :: Imperativ 5
- Algorithmische Ebene und Synthese :: Funktional 8
- Higher-Level-Synthese (I) 10
- Higher-Level-Synthese (II) 13
- Ein Beispiel: Transputer-Architektur & OCCAM Programmiersprache 14

Regelbasierte Syntheseverfahren 16
- Regelbasierte Syntheseverfahren 16
- Higher-Level-Synthese :: Regelbasierte Synthese (I) 17
- Higher-Level-Synthese :: Regelbasierte Synthese (II) 22

Multiprozeß-Modell und Interprozeß-Kommunikation 27
- Multiprozeß-Modell und Interprozeß-Kommunikation 27
- Multiprozeß-Modell 28
- Interprozeß-Kommunikation 31
- Interprozeßkommunikation :: Abstrakte Datentypen 32
- Multiprozeß-Modell :: Hardware-Implementierung 41

ConPro: Higher-Level-Synthese Werkzeug und Programmiersprache 43
- ConPro: Higher-Level-Synthese Werkzeug und Programmiersprache 43
- ConPro: Higher-Level-Synthese 44

Inhaltsverzeichnis

ConPro :: Multiprozeß-Modell :: Hardware-Implementierung 47
ConPro: Higher-Level-Synthese :: Objekte 50
ConPro: Higher-Level-Synthese :: Produktbildungstypen 55
ConPro: Higher-Level-Synthese :: Summenbildungstypen 56
ConPro: Higher-Level-Synthese :: Kontrollstrukturen 57
ConPro: Higher-Level-Synthese :: Abstrakte Daten Objekte 59
ConPro: Higher-Level-Synthese :: Funktionen 61
ConPro: Higher-Level-Synthese :: Interconnect-Architektur 63
ConPro: Higher-Level-Synthese :: Synthese 65
ConPro: Higher-Level-Synthese :: Synthese (II) 68
ConPro: Higher-Level-Synthese :: µCode 70
ConPro: Higher-Level-Synthese :: Referenz Stack 73
ConPro: Higher-Level-Synthese :: Basisblock Scheduler 76
ConPro: Higher-Level-Synthese :: Beispiel 79

1. Überblick und Einführung von Syntheseverfahren für Register-Transfer Systeme

Frage

Wie kommt der Algorithmus auf den Mikrochip?

- Register-Transfer Modell
- Algorithmische Ebene: Imperative und Funktionale Modelle
- Higher-Level Synthese
- Ein Beispiel: Transputer-Architektur und OCCAM Programmiersprache

Synthese von Digitallogikschaltungen

Es gibt zwei wesentliche Beschreibungs- und Modellierungsebenen für die Entwicklung von Digitallogikschaltungen:

Register-Transfer Ebene RTL

- Beschreibung der Digitallogikschaltungen mit getrennten Daten- und Kontrollpfad (Δ und Γ).
- Der Datenpfad besteht aus Funktionsblöcken Π, Datenpfadselektoren Σ und Registern $\Re$.
- Der Kontrollpfad besteht aus einem endlichen Zustandsautomaten FSM mit einer endlichen Menge von Zuständen $\Phi=\{\varphi_1,\varphi_2,..\}$, der den Datenpfad spatial und zeitlich steuert.

Algorithmische Ebene ALG

Eine Sequenz von Anweisungen, die mit Mitteln einer imperativen oder funktionalen Programmiersprache implementiert werden, beschreibt das (schrittweise) funktionale Verhalten als Relation zwischen Eingabe- und Ausgabedaten $R \subseteq E \times A$ der Digitallogikschaltung.
Daten- und Kontrollpfad (Δ und Γ) werden explizit aber kombiniert durch die Anweisungssequenz beschrieben, gekapselt durch Anweisungsblöcke B.

Synthese von Digitallogikschaltungen :: RTL (I)

- RTL-Modellierung entspricht einer Spezifikation als **gerichteter Graph**, deren Knoten Schaltungsblöcke (Π,Σ,$\Re$) und die Kanten Verbindungsleitungen darstellen.
- Die RTL-Spezifikation wird dann in eine Hardware-Verhaltens-Beschreibung (VHDL,Verilog) umgesetzt: 1. per Hand, 2. automatisch als Synthese-Verfahren.
- Obwohl alle (imperativen) Algorithmen mit dieser Methode modelliert und spezifiziert werden können, steigt die **Komplexität** überlinear (exponentiell) mit der Algorithmus- und Systemkomplexität.
- Diese stark ansteigende Entwurfskomplexität resultiert in einem entsprechenden Anstieg an Entwicklungszeit- und Ressourcen.
- Zudem steigt die **Fehleranfälligkeit** gleichzeitig mit einer reduzierten Möglichkeit, diese Fehler überhaupt feststellen und testen zu können.
- Die RTL-Ebene bietet **inherent** maximale **Parallelität** - die aber vom Entwickler explizit formuliert werden muß. Parallelität bedeut aber auch **Synchronisation** - diese muß ebenfalls explizit modelliert werden und ist nicht Bestandteil des Entwurfsmodells.
- Die RTL-Ebene setzt weiterhin explizite Modellierung des zeitlichen Ablaufs und aller Ressourcen voraus $\Rightarrow$ Diskretisierung des Kontrollpfades $\Gamma \Rightarrow$ **Scheduling und Allokation**.

Synthese von Digitallogikschaltungen :: RTL (II)

RTL-Implementierung aus algorithmischer Beschreibung bedeutet:

Scheduling

Zuordnung von Operationen zu Zeitschritten. *Optimierung*: Minimierung der Zeitschritte.

Ressourcen-Allokation

Bestimmung von Typ und Anzahl der erforderlichen Hardware-Ressourcen wie Funktionselemente, Speicher, Busse. *Optimierung*: Minimierung der Funktionselemente, z.B. durch ALU-Blöcke, die über Multiplexern mit mehreren arithmetischen Operationen verwendet werden können (Ressource-Sharing).

- Erzeugung einer RTL bedeutet das Abbilden von Daten- und Kontrollfluß in zwei Dimensionen: Zeit ≡ FSM und Fläche ≡ Hardware ≡ Logik.

Ressourcen-Zuweisung

A. Zuordnung von Funktionselementen zu einzelnen Instanzen und Operationen.

B. Abbildung auf eine Hardware-Verhaltensbeschreibung

- starke Wechselwirkung zwischen Scheduling und Ressourcen-Allokation
- RTL-Scheduling- und Allokation kann per Hand, aber auch regel- und modellbasiert mit automatischer **High-Level-Synthese** erzeugt werden.

Algorithmische Ebene und Synthese :: Imperativ

Algorithmische Ebene kann durch zwei verschiedene Beschreibungsmodelle ausgedrückt werden:

Imperativ oder Prozedural

- Anweisungen erzeugen schrittweise und sequenziell extern sichtbare Ergebniswerte, i.A. über Zwischenwerte die nur intern sichtbar sind.

Table 1. Sequenzielle Ausführung von Anweisungen

Anweisungsmenge	Ausführungsmodell
$A_1;A_2;A_3..$	❶A_1➡❷A_2➡❸$A_3..$

- Anweisungen sind Bestandteil des Daten- und Kontrollpades.
- Neben Datenanweisungen gibt es explizite Kontrollanweisungen, die den Kontrollfluß direkt und sichtbar beeinflußen.
- Anweisungen werden in Blöcken gekapselt. Es gibt i .A. benannte Blöcke ⇒ Prozeduren/ Funktionen 1. Ordnung.
- RTL läßt sich direkt aus imperativer Beschreibung ableiten, i.A. prozessororientiert eindeutig bestimmt bezüglich Scheduling und Allokation.
- Keine inherente Parallelität!

- Explizite Parallelisierung z.B. mit dem geläufigen Programmiermodell Multi-Threading möglich:
- Thread ➡ Leichtgewichtiger Prozeß ➡ Nebenläufigkeit von Prozessen
- Programmiermodell: **Kommunizierende Sequenzielle Prozesse** [C.A.R. Hoare, 1985]

Table 2. Parallele Ausführung von Anweisungen

Anweisungsmenge	Ausführungsmodell
$A_1,A_2,A_3..$	❶$\langle A_1 ▮ A_2 ▮ A_3.. \rangle$

- Implizite Parallelisierung nur durch Synthese-Werkzeug oder durch Erweiterung des Programmiermodells (parametrisiertes Multi-Prozeß-System) möglich.
- Extrahierte Parallelität aus

Schleifen

Abrollen des Schleifenkörpers B ➡ Replikation der Anweisungen in B

Basisblöcken

Bindung und Ausführung mehrerer datenunabhängiger Anweisungen in einem Zeitschritt)

Beispiele:

Basisblock Parallelität

Example 1. Transformation sequenzielle nach parallele Ausführung

```
reg x,y,z: int[8];
begin                ⇒  begin
  x←0;                    x←0 , y←1;
  y←1;                    z←x+y;
  z←x+y;                end;
end;
```

Abrollen von Schleifen (1) und Symbolische Analyse (2)

Example 2. Block einer Schleife mit konstanten Grenzen wird repliziert (1) und reduziert (2)

```
reg x: int[8];
x←100;
for i = 1 to 3 do
begin       ⇒(1) begin      ⇒(2) begin
  x←x+1;           x←x+1;          x←103;
end;               x←x+1;        end;
                   x←x+1;
                 end;
```

Algorithmische Ebene und Synthese :: Funktional

(Forts.) Algorithmische Ebene kann durch zwei verschiedene Beschreibungsmodelle ausgedrückt werden:

Funktional

- Bei der funktionalen Beschreibung wird nur der Zusammenhang zwischen Ein- und Ausgabedaten beschrieben, nicht aber der sequenzielle Ablauf. Der Daten- und Kontrollpfad ist transparent.
- Anweisungen sind Funktionen, die Eingabe- auf Ausgabewerte abbilden, und nur eine Relation $R \subseteq E \times A$ beschreiben.
- Inherente Parallelität vorhanden, z.B. bei der Evaluierung von Funktionsargumenten!
- Funktionale Programmierung setzt sich aus einer Menge von **Funktionsdefinitionen, Funktionsapplikationen und Funktionskompositionen** zusammen.
- **Funktionskomposition erlaubt die direkte Abbildung auf Hardware-Blöcke.**
- Das Scheduling und die Allokation ist transparent und ist dem Synthese-Modell und Synthese-Werkzeug überlassen.

Beispiele:

Example 3. Funktionsdefinition und Applikation

```
let f (x,y) =
  if x < 0 then y-1
  else x+y
let z = f (u+1,s-1)
```

Example 4. Transformation funktionales Modell auf imperative RTL

```
reg u,s,z: int[8];
reg f_x,f_y: int[8];
begin
  ...
  f_x←u+1 , f_y←s-1;
  ... (1)                      (2)
  if f_x < 0 then              z←SELECT{f_x<0:f_y-1∨f_x≥0:f_x+f_y};
    f_res←f_y-1
  else
    f_res←f_x+f_y;
  ...
  z←f_res;
end;
```

Higher-Level-Synthese (I)

1. Die Beschreibung eines Digitallogik-Schaltkreises mit der abstrahierten algorithmischen Ebene erfordert regelbasierte Abbildung von sequenziellen Anweisungsblöcken auf RTL
   - **Higher-Level-Synthese** ≡ Programmierebene ⇒ RTL-Ebene
   - RTL-Ebene ≡ Daten- und Kontrollfluß
   - Daten- und Kontrollfluß ≡ zeitdiskrete Ausführung von Anweisungen

   Die Synthese-Regeln und die Systemarchitektur sind eng verknüpft mit der verwendeten Programmiersprache und deren Programmiermodell ⇒ z.B. C oder Pascal/Modula

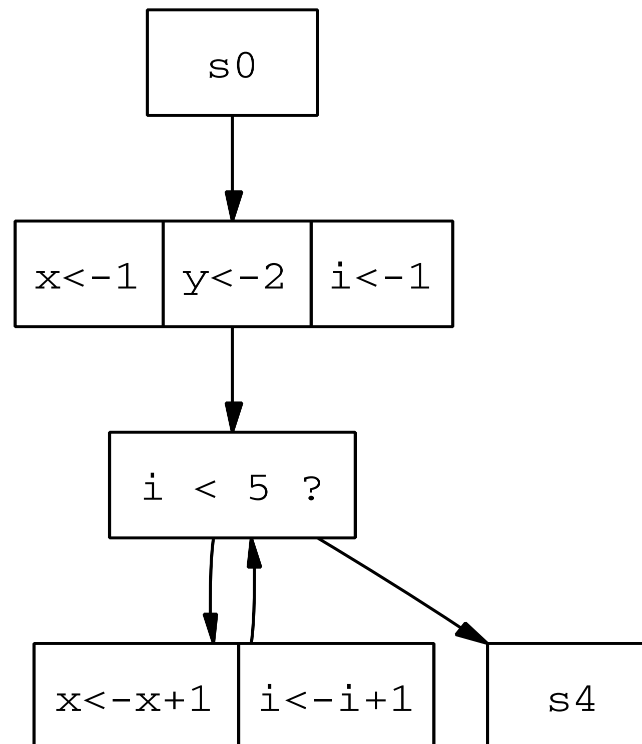
Table 3. Parallele Ausführung von Anweisungen

Anweisungsmenge	Ausführungsmodell
`concurrent ::= ','` $A_1,A_2,A_3..$	❶$\langle A_1 \blacksquare A_2 \blacksquare A_3.. \rangle$
`next-state ::= ';'` $A_1;A_2;A_3..$	❶A_1➡❷A_2➡❸$A_3..$
`jump-state ::= 'jump'` A_1;jump $A_i;A_3..$	❶A_1➡❷A_i➡❸$A_{i+1}..$

Example 5. Beispiel einer programmorientierten Higher-Level RTL Beschreibung

```
x←1,y←2,i←1;
loop: if i < 5 then
       {x←x+1,i←i+1;jump loop};
```

Figure 1. RTL Daten- und Kontrollpfad aus dem Beispiel



2. Die Abbildung von RTL auf eine Hardware-Verhaltensbeschreibung (VHDL) ist ein Zwischenschritt ⇒ **High-Level-Synthese**

Example 6. VHDL Beschreibung eines Zustandsautomaten für Beispiel RTL

```
process fsm()
begin
 if clk'event ... then
  case state is
   when S0 => state <= S1;
   when S1 => x <= "0001"; y <= "0010"; i <= "0001";
              state <= S2;
   when S2 => if i < 5 then state <= S3 else state <= S4 ...
   when S3 => x <= x+1; i <= i + 1;
              state <= S2;
   when S4 => ...
end;
```

3. Die Abbildung der Hardware-Verhaltenbeschreibung auf einen konkreten Schaltkreis (Netzliste von Logikgattern) ist der letzte Schritte ⇒ **Logik-Synthese**

Higher-Level-Synthese (II)

- Die Verwendung einer allgemeinen imperativen Programmiersprache wie C bedingt ein zunächst rein sequenzielles Programmier- und zentralistisches Systemarchitekturmodell!
- Parallelisierung als Motivation des anwendungs-spezifischen Schaltkreisentwurfs zunächst nicht vorhanden.
- Entweder implizite Parallelisierung durch das Synthese-Werkzeug oder Erweiterung des Programmiermodells mit expliziter Parallelisierung, z.B. konkurrierende Prozeße ⇒ Multi-Threading.
- Existierende imperative und funktionale Programmiersprachen unterstützen keine bit-skalierbaren Datentypen und Datenobjekte ⇒ nur wenige statische verfügbare Datenwortbreiten.
- Vorteil einer weit verbreiteten Programmiersprache aus der "Software-Welt" für den anwendungs-spezifischen Schaltkreisentwurf: große Akzeptanz bei Software-Entwicklern, einfache Adaption bisheriger Implementierungen von Algorithmen.
- Jedoch: rein automatisches Scheduling und insbesondere Allokation (d.h. Inferenz, nicht nur Mapping) von Hardware-Ressourcen - die sonst bei reinen Software-Ansatz entfällt - führt i.A. zu unzufriedenstellenden Ergebnissen ⇒ Performanz & Ressourcen- Effizienz

