
**High-Level-RTL-Synthese mit ei-
nem
Multi-Prozess-Modell**

Von der imperativen algorithmischen Ebene zu RTL

Dr. Stefan Bosse

-

27.11.2009

Einführung

1. Überblick von Syntheseverfahren für Register-Transfer-Logik
2. Das Multi-Prozeß-Modell als Programmiermodell
3. Interprozeßkommunikation
4. Das Multi-Prozeß-Modell als Hardwaremodell

Synthese von Digitallogikschaltungen

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

Register-Transfer-Logik Ebene

Beschreibung der Digitallogikschaltungen mit getrennten Daten- und Kontrollpfad (Δ und Γ).

Der Datenpfad besteht aus Funktionsblöcken Π , Datenpfadselektoren Σ und Registern \mathfrak{R} .

Der Kontrollpfad besteht aus einem endlichen Zustandsautomaten FSM Φ , der den Datenpfad spatial und zeitlich steuert.

Algorithmischen Ebene

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, \mathcal{R}) 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 bedeutet 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 ➤ Diskretisierung des Kontrollpfades Γ ➤ **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 (Resource-Sharing).

- ↳ Erzeugung einer RTL bedeutet das Abbilden von Daten- und Kontrollfluß in zwei Dimensionen: Zeit \equiv FSM und Fläche \equiv Hardware \equiv 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 (I)

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.
- 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. prozessorientiert eindeutig bestimmt bezüglich Scheduling und Allokation.
- Keine inherente Parallelität!
- Explizite Parallelisierung z.B. mit dem Programmiermodell Multi-Threading möglich
- Implizite Parallelisierung nur durch Synthese-Werkzeug oder durch Erweiterung des Programmiermodells (Multi-Prozeß-System) möglich.
- Extrahierte Parallelität aus Schleifen (➔ Abrollen) und Basisblöcken (➔ Ausführung mehrerer datenunabhängiger Anweisungen in einem Zeitschritt)

Algorithmische Ebene und Synthese :: Imperativ (II)

Beispiele:

Basisblock Parallelität

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

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

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

Algorithmische Ebene und Synthese :: Funktional (I)

(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.

Algorithmische Ebene und Synthese :: Funktional (II)

Beispiele:

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

[Funktional \Rightarrow Imperativ-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;
```

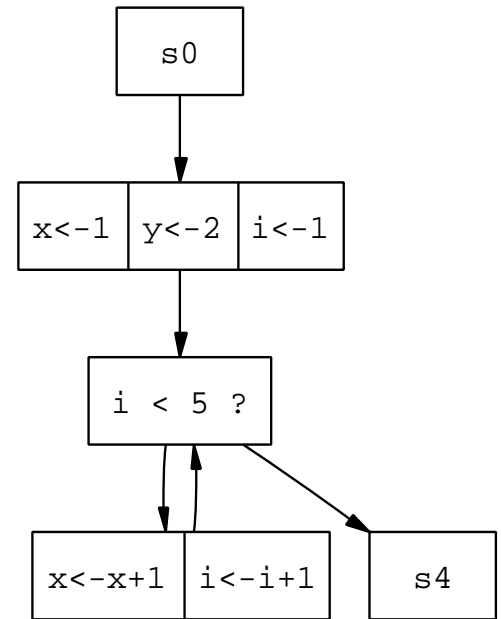
Higher-Level-Synthese (I)

1. Die Beschreibung eines Digitallogik-Schaltkreises mit der abstrahierten algorithmischen Ebene erfordert regelbasierte Abbildung von sequenziellen Anweisungsblöcken auf RTL

- ▼ **Higher-Level-Synthese** \equiv Programmiererebene
 - RTL-Ebene
- ▼ RTL-Ebene \equiv Daten- und Kontrollfluß
- ▼ Daten- und Kontrollfluß \equiv 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

Beispiel einer RTL-Beschreibung:
Nomenklatur: next-state ::= ';' .
label ::= name ':' .
concurrent ::= ',' .
x←-1,y←-2,i←-1;
loop: if i < 5 then
 {x←-x+1,i←-i+1;jump loop};



Higher-Level-Synthese (II)

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

```
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 case;
  end if;
end;
```

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

Higher-Level-Synthese (III)

- 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

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

Einfachstes Scheduling- und Allokationsmodell: Grundregeln

- ▼ **Regel I:** Es gibt nur Register als Datenspeicher mit beliebiger Datenwortbreite.

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

- ▼ **Regel II:** Kein Ressource-Sharing: jedes Register was definiert wurde wird auch als Hardware-Block $\omega^* : \text{REG}(\downarrow\text{WR}, \downarrow\text{WE}, \uparrow\text{RD})$ implementiert.

```
 $\kappa : \text{reg } x,y,z \Rightarrow \omega : \text{REG} \rightarrow \text{REG}_x(\downarrow\text{REG}_x\text{WR}, \downarrow\text{REG}_x\text{WE}, \uparrow\text{REG}_x\text{RD}), \dots$ 
```

- ▼ **Regel III:** Jede elementare Anweisung κ wird auf einen Zustand σ des Kontrollflusses Σ /des RTL-FSM abgebildet:

```
 $\kappa : x \leftarrow 1; \Rightarrow \sigma : S\_assign\_1$   
 $\kappa : y \leftarrow 2; \Rightarrow \sigma : S\_assign\_2$   
 $\kappa : i \leftarrow 1; \Rightarrow \sigma : S\_assign\_3$   
 $\kappa : \text{while } i < 5 \text{ do} \Rightarrow \sigma : S\_loop\_1$   
     $\kappa : i \leftarrow i+1 \Rightarrow \sigma : S\_assign\_4$ 
```

- ▼ **Regel IV:** Der Datenpfad Π wird in Abhängigkeit der Zustände Σ zerlegt:

