technische universität
dortmund

fakultät für informatik
informatik 12

Synthese Eingebetteter Systeme

Wintersemester 2012/13

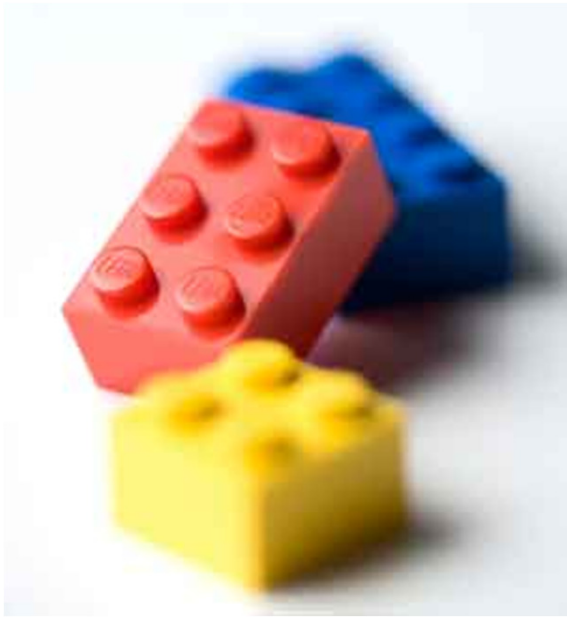
10 – Synthese: Code- und Datenflussanalyse

Michael Engel
Informatik 12
TU Dortmund

2012/12/03

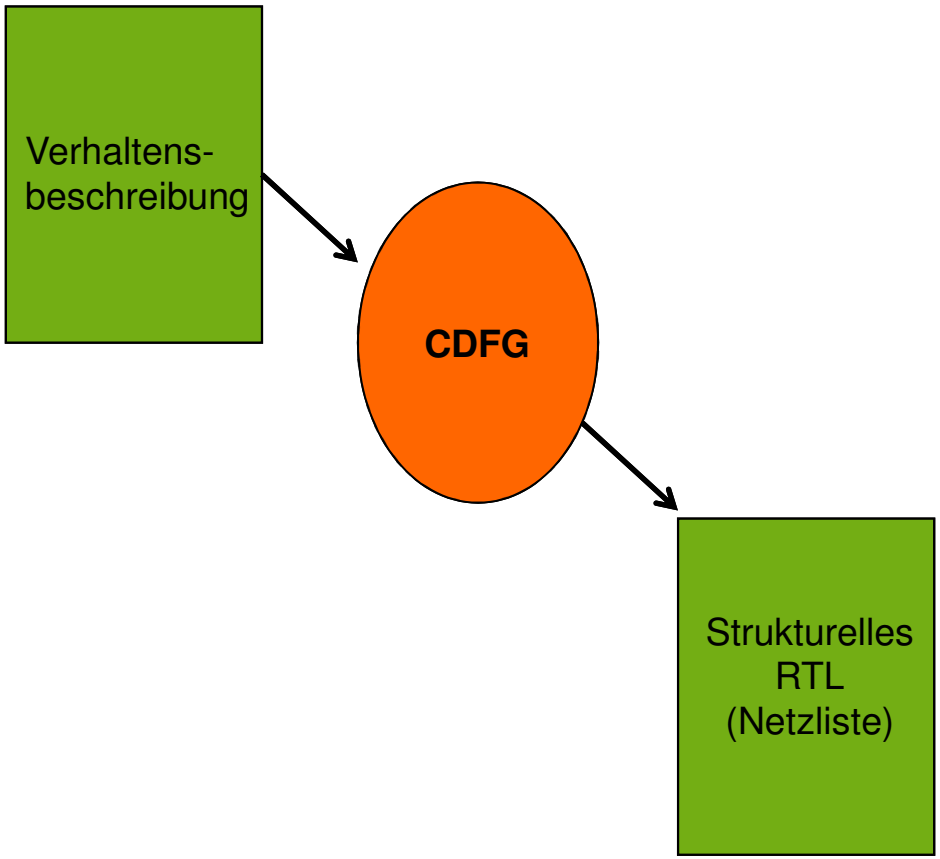
Synthese: Code- und Datenflussanalyse

- Übersetzung
  - AST
  - CDFG und FSMD
- Parallelisierung
  - reale und falsche Abhängigkeiten
  - SSA
- Übersetzungs- und Synthesefluss
  - AST
  - Zwischendarstellung