Ein Beispiel: Transputer-Architektur & OCCAM Programmiersprache

Transputer: massiv paralleles Rechnersystem [Inmos, 1980iger]

OCCAM: Parallele Programmiersprache

- Grundmodell: Multi-Prozeß
- Prozeß: beliebig fein granuliertes Modell und beliebig tief schachtelbar ➡ Baumstruktur von Prozessen
- Prozeß-Definition: sequentielle oder parallele Komposition von Komponenten (Prozessen oder elementare Anweisungen)
- Elementare Anweisungen: primitive Prozesse, Datenpfadoperationen
- Kommunikation: nachrichtenbasiert über synchrone Kanäle (Channels)
- Operationen: Read `‘?’`, Write `‘!’`.
- Netzwerkstruktur: zwei-dimensionales Maschennetzwerk, jeder Knoten (Transputer) besitzt Verbindungen zu seinen vier Nachbarn
- Synchronisation: nur über Nachrichtenaustausch

Table 4. Prozesse und Sparchelemente (Konstruktoren) in OCCAM

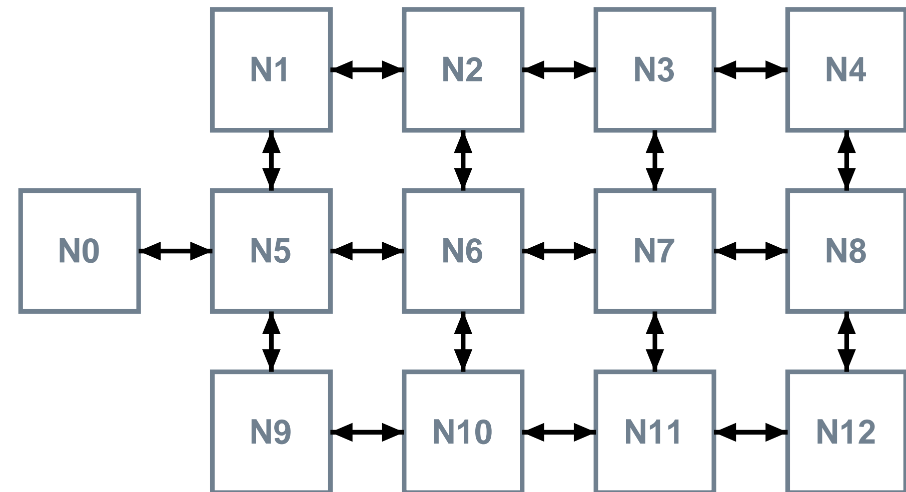
Prozeß-Definition	Ausführungsmodell
`concurrent ::= ‘PAR’`	Paralleles Konstrukt: führt alle Komponenten dieses Prozesses nebenläufig aus.

Prozeß-Definition	Ausführungsmodell
`sequential ::= 'SEQ'`	Sequentielles Konstrukt: führt alle Komponenten dieses Prozesses in der Reihenfolge sequentiell aus.
`replicated-seq ::= 'SEQ' index '=' base 'FOR' count`	Ein Array von Prozessen wird erzeugt, die sequentiell ausgeführt werden.
`replicated-par ::= 'PAR' index '=' base 'FOR' count`	Ein Array von parallelen Prozessen wird erzeugt.

Example 7. OCCAM Beispiel

```
1 PAR
2   SEQ
3     chan1 ? x
4     x := x + 1
5     chan2 ! x
6   SEQ
7     chan2 ? y
8     y := y + 1
9     chan1 ! y
```

Figure 2. Transputer Netzwerk



2. Regelbasierte Syntheseverfahren

Frage

Wie kommt der Algorithmus auf den Mikrochip?

1. Einfachtes Scheduling- und Allokationsmodell: Regeln
2. Optimierung

Higher-Level-Synthese :: Regelbasierte Synthese (I)

Einfachstes Scheduling- und Allokationsmodell: Grundregeln

Regel I

Es gibt nur Register als Datenspeicher mit beliebiger Datenwortbreite.

Regel II

Kein Ressource-Sharing: jedes Register $\Re$ was definiert wurde wird auch als Hardware-Block ω implementiert.

Regel III

Jede elementare Anweisung κ wird auf einen Zustand φ des Kontrollflußes $\Gamma=(\Phi,E)$ als gerichteter Graph bzw. der Zustandsmenge Φ des RTL-FSM abgebildet.

Regel IV

Der Datenpfad Δ wird in Abhängigkeit der Zustände Φ zerlegt.

Regel V

Kein Ressource-Sharing: jeder funktionale Operator (arithmetisch, relational und boolsch) Π wird mit eigenen Hardware-Blöcken ω implementiert.

Regel VI

Komplexe Anweisungen ε (z.B. Schleifen) müßen in eine Menge $\{\kappa\}$ von elementaren Anweisungen mit Transformationsregeln zerlegt werden.

Regel VII

Zu jedem Zustand φ gehört ein Folgezustand $\varphi+$, der bedingt oder unbedingt definiert sein kann.

Regel I

Es gibt nur Register als Datenspeicher mit beliebiger Datenwortbreite.

Definition 1. Register

```
reg name: type[size];
reg x,y,z: int[8];
```

Regel II

Kein Ressource-Sharing: jedes Register $\Re$ was definiert wurde wird auch als Hardware-Block ω implementiert.

Definition 2. Register Allokation

```
DEF ALLOCATE : ℜ={R1,R2,...} ➡ {ω}={ω1,ω2,...}
```

Regel III

Jede elementare Anweisung $\kappa \in K$ wird auf einen Zustand $\varphi \in \Phi$ des Kontrollflußes Γ bzw. der Zustandsmenge Φ des RTL-FSM abgebildet.

Table 5. Elementare Anweisungen

Elementare Anweisung	Wirkung
`operator ::= '+'|'-'|...`	Binärer arithmetischer, logsicher oder boolescher Operator (zwei Operanden)
`expression ::= (ident operator ident) | ident`	Einfacher Ausdruck
`assign ::= ident '<-' expression`	Wertzuweisung an Register (Datentransfer)
`jump ::= 'jump' label;`	Verzweigung im Kontrollpfad, unbedingt
`falsejump ::= 'falsejump' label;`	Verzweigung im Kontrollpfad, bedingt

Definition 3. Scheduling Anweisungen

DEF SCHEDULE : $\{\kappa\}=\{\kappa 1;\kappa 2;...\} \Rightarrow \Gamma=\{\varphi 1;\varphi 2;...\}$

Regel IV

Der Datenpfad Δ wird in Abhängigkeit der Zustände Φ zerlegt.

Regel V

Kein Ressource-Sharing: jeder funktionale Operator (arithmetisch, relational und boolsch) π∈∏ wird mit eigenen Hardware-Blöcken ω implementiert.

Definition 4. Allokation Operatoren

```
DEF ALLOCATE : {π}={π1,π2,...} ➡ {ω}={ω1,ω2,...}
```

Regel VI

Komplexe Anweisungen ε (z.B. Schleifen) müßen in eine Menge {κ} von elementaren Anweisungen mit Transformationsregeln zerlegt werden.

Table 6. Zerlegung von komplexen Anweisungen

Komplexe Anweisung	Transformation

`FOR i = a TO b DO B`	`DEF ALLOCATE: i ➡ R ➡ ω`
	`DEF TRANSFORM: i <- a;`
	`LOOP: i < b?;`
	`falsejump END;`
	`B;`
	`i <- i + 1;`
	`jump LOOP;`
	`END: ...`

`IF B THEN B1 ELSE B2`	`DEF TRANSFORM: EVAL(B);`
	`falsejump LB2;`
	`B1;`
	`jump END;`
	`LB2: B2;`
	`END: ...`

Regel VII

Zu jedem Zustand φ gehört ein Folgezustand φ+, der bedingt oder unbedingt definiert sein kann. => Kanten des Zustandsgraphen: {NEXT(φ→φ+),CONDITIONAL(B:φ→φ+)}

Higher-Level-Synthese :: Regelbasierte Synthese (II)

Verbessertes Scheduling- und Allokationsmodell: Optimierungsregeln

Regel OI

Analyse des Daten- und Kontrollpfades bezüglich **Basisblöcken** mittels **Daten-Abhängigkeitsgraphen (DDG)**.

Ein Basisblock β ist ein zusammenhänger Anweisungsblock mit nur einem Kontrollpfad-Zugang am Kopf und ohne Seiteneffekte.

➥ Scheduling: Bindung, Reduktion der Zeitschritte, keinen Einfluß auf Allokation

---

Algorithm 1. Basisblock und Datenabhängigkeit Analyse

```
DEF TRANSFORM: K={κ} ➡ B={β}
  b=[]; B=[];
  WHILE K ≠ [] DO
    k ← HEAD(K), K ← TAIL(K);
    IF KIND(k) = DATA THEN b ← b + [k] ELSE B ← B + b, b ← [];

DEF SCHEDULE: B={β} ➡ K'={κ}
  K=[];
  FOREACH b ∈ B DO
    DDG ← BUILD DDG OF LIST (b);
    FOREACH INDEPENDENT SUBSET k ⊆ DDG DO K ← K + [BIND(k)];
```

Example 8. DDG und Basisblock-Scheduling mit ASAP-Scheduler

```
K=[
  ❶ x <- 1;                         B=[                  DDGs=[
  ❷ y <- 2;                             [❶❷]                 x:[❶],y:[❷]
  ❸ IF x < y THEN y <- x; ➡             [❸]       ➡          [❸]
  ❹ z <- x+y;                           [❹❺❻]                z:[❹➡❻],x:[❹➡❺],
  ❺ x <- x-1;                                                          y:[❹➡❻]
  ❻ y <- z+1;]                        ]                    ]
➡
K'=[
  ①➡ BIND(❶❷)
  ②➡ ❸
  ③➡ ❹
  ④➡ BIND(❺❻)]
```

Algorithm 2. ASAP Scheduler

```
DEF SCHEDULE: Ds={DDGs} ➡ K'={κ}
  K ← [];
    FOREACH D ∈ Ds DO
    k ← [];
    FOREACH d ∈ D DO IF h=HEAD(d) ∉ k AND h ∉ TAIL(d') ∀ d' ∈ (D' = D\[d]))
                      THEN k ← k + [h],d ← TAIL(d);
    K ← K + [BIND(k)];
```

Regel OII

Ressourcen-Sharing: Analyse des Daten- und Kontrollpfades bezüglich **Basisblöcken** mittels **Daten-Fortpflanzungsgraphen (DPG)**, 2. Symbolische Analyse der sequenziellen Entwicklung von Ausdrücken ⇒ Registeroptimierung

➥ Scheduling: kein Einfluß, Allokation: Reduktion von Ressourcen

Algorithm 3. Daten-Fortpflanzungsanalyse und Ressourcen-Sharing

```
DEF TRANSFORM: B={β},W={ω} ➡ {DPG}
  FOREACH b ∈ B DO
    DPG(w) ← [];
    FOREACH k ∈ b DO IF w ∈ k THEN DPG(w) ← DPG(w) + [k];
DEF SUBSTITUTE: D={DPG},W={ω} ➡ W'={ω,ω°}
  FOREACH NON OVERLAPPING SET d({ω}) {⊆} D DO: {ω} ➡ ω°
```

Example 9. DPG Analyse

```
B=[❶a←1;❷b←a-1;❸c←0;❹d←c+1;]
➡ W=[a,b,c,d]
➡ D=[①DPG(a)=[❶➡❷],②DPG(b)=[❷],③DPG(c)=[❸➡❹],④DPG(d)=[❹]]
①,③:[a,c]➡ S1, ②,④:[b,d] ➡ S2]
➡ W'=[S1,S2]
```

Regel OIII

Ressourcen- und Zeitschrittreduktion: Symbolische Analyse mit **Referenzstackmethode** T der sequenziellen Entwicklung von Ausdrücken und Wertfortpflazung ⇒ Registeroptimierung

➥ Scheduling: Reduktion von Zeitschritten, Allokation: Reduktion von Ressourcen

Algorithm 4. Referenzstack: Analyse

```
DEF ANALYSIS: K={κ},W={ω} ➡ {T}
  FOREACH w ∈ W DO T(w) ← [];
  FOREACH k ∈ K DO
    FOREACH w ∈ LHS(k) DO
      ε ← [];
      FOREACH w' ∈ RHS(k) DO
        IF VALUE(w') THEN ε ← ε + [ν]
        ELSE IF T(w') ≠ [] THEN ε ← ε + [REFERENCE(T(w'),ACTUAL TOP)]
        ELSE ε ← ε + [OBJECT(w')];
      T(w) ← [ε] + T(w);
```

Algorithm 5. Referenzstack: Rückwärts-Substitution

```
DEF TRANSFORM: K={κ},W={ω} ➡ K'={κ},W'={ω}
  FOREACH k ∈ K DO
    FOREACH w ∈ RHS(k) DO
      w ➡ RESOLVE(T(w),TOP)
```

```
DEF RESOLVE: T(w),nth ➡ w'
  x ← NTH(T(w),nth];
  IF VALUE(x) THEN x ELSE
  IF REFERENCE(x) THEN RESOLVE(x) ELSE x
```

Example 10. Referenzstack

```
  κ1: x←1;
  κ2: a←x+1;
  κ3: a←a*2;
  κ4: b←x+a;  ... x wird nicht weiter verwendet

[κ],[ω]
  K: {κ1,κ2,κ3,κ4} mit W:{ω1=x,ω2=a,ω3=b}
➡
  T(x):{ν1} ⇒ x←T(x,1)≡ν1;
  T(a):{ε(T(x,1),ν2),ε(T(a,1),ν3)} ⇒ a←T(a,2)≡ε(ν1,ν2)=ν4
  T(b):{ε(T(x,1),T(a,2)} ⇒ b←T(b,1)≡ε(ν1,ν4)=ν5
➡
[κ'],[ω']
  κ1': a←2;  ω'={ω1'=a,ω2'=b}
  κ2': b←5;
```

3. Multiprozeß-Modell und Interprozeß-Kommunikation

Frage

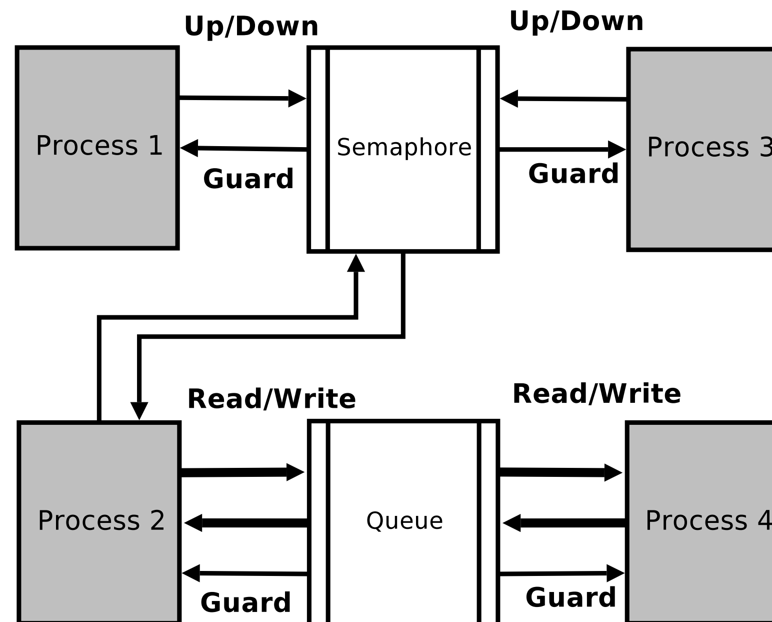
Wie kann Nebenläufigkeit modelliert werden?