```
 $\kappa \Leftrightarrow \pi \quad S\_assign\_1: x \leftarrow 1; S\_assign2: y \leftarrow 1; \dots$ 
```

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

Einfachstes Scheduling- und Allokationsmodell: Grundregeln

- ▼ **Regel V:** Kein Ressource-Sharing: jeder funktionale Operator (arithmetisch, relational und boolesch) wird mit eigenen Hardware-Blöcken ω implementiert:

$$\begin{aligned}\omega^*: & \text{ADD}(\uparrow\text{DST}, \uparrow\text{DST_EN}, \downarrow\text{SRC1}, \downarrow\text{SRC2}) \\ \kappa: & i \leftarrow i+1; \Rightarrow \omega: \text{ADD} \rightarrow \text{ADD1}, \pi: \text{ADD1}(\uparrow\text{REG_i_WR}, \uparrow\text{REG_i_WE}, \downarrow\text{REG_i_RD}, \downarrow 1) \\ \kappa: & x \leftarrow x+1; \Rightarrow \omega: \text{ADD} \rightarrow \text{ADD2}, \pi: \text{ADD2}(\uparrow\text{REG_x_WR}, \uparrow\text{REG_x_WE}, \downarrow\text{REG_x_RD}, \downarrow 1)\end{aligned}$$

- ▼ **Regel VI:** Komplexe Anweisungen **K** (z.B. Schleifen) müssen in eine Menge $\{\kappa\}$ von elementaren Anweisungen mit Transformationsregeln zerlegt werden:

$$\begin{aligned}\mathbf{K}: & \text{for } i = 1 \text{ to } 10 \text{ do} \\ \kappa: & \quad x \leftarrow x+1; \\ [\mathbf{K}, \kappa \Rightarrow \kappa', \sigma] \\ \kappa: & i \leftarrow 1 \quad \Rightarrow \sigma: \text{S_assign_1}, \sigma+: \text{S_loop_cond} \\ \kappa: & i=1 \dots 10? \Rightarrow \sigma: \text{S_loop_cond} \\ & \quad \sigma+: \text{if } i=10 \text{ then } \text{S_loop_end} \text{ else } \text{S_assign_2} \\ \kappa: & x \leftarrow x+1; \quad \Rightarrow \sigma: \text{S_assign_2}, \sigma+: \text{S_assign_3} \\ \kappa: & i \leftarrow i+1; \quad \Rightarrow \sigma: \text{S_assign_3}, \sigma+: \text{S_loop_cond}\end{aligned}$$

- ▼ **Regel VII:** Zu jedem Zustand σ gehört ein Folgezustand $\sigma+$, der bedingt oder unbedingt definiert sein kann.

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

Verbessertes Scheduling- und Allokationsmodell: Optimierungsregeln

- ▼ **Regel OI:** Analyse des Daten- und Kontrollpfades bezüglich **Basisblöcken** mittels **Daten-Abhängigkeitsgraphen (DDG)**. Ein Basisblock β ist ein zusammenhängender Anweisungsblock mit nur einem Kontrollpfad-Zugang am Kopf und ohne Seiteneffekte.
➔ Scheduling: Reduktion der Zeitschritte, keinen Einfluß auf Allokation

$$\begin{aligned}\kappa 1: & x \leftarrow 1; \\ \kappa 2: & y \leftarrow 1; \\ \kappa 3: & z \leftarrow x+y; \\ \kappa 4: & \text{if } \dots\end{aligned}$$
$$\begin{aligned}[\kappa \Rightarrow \kappa'] \\ \beta: & \{\kappa 1, \kappa 2, \kappa 3\}\end{aligned}$$
$$\begin{aligned}[\beta \Rightarrow \text{DDG}] \\ \text{DDG1}: & \{\kappa 1 \rightarrow \kappa 3\} \\ \text{DDG2}: & \{\kappa 2 \rightarrow \kappa 3\}\end{aligned}$$
$$\begin{aligned}[\beta \Rightarrow \text{DDG} \Rightarrow \kappa'] \text{ mit ASAP-Scheduler} \\ \kappa 1' = \{\kappa 1, \kappa 2\}: & x \leftarrow 1, y \leftarrow 1; \\ \kappa 2' = \{\kappa 3\}: & z \leftarrow x+y;\end{aligned}$$

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

Verbessertes Scheduling- und Allokationsmodell: Optimierungsregeln

- ▼ **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

$\kappa 1: x \leftarrow 1;$
 $\kappa 2: a \leftarrow x + 1;$
 $\kappa 3: y \leftarrow a * 2;$
 $\kappa 4: b \leftarrow y + a;$

$[\omega \Rightarrow \omega']$
 $\beta: \{\kappa 1, \kappa 2, \kappa 3, \kappa 4\}$ mit $\omega: \{\omega 1 = x, \omega 2 = y, \omega 3 = a, \omega 4 = b\}$

$[\beta \Rightarrow \text{DPG}]$
 $\text{DPG}(x): \{\kappa 1 \rightarrow \kappa 2\}, \text{DPG}(y): \{\kappa 3 \rightarrow \kappa 4\}$
 $\text{DPG}(a): \{\kappa 2 \rightarrow \kappa 3 \rightarrow \kappa 4\}, \text{DPG}(b): \{\kappa 4\}$

$[\beta \Rightarrow \text{DPG} \Rightarrow \omega']$
 $\omega 1': t1 \equiv \{x, y\}, \omega 2' = a, \omega 3' = b$
 $t1 \leftarrow 1; a \leftarrow t1 + 1; t1 \leftarrow a * 2; b \leftarrow t1 + a;$

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

Verbessertes Scheduling- und Allokationsmodell: Optimierungsregeln

- ▼ **Regel OIII:** Ressourcen- und Zeitschrittreduktion: Symbolische Analyse mit **Referenzstackmethode** T der sequenziellen Entwicklung von Ausdrücken und Wertfortpflanzung
► Registeroptimierung
↳ Scheduling: Reduktion von Zeitschritten, Allokation: Reduktion von Ressourcen

$\kappa 1: x \leftarrow 1;$
 $\kappa 2: a \leftarrow x + 1;$
 $\kappa 3: a \leftarrow a * 2;$
 $\kappa 4: b \leftarrow x + a; \dots x \text{ wird nicht weiter verwendet}$

$[\kappa, \omega \Rightarrow \omega']$
 $\beta: \{\kappa 1, \kappa 2, \kappa 3, \kappa 4\}$ mit $\omega: \{\omega 1 = x, \omega 2 = a, \omega 3 = b\}$

$[\beta \Rightarrow \text{T}]$ mit $v: \text{Wert}, \varepsilon: \text{Ausdruck und Rückwärts substituierung}$
 $T(x): \{v1\} \Rightarrow x \leftarrow T(x, 1) \equiv v1;$
 $T(a): \{\varepsilon(T(x, 1), v2), \varepsilon(T(a, 1), v3)\} \Rightarrow a \leftarrow T(a, 2) \equiv \varepsilon(v1, v2) = v4$
 $T(b): \{\varepsilon(T(x, 1), T(a, 2))\} \Rightarrow b \leftarrow T(b, 1) \equiv \varepsilon(v1, v4) = v5$

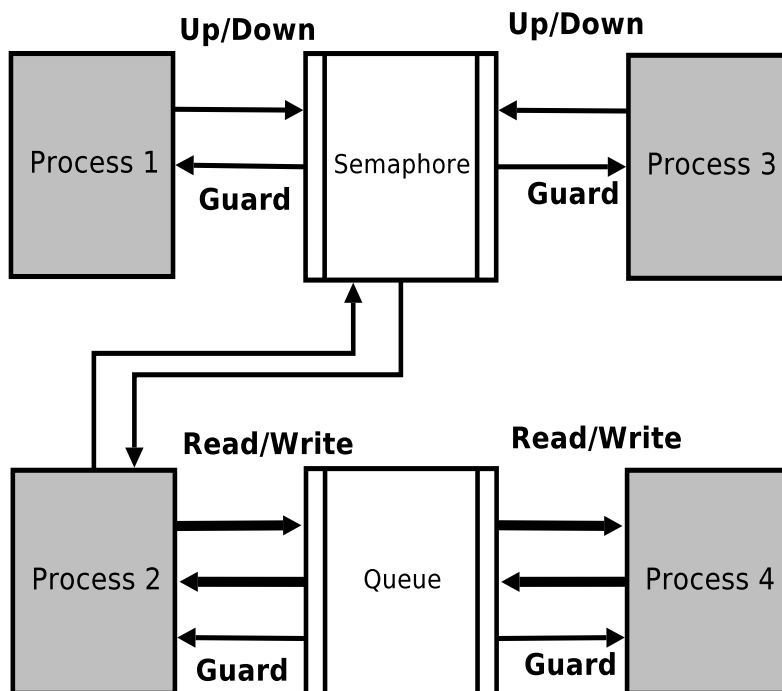
$[\beta \Rightarrow \text{T} \Rightarrow \kappa', \omega']$
 $\kappa 1': a \leftarrow 2; \omega 1' = a, \omega 2' = b$
 $\kappa 2': b \leftarrow 5;$

Multiprozeß-Modell (I)

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

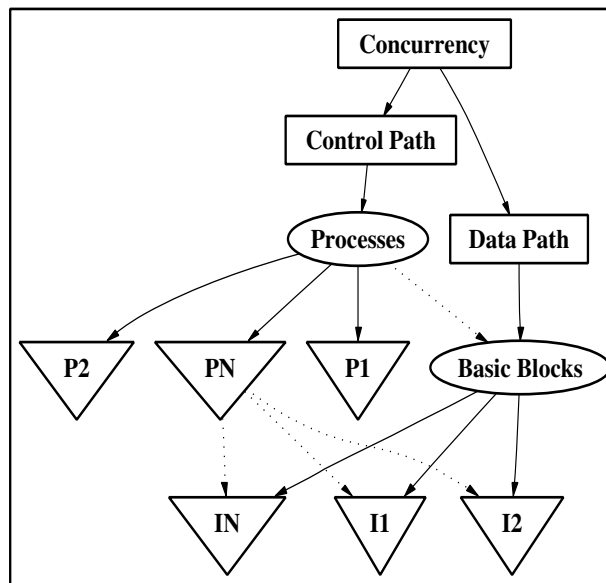
Multiprozeß-Modell (II)

- Multi-Prozeß-Modell mit Interprozeßkommunikation über Semaphoren und Queues



Multiprozeß-Modell (III)

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



Multiprozeß-Modell :: Interprozeßkommunikation (I)

Mutual Exclusion Lock (μ :Mutex)

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

LOCK(σ :S_UNLOCKED)

Ein Prozeß ρ_1 erlangt durch eine LOCK-Operation die ausschließlichen Eigentumsrechte der Mutex.

Zustandsübergang: $\sigma+(\mu): S_UNLOCKED \rightarrow S_LOCKED$

LOCK(σ :S_LOCKED)

Ein Prozeß ρ_2 führt eine LOCK-Operation durch, wird aber blockiert bis ρ_1 die Mutex freigibt.

Zustandsübergang: $\sigma+(\rho_2): S_RUN \rightarrow S_BLOCKED$

UNLOCK(σ :S_LOCKED)

Der Prozeß ρ_1 gibt die Mutex wieder frei.

Zustandsübergang: $\sigma+(\mu): S_LOCKED \rightarrow S_UNLOCKED$

Zustandsübergang: $\sigma+(\rho_2): S_BLOCKED \rightarrow S_RUN$

Multiprozeß-Modell :: Interprozeßkommunikation (II)

Semaphore (μ :Sem)

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

SET($\phi=0$)

Ein Prozeß ρ_1 ändert den Wert der Semaphore.

Wertänderung: $\phi \leftarrow \phi_0$

DOWN($\sigma: S_UNLOCKED, \phi > 0$)

Ein Prozeß ρ_1 ändert den Wert der Semaphore.

Wertänderung: $\phi \leftarrow \phi - 1$