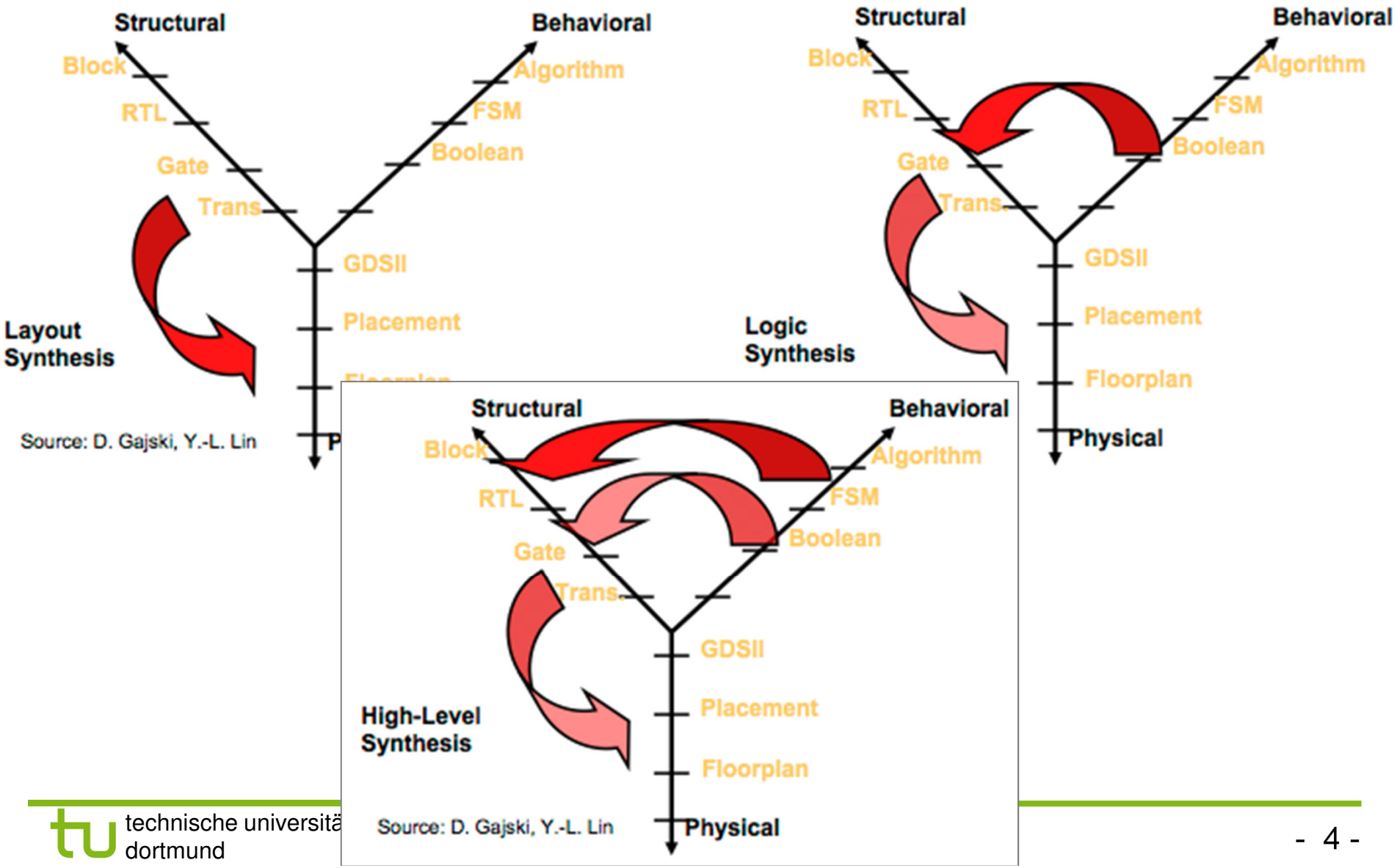


Synthese

- Übersetzung: High-Level-Beschreibung ⇢ RTL-Netzliste



Synthese


Structural
Behavioral
Block
Algorithm
RTL
FSM
Gate
Boolean
Trans
GDSII
Placement
Floorplan
Physical
Layout Synthesis
Logic Synthesis
High-Level Synthesis
Source: D. Gajski, Y.-L. Lin

Parallelisierung

Ziel:

- Synthese paralleler Hardwarestrukturen aus C/C++-Code

Problem:

- Keine Sprachkonstrukte für Parallelität in C(++)
- C(++)-Programmierer “denken” sequentiell

Lösungansätze:

- Erweiterung der Sprache um explizite Parallelität
  - SystemC: SC_METHODS, SC_THREADS
- Finden von Parallelität in sequentiellem Code
  - “normaler” C(++)-Code

Parallelisierung

- Ausnutzung von Parallelität auf Befehlsebene [3],
- Beispiele:
  - Drei Befehle: gleichzeitig ausführbar, da sie nicht von den Ergebnissen der anderen Befehle abhängen:

```
r1      = 1;
[r9]r2  = 17;
4[r3]   = r6;
```

    - *Parallelität = 3*
  - Abhängigkeiten verhindern parallele Ausführung:

```
r1     = 0[r9];
r2     = r1+17;
4[r2]  = r6;
```

    - *Parallelität =1*

Def. Parallelität

- Def: *Parallelität*
  - Parallelität := $\frac{\text{Anzahl Instruktionen}}{\text{Anzahl benötigter Zyklen}}$