1. Multiprozeß-Modell
2. Interprozeß-Kommunikation
3. Multiprozeß-Modell: Hardware-Implementierung

Multiprozeß-Modell

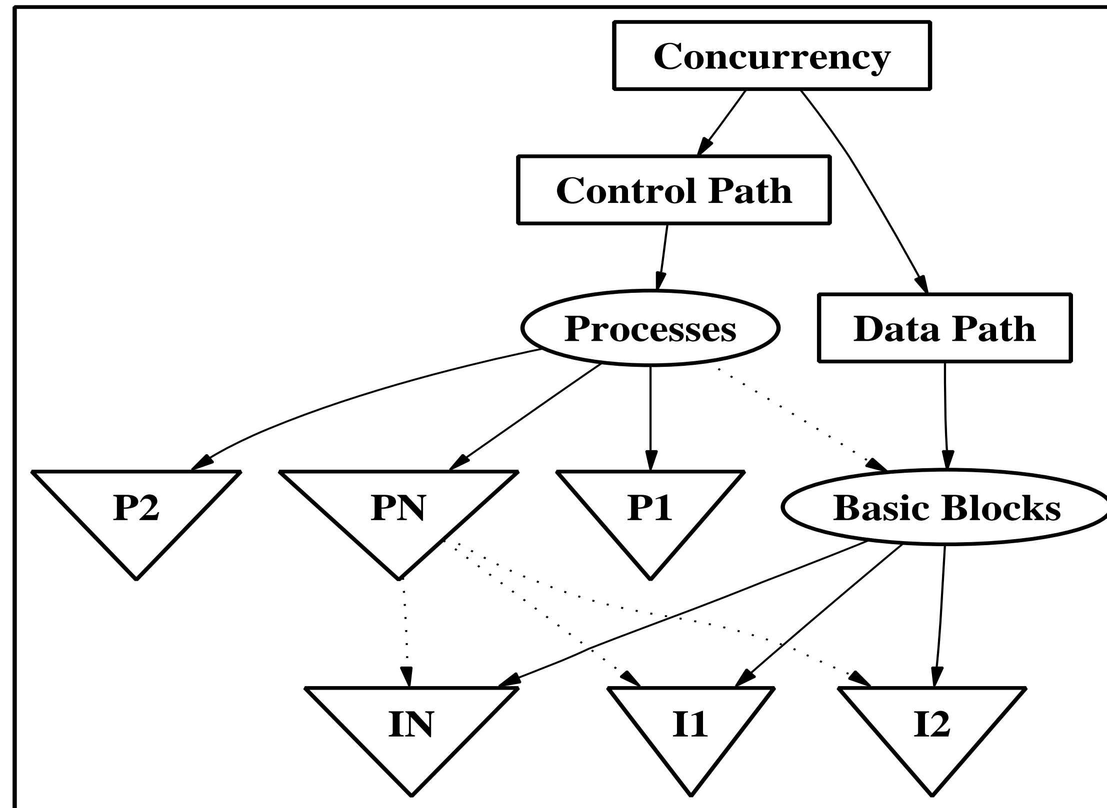
- Programmier- und Architekturmodell: Parallelisierung durch Partitionierung des Gesamtsystems in eine Vielzahl N von zunächst unabhängigen Prozessen $P=[\rho]$, die konkurrierend zueinander arbeiten.
- Jeder Prozeß ρ wird durch einen eigenen endlichen Zustandsautomaten FSM modelliert.
- Es gibt Prozeß-lokale Ressourcen ϖ, und globale von allen (oder einigen) Prozessen geteilte Ressourcen ω.
- Die Prozesse kommunizieren mit- und untereinander: Interprozeßkommunikation ≡ Synchronisation ist erforderlich.
- **Parallelisierung und Synchronisation**:
  - grob ganuliert
  - nicht hierarchisch
  - explizit
  - nicht transparant
  - Bestandteil des Programmiermodells
- Ein Prozeß kann sich in folgenden Zuständen befinden:
  ψ=**`{S_START,S_RUN,S_END,S_BLOCKED}`**
- Multi-Prozeß-Modell mit Interprozeßkommunikation über Semaphoren und Queues

Figure 3. Prozesse und Kommunikation



- Parallelität bei dem Multi-Prozeß-Modell auf Prozeßebene (Kontrollpfad) ⇒ MIMD-Architektur
- Parallelität auf Datenpfadebene ⇒ MIMD2-Modell

Figure 4. Nebenläufigkeit auf Kontroll- und Datenpfadebene beim Multiprozeß-Modell



Interprozeß-Kommunikation

- Interprozeß-Kommunikation (IPC) beeinflußt Kontrollpfad Γ der Prozeß-Menge $P=[\rho]$ und deren aktuellen Zustände ψ`={S_START,S_RUN,S_END,S_BLOCKED}`.
- IPC läßt sich als Operation auf einem abstrakten Datentypobjekt (ADTO) TYPE Θ modellieren. Nur die Wirkung von Operationen $\{\theta\}$, nicht die Implementierung des ADTO ist sichtbar.
- Basiselement: Mutex zum Schutz gemeinsam geteilter Ressource, auch ein IPC-ADTO ist selbst eine geteilte Ressource!
- Menge der IPC Objekte $\{\Theta\}$=

  A. Mutex

  B. Semaphore

  C. Event

  D. Timer

  E. Barriere

  F. Queue

  G. Channel

Interprozeßkommunikation :: Abstrakte Datentypen

A. Mutual Exclusion Lock (TYPE Mutex)

- Eine Mutex wird verwendet wenn mehrere Prozesse auf eine geteilte/globale Ressource ω zugreifen, z.B. Register oder RAM-Blöcke,oder IPC-Objekte selbst!
- Es gibt implizite und explizite Mutex: explizit durch den Programmierer/Entwickler für ein nicht atomares ObjeKt (z.B. verkettete Liste), implizit für jedes geteilte Objekt als Bestandteil des Implementierungsmodells!
- Eine Mutex kann zwei Zustände besitzen: ψ **={S_UNLOCKED,S_LOCKED}**
- Es gibt zwei Operationen auf eine Mutex: **{θ}={LOCK,UNLOCK}**

θ=LOCK ▮ φ=S_UNLOCKED

Ein Prozeß $\rho 1$ erlangt durch eine LOCK-Operation die ausschließlichen Eigentumsrechte der Mutex.

Zustandsübergang: φ+(Θ): **S_UNLOCKED $\rightarrow$ S_LOCKED**

θ=LOCK ▮ φ=S_LOCKED

Ein Prozeß $\rho 2$ führt eine LOCK-Operation durch, wird aber blockiert bis $\rho 1$ die Mutex freigibt.

Zustandsübergang: φ+(ρ**2**): **S_RUN $\rightarrow$ S_BLOCKED ▮ φ=S_LOCKED**

θ=UNLOCK ▮ φ=S_LOCKED

Der Prozeß $\rho 1$ gibt die Mutex wieder frei.

Zustandsübergang: φ+(Θ): **S_LOCKED $\rightarrow$ S_UNLOCKED**

Zustandsübergang: φ+(ρ**2**): **S_BLOCKED $\rightarrow$ S_RUN**

B. Semaphore (TYPE Sema)

- Eine Semaphore wird für die Synchronisation bei Producer-Consumer Algorithmen eingesetzt.
- Eine Semaphore ist ein geschützter Zähler ϕ. Regel: $\phi \geq 0$
- Eine Semaphore kann zwei Zustände besitzen: ψ **={S_UNLOCKED,S_LOCKED}**
- Es gibt drei Operationen auf eine Semaphore: $\{\theta\}$ **={DOWN,UP,SET}**
  Die Wirkung ist abhängig von dem Zustand und dem Wert der Semaphore.

θ=SET ▮ φ=S_UNLOCKED

Ein Prozeß $\rho 1$ ändert den Wert der Semaphore.
Wertänderung: $\phi \leftarrow \phi$**0**

θ=DOWN ▮ φ=S_UNLOCKED,ϕ>0

Ein Prozeß $\rho 1$ ändert den Wert der Semaphore.
Wertänderung: $\phi \leftarrow \phi$**-1**

θ=DOWN ▮ φ=S_UNLOCKED,ϕ=0

Ein Prozeß $\rho 2$ führt eine DOWN-Operation durch, wird aber blockiert bis ein beliebiger Prozeß ρ die Semaphorenwert ϕ erhöht. Danach wird die DOWN-Operation wirksam, so daß als Resultat ϕ=0 bleibt!

Zustandsübergang: φ**+(Θ): S_UNLOCKED→ S_LOCKED**
Zustandsübergang: φ**+(ρ2): S_RUN→ S_BLOCKED**

θ=DOWN ▮ φ=S_LOCKED

Ein Prozeß $\rho 2$ führt eine DOWN-Operation durch, wird aber blockiert bis ein beliebiger Prozeß ρ die Semaphorenwert ϕ erhöht. Danach wird die DOWN-Operation wirksam, so daß als Resultat ϕ=0 bleibt! .

Zustandsübergang: φ**+(ρ2):** `S_RUN` $\rightarrow$ `S_BLOCKED` ▮ φ**=S_LOCKED**

θ=UP ▮ φ=S_UNLOCKED,$\phi \geq 0$

Der Prozeß ρ verändert den Wert der Semaphore.
Wertänderung: $\phi \leftarrow \phi$**+1**

θ=UP ▮ φ=S_LOCKED,ϕ=0

Der Prozeß $\rho 1$ verändert den Wert der Semaphore. Ein wartender Prozeß$\rho 2$ mit ausstehender DOWN-Operation wird freigegeben.

Keine Wertänderung: $\phi \leftarrow$**0**
Zustandsübergang: φ**+(Θ):** `#ρ(S_BLOCKED)=1? S_LOCKED` $\rightarrow$ `S_UNLOCKED`
Zustandsübergang: φ**+(ρ2):** `S_BLOCKED`$\rightarrow$ `S_RUN`

C. Event (TYPE Event)

- Mit einem Event wird der Kontrollfluß einer Gruppe aus mehreren konkurrierend laufende Prozessen synchronisiert.
- Es gibt zwei Operationen auf ein Event: $\{\theta\}$= **{AWAIT,WAKEUP}**

θ=AWAIT

Prozesse ρ1…ρN warten auf das Event. Die Prozesse werden bis zum Eintreten blockiert.
Zustandsübergang: φ**+(**ρ**1,**ρ**2,**…**):** **S_RUN**→ **S_BLOCKED**

θ=WAKEUP

Ein belibieger Prozeß ρ signalisiert das Event. Alle blockierten und wartenden Prozesse werden frei gegeben.
Zustandsübergang: φ**+(**ρ**1,**ρ**2,**…**):** **S_BLOCKED**→ **S_RUN**

D. Zeitzähler (TYPE Timer)

- Wie bei einem Event wird eine Gruppe aus mehreren konkurrierend laufende Prozessen synchronisiert, hier jedoch wird das Ereignis durch einen (periodischen) Timer im Zeitintervall $t=\tau$ ausgelöst.
- Es gibt vier Operationen auf ein Event: $\{\theta\}$**={AWAIT,START,STOP,SET}**

θ=AWAIT ▮ $t<\tau$

Prozesse $\rho 1 \ldots \rho N$ warten auf das Timer-Event. Die Prozesse werden bis zum Eintreten blockiert.

Zustandsübergang: $\varphi+(\rho 1,\rho 2,\ldots)$: **S_RUN→ S_BLOCKED**

θ=AWAIT ▮ $t=\tau$

Die wartenden Prozesse werden gleichzeitig freigegeben.

Zustandsübergang: $\varphi+(\rho 1,\rho 2,\ldots)$: **S_BLOCKED→ S_RUN**

```
t ← 0 ∧ start process timer again
```

θ=SET(t)

```
τ ← t
```

θ=START

```
t ← 0 ∧ start process timer: while t<τ do t←t+δ; wakeup
```

θ=STOP

```
t ← 0 ∧ stop process timer
```

E. Barriere (TYPE Barrier)

- Mit einer Barriere wird eine Gruppe aus N konkurrierend laufende Prozessen synchronisiert. Aber im Gegensatz zum Event erzeugt die Gruppe selbst das Ergeinis, auf das sie warten.
- Die Barriere ist ein Zähler mit dem Startwert 0. Jeder Prozeß der auf die Barriere wartet erhöht die Barriere um den Wert 1. Ist der Zähler ϕ=N, wird die Barriere wieder auf 0 gesetzt, und alle blockierten Prozesse werden freigegeben.
- Es gibt zwei Operationen auf ein Event: $\{\theta\}$**={INIT,AWAIT}**

θ=INIT(n)

$N \leftarrow n$
$\phi \leftarrow 0$

θ=AWAIT ▮ ϕ+1<N

Prozesse $\rho 1 \ldots \rho(N-1)$ warten auf das Event. Die Prozesse werden bis zum Eintreten der Bedingung ϕ=N blockiert.

Zustandsübergang: φ**+(ρ1,ρ2,…):** **S_RUN** → **S_BLOCKED**
$\phi \leftarrow \phi + 1$

θ=AWAIT ▮ ϕ+1=N

Die Prozesse werden freigegeben.
Zustandsübergang: φ**+(ρ1,ρ2,…):** **S_BLOCKED** → **S_RUN**
$\phi \leftarrow 0$

F. Queue (TYPE Queue) / Channel (TYPE Channel)

- Eine Queue stellt eine Möglichkeit der datenabhängigen Synchronisation und dem Datenaustausch von einer Gruppe von Prozessen dar.
- Eine Queue stellt einen Datenkontainer $\Re$ mit Daten eines bestimmten Datentyps dar, die in der Reihenfolge ausgelesen werden wie sie eingelesen wurden (FIFO).
- Eine Queue ist ein Zwischenspeicher, der bis zu ϕ<N Datenobjekte aufnehemen kann, und die drei Zustände Ψ={EMPTY,FILLED,FULL} besitzen kann:

Table 7. Zustände einer Queue

Zustand φ	Belegung ϕ
EMPTY	$\phi=0$
FILLED	$N>\phi>0$
FULL	$\phi=N$

- Es gibt zwei Operationen auf eine Queue: **{θ}={WRITE,READ}**

θ=READ ▮ φ=EMPTY

Ein Prozeß ρ1 führt eine READ-Operation durch, wird aber blockiert bis ein beliebiger Prozeß ρ2 ein Datenwort in die Queue schreibt.

Zustandsübergang: **φ+(ρ1):** **S_RUN→ S_BLOCKED**

θ=READ ▮ φ=FILLED

Ein Prozeß ρ1 führt eine READ-Operation durch, und das älteste Datenwort wird aus der Queue entfernt:

$\phi \leftarrow \phi - 1$

$\Re = [x1,x2,\ldots,xM] \rightarrow \Re = [x2,\ldots,xM] \rightarrow x1 \mid M < N$

Zustandsübergang: φ+(Θ): **S_FILLED** → **S_EMPTY** | $\phi = 0$

θ=READ ▮ φ=FULL

Ein Prozeß ρ1 führt eine READ-Operation durch, und das älteste Datenwort wird aus der Queue entfernt:

$\phi \leftarrow \phi - 1$

$\Re = [x1;x2;\ldots;xN] \rightarrow \Re = [x2;\ldots;xN] \rightarrow x1 \mid M = N$

Zustandsübergang: φ+(Θ): **S_FULL** → **S_FILLED** | **#ρ(S_BLOCKED)=0**

Wenn es blockierte schreibende Prozesse gibt (φ=FULL), dann wird ein neues Datenwort in die Queue geschrieben, und ein Prozeß ρ2 wieder freigegeben.

$\Re = [x1;x2;\ldots;xN-1;xN] \rightarrow \Re = [x2,\ldots,xN,xN+1] \rightarrow x1$

Zustandsübergang: φ+(ρ2): **S_BLOCKED** → **S_RUN**

θ=WRITE ▮ φ=FILLED

Ein Prozeß ρ1 führt eine WRITE-Operation durch.