DOWN($\sigma: S_UNLOCKED, \phi = 0$)

Ein Prozeß ρ_2 führt eine DOWN-Operation durch, wird aber blockiert bis ein beliebiger Prozeß ρ die Semaphorewert ϕ erhöht. Danach wird die DOWN-Operation wirksam, so daß als Resultat $\phi=0$ bleibt!

Zustandsübergang: $\sigma + (\mu): S_UNLOCKED \rightarrow S_LOCKED$

Zustandsübergang: $\sigma + (\rho_2): S_RUN \rightarrow S_BLOCKED$

Multiprozeß-Modell :: Interprozeßkommunikation (III)

Semaphore (μ :Sem) Fort.

➤ DOWN($\sigma: S_LOCKED, \phi = 0$)

Ein Prozeß ρ_2 führt eine DOWN-Operation durch, wird aber blockiert bis ein beliebiger Prozeß ρ die Semaphorewert ϕ erhöht. Danach wird die DOWN-Operation wirksam, so daß als Resultat $\phi=0$ bleibt!

Zustandsübergang: $\sigma + (\rho_2): S_RUN \rightarrow S_BLOCKED$

UP($\sigma: S_UNLOCKED, \phi \geq 0$)

Der Prozeß ρ verändert den Wert der Semaphore.

Wertänderung: $\phi \leftarrow \phi + 1$

UP($\sigma: S_LOCKED, \phi = 0$)

Der Prozeß ρ_1 verändert den Wert der Semaphore. Ein wartender Prozeß ρ_2 mit ausstehender DOWN-Operation wird freigegeben.

Keine Wertänderung: $\phi \leftarrow 0$

Zustandsübergang: $\sigma + (\mu): \# \rho(S_BLOCKED) = 0: S_LOCKED \rightarrow S_UNLOCKED$

Zustandsübergang: $\sigma + (\rho_2): S_BLOCKED \rightarrow S_RUN$

Multiprozeß-Modell :: Interprozeßkommunikation (IV)

Event (μ :Event)

- Mit einem Event wird eine Gruppe aus mehreren konkurrierend laufende Prozessen synchronisiert.
- Es gibt zwei Operationen auf ein Event: $\Theta = \{\text{AWAIT}, \text{WAKEUP}\}$

AWAIT

Prozesse $\rho_1 \dots \rho_N$ warten auf das Event. Die Prozesse werden bis zum Eintreten blockiert.

Zustandsübergang: $\sigma+(\rho_1, \rho_2, \dots): S_RUN \rightarrow S_BLOCKED$

WAKEUP

Ein beliebiger Prozeß ρ signalisiert das Event. Alle blockierten und wartenden Prozesse werden frei gegeben.

Zustandsübergang: $\sigma+(\rho_1, \rho_2, \dots): S_BLOCKED \rightarrow S_RUN$

Multiprozeß-Modell :: Interprozeßkommunikation (IVb)

Zeitzähler (μ :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: $\Theta = \{\text{AWAIT}, \text{START}, \text{STOP}, \text{SET}\}$

AWAIT($t < \tau$)

Prozesse $\rho_1 \dots \rho_N$ warten auf das Timer-Event. Die Prozesse werden bis zum Eintreten blockiert.

Zustandsübergang: $\sigma+(\rho_1, \rho_2, \dots): S_RUN \rightarrow S_BLOCKED$

AWAIT($t = \tau$)

Die wartenden Prozesse werden gleichzeitig freigegeben.

Zustandsübergang: $\sigma+(\rho_1, \rho_2, \dots): S_BLOCKED \rightarrow S_RUN$

$t \leftarrow 0 \wedge \text{start process timer again}$

SET(t)

$\tau \leftarrow t$

START

$t \leftarrow 0 \wedge \text{start process timer: while } t < \tau \text{ do } t \leftarrow t + \delta; \text{ wakeup}$

STOP

$t \leftarrow 0 \wedge \text{stop process timer}$

Multiprozeß-Modell :: Interprozeßkommunikation (V)

Barriere (μ :Barrier)

- Mit einer Barriere wird eine Gruppe aus N konkurrierend laufende Prozessen synchronisiert. Aber im Gegensatz zum Event erzeugt die Gruppe selbst das Ergebnis, 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 $\phi=N$, wird die Barriere wieder auf 0 gesetzt, und alle blockierten Prozesse werden freigegeben.
- Es gibt zwei Operationen auf ein Event: $\Theta=\{\text{INIT}, \text{AWAIT}\}$

INIT(n)

$N \leftarrow n$

$\phi \leftarrow 0$

AWAIT($\mu+1 < N$)

Prozesse $\rho_1 \dots \rho_{(N-1)}$ warten auf das Event. Die Prozesse werden bis zum Eintreten der Bedingung $\mu=N$ blockiert.

Zustandsübergang: $\sigma+(\rho_1, \rho_2, \dots): S_RUN \rightarrow S_BLOCKED$

$\phi \leftarrow \phi + 1$

AWAIT($\mu+1 = N$)

Die Prozesse werden freigegeben.

Zustandsübergang: $\sigma+(\rho_1, \rho_2, \dots): S_BLOCKED \rightarrow S_RUN$

$\phi \leftarrow 0$

Multiprozeß-Modell :: Interprozeßkommunikation (VI)

Queue (μ :Queue)

- Eine Queue stellt eine Möglichkeit der datenabhängigen Synchronisation und dem Datenaustausch von einer Gruppe von Prozessen dar.
- Eine Queue stellt einen Datenkontainer 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 $\phi < N$ Datenobjekte aufnehmen kann, und die drei Zustände $\Sigma=\{\text{EMPTY}, \text{FILLED}, \text{FULL}\}$ besitzen kann:

$\sigma(\mu)=\text{EMPTY} \equiv \phi=0$

$\sigma(\mu)=\text{FILLED} \equiv N > \phi > 0$

$\sigma(\mu)=\text{FULL} \equiv \phi=N$

- Es gibt zwei Operationen auf ein Event: $\Theta=\{\text{WRITE}, \text{READ}\}$

READ($\sigma:\text{EMPTY}$)

Ein Prozeß ρ_1 führt eine READ-Operation durch, wird aber blockiert bis ein beliebiger Prozeß ρ_2 ein Datenwort in die Queue schreibt.

Zustandsübergang: $\sigma+(\rho_1): S_RUN \rightarrow S_BLOCKED$

Multiprozeß-Modell :: Interprozeßkommunikation (VII)

Queue (μ :Queue)

► READ(σ :FILLED)

Ein Prozeß ρ_1 führt eine READ-Operation durch, und das älteste Datenwort wird aus der Queue entfernt:

$$\phi \leftarrow \phi - 1$$

$$\mu = [x_1; x_2; \dots; x_M] \rightarrow \mu = [x_2; \dots; x_M] \rightarrow x_1 \mid M < N$$

Zustandsübergang: $\sigma + (\mu): S_FILLED \rightarrow S_EMPTY \mid \phi = 0$

READ(σ :FULL)

Ein Prozeß ρ_1 führt eine READ-Operation durch, und das älteste Datenwort wird aus der Queue entfernt:

$$\phi \leftarrow \phi - 1$$

$$\mu = [x_1; x_2; \dots; x_N] \rightarrow \mu = [x_2; \dots; x_N] \rightarrow x_1 \mid M = N$$

Zustandsübergang: $\sigma + (\mu): S_FULL \rightarrow S_FILLED \mid \# \rho(S_BLOCKED) = 0$

Wenn es blockierte schreibende Prozesse gibt (σ :FULL), dann wird ein neues Datenwort in die Queue geschrieben, und ein Prozeß ρ_2 wieder freigegeben.

$$\mu = [x_1; x_2; \dots; x_{N-1}; x_N] \rightarrow \mu = [x_2; \dots; x_N, x_{N+1}^{WRITE}] \rightarrow x_1$$

Zustandsübergang: $\sigma + (\rho_2): S_BLOCKED \rightarrow S_RUN$

Multiprozeß-Modell :: Interprozeßkommunikation (VIII)

Queue (μ :Queue)

► WRITE(σ :FILLED)

Ein Prozeß ρ_1 führt eine WRITE-Operation durch.

$$\phi \leftarrow \phi + 1$$

$$x_{M+1} \rightarrow \mu = [1; 2; \dots; x_M] \rightarrow \mu = [1; 2; \dots; x_M; x_{M+1}] \mid M < N$$

Zustandsübergang: $\sigma + (\mu): S_FILLED \rightarrow S_FULL \mid \phi = N$

WRITE(σ :FULL)

Ein Prozeß ρ_1 führt eine WRITE-Operation durch, wird aber blockiert solange blockiert, bis ein anderer Prozeß eine READ-Operation durchführt.

Zustandsübergang: $\sigma + (\rho_1): S_RUN \rightarrow S_BLOCKED$

Multiprozeß-Modell :: Hardware-Implementierung (I)

Prozeß

- Ein Prozeß als Ausführung und Abbildung eines Programms kann auf zwei Arten mit Digitallogik implementiert werden:
 1. Mit einem programmgesteuerten Prozessor (klassisch von-Neumann oder Harvard-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 θ des Programms bzw. des Algorithmus:

```
process p:
begin
  1: x ← 1;
  2: a ← a + x;
  3: while a > 0 do x ← x + 1;
end;
⇒ Kosten  $\kappa = (+) \oplus (\leftarrow) \oplus (>) \oplus (\text{ROM}) \oplus (\text{RAM}) \oplus (\text{CODEINTP})$  ⇒  $\mu\text{P}$ 
```

Multiprozeß-Modell :: Hardware-Implementierung (II)

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;
  3: while a > 0 do x ← x + 1;
end;
⇒ Kosten  $\kappa = 2 \otimes (+) \oplus 3 \otimes (\leftarrow) \oplus (>) \oplus (\text{while}) \oplus (\mathcal{R}x) \oplus (\mathcal{R}a) \oplus (\text{FSM}) \oplus (\text{MUX}) \oplus (\text{DEMUX})$  ⇒ FSM-RTL
```

- (1)
 - ↳ Kosten $\kappa(1) < \kappa(2)$ wenn Komplexität θ groß ist
 - Keine oder nur geringe/spezielle Datenpfadparallelität möglich
- (2)
 - ↳ Kosten $\kappa(2) < \kappa(1)$ wenn Komplexität θ klein bis mittel ist
 - ↳ Datenpfadparallelität möglich

ConPro: Higher-Level-Synthese (I)

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;
end;                      end;
```

- Interprozesskommunikation:
 - ➔ Mutex, Semaphore, Barrier, Event, Timer, Queue,
 - ➔ Geschützte globale Register, Variablen (shared resources)

ConPro: Higher-Level-Synthese (II)

Imperativer Multi-Prozeß-Ansatz

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

```
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;
```

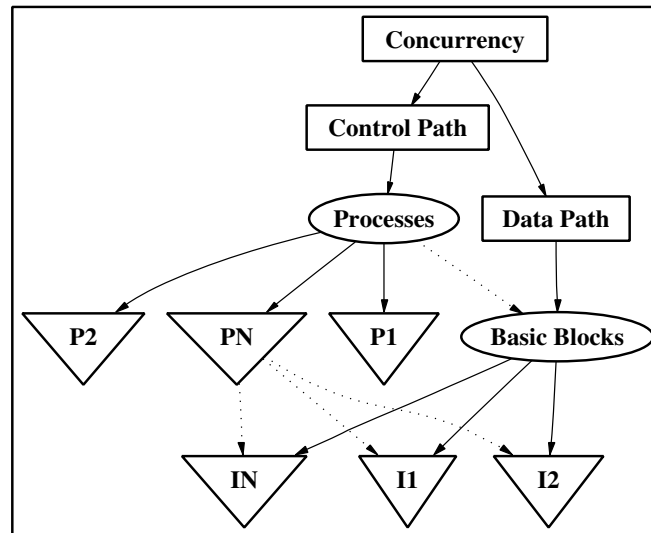
Anweisungen (Ausdrücke und Wertzuweisungen ←), die keine Datenabhängigkeit zueinander besitzen, können im gleichen Zeitschritt ausgeführt werden.

- ➔ Aber: maximal nur ein geschützter Objektzugriff in einem BB.
- ➔ Geschützter Objektzugriff beeinflussen Kontrollpfad (blockierende Operation)!
- BB-Parallelität kann auch automatisch mit Basisblock-Scheduler extrahiert werden.

ConPro: Higher-Level-Synthese (III)

Imperativer Multi-Prozeß-Ansatz

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



ConPro :: Multiprozess-Modell :: Hardware-Implementierung (I)

Die Prozeß-Architektur

- Prozesse werden als reiner Zustandsautomat FSM und RTL implementiert
- Anweisungssequenzen $\mathfrak{J}=\{I1,I2,\dots\}$ werden als Folge von Zuständen $\Sigma=\{\sigma1,\sigma2,\dots\}$ 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), takt-synchron $\equiv \Gamma: \mathfrak{J} \rightarrow \Sigma$

Abhängigkeit und Wirkung: $\downarrow: \sigma, \mathfrak{K}_{\text{LOK}}, \mathfrak{K}_{\text{GLO}}, \Pi, \mu \uparrow: \sigma$

Datenblock Kombinatorisch

Implementierung der Funktionsblöcke sowie Datenpfadselektoren als rein kombinatorische Logik und Beschaltung mit externen globalen Objekten \equiv globale Ausdrücke

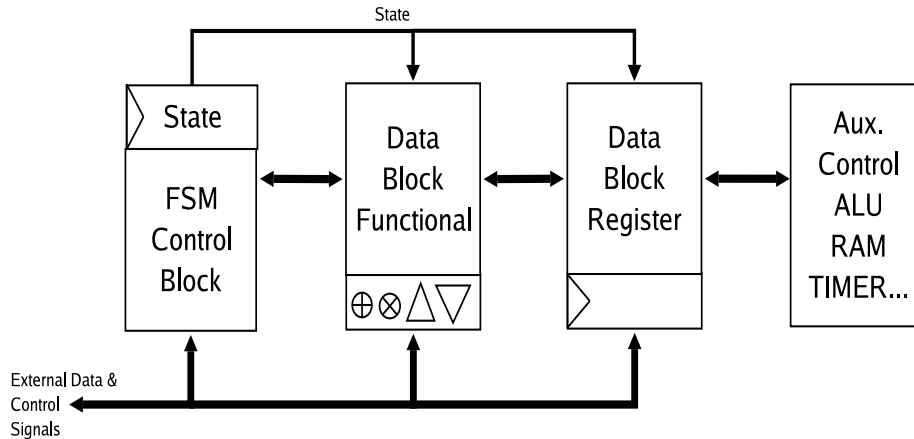
Abhängigkeit und Wirkung: $\downarrow: \sigma, \mathfrak{K}_{\text{LOK}}, \mathfrak{K}_{\text{GLO}}, \Pi, \mu \uparrow: \mathfrak{K}_{\text{GLO}}, \mu$

ConPro :: Multiprozeß-Modell :: Hardware-Implementierung (II)

Die Prozeß-Architektur

Datenblock Transitorisch

Implementierung von lokalen Speicherobjekten wie Register \mathcal{R} und Funktionsblöcken
 \equiv lokale Ausdrücke
 Abhängigkeiten und Wirkung: $\downarrow: \sigma, \mathcal{R}_{\text{LOK}}, \mathcal{R}_{\text{GLO}}, \Pi \uparrow: \mathcal{R}_{\text{LOK}}$

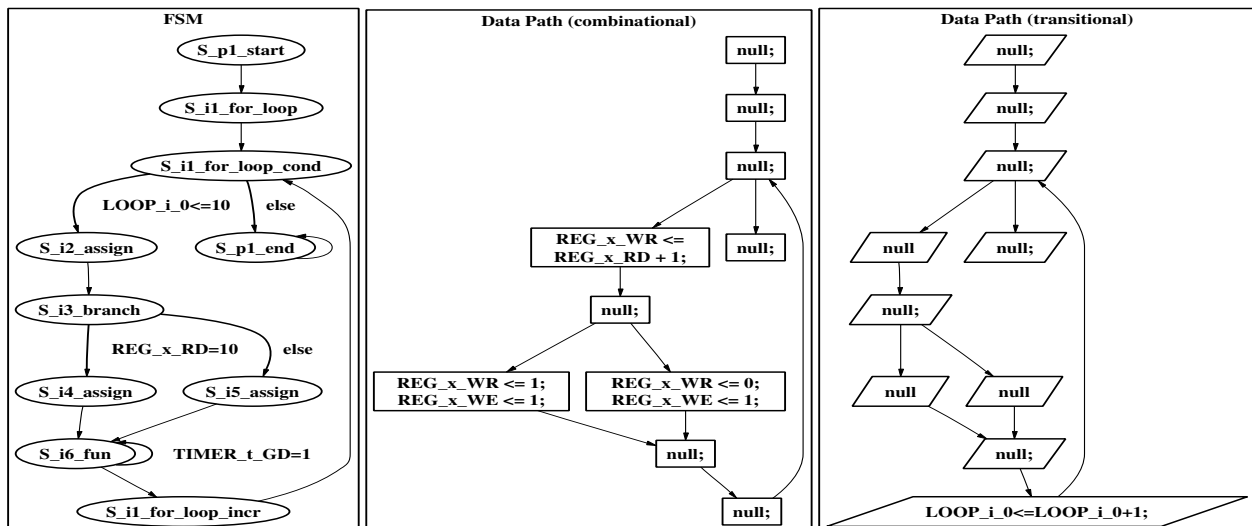


ConPro :: Multiprozeß-Modell :: Hardware-Implementierung (III)

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;
end;
    
```

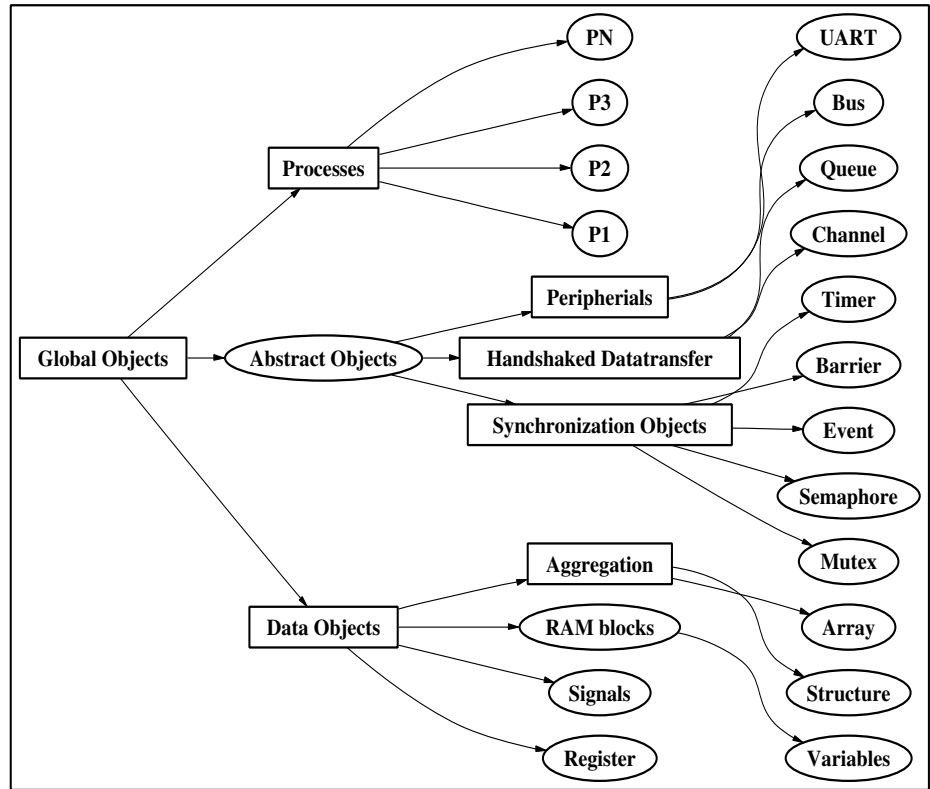


ConPro: Higher-Level-Synthese :: Objekte (I)

Übersicht

Globale Modulebene

- ▶ Prozesse P
- ▶ Register \mathcal{R}
- ▶ Variablen V
- ▶ RAM Blöcke \wp
- ▶ IPC
- ▶ Kommunikationsschnittstellen
- ▶ Signale (wire)

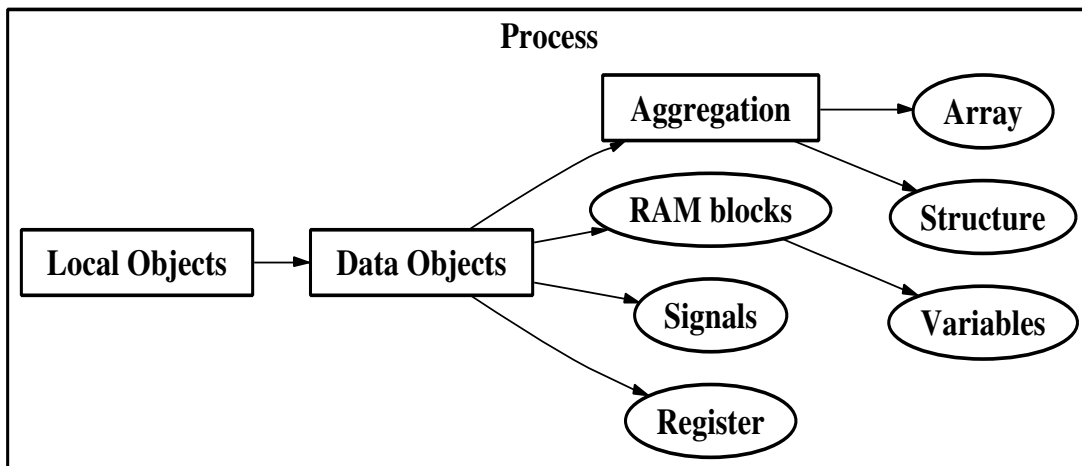


ConPro: Higher-Level-Synthese :: Objekte (I)

Übersicht

Lokale Prozeßebene

- ▶ Register \mathcal{R}
- ▶ Variablen V
- ▶ RAM Blöcke \wp
- ▶ Kommunikationsschnittstellen
- ▶ Signale (wire)



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

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: logic, logic[N], int[N], bool, char
- Verschiedene einzelne Register sind unabhängig voneinander

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 VT IS logic | logic[N] | int[N] | bool | char
- Variablen werden in RAM-Blöcken implementiert, bis zu M (verschiedene: N/VT) Variablen

ConPro: Higher-Level-Synthese :: Objekte (III)

Variablen (Forts.)