$\phi \leftarrow \phi + 1$

$xM+1 \rightarrow \Re=[x1;x2;...,xM] \rightarrow \Re=[x1;x2;...;xM;xM+1] \mid M < N$

Zustandsübergang: φ+(Θ): $S_FILLED \rightarrow S_FULL \mid \phi = N$

θ=WRITE ▮ φ=FULL

Ein Prozeß ρ1 führt eine WRITE-Operation durch, wird aber blockiert solange blockiert, bis ein anderer Prozeß eine READ-Operation durchführt.

Zustandsübergang:σ+(ρ1): $S_RUN \rightarrow S_BLOCKED$

- Ein Kommunikations-Kanal (Channel) ist eine Queue mit N=1 und kann vom Typ buffered oder unbuffered sein.

Multiprozeß-Modell :: Hardware-Implementierung

Prozeß

- Ein Prozeß als Ausführung und Abbildung eines Programms kann auf zwei Arten mit Digitallogik implementiert werden:

(1) Mit einem programmgesteurten Prozessor (klassich von-Neumann oder Havard-Rechner-Architektur)

(2) Mit einem Zustandsautomaten FSM und reiner Register-Transfer-Logik RTL

- Der Hardware-Ressource-Aufwand hängt bei (1) im wesentlichen nur von der Art der Operationen (Datenwortbreite und Kosten einer Elementaroperation wie Addition) und der Menge verschiedener Operationen ab, nicht aber von der Komplexität $|\kappa|$ des Programms bzw. des Algorithmus:

```
process p:
begin
  1: x ← 1;
  2: a ← a + x;                          ⇒ µP
  3: while a > 0 do x ← x + 1;
end;
```

- Kosten: $\chi = \Sigma$ { (+), (←), (>), (ROM), (RAM), (CODEINTP) }

Prozeß

- Der Hardware-Ressource-Aufwand hängt bei (2) im wesentlichen von der Art der Operationen (Datenwortbreite und Kosten einer Elementaroperation wie Addition) und der Menge aller sequenziellen Operationen ab, und steigt mit der Größe und Komplexität θ des Programms bzw. des Algorithmus!

```
process p:
begin
  1: x ← 1;
  2: a ← a + x;                    ⇒  FSM-RTL
  3: while a > 0 do x ← x + 1;
end;
```

- Kosten χ = Σ {**2** (+) , **3** (←) , (>) , (while) , (ℜx) , (ℜa) , (FSM) , (MUX) , (DEMUX) }

(1)

Kosten $\chi(1) < \chi(2)$ wenn Komplexität $|\kappa|$ groß ist
Keine oder nur geringe/spezielle Datenpfadparallelität möglich

(2)

Kosten $\chi(2) < \chi(1)$ wenn Komplexität $|\kappa|$ klein bis mittel ist
Datenpfadparalleliät möglich

4. ConPro: Higher-Level-Synthese Werkzeug und Programmiersprache

Frage

Wie wird eine Programmiersprache auf einen Mikro-Chip abgebildet?

1. Multiprozeß-Modell
2. Interprozeß-Kommunikation
3. Multiprozeß-Modell: Hardware-Implementierung

ConPro: Higher-Level-Synthese

Imperativer Multi-Prozeß-Ansatz

- Abbildung: Imperative Programmier-/Programmebene → RTL
- Explizite Modellierung von Parallelität mit Multi-Prozeß-Architektur ↲
  Partitionierung des Algorithmus auf N Prozesse, die nebenläufig zueinander ausgeführt werden
- Beispiel zweier Prozesse, die ein globales Register `a` konkurrierend verändern und lesen. Der Zugriff auf das Register ist durch implizite Mutex (Guard) geschützt, aber explizite Mutex `lock_a` für Zugriff LHS ↔ RHS erforderlich!

```
process p1:                 process p2:
begin                       begin
  for i = 1 to 10 do        for i = 1 to 10 do
  begin                     begin
    lock_a.lock ();           lock_a.lock ();
    a ← a + i;               a ← a - i;
    lock_a.unlock ();         lock_a.unlock ();
end;                        end;
```

- Interprozeßkommunikation:
  - ➤ `Mutex, Semaphore, Barrier, Event, Timer, Queue,`
  - ➤ `Geschützte globale Register, Variablen (shared ressources)`

Imperativer Multi-Prozeß-Ansatz

- Parallelität auf Kontrollpfadebene durch Multi-Prozeß-Modell
- Parallelität auf Datenpfadebene durch **bounded/basic blocks (BB)**:

Example 11. Bounded Blocks

```
process p:
begin
  reg x,y,z: int[8];
  for i = 1 to 5 do
  begin
    x←x+i;        ⇔  x←x+i,y←y+i,z←z+i;
    y←z+i;
    z←z+i;
  end with bind;
```

Anweisungen (Ausdrücke und Wertzuweisungen ←), die keine Datenabhängigkeit zueinander besitzen, können im gleichen Zeitschritt ausgeführt werden.

➤ Aber: maximal nur ein geschützter Objektzugriff in einem BB.

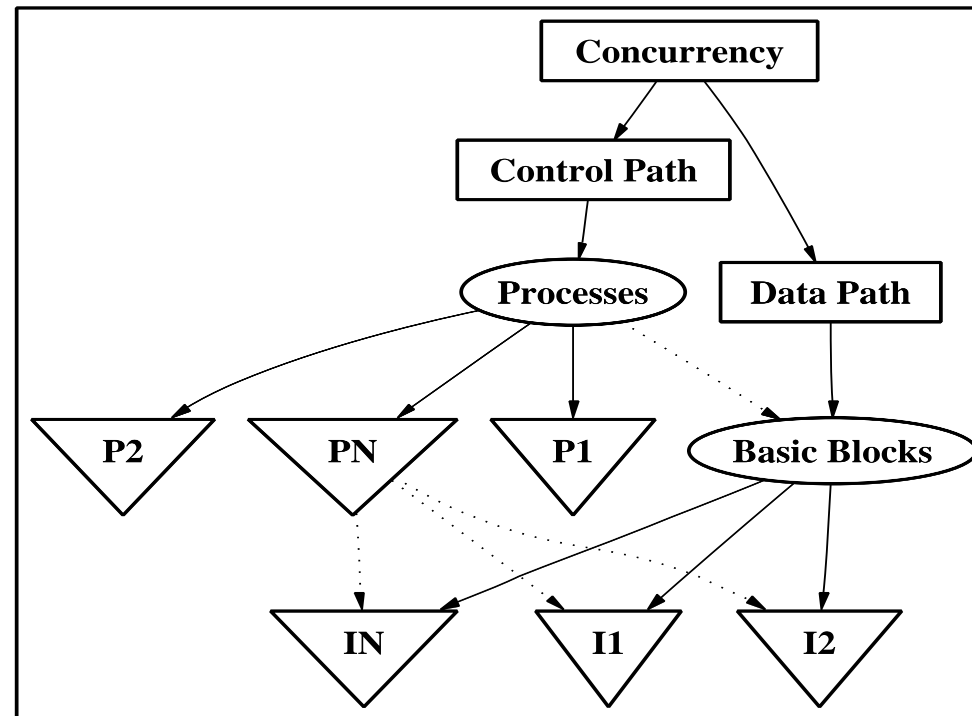
➤ Geschützter Objektzugriff beeinflußt Kontrollpfad (blockierende Operation)!

- BB-Parallelität kann auch automatisch mit Basisblock-Scheduler extrahiert werden.

Imperativer Multi-Prozeß-Ansatz

- Parallelität auf Kontrollpfadebene durch Multi-Prozeß-Modell
  ➤ grobe Granularität
- Parallelität auf Datenpfadebene durch **bounded/basic blocks (BB)**:
  ➤ feine Granularität

Figure 5. Ebenen der Nebenläufigkeit



ConPro :: Multiprozeß-Modell :: Hardware-Implementierung

Die Prozeß-Architektur

- Prozesse werden als reiner Zustandsautomat FSM und RTL implementiert
- Anweisungssequenzen $\Im=\{\kappa\}=\{\kappa1,\kappa2,\ldots\}$ werden als Folge von Zuständen $\psi=\{\varphi1,\varphi2,\ldots\}$ abgebildet
- Anweisungen gehören entweder zum Datenpfad Δ und/oder zum Kontroll/Steuerungspfad Γ

```
Γ: if-then-else,for do,while do,function call,
   guarded object access
Δ: assignment, expression
```

- Die Prozeß-Architektur besteht aus drei Hardware-Blöcken (die in VHDL mit jeweils einem VHDL-Prozeß abgebildet werden):

Steuerblock

Implementierung der Kontrollpfades als Zustandsautomat
FSM (moore), taktsynchron $\equiv \Gamma\text{:}\ \{\kappa\} \rightarrow \Sigma$
Abhängigkeit und Wirkung: $\downarrow$:$\varphi,\varpi,\omega,\Pi,\Theta$ $\uparrow$:$\varphi+$

Datenblock Kombinatorisch

Implementierung der Funktionsblöcke sowie Datenpfadselektoren als rein kombinatorische Logik und Beschaltung mit externen globalen Objekten $\equiv$ globale Ausdrücke
Abhängigkeit und Wirkung: $\downarrow$:$\varphi,\varpi,\omega,\Pi,\Theta$ $\uparrow$:ω,Θ

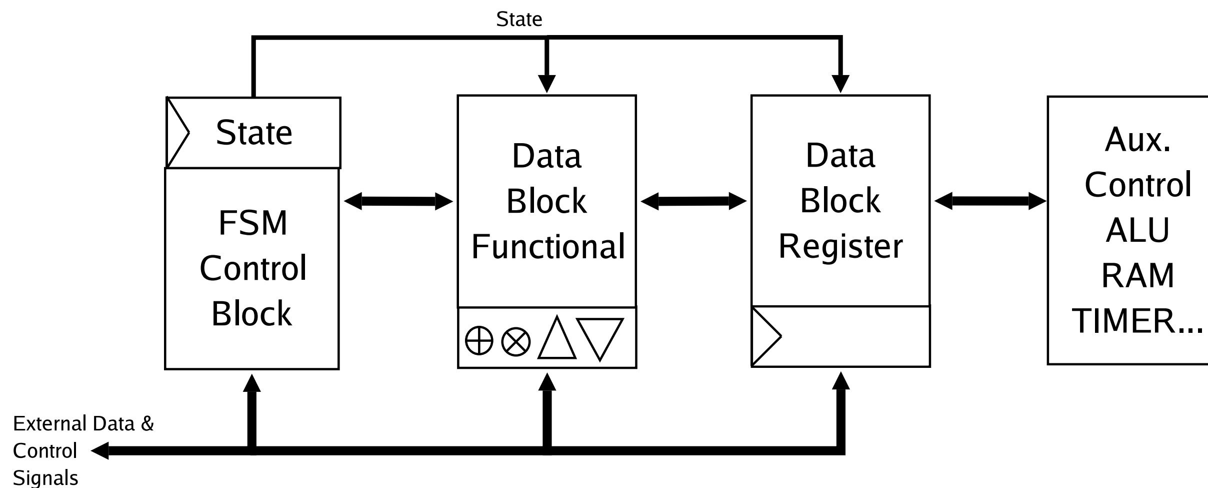
Die Prozeß-Architektur

Datenblock Transitorisch

Implementierung von lokalen Speicherobjekten wie Register $\Re$ und Funktionsblöcken ≡ loake Ausdrücke

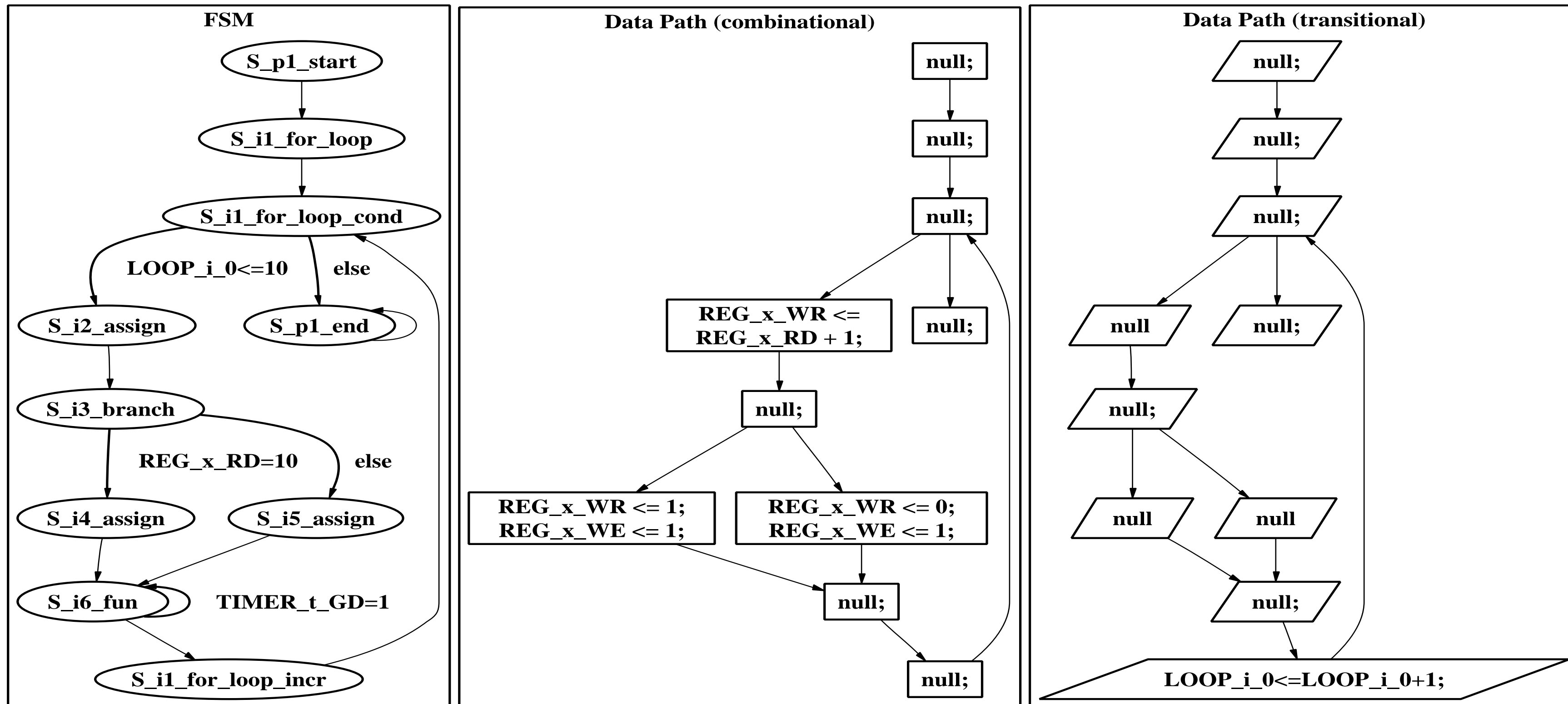
Abhängigkeiten und Wirkung: ↓:φ,ω,ϖ,Π ↑:ϖ

Figure 6. Systemarchitektur eines ConPro Prozesses



Die Prozeß-Architektur :: Beispiel

```
reg x: logic[8]; object t: timer;
process p1:
begin
  for i = 1 to 10 do begin
    x ← x + 1; if x = 10 then x ← 1 else x ← 0;
    t.await (); end;
end;
```