- Automatische Datentypenanpassung in Ausdrücken
TYPE RAM IS logic[ω]
 $\omega = \max(\text{size}(V_i) \ \forall \ i=1..M)$
- Beliebige viele RAM-Blöcke möglich

Beispiele

```
reg xs: int[10];
array vs: var[10] of int[31];
block ram with arch="single-port" and mode="read-first";
array vs2: var[10] of int[31] in ram;
object lock_a: mutex;
object ser1: uart;

process p:
begin
  reg x,y,z: int[8];
  array v: var[10] of int[12];
```

ConPro: Higher-Level-Synthese :: Produktbildungstypen

Produktbildung :: einsortig

- Arrays
 - ↳ Objekttypen: `reg|var|sig|object`
 - ↳ Datentypen: `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: `reg|var|sig`
 - ↳ Datentypen: `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: `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 (I)

Bedingte Verzweigung

- Ausführung von Anweisungsblöcken $\mathfrak{S}_{\text{TRUE}}^B$ und $\mathfrak{S}_{\text{FALSE}}^B$ 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 $\mathfrak{S}_i^B \forall i=1\dots n$ in Abhängigkeit der Evaluierung eines Ausdrucks E (atomare und skalare Elementartyp) und Vergleich mit konstanten Werten aus einer endlichen Menge $V=\{v_1, v_2, \dots\}$.

```
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;
```

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

Bedingte Schleife

- Wiederholte Ausführung eines Anweisungsblockes $\mathfrak{S}_{\text{LOOP}}^B$ 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 $\mathfrak{S}_{\text{LOOP}}^B$ in Abhängigkeit der Evaluierung eines impliziten booleschen Ausdrucks E mit einem Indexregister i im Intervall $i = \{ a \text{ to } \mid \text{downto } b \}$

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

Funktionen

- Wiederholte Ausführung eines benannten Anweisungsblockes $\mathfrak{S}_{\text{FUN}}^B$, 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 (I)

Konzept

- Methodenbasierter Zugriff auf abstrakte Objekte (ADTO)
- Ein ADTO ist gekennzeichnet durch:
 1. Ein abstrakter Datentyp α
 2. Ein reaktives Verhaltensmodell
 3. Eine Menge von Operation \mathfrak{O} auf den ADTO α
 4. Eine Hardwarebeschreibung
 5. Steuersignale

Arten

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

ConPro: Higher-Level-Synthese :: Abstrakte Daten Objekte (II)

ADTOs 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 (I)

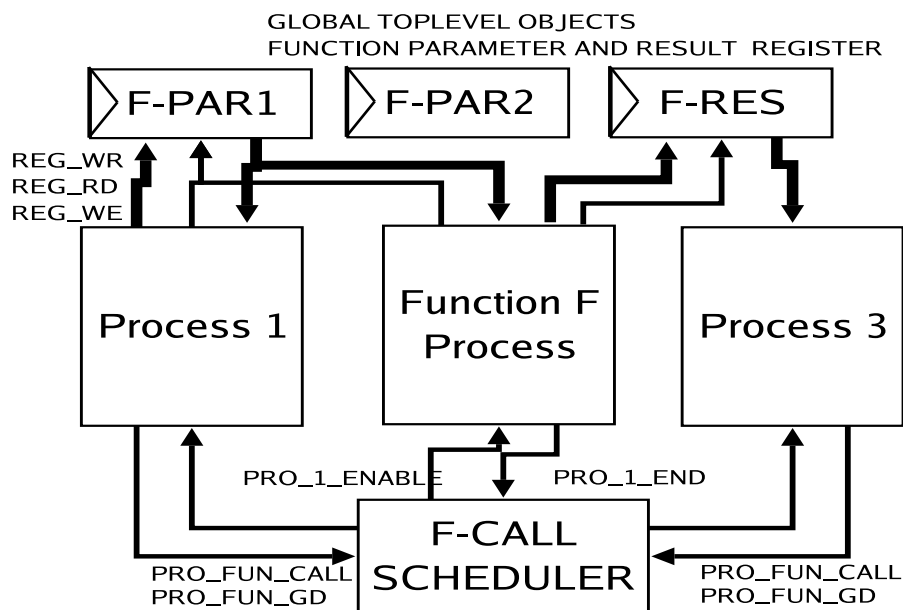
Konzept

- 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 0xFFFFFFFFE;
    end;
  end;
  z ← q[0 to div_n1];
end;
```

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

Implementierung



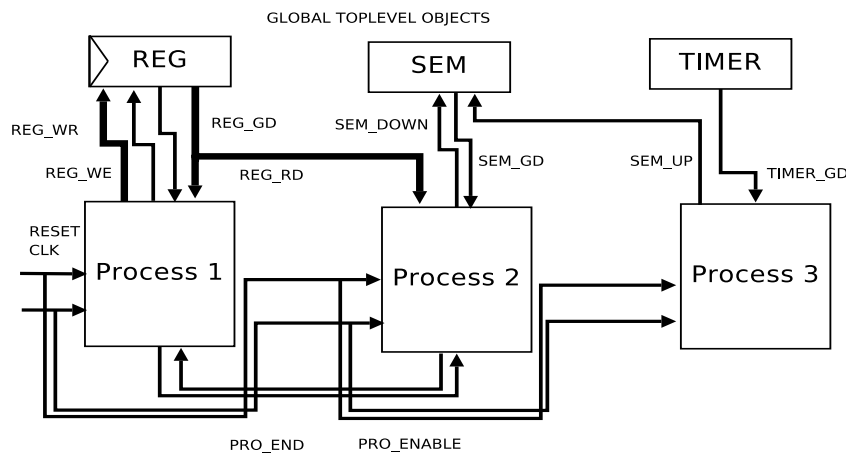
ConPro: Higher-Level-Synthese :: Interconnect-Architektur (I)

Steuersignale

- Objekte werden über Steuersignale $\wp \rightarrow$ Ausgang und Eingang $\rightarrow \wp$, angesprochen
- Datenaustausch über Lese- und Schreibports \aleph
- Beispiel Register:

```

 $\mathfrak{J}$ : reg x: int[8];  $\Rightarrow$ 
 $\wp$ : REG_x_WE:std_logic $\rightarrow$ REG_x_GD:std_logic
 $\aleph$ :  $\rightarrow$ REG_x_RD:std_logic_vector(7 downto 0)
 $\aleph$ : REG_x_WR:std_logic_vector(7 downto 0) $\rightarrow$ 
    