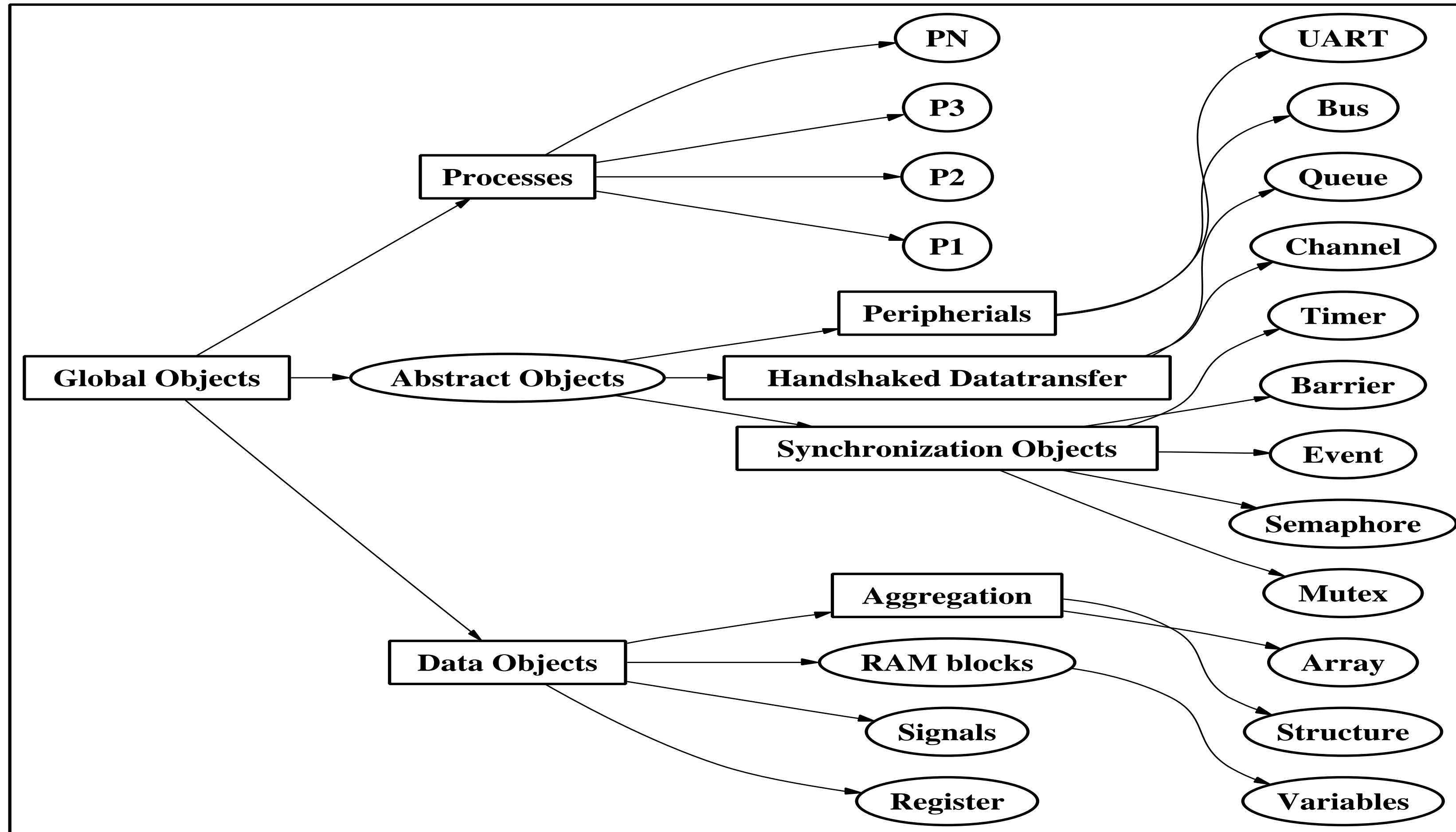
ConPro: Higher-Level-Synthese :: Objekte

Übersicht

Globale Modulebene

- Prozesse P
- Register $\Re$
- Variablen V
- RAM Blöcke $\wp$
- IPC Θ
- Kommunikationsschnittstellen Θ
- Signale (wire)

Figure 7. Globale Objekt und Typen Hierarchie

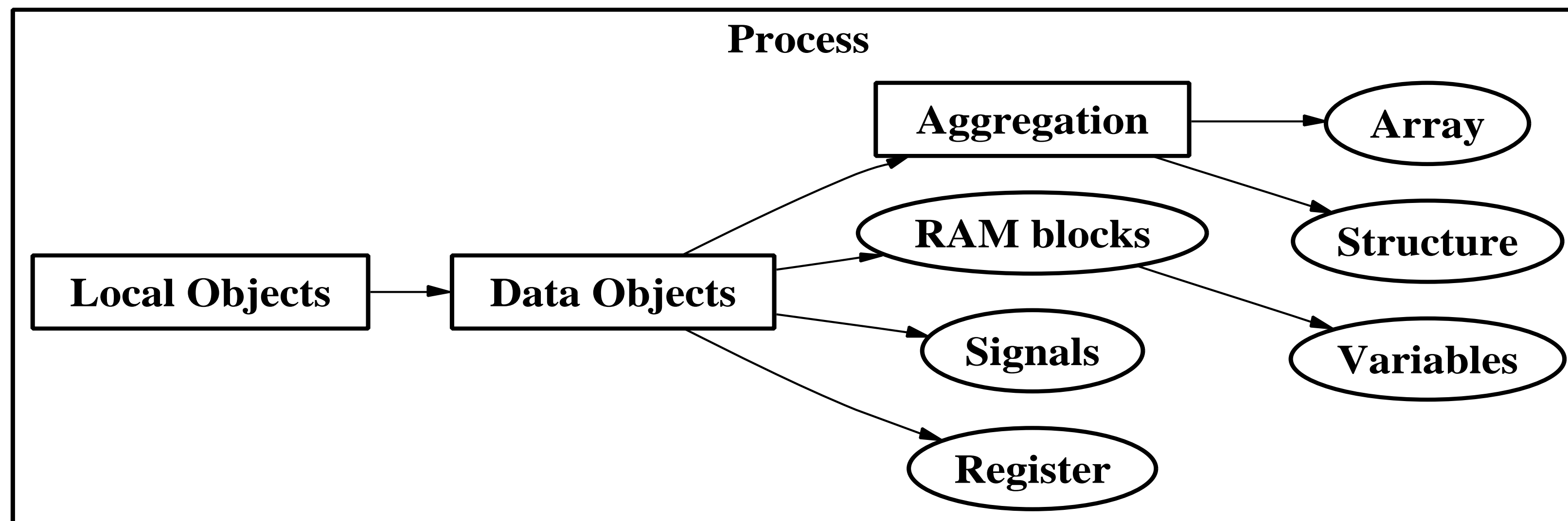


Übersicht

Lokale Prozeßebene

- Register $\Re$
- Variablen V
- RAM Blöcke $\wp$
- Kommunikationsschnittstellen Θ
- Signale (wire)

Figure 8. Lokale Objekt und Typen Hierarchie



Register

- Globale geteilte und geschützte Register mit CREW-Modell.
  - ➤ Der Lesezugriff kann von N Prozessen nebenläufig ohne Blockierung erfolgen
  - ➤ Der Schreibzugriff ist geschützt (Mutex mit Schedule) und wird sequenziell ausgeführt
- Prozeß-lokale Register
- Datenbreite: 1..64 Bit
- Datentypen: `TYPE DT IS logic,logic[N],int[N],bool,char`
- Verschiedene einzelne Register sind unabhängigen Hardware-Blöcken zugeordnet.

Variablen

- Globale geteilte und geschützte Variablen mit EREW-Modell.
  - ➤ Der Lese- und Schreibzugriff ist geschützt (Mutex mit Schedule) und wird sequenziell ausgeführt
- Prozeß-lokale Variablen
- Datenbreite: N=1..64 Bit
- Datentypen: `TYPE DT IS logic,logic[N],int[N],bool,char`
- Variablen werden in RAM-Blöcken implementiert, bis zu M (verschiedene: N/VT) Variablen

Variablen (Forts.)

- Automatische Datentypanpassung in Ausdrücken

```
TYPE RAM IS logic[N]
N=max(size(Vi) ∀ i=1...M)
```

- Beliebieg viele RAM-Blöcke möglich

Beispiele

```
reg xs: int[10];
array vs: reg[10] of int[31];
block ram with arch="single-port" and mode="read-first";
array vs2: var[10] of int[31] in ram;
type complex = {
  rx: int[20];
  ix: int[20];
};
reg c1,c2: complex;
process p:
begin
  reg x,y,z: int[8];
  array v: var[10] of int[12];
  for i = 0 to 9 do v.[i] <- 0;
  for j = 1 to 5 do
  begin
    x <- v.[j] + c1.rx;
    v.[j] <- vs.[j+1];
    vs.[j+1] <- x;
  end;
end;
```

ConPro: Higher-Level-Synthese :: Produktbildungstypen

Produktbildung :: einsortig

- Arrays
  - ➤ Objekttypen: `TYPE OT IS reg, var, sig, object`
  - ➤ Datentypen: `TYPE DT IS logic[n], int[n], bool, char, ADTO`
  - ➤ Selection: *aname*.[*index*] ∀ *index* =0...size-1

```
array aname: objtype[size] of datatype;
array a: reg[100] of logic[9];
array c: object[4] of semaphore;
```

Produktbildung :: mehrsortig

- Strukturen
  - ➤ Objekttypen: `TYPE OT IS reg, var, sig`
  - ➤ Datentypen: `TYPE DT IS logic[n], int[n], bool, char`
  - ➤ Selection: *sname.selem*

```
type sname : {
  el1: el1type;
  el2: el2type ... };    →  objtype soname: sname;
type hashentry : {
  key: int[8];
  data: char;
  next: int[8] };        →  array hash: reg[100] of hashentry;
```

ConPro: Higher-Level-Synthese :: Summenbildungstypen

Summenbildung

- Symbolische Aufzählung, Aufzählungstyp χ
  - ➤ Objekttypen: `TYPE OT IS reg,var`
  - ➤ Abbildung: f: $\chi \rightarrow$ integer

```
type ename : {
    el1;
    el2; ... };
type states : {
  S_start;
  S_scan;
  S_read;
  S_end;
};

reg state,next_state: states;
...
  match state with
  begin
    when S_start: state ← S_scan;
    when S_scan: state ← S_read;
    when S_read: state ← S_start;
  end;
```

ConPro: Higher-Level-Synthese :: Kontrollstrukturen

Bedingte Verzweigung

- Ausführung von Anweisungsblöcken $\Im_{TRUEB}$ und $\Im_{FALSEB}$ in Abhängigkeit der Evaluierung eines Booleschen Ausdrucks E

```
if E then B1;
if E then B1 else B0;
if a<b and a=0 then a←a+1;
```

Mehrfachauswahl

- Ausführung von Anweisungsblöcken $\Im_{iB}$ $\forall$ i=1…n in Abhängigkeit der Evaluierung eines Ausdrucks E (atomare und skalarer Elementardatentyp) und Vergleich mit konstanten Werten aus einer endlichen Menge V={v1,v2,…}.

```
match a-1 with
begin
  when 1: a←a+1;
  when 2,3,4: a←a-1;
  when others:
  begin
    a←0;
    b←1;
  end;
```

Bedingte Schleife

- Wiederholte Ausführung eines Anweisungsblockes $\Im_{LOOPB}$ in Abhängigkeit der Evaluierung eines Booleschen Ausdrucks E

```
while E do B;
i ← 1;
while i < 10 do i ← i + 1;
```

Zählschleife

- Wiederholte Ausführung eines Anweisungsblockes $\Im_{LOOPB}$ in Abhängigkeit der Evaluierung eines impliziten booleschen Ausdrucks E mit einem Indexregister i im Intervall `i = { a to | downto b}`

```
i ← 1; for j = 1 to 10 do i ← i + 1;
```

Funktionen

- Wiederholte Ausführung eines benannten Anweisungsblockes $\Im_{FUNB}$, entweder als Ressourcenduplikat (inline, Makro) oder als geteilter Funktionsblock.

```
function sq(x:int[8]) return (s:int[8]):
begin
  s ← x * x;
end;
  ... a ← sq(b);
```

ConPro: Higher-Level-Synthese :: Abstrakte Daten Objekte

Konzept

- Methodenbasierter Zugriff auf abstrakte Objekte (ADTO)
- Ein ADTO ist gekennzeichnet durch:

I. Ein abstrakter Datentyp Θ
II. Ein reaktives Verhaltensmodell
III. Eine Menge von Operation (Methoden) θ auf den ADTO Θ
IV. Eine Hardwarebeschreibung (External Module Interface: Interpretativer Subset VHDL)
V. Steuersignale (Request,Reply,Guard)

Arten

1. Interprozeßkommunikation ohne Datenaustausch:
   Mutex, Semaphore, Event, Barrier, Timer
2. Interprozeßkommunikation mit Datenaustausch:
   Queue, Channel, RAM, ROM
3. Prozesse!

Table 8. Verschiedene Abstrakte Datentyp Objekte und deren Methoden

ADTO	Methoden
process	{start,stop,call}
mutex	{lock,unlock}
semaphore	{init,down,up,level}
event	{await,wakeup}
timer	{set,start,stop,await}
queue	{read,write,empty,full}
uart	{read,write,baud,start,stop,interface}

Definition

```
open Omodule;                        ⇔     open Mutex;
object oname: otype;                       object mu1,mu2:mutex;
object oname: otype with params;           object mu1:mutex with
                                                  scheduler="fifo";
```

Methodenaufruf

```
oname.ometh(args);                   ⇔     mu1.lock();
                                           timer1.set(1 millisec);
```

ConPro: Higher-Level-Synthese :: Funktionen

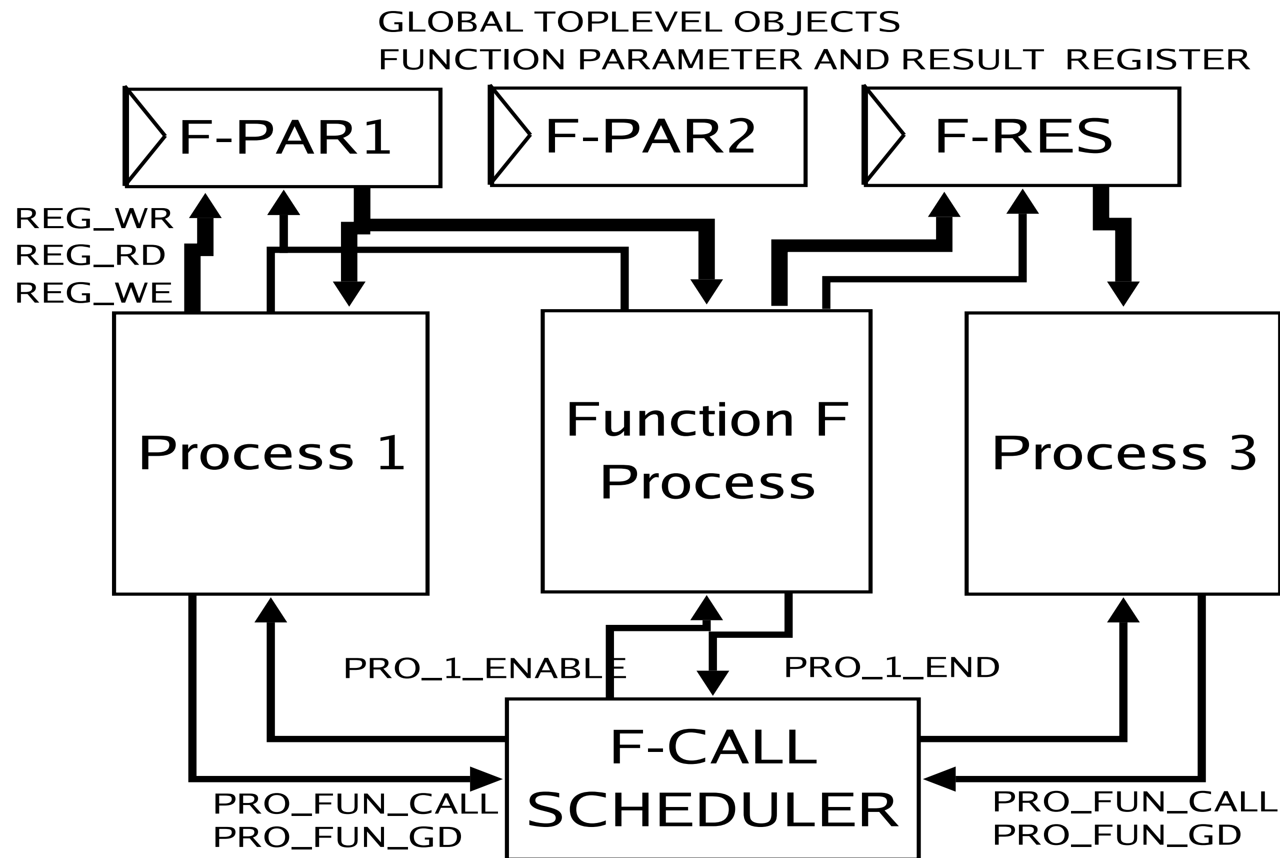
Konzept

- Nur nicht-rekursive Funktionen definierbar
- Implementierung von Funktionen mit Prozessen und einem Funktionsaufruf-Scheduler, der auch Funktionsargumente behandelt.
- Funktionsargumente werden über globale Register übergeben (nur atomare und skalare Elementardatentypen möglich)
- Rückgabewert ebenfalls mit Register umgesetzt

```
function div (a:logic[div_n],b:logic[div_n]) return(z:logic[div_n]):
begin
  reg q,b2: logic[div2_n];reg i: logic[5];const l0: logic[1] := 0;
  q ← a; b2 ← b lsl div_n; i ← 0;
  while i < div_n do
  begin
    begin
      q ← ((q lsl 1)-b2) lor 1; i ← i + 1;
    end with bind;
    if q[div2_n1] = 1 then
      q ← (q + b2) land 0xFFFFFFFE;
  end;
  z ← q[0 to div_n1];
end;
```