```



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

Konzept

- Regelbasierte Abbildung von (komplexen) Prozeß-Instruktionen \mathfrak{J} auf Register-Transfer-Logik:

$$S : \mathfrak{J} \rightarrow \Delta = (\Sigma, \Pi, \aleph) \times \Gamma, \forall \iota \in \mathfrak{J}(P) \wedge t \in T \Rightarrow t : \iota \rightarrow (\sigma, \pi, r) \times \gamma \quad (1)$$

mit: Δ : Datenpfad (Σ : Datenpfadselektoren, Π : Funktionsblöcke, \aleph : Register) und Γ : Zustandsmenge des FSM, T : Abbildungsregelsatz

- Es stehen verschiedene (Mengen von) Abbildungsregeln t 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 $G_P(\mathfrak{J})$ beschrieben.
- RTL wird nicht direkt aus dem Syntaxgraphen erzeugt
 - Die komplexen Prozeß-Instruktionen des Syntaxgraphens G werden in lineare Liste von einfachen Mikrocode-Instruktionen als Zwischenrepräsentation transformiert:

$$S : \mathfrak{J} \rightarrow \Omega, \forall \iota \in \mathfrak{J}(P) \wedge t \in T \Rightarrow t : \iota \rightarrow \mu \quad (2)$$

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

Scheduling & Optimierung

- Auf Syntaxgraph G-Ebene: **Referenzstack-Scheduler**
 - ↳ Symbolische Syntaxgraph-Analyse bezüglich Fortpflanzung und Entwicklung von Wertzuweisungen und Wertaufwurf von Datenobjekten (Register und Variable)
 - ↳ Rückwärtssubstitution und Konstantenfaltung ermöglichen reduzierte Ausdrücke und Anweisungen
 - ↳ Reduktion von N Datenpfadanweisungen mit J Registern \mathfrak{R} und X Funktionsblöcken (Σ, Π) ursprünglich in τ_1 Zeitschritten auf $M < N$ Datenpfadanweisungen in $\tau_2 < \tau_1$ Zeitschritten mit $K < J$ Registern \mathfrak{R} 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 τ_1 Zeitschritten auf $\tau_2 < \tau_1$ Zeitschritte
- Auf RTL-Zustandsebene Γ : Komprimierung von Zuständen
 - ↳ Zusammenfassung von Kontroll- und Datenpfadanweisungen
 - ↳ Entfernung überflüssiger Zustände

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

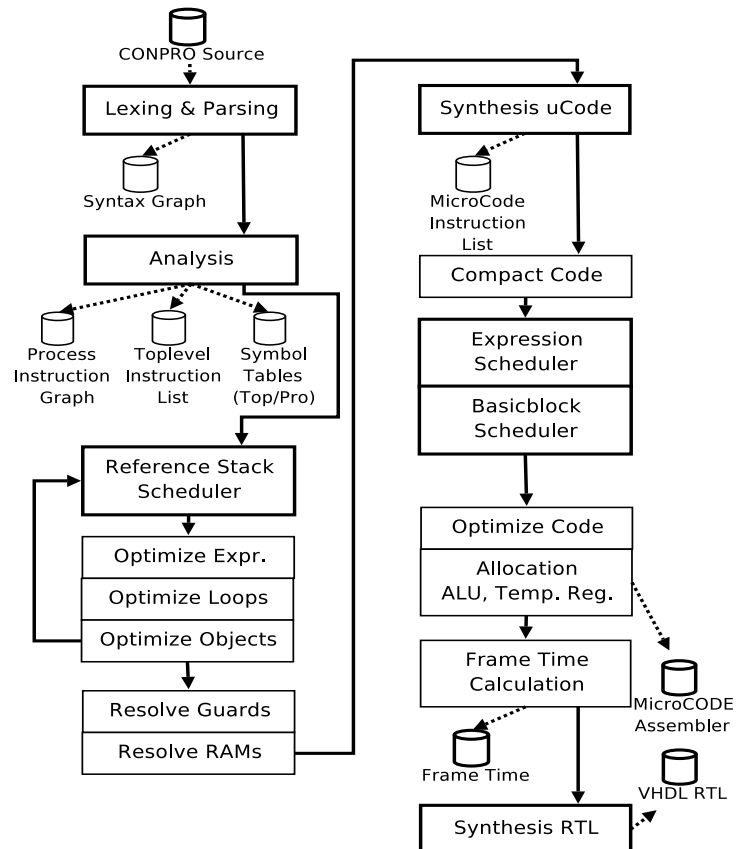
Allokation

- 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 Ausdrücken mit **Ausdrucks-Scheduler**
 - ↳ Randbedingung parametrisierbar: Zeit/Latenz τ für Einheitsoperationen
 - ↳ Ausdrücke werden in binären Teilausdrücke zerlegt und wieder zusammengesetzt, solange zeitliche kummulative Randbedingung $\sum \tau_i < \tau_{CLK}$ eingehalten wird

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

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



ConPro: Higher-Level-Synthese :: μ Code (I)

Konzept

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

Mnemonic	Beschreibung
move (dst,src)	Δ : Datentransfer $dst \leftarrow src$, Γ : $\sigma+1 \sigma(GD)$
expr (dst,op1,op,op2)	Δ : Ausdruck $E = op1 \ op \ op2$ und Datentransfer $dst \leftarrow E$, Γ : $\sigma+1 \sigma(GD)$
falsejump (expr,label)	Bedingte Verzweigung Γ : $\sigma+1 label(expr)$ if $expr = true$
jump (label)	Unbedingte Verzweigung Γ : label
bind (n)	Δ : Bindung von Anweisungen in einem Zeitschritt, Γ : $\sigma+1 \sigma(GD)$

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

Beispiel

```
reg x,y,z: int[8];
process p1:
begin
  reg a: int[8];
  for i = 1 to 10 do
  begin
    a  $\leftarrow$  i*2;
    x  $\leftarrow$  x + a;
  end;
end;
↓
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 (I)

Konzept

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

```
reg x,y:int[8];
1: x $\leftarrow$ 1;       $\Rightarrow$  DEF x1=1
2: x $\leftarrow$ x+1;     $\Rightarrow$  DEF x2=x1+1
3: y $\leftarrow$ x-1;     $\Rightarrow$  DEF y1=x2-1
```

- Bei der Analyse des Kontrollpfades wird jeder neue Ausdruck E_i von \mathfrak{X} auf dem Stack $T(\mathfrak{X})=[E_i; \dots; E_2; E_1]$ abgelegt.

```
reg x,y:int[8];  $\Rightarrow$  T(x)=[?] T(y)=[?]
1: x $\leftarrow$ 1;       $\Rightarrow$  T(x)=[1;?]
2: x $\leftarrow$ x+1;     $\Rightarrow$  T(x)=[T(x,2)+1;1;?]
3: y $\leftarrow$ x-1;     $\Rightarrow$  T(y)=[T(x,3)-1;?]
```

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

Scheduling

- ▶ Wertzuweisungen an Speicherobjekte \mathfrak{X} werden
 1. bis zum Ende des aktuellen Rahmens,
 2. bis zum Auftreten einer Verzweigung im Kontrollpfad (bedingte Verzweigung und Schleifen)
 3. bis zum Auftreten eines Funktionsaufrufes verzögert.
- ↳ ALAP-Scheduling
- ▶ Wird eine Zuweisung für ein Objekt \mathfrak{X} ausgeführt (scheduled), wird der oberste Wert/Ausdruck vom Stack verwendet.
- ▶ Durch Rückwärtssubstitution von weiteren Referenzen $T(\mathfrak{X},i)$ und Konstantenfaltung kann der Ausdruck vereinfacht werden:
$$\mathbf{T(x)}=[T(x,2)+1;1;?] \quad \mathbf{T(y)}=[T(x,3)-1;?]$$
$$\mathbf{x} \leftarrow T(x,2)+1 \leftarrow 1+1 \leftarrow 2;$$
$$\mathbf{y} \leftarrow T(x,3)-1 \leftarrow T(x,2)+1-1 \leftarrow 1+1-1 \leftarrow 1;$$
- ▶ Bei bedingten Verzweigungen und Mehrfachauswahl kann weiterhin Invarianz von Objekten getestet werden. Für jeden Zweig des Kontrollpfades wird ein Sub-Stack angelegt. Enthielten diese nach der Evaluierung der Verzweigung alle den gleichen Ausdruck, kann eine Wertzuweisung entweder weiter verzögert werden oder vor die Verzweigung einmalig erfolgen.