Implementierung

Figure 9. Funktionsblock und Prozeß-Interaktion über globale Parameter-Register und einem Funktionsaufruf-Scheduler



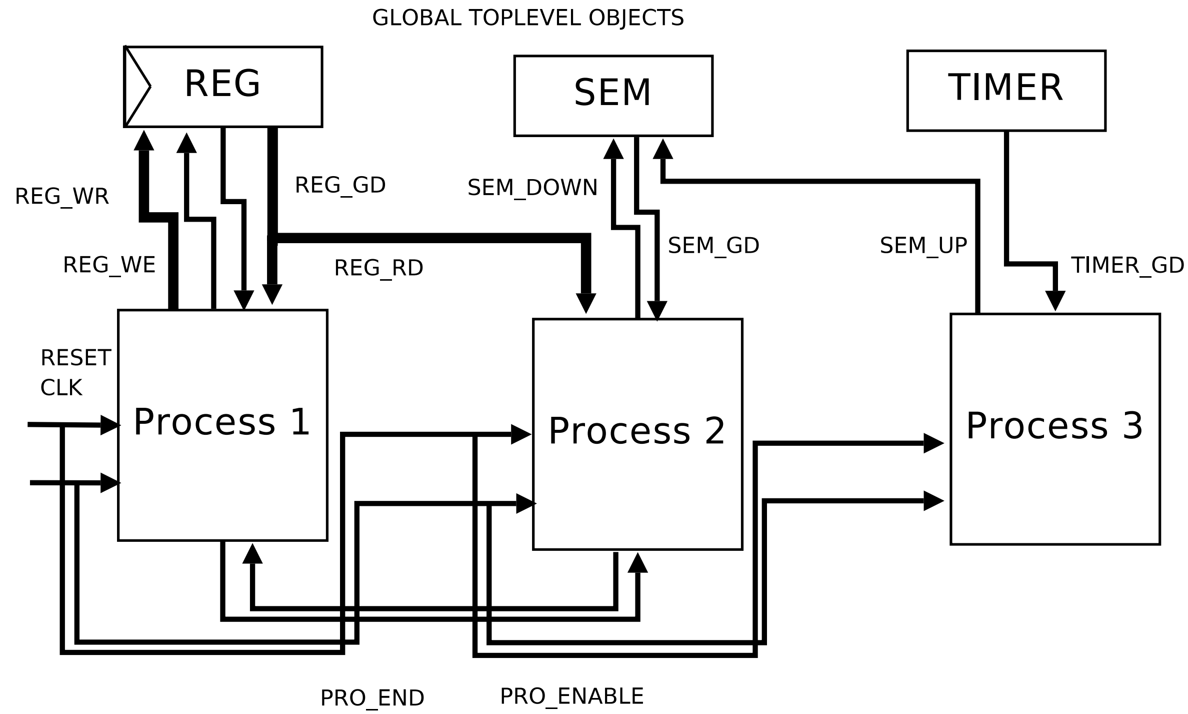
ConPro: Higher-Level-Synthese :: Interconnect-Architektur

Steuersignale

- Objekte werden über Steuersignale, Ausgang $\wp\uparrow$ und Eingang $\wp\downarrow$ aus der Sicht einese Prozesses, angesprochen
- Datenaustausch über Lese- und Schreibeports $\aleph\uparrow\downarrow$
- Beispiel Register:

$\Re$: `reg x: int[8];`
$\wp\uparrow$: `REG_x_WE:std_logic, REG_x_GD:std_logic`
$\wp\downarrow$: `REG_x_GD:std_logic`
$\aleph\downarrow$: `REG_x_RD:std_logic_vector(7 downto 0)`
$\aleph\uparrow$: `REG_x_WR:std_logic_vector(7 downto 0)`

Figure 10. Interconnect und Handshake-Architektur



ConPro: Higher-Level-Synthese :: Synthese

Konzept

- Regelbasierte Abbildung von (komplexen) Prozeß-Instruktionen $\Im$ auf Register-Transfer-Logik:

Equation 1. Regelbasierte Synthesefunktion

$$S:(\Im \rightarrow \Delta \times \Gamma)$$

$$\Delta = (\Sigma, \Pi, \Re)$$

$$\forall(\kappa \in \Im(P)) \Rightarrow (\tau \in \mathrm{T}):(\kappa \rightarrow (\pi, \sigma, r) \times \varphi)$$

mit: Δ: Datenpfad (Σ: Datenpfadselektoren, Π: Funktionsblöcke, $\Re$: Register) und Φ: Zustandsmenge des FSM, T: Abbildungsregelsatz

- Es stehen verschiedene (Mengen von) Abbildungsregeln $\{\tau\}$ zur Verfügung, die explizit vom Programmierer aktiviert werden müssen, und Scheduling und/oder Allokation beeinflussen.
- Ein Prozeß wird durch Objektdefinitionen und einem komplexen Syntax-Graphen G($\Im$) beschrieben.
- RTL wird nicht direkt aus dem Syntaxgraphen erzeugt
  ➤ Die komplexen Prozeß-Instruktionen des Syntaxgraphens G werden in lineare Liste von

einfachen Mikrokode-Instruktionen als Zwischenrepräsentation transformiert:

Equation 2. Regelbasierte Synthesefunktion: Zwischendarstellung (IR)

$$S:((\Im \rightarrow \Omega) \rightarrow \Delta \times \Gamma)$$

$$\forall(\kappa \in \Im(P)) \Rightarrow (\tau \in \mathrm{T}_{\Omega}):(\kappa \rightarrow \mu)$$

$$\forall(\mu \in \Omega) \Rightarrow (\tau \in \mathrm{T}):(\mu \rightarrow (\pi, \sigma, r) \times \varphi)$$

Scheduling & Optimierung

- Auf Syntaxgraph G-Ebene: **Referenzstack-Scheduler**
  ➤ Symbolische Syntaxgraph-Analyse bezüglich Fortpflanzung und Entwicklung von Wertzuweisungen und Wertaufruf von Datenobjekten (Register und Variable)
  ➤ Rückwärtssubstitution und Konstantenfaltung ermöglichen reduzierte Ausdrücke und Anweisungen
  ➤ Reduktion von N Datenpfadanweisungen mit J Registern $\Re$ und X Funktionsblöcken (Σ,Π) ursprünglich in $\tau 1$ Zeitschritten auf M<N Datenpfadanweisungen in $\tau 2<\tau 1$ Zeitschritte mit K<J Registern $\Re$ und Y<X Funktionsblöcken (Σ,Π)
- Auf Mikrokode μ-Ebene: **Basisblock-Scheduler**
  ➤ Zusammenfassungen von Datenpfad-Anweisungen in gebundene Blöcke, die in einem elementaren Zeitschritt ausgeführt werden
  ➤ Reduktion von N Datenpfadanweisungen ursprünglich in $\tau 1$ Zeitschritten auf $\tau 2<\tau 1$

Zeitschritte

- Auf RTL-Zustandsebene Γ: Komprimierung von Zuständen
  ➤ Zusammenfassung von Kontroll- und Datenpfadanweisungen
  ➤ Entfernung überflüssiger Zustände

Allokation

- Allokationsmodell wird durch parametrisierbares Ausdrucksmodell bestimmt

flach

direkte Abbildung von Funktionsblöcken auf Hardwareblöcke,
➤ keine Zerlegung komplexer Ausdrücke

binär

direkte Abbildung von Funktionsblöcken auf Hardwareblöcke
➤ Zerlegung von komplexen Ausdrücken in Teilausdrücke mit nur einem Operator und Inferenz von temporären Registern

shared/ALU

Abbildung von Funktionsblöcken auf geteilte Hardwareblöcke
➤ Zerlegung von komplexen Ausdrücken in Teilausdrücke mit nur einem Operator und Inferenz von temporären Registern

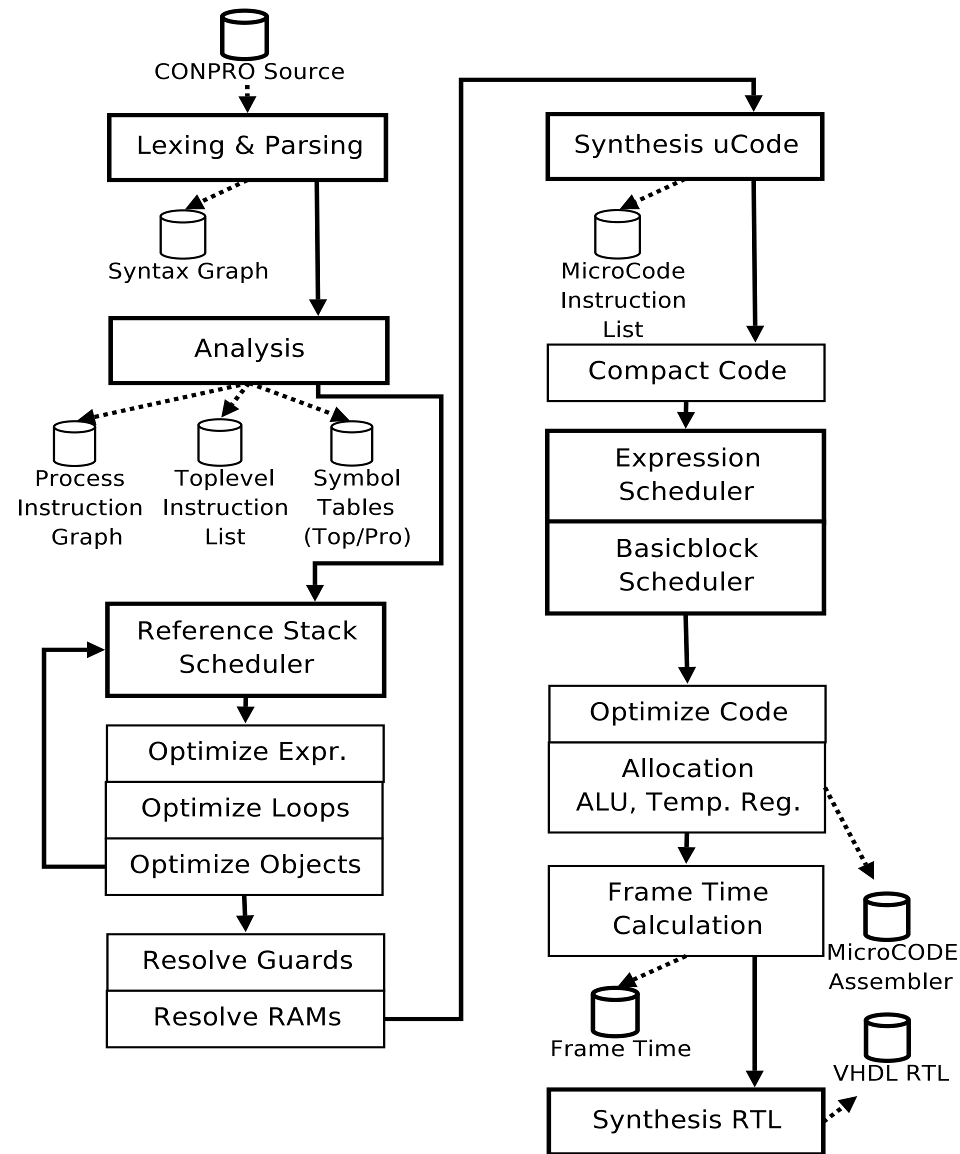
- Granularität der Parametrisierung: Block-Ebene
- **Automatische Zerlegung** von komplexen Ausdrücke mit**Ausdrucks-Scheduler**
  ➤ Randbedingung parametrisierbar: Zeit/Latenz τ für Einheitsoperationen
  ➤ Ausdrücke werden in binären Teilausdrücke zerlegt und wieder zusammengesetzt, solange zeitliche kummulative Randbedingung $\Sigma\tau i < \tau CLK$ eingehalten wird

ConPro: Higher-Level-Synthese :: Synthese (II)

Synthese-Pfad und Ablauf

1. Lexikalische und Syntaktische Analyse
2. Semantische Analyse
3. Referenz-Stack Scheduler
4. Optimierung von Ausdrücken, Schleifen und Objekten
5. Auflösung von Objektschutz und RAM Blöcken
6. Transformation und Kompaktierung μ-Code
7. Ausdrucks Scheduler
8. Basisblock Scheduler
9. Optimierung μ-Code
10. Allokation und Bindung von Ressourcen
11. Blockaufgelöste Berechnung Zeitschritte
12. RTL Zustands-Synthese

Figure 11. ConPro Synthese



ConPro: Higher-Level-Synthese :: μCode

Konzept

- Abbildung des Syntax-Programm-Graphens mit komplexen Anweisungen auf lineare Liste von einfachen Anweisungen als Zwischenrepräsentation: μCode
- Regelbasierte Abbildung stellt erstes Pre-Scheduling dar, und grenzt den Entwurfsraum der folgenden RTL ein.
- Aus dem μCode wird schließlich RTL abegeleitet.
- Optimierungen werden auf μCode-Ebene vollzogen.
- μCode

Table 9. μCode

Mnemonic	Beschreibung und Wirkung
move (dst,src)	Δ: Datentransfer dst←src, Γ: φ+1 // φ|GD=1
expr (dst,op1,op,op2)	Δ: Ausdruck E=op1 op op2 und Datentransfer dst←E, Γ: φ+1 // φ|GD=1
falsejump (expr,label)	Bedingte Verzweigung Γ: φ+1 // label(expr) if expr=true

Mnemonic	Beschreibung und Wirkung
jump (label)	Unbedingte Verzweigung Γ: label
bind (n)	Δ: Bindung von Anweisungen in einem Zeitschritt, Γ:φ+1 // φ|GD=1

Beispiel

```
reg x,y,z: int[8];
process p1:
begin
  reg a: int[8];
  for i = 1 to 10 do
  begin
    a ← i*2;
    x ← x + a;
  end;
end;
```

Transformation Conpro Programmmodell nach µCode

```
i1_for_loop:       move (LOOP_i_0,1) with ET=I[5]
i1_for_loop_cond: bind (2)
                   expr ($immed.[1],10,>=,LOOP_i_0) with ET=I[5]
                   falsejump ($immed.[1],i1_for_loop_end)
```

```
i2_assign:         expr (a,LOOP_i_0,*,2) with ET=I[8]
i3_assign:         expr (x,x,+,a) with ET=I[8]
i1_for_loop_incr: bind (3)
                   expr (LOOP_i_0,LOOP_i_0,+,1) with ET=I[5]
                   jump (i1_for_loop_cond)
i1_for_loop_end:
```

ConPro: Higher-Level-Synthese :: Referenz Stack

Konzept

- Wertzuweisungen von Objekten (Register & Variablen) werden mit einem Stack entlang des Kontrollpfades verfolgt, und Referenz von Objekten bestimmt.
- Für jedes Objekt $\mathfrak{R}$ existiert ein eigener Stack T($\mathfrak{R}$).
- Eine Folge von Zuweisungen an Speicherobjekte der Form $\mathfrak{R} \leftarrow \varepsilon_i$ für i=1…N werden als Definition & Wertzuweisung jeweils neuer konstanter Objekte $\mathfrak{R}_i$ aufgefaßt, so daß Speicherobjekte in unveränderliche symbolische Variablen transformiert werden:

```
reg x,y:int[8];
1: x←1;          ⇒ DEF x1=1
2: x←x+1;        ⇒ DEF x2=x1+1
3: y←x-1;        ⇒ DEF y1=x2-1
```

- Bei der Analyse des Kontrollpfades wird jeder neue Ausdruck ε_i von $\mathfrak{R}$ auf dem Stack T($\mathfrak{R}$)=[ε_i;…;ε_2;ε_1] abgelegt (Top ist erstes Element) .

```
reg x,y:int[8];  ⇒ T(x)=[?] T(y)=[?]
1: x←1;          ⇒ T(x)=[1;?]
2: x←x+1;        ⇒ T(x)=[T(x,2)+1;1;?]
3: y←x-1;        ⇒ T(y)=[T(x,3)-1;?]
```

Scheduling

- Wertzuweisungen an Speicherobjekte $\mathfrak{R}$ werden

I. bis zum Ende des aktuellen Rahmens,

II. bis zum Auftreten einer Verzweigung im Kontrollpfad (bedingte Verzweigung und Schleif-

en)

III. bis zum Auftreten eines Funktionsaufrufes verzögert.
➤ As-Last-As-Possible (ALAP) Scheduling

- Wird eine Zuweisung für ein Objekt $\Re$ ausgeführt (scheduled), wird der oberste Wert/Ausdruck vom Stack verwendet.
- Durch Rückwärtssubstitution von weiteren Refernzen T($\Re$,i) und Konstantenfaltung kann der Ausdruck vereinfacht werden:

```
T(x)=[T(x,2)+1;1;?] T(y)=[T(x,3)-1;?]
x←T(x,2)+1←1+1←2;
y←T(x,3)-1←T(x,2)+1-1←1+1-1←1;
```

- Bei bedingten Verzweigungen und Mehrfachauswahl kann weiterhin Invarianz von Objekten getestet werde. Für jeden Zweig des Kontrollpfades wird ein Sub-Stack angelegt. Enthaten diese nach der Evaluierung der Verzweigung alle den gliechen Ausdruck, kann eine Wertzuweisung entweder weiter verzögert werden oder vor die Verzweigung einmalig erfolgen.

Erweiterung

- Beispiel bedingte Verzweigung mit invarianter Wertzuweisung:

```
reg x,a:int[8];
x←1;
x←x+1;         ⇒ T(x)=[T(x,2)+1;1;?]
if a>0 then
```

```
begin
  ...
  x←x-1;      ⇒ T(x)=[⟨[T(x,3)-1]⟩;T(x,2)+1;1;?]
end
else
begin
  ...
  x←x-1;      ⇒ T(x)=[⟨[T(x,3)-1] | [T(x,3)-1]⟩;T(x,2)+1;1;?]
end;
```

```
if a >0 then ...
x←T(x,3)-1←T(x,2)+1-1←1+1-1←1;
```

ConPro: Higher-Level-Synthese :: Basisblock Scheduler

Konzept

- Der Daten- und Kontrollpfad ≡ Instruktionssequenz $\Im=\{\iota 1,\ldots\}$ wird in Basisblöcke zerlegt.
- Ein Basisblock ist gekennzeichnet durch

I. eine Menge von reinen Datenanweisungen und Ausdrücken,
II. nur einen einzigen Zugang im Kontrollpfad am Kopf,
III. und nur einen Ausgang im Kontrollpfad am Ende,
IV. sowie keine Seitensprünge innerhalb dieses Blockes.

- Die Basisblock wird Major-Block MB genannt, er ist entweder vom Typ DATA Δ (reine Datenanweisungen) oder CONTROL Γ:
- Der Major-Block wird weiter in Minor-Blöcke mb zerlegt. Ein Minor-Block besteht aus einer oder mehreren gebundenen Anweisungen, die innerhalb eines Zeitschrittes (oder atomaren Zeitraums) ausgeführt werden.

Datenabhängigkeitsgraphen

- Die Minor-Blöcke werden in Datenabhängigkeitsgraphen DDG strukturiert.
- Es entsteht eine Menge Λ von DDGs, die untereinander unabhängig sind.
- Beispiel:

```
1: move(x,0)                ⇒ MB1:DATA={1,2,3,4}
2: move(y,0)
3: expr(z,x-y)
4: move(x,1)
5: expr($immed.[0],z,=,0)   ⇒ MB2:CONT={5}
```

```
   falsejump ($immed.[0],8)
6: move(x,1)                ⇒ MB3:DATA={7}
7: jump (10)                ⇒ MB4:CONT={8}
8: move(x,0)                ⇒ MB5:DATA={9,10}
9: move(y,0)
10: ...
```

Datenabhängigkeitsgraphen

- Die Minor-Blöcke werden in Datenabhängigkeitsgraphen DDG strukturiert.
- Es entsteht eine Menge Λ von DDGs, die untereinander unabhängig sind.

Equation 3. Datenabhängiskeitsgraphen (DDG) strukturieren Minorblöcke

$$\mathrm{DDG}_i^{\Delta} \rightarrow \{\mathrm{mb}_a, \mathrm{mb}_b, \ldots, \mathrm{mb}_x\}$$

$$\mathrm{mb}_i = \{\kappa_1, \kappa_2, \ldots, \kappa_n\}$$

- Einfügen von mbs in DDG(s) durch Datenabhängigkeit.
- Partitionierung der MBs in mbs und DDGs liefert:

```
MB1={1,2,3,4} ⇒ mb1_1={1},mb1_2={2},...
MB2:CONT={5,6}
MB3:DATA={7}
MB4:CONT={8}
MB5:DATA={9,10}
```

```
DDG1_1={mb1_1={1},mb1_3={3},mb1_4={4}}≡DDG(x)
DDG1_2={mb1_2={2},mb1_3={3}}≡DDG(y)
```

Scheduling

- Scheduling:
  I. So schnell wie möglich: ASAP, und
  II. in jedem Zeitschritt werden innerhalb eines MB aus jedem DDG eine Anweisung entnommen und zu diesem Zeitschritt gebunden, sofern diese noch nicht ausgeführt wurde.
- Ergebnis:

```
1: move(x,0),move(y,0)
2: expr(z,x-y)
3: move(x,1)
5: expr($immed.[0],z,=,0)
   falsejump ($immed.[0],8)
6: move(x,1)
7: jump (9)
8: move(x,0),move(y,0)
9: ...
```

- Ziel: Reduktion der Zeitschritte, Verkleinerung des FSM (sowohl Zustandsregister als auch Schaltnetzwerk)

ConPro: Higher-Level-Synthese :: Beispiel

Dining Philosopher Problem

- Problemstellung:
  Five philosophers sit around a circular table. Each philosopher spends his life alternately thinking and eating. In the center of the table is a large platter of spaghetti. Each philosopher needs two forks two eat. But there are only five forks for all. One fork is placed between each pair of philosophers, and they agree that each will use only the forks to the immeadiate left and right.
  [Andrews 2000, Multihtreaded, Parallel, and Distributed Programming]
- Lösung:

I. Jeder Philosoph wird durch einen Prozeß `P` abgebildet
II. Geteilte Ressource: die 5 Gabeln
III. Jede Gabel wird durch eine Semaphore `S` abgebildet, Initialwert `S=1`
IV. Ein Philosoph befindet sich entweder im Zustand φ=`thinking` oder `eating`.
V. Der Zustandsübergang φ:`thinking`→`eating` des i-ten Philosophen erfordert gleichzeitig Ressourcenanforderung von Si und Si+1 (down).
VI. Nachdem ein Philosoph gegessen hat, geht er wieder in den Zustand `thinking` über: φ:`eating`→`thinking`, und gibt die Ressourcen Si und Si+1 (up) wieder frei.

- Higher-Level Modellierung und Implementierung: ConPro
- Prozeß-Array mit 5 Prozessen
- # ist Prozeß-Nummer-Index: {0..4}
- Die bedingten Verzweigung werden mittels Konstantenfaltung zur Synthese-Zeit aufgelöst
- Semaphore-Array:

Example 12. Conpro Definition des Prozeßarrays `philosopher`

```
array fork: object
  semaphore[5] with
  depth=8 and init=1 and
  scheduler="fifo";
```

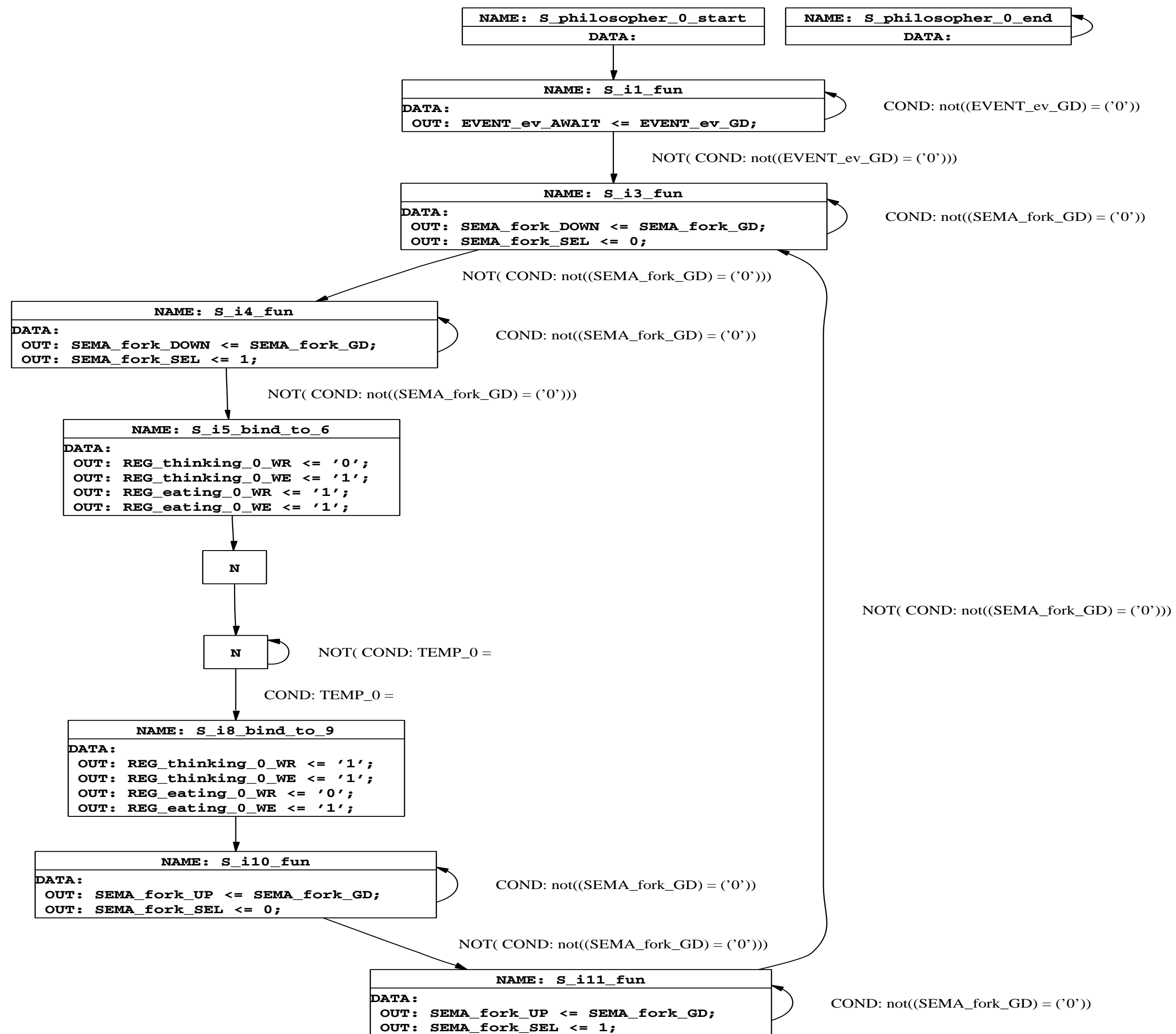
```
array philosopher: process[5] of
begin
  if # < 4 then begin
   ev.await ();
   always do begin
     -- get left fork then right
     fork.[#].down ();
     fork.[#+1].down ();
     eat (#);
     fork.[#].up ();
     fork.[#+1].up ();
   end;
  end
  else begin
   always do begin
     -- get right fork then left
     fork.[4].down ();
     fork.[0].down ();
     eat (#);
     fork.[4].up ();
     fork.[0].up ();
   end;
  end;
end;
```

- Higher-Level: µCode
- gekürzt
- nop-Instruktionen entstanden aus Optimierung

Example 13. Synthetisierter µCode für Prozeß `philiosopher`

```
i1_fun:   fun ev.await()
          jump (i3_fun)
i2_loop: nop
i3_fun:   fun fork.[0].down()
i4_fun:   fun fork.[1].down()
i5_bind_to_6: bind(2)
 move (eating_0,1) with ET=L[1]
 move (thinking_0,0) with ET=L[1]
i5_bind_to_6_end: nop
i7_waitfor:move ($tmp.[waitfor_count],3)
             with ET=L[4]
i7_waitfor_loop: bind(4)
   expr ($tmp.[waitfor_count],
         $tmp.[waitfor_count],-,1)
         with ET=L[4]
   expr ($immed.[1],
         $tmp.[waitfor_count],=,0)
         with ET=L[4]
   falsejump ($immed.[1],i7_waitfor_loop)
i7_waitfor_loop_end:jump (i8_bind_to_9)
i8_bind_to_9: bind (2)
    move (eating_0,0) with ET=L[1]
    move (thinking_0,1) with ET=L[1]
i10_fun: fun fork.[0].up()
         fun fork.[1].up()
         jump (i3_fun)
```

- High-Level: RTL
- Zustandsdiagramm vom Prozeß philosopher



- High-Level: VHDL
- Prozeß philosopher_0
- Port
- Architecture

Example 14. Synthetisiertes VHDL-Entity-Port für Prozeß `philosopher_0`

```
port(
  signal EVENT_ev_AWAIT: out std_logic;
  signal EVENT_ev_GD: in std_logic;
  signal REG_thinking_0_WR: out std_logic;
  signal REG_thinking_0_WE: out std_logic;
  signal SEMA_fork_DOWN: out std_logic;
  signal SEMA_fork_UP: out std_logic;
  signal SEMA_fork_GD: in std_logic;
  signal SEMA_fork_SEL: out integer;
  signal REG_eating_0_WR: out std_logic;
  signal REG_eating_0_WE: out std_logic;
  signal PRO_philosopher_0_ENABLE: in std_logic;
  signal PRO_philosopher_0_END: out std_logic;
  signal conpro_system_clk: in std_logic;
  signal conpro_system_reset: in std_logic
);
```