ConPro: Higher-Level-Synthese :: Referenz Stack (III)

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 (I)

Konzept

- ▶ Der Daten- und Kontrollpfad $\mathfrak{J} = \{\iota_1, \dots\}$ wird in Basisblöcke zerlegt.
- ▶ Ein Basisblock ist gekennzeichnet durch
 1. eine Menge von reinen Datenanweisungen und Ausdrücken,
 2. nur einen einzigen Zugang im Kontrollpfad am Kopf,
 3. und nur einen Ausgang im Kontrollpfad am Ende,
 4. 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 :

$$\mathfrak{J} \rightarrow \{MB_1, MB_2, \dots, MB_N\}, MB_i^\Delta = \{\iota_{\Delta 1}, \dots\} | MB_i^\Gamma = \{\iota_{\Gamma 1}, \dots\} \quad (3)$$

- ▶ 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.

$$MB_i^\Delta \rightarrow \{mb_1, mb_2, \dots, mb_N\}, mb_i = \{\iota_{\Delta 1}, \dots\} \quad (4)$$

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

Datenabhängigkeitsgraphen

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

$$DDG_i^\Delta \rightarrow \{mb_a \rightarrow mb_b \rightarrow \dots \rightarrow mb_x\}, mb_i = \{\iota_{\Delta 1}, \dots\} \quad (5)$$

- ▶ Beispiel:

1: move(x, 0)	\Rightarrow 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)	\Rightarrow MB2:CONT={5}
falsejump (\$immed.[0], 8)	
6: move(x, 1)	\Rightarrow MB3:DATA={7}
7: jump (10)	\Rightarrow MB4:CONT={8}
8: move(x, 0)	\Rightarrow MB5:DATA={9, 10}
9: move(y, 0)	
10: ...	

ConPro: Higher-Level-Synthese :: Basisblock Scheduler (III)

Datenabhängigkeitsgraphen

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

$$\text{DDG}_i^\Lambda \rightarrow \{\text{mb}_a \rightarrow \text{mb}_b \rightarrow \dots \rightarrow \text{mb}_x\}, \text{mb}_i = \{\iota_{\Delta 1}, \dots\} \quad (6)$$

- 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)
```

ConPro: Higher-Level-Synthese :: Basisblock Scheduler (IV)

Scheduling

- Scheduling:
 1. So schnell wie möglich: ASAP, und
 2. 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 (I)

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 to 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 immediate left and right.

[Andrews 2000, Multithreaded, Parallel, and Distributed Programming]

➤ Lösung:

1. Jeder Philosoph wird durch einen Prozeß P abgebildet
2. Geteilte Ressource: die 5 Gabeln
3. Jede Gabel wird durch eine Semaphore S abgebildet, Initialwert S=1
4. Ein Philosoph befindet sich entweder im Zustand σ =thinking oder eating.
5. Der Zustandsübergang σ :thinking→eating des i-ten Philosophen erfordert gleichzeitig Ressourcenanforderung von S_i und S_{i+1} (down).
6. Nachdem ein Philosoph gegessen hat, geht er wieder in den Zustand thinking über: σ :eating→thinking, und gibt die Ressourcen S_i und S_{i+1} (up) wieder frei.

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

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

```
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;
```

ConPro: Higher-Level-Synthese :: Beispiel (III)

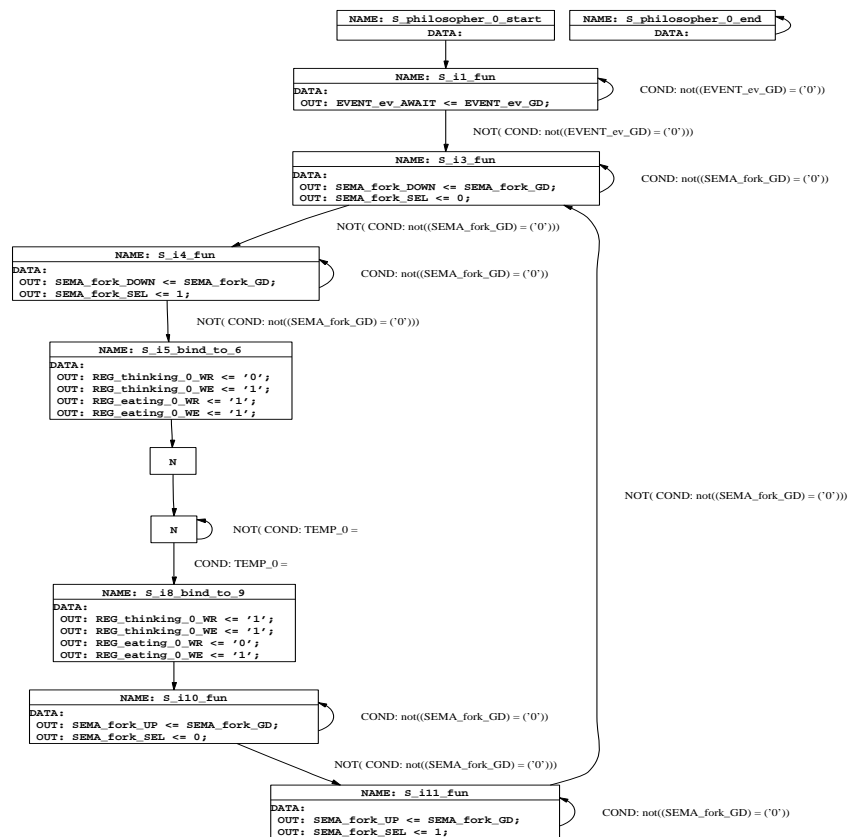
- Higher-Level: μ Code
- gekürzt
- nop-Instruktionen entstanden aus Optimierung

```

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)
    
```

ConPro: Higher-Level-Synthese :: Beispiel (IV)

- High-Level: RTL
- Zustandsdiagramm vom Prozeß philosopher



ConPro: Higher-Level-Synthese :: Beispiel (V)

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

```
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
```

```
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
);
```

ConPro: Higher-Level-Synthese :: Beispiel (VI)

- High-Level: VHDL
- Prozeß philosopher_0
- Γ

```
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;
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;
```

ConPro: Higher-Level-Synthese :: Beispiel (VII)

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

```
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;
    end case;
end process data_path;
```

```
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;
```

ConPro: Higher-Level-Synthese :: Beispiel (VIII)

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

```
-- 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;
            end case;
        end if;
    end if;
end process data_trans;
```

```
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: Higher-Level-Synthese :: Beispiel (iX)

- Synthesis-Summary
- ConPro

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

ConPro: Higher-Level-Synthese :: Beispiel (iX)

- VHDL Synthese
- SXLIB Standardzellenbibliothek SXLIB
- Leonardo Spectrum Mentor

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
nm2_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

ConPro: Higher-Level-Synthese :: Beispiel (iX)

- VHDL Synthese
- SXLIB Standardzellenbibliothek SXLIB
- Leonardo Spectrum Mentor

```

nrx2_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

```

ConPro: Higher-Level-Synthese :: Beispiel (X)

- Simulation gate-level
- SXLIB Standardzellenbibliothek SXLIB
- Alliance Asimut
- Pattern-generator: C

```

#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 *) mbcalloc(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<...

```

ConPro: Higher-Level-Synthese :: Beispiel (XI)

```

● Simulation gate-level
● SXLIB Standardzellen-
bibliothek SXLIB
● Alliance Asimut
● Pattern-Datei
-- 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 s
i i i i i t t t t t k e
n n n n n i i i i i e t
k k k k k n n n n n
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
_ _ _ _ _ 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 ?u ?u 1 1 ;
< 5000 ps> : ?u ?u ?u ?u ?u ?u ?u ?u ?u ?u ?u ?u 1 1 ;
< 10000 ps> : ?u ?u ?u ?u ?u ?u ?u ?u ?u ?u ?u ?u 1 1 ;
< 15000 ps> : ?u ?u ?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 ;
...

```

ConPro: Higher-Level-Synthese :: Beispiel (XII)

- Simulation gate-level
- SXLIB Standardzellen
- bibliothek SXLIB
- Alliance Asimut
- Pattern Xpat