Example 15. Synthetisierte Architecture für ConPro Prozeß `philosopher_0`

```
architecture main of dining_philosopher_0 is
  -- Local and temporary data objects
  signal TEMP_0: std_logic_vector(3 downto 0);
  -- Auxilliary ALU signals
  -- State Processing
  type pro_states is (
    S_philosopher_0_start, -- PROCESS0[:0]
    S_i1_fun, -- FUN95127[dining.cp:53]
    S_i3_fun, -- FUN98283[dining.cp:57]
    S_i4_fun, -- FUN68501[dining.cp:58]
    S_i5_bind_to_6, -- ASSIGN46811[dining.cp:40]
    S_i7_waitfor, -- COND_LOOP7669[dining.cp:42]
    S_i7_waitfor_loop, -- COND_LOOP7669[dining.cp:42]
    S_i8_bind_to_9, -- ASSIGN15578[dining.cp:45]
    S_i10_fun, -- FUN91716[dining.cp:60]
    S_i11_fun, -- FUN60409[dining.cp:61]
    S_philosopher_0_end -- PROCESS0[:0]
    );
  signal pro_state: pro_states := S_philosopher_0_start;
  signal pro_state_next: pro_states := S_philosopher_0_start;
  -- Auxilliary toplevel definitions
begin
```

- High-Level: VHDL
- Prozeß philosopher_0
- Γ

Example 16. VHDL Prozeß Zustandsübergang für ConPro Prozeß `philosopher_0`

```
state_transition: process(
        PRO_philosopher_0_ENABLE,
        pro_state_next,
        conpro_system_clk,
        conpro_system_reset
)
begin
  if conpro_system_clk'event and conpro_system_clk='1' then
    if conpro_system_reset='1' or PRO_philosopher_0_ENABLE='0' then
      pro_state <= S_philosopher_0_start;
    else
      pro_state <= pro_state_next;
    end if;
  end if;
end process state_transition;
```

Example 17. VHDL Prozeß Zustandsübergangsnetzwerk für ConPro Prozeß `philosopher_0`

```
    control_path: process(
            EVENT_ev_GD,
            SEMA_fork_GD,
            TEMP_0,
            pro_state
            )
    begin
      PRO_philosopher_0_END <= '0';
      case pro_state is
        when S_philosopher_0_start => -- PROCESS0[:0]
          pro_state_next <= S_i1_fun;
        when S_i1_fun => -- FUN95127[dining.cp:53]
          if not((EVENT_ev_GD) = ('0')) then
            pro_state_next <= S_i1_fun;
          else
            pro_state_next <= S_i3_fun;
          end if;
        when S_i3_fun => -- FUN98283[dining.cp:57]
          if not((SEMA_fork_GD) = ('0')) then
            pro_state_next <= S_i3_fun;
          else
            pro_state_next <= S_i4_fun;
          end if;
        when S_i4_fun => -- FUN68501[dining.cp:58]
          if not((SEMA_fork_GD) = ('0')) then
```

```
          pro_state_next <= S_i4_fun;
        else
          pro_state_next <= S_i5_bind_to_6;
        end if;
...
      when S_philosopher_0_end => -- PROCESS0[:0]
        pro_state_next <= S_philosopher_0_end;
        PRO_philosopher_0_END <= '1';
    end case;
  end process control_path;
```

- High-Level: VHDL
- Prozeß philosopher_0
- Δ: kombinatorisch

Example 18. VHDL Prozeß Datenpfad, kombinatorisch, für ConPro Prozeß philosopher_0

```
data_path: process(
        pro_state
        )
begin
  -- Default values
  EVENT_ev_AWAIT <= '0';
  SEMA_fork_DOWN <= '0';
  SEMA_fork_SEL <= 0;
  REG_thinking_0_WR <= '0';
  REG_thinking_0_WE <= '0';
  REG_eating_0_WR <= '0';
  REG_eating_0_WE <= '0';
  SEMA_fork_UP <= '0';
  case pro_state is
    when S_philosopher_0_start => -- PROCESS0[:0]
      null;
    when S_i1_fun => -- FUN95127[dining.cp:53]
      EVENT_ev_AWAIT <= EVENT_ev_GD;
```

```
      when S_i3_fun => -- FUN98283[dining.cp:57]
        SEMA_fork_DOWN <= SEMA_fork_GD;
        SEMA_fork_SEL <= 0;
      when S_i4_fun => -- FUN68501[dining.cp:58]
        SEMA_fork_DOWN <= SEMA_fork_GD;
        SEMA_fork_SEL <= 1;
      when S_i5_bind_to_6 => -- ASSIGN46811[dining.cp:40]
        REG_thinking_0_WR <= '0';
        REG_thinking_0_WE <= '1';
        REG_eating_0_WR <= '1';
        REG_eating_0_WE <= '1';
      when S_i7_waitfor => -- COND_LOOP7669[dining.cp:42]
        null;
      when S_i7_waitfor_loop => -- COND_LOOP7669[dining.cp:42]
        null;
      when S_i8_bind_to_9 => -- ASSIGN15578[dining.cp:45]
        REG_thinking_0_WR <= '1';
        REG_thinking_0_WE <= '1';
        REG_eating_0_WR <= '0';
        REG_eating_0_WE <= '1';
      when S_i10_fun => -- FUN91716[dining.cp:60]
        SEMA_fork_UP <= SEMA_fork_GD;
        SEMA_fork_SEL <= 0;
      when S_i11_fun => -- FUN60409[dining.cp:61]
        SEMA_fork_UP <= SEMA_fork_GD;
        SEMA_fork_SEL <= 1;
```

```
        when S_philosopher_0_end => -- PROCESS0[:0]
          null;
      end case;
    end process data_path;
```

- High-Level: VHDL
- Prozeß philosopher_0
- Δ: transitorisch

Example 19. VHDL Prozeß Datenpfad, transitorisch, für ConPro Prozeß `philosopher_0`

```
-- Instruction Datapath Transitional Unit
data_trans: process(
        TEMP_0,
        conpro_system_clk,
        conpro_system_reset,
        pro_state
        )
begin
  if conpro_system_clk'event and conpro_system_clk='1' then
    if conpro_system_reset = '1' then
      TEMP_0 <= "0000";
    else
      case pro_state is
        when S_philosopher_0_start => -- PROCESS0[:0]
          null;
        when S_i1_fun => -- FUN95127[dining.cp:53]
          null;
        when S_i3_fun => -- FUN98283[dining.cp:57]
```

```
          null;
        when S_i4_fun => -- FUN68501[dining.cp:58]
          null;
        when S_i5_bind_to_6 => -- ASSIGN46811[dining.cp:40]
          null;
        when S_i7_waitfor => -- COND_LOOP7669[dining.cp:42]
          TEMP_0 <= "0011";
        when S_i7_waitfor_loop => -- COND_LOOP7669[dining.cp:42]
          TEMP_0 <= TEMP_0 - "0001";
        when S_i8_bind_to_9 => -- ASSIGN15578[dining.cp:45]
          null;
        when S_i10_fun => -- FUN91716[dining.cp:60]
          null;
        when S_i11_fun => -- FUN60409[dining.cp:61]
          null;
        when S_philosopher_0_end => -- PROCESS0[:0]
          null;
      end case;
    end if;
  end if;
end process data_trans;
```

- ConPro Synthese Ergebnis
- ConPro Level

Example 20. ConPro Ressourcen Bericht

```
+--------------------------- SYNTHESIS SUMMARY ---------------------------+
+-------------------------------------------------------------------------+
| DESCRIPTION                                VALUE                         |
+-------------------------------------------------------------------------+
| arithmetic unit                            99                            |
|  -> adder                                  (62)                          |
|  -> comparator                             (7)                           |
|  -> subtractor                             (30)                          |
| object                                     6                             |
|  -> event                                  (1)                           |
|  -> semaphore                              (5)                           |
| process                                    7                             |
| process.init                                                             |
|  -> port-width [bit]                       8                             |
|  -> register                               1                             |
|  -> states                                 7                             |
| process.main                                                             |
|  -> port-width [bit]                       10                            |
|  -> register                               1                             |
|  -> states                                 8                             |
```

```
| process.philosopher_0                                              |
|  -> port-width [bit]                     13                        |
|  -> register                             1                         |
|  -> states                               11                        |
| ...
| register-local                           7                         |
|  -> signed(3 downto 0)                   (2)                       |
|  -> std_logic_vector(3 downto 0)         (5)                       |
| register-shared                          10                        |
|  -> std_logic                            (10)                      |
| synthesis time [sec]                     4                         |
+--------------------------------------------------------------------+
```

- VHDL Gatelevel Synthese
- Zieltechnologie: Standardzellenbibliothek SXLIB
- Gatelevel Synthesewerkzeug: Leonardo Spectrum, Mentor Graphics

Example 21. Ressourcen-Bereicht Gatelevel-Synthese

```
 Cell                Library  References      Total Area

a2_x2               sxlib      3 x       2      5 gates
a3_x2               sxlib      2 x       2      4 gates
a4_x2               sxlib      3 x       2      7 gates
an12_x1             sxlib     71 x       2    121 gates
an12_x4             sxlib      1 x       3      3 gates
ao22_x2             sxlib     15 x       2     30 gates
inv_x1              sxlib    217 x       1    217 gates
inv_x2              sxlib      6 x       1      6 gates
inv_x4              sxlib      2 x       1      3 gates
na2_x1              sxlib     76 x       1     99 gates
na2_x4              sxlib      4 x       2      9 gates
na3_x1              sxlib     71 x       2    121 gates
na3_x4              sxlib      4 x       3     11 gates
na4_x1              sxlib     44 x       2     88 gates
na4_x4              sxlib      2 x       3      7 gates
nao22_x1            sxlib     26 x       2     52 gates
nao22_x4            sxlib      1 x       3      3 gates
```

```
nao2o22_x1      sxlib   12 x    2     28 gates
nmx2_x1         sxlib   57 x    2    131 gates
no2_x1          sxlib  141 x    1    183 gates
no3_x1          sxlib   61 x    2    104 gates
no3_x4          sxlib    1 x    3      3 gates
no4_x1          sxlib   44 x    2     88 gates
no4_x4          sxlib    1 x    3      3 gates
noa22_x1        sxlib   57 x    2    114 gates
noa22_x4        sxlib    3 x    3     10 gates
noa2a22_x1      sxlib   18 x    2     41 gates
noa2a2a23_x1    sxlib    1 x    3      3 gates
noa2ao222_x1    sxlib   16 x    2     37 gates
noa3ao322_x1    sxlib    1 x    3      3 gates
noa3ao322_x4    sxlib    1 x    4      4 gates
nxr2_x1         sxlib  114 x    3    342 gates
o2_x2           sxlib   10 x    2     17 gates
o3_x2           sxlib    4 x    2      8 gates
o4_x2           sxlib    3 x    2      7 gates
oa22_x2         sxlib   22 x    2     44 gates
oa3ao322_x2     sxlib    2 x    4      7 gates
on12_x1         sxlib   62 x    2    105 gates
on12_x4         sxlib    2 x    3      5 gates
one_x0          sxlib    1 x    1      1 gates
sff1_x4         sxlib   74 x    6    444 gates
sff2_x4         sxlib  161 x    8   1288 gates
xr2_x1          sxlib   34 x    3    102 gates
```

```
zero_x0          sxlib     1 x      1      1 gates

 Number of ports :                      12
 Number of nets :                     1454
 Number of instances :                1452
 Number of references to this view :     0

Total accumulated area :
 Number of gates :                    3909
 Number of accumulated instances :    1452
Using default wire table: small



                        Clock Frequency Report
      ------------------------------------

        CLK                  : 57.8 MHz
```

- Simulation: gate-level
- Zieltechnologie: Standardzellenbibliothek SXLIB
- Gatelevel-Simulator: Alliance, Asimut
- Stimuli Patterngenerator: Alliance, lib genpat, C

Example 22. C Source-Code für Stimuli Patterngenerator

```
#include <stdio.h>
#include <genpat.h>
#define PERIOD       100
#define RES          5
#define CLKDIV       (PERIOD/(2*RES))
#define NS(clk,n)    ((clk*2)*PERIOD*500+(n*PERIOD*500)/CLKDIV)
#define NS2(clk,n)   ((clk*2+1)*PERIOD*500+(n*PERIOD*500)/CLKDIV)
#define N            500
char *i2s(int v)
{
  char *str;
  str=(char *) mbkalloc(32);
  sprintf(str,"%d",v);
  return str;
};
int main(int argc,char **argv)
{
  int clk;
```

```
int i,j;
DEF_GENPAT("dining");
DECLAR("thinking_0_rd",":2","B",OUT,"","");
DECLAR("thinking_1_rd",":2","B",OUT,"","");
DECLAR("thinking_2_rd",":2","B",OUT,"","");
DECLAR("thinking_3_rd",":2","B",OUT,"","");
DECLAR("thinking_4_rd",":2","B",OUT,"","");
DECLAR("eating_0_rd",":2","B",OUT,"","");
DECLAR("eating_1_rd",":2","B",OUT,"","");
DECLAR("eating_2_rd",":2","B",OUT,"","");
DECLAR("eating_3_rd",":2","B",OUT,"","");
DECLAR("eating_4_rd",":2","B",OUT,"","");
DECLAR("clk",":2","B",IN,"","");
DECLAR("reset",":2","B",IN,"","");
for(clk=0;clk<N*2;clk++){
  for(j=0;j<...
```

- Gate-level Simulation
- Zieltechnologie: Standardzellenbibliothek SXLIB
- Simulator: Alliance, Asimut
- Pattern-Datei

Example 23. Vom Patterngenerator (C-Code) erzeugte Stimuli-Muster

```
-- Pattern description :
--                   t   t   t   t   t   e   e   e   e   e  c  r
--                   h   h   h   h   h   a   a   a   a   a  l  e
--                   i   i   i   i   i   t   t   t   t   t  k  s
--                   n   n   n   n   n   i   i   i   i   i     e
--                   k   k   k   k   k   n   n   n   n   n     t
--                   i   i   i   i   i   g   g   g   g   g
--                   n   n   n   n   n   _   _   _   _   _
--                   g   g   g   g   g   0   1   2   3   4
--                   _   _   _   _   _   _   _   _   _   _
--                   0   1   2   3   4   r   r   r   r   r
--
--                   _   _   _   _   _   d   d   d   d   d
--                   r   r   r   r   r
--                   d   d   d   d   d
<          0 ps>              : ?u  ?u  ?u  ?u  ?u  ?u  ?u  ?u  ?u  ?u  1  1  ;
<       5000 ps>              : ?u  ?u  ?u  ?u  ?u  ?u  ?u  ?u  ?u  ?u  1  1  ;
<      10000 ps>              : ?u  ?u  ?u  ?u  ?u  ?u  ?u  ?u  ?u  ?u  1  1  ;
```

```
<      15000 ps>                 : ?u  ?u  ?u  ?u  ?u  ?u  ?u  ?u  ?u  ?u  1  1   ;
...
<    7135000 ps>                 : ?0  ?0  ?0  ?0  ?1  ?0  ?0  ?0  ?1  ?0  1  0   ;
<    7140000 ps>                 : ?0  ?0  ?0  ?0  ?1  ?0  ?0  ?0  ?1  ?0  1  0   ;
<    7145000 ps>                 : ?0  ?0  ?0  ?0  ?1  ?0  ?0  ?0  ?1  ?0  1  0   ;
<    7150000 ps>                 : ?0  ?0  ?0  ?0  ?1  ?0  ?0  ?0  ?1  ?0  0  0   ;
<    7155000 ps>                 : ?0  ?0  ?0  ?0  ?1  ?0  ?0  ?0  ?1  ?0  0  0   ;
<    7160000 ps>                 : ?0  ?0  ?0  ?0  ?1  ?0  ?0  ?0  ?1  ?0  0  0   ;
<    7165000 ps>                 : ?0  ?0  ?0  ?0  ?1  ?0  ?0  ?0  ?1  ?0  0  0   ;
<    7170000 ps>                 : ?0  ?0  ?0  ?0  ?1  ?0  ?0  ?0  ?1  ?0  0  0   ;
...
```

- Gate-level Simulation
- Pattern-Viewer: Xpat