- Wie viel Parallelität ist üblicherweise vorhanden?
  - Einschlägige Literatur sagt aus, dass die Parallelität innerhalb eines Basisblocks im Mittel selten den Faktor 3–4 übersteigt
  - Größere Werte z.B. bei bestimmten numerischen Anwendungen erreichbar

Parallelisierung

- **Ziel: Erhöhung der Parallelität innerhalb von Blöcken**
- Parallelität innerhalb eines Basisblocks beschränkt durch paarweise Abhängigkeiten zwischen Instruktionen
  - Einige dieser Abhängigkeiten sind *real* und stellen den Datenfluss im Programm dar
  - Andere Abhängigkeiten sind *falsche Abhängigkeiten,* die aus Mangel an genauem Wissen über den Datenfluss entstehen

Falsche Abhängigkeiten

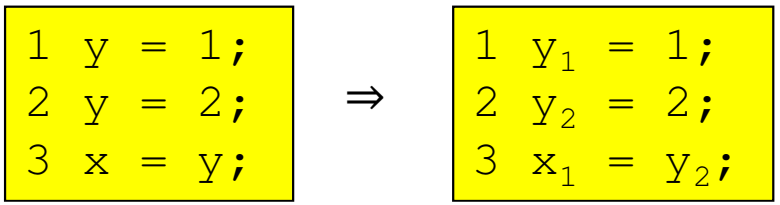
- Registerzuweisungen eines Compilers (z.B. in einer IR), die eine traditionalle skalare Architektur annehmen, können zu falschen Abhängigkeiten führen
- Für die Instruktionen

```
1 r0 = 0[r9];
2 r2 = r1 + 1;
3 r1 = 9;
```

- müssen die Instruktionen 2 und 3 in dieser Reihenfolge ausgeführt werden, da die dritte Instruktion den Wert von r1 ändert
  - Wenn der Compiler an Stelle von r1 das Register r3 verwendet hätte, wären die beiden Befehle unabhängig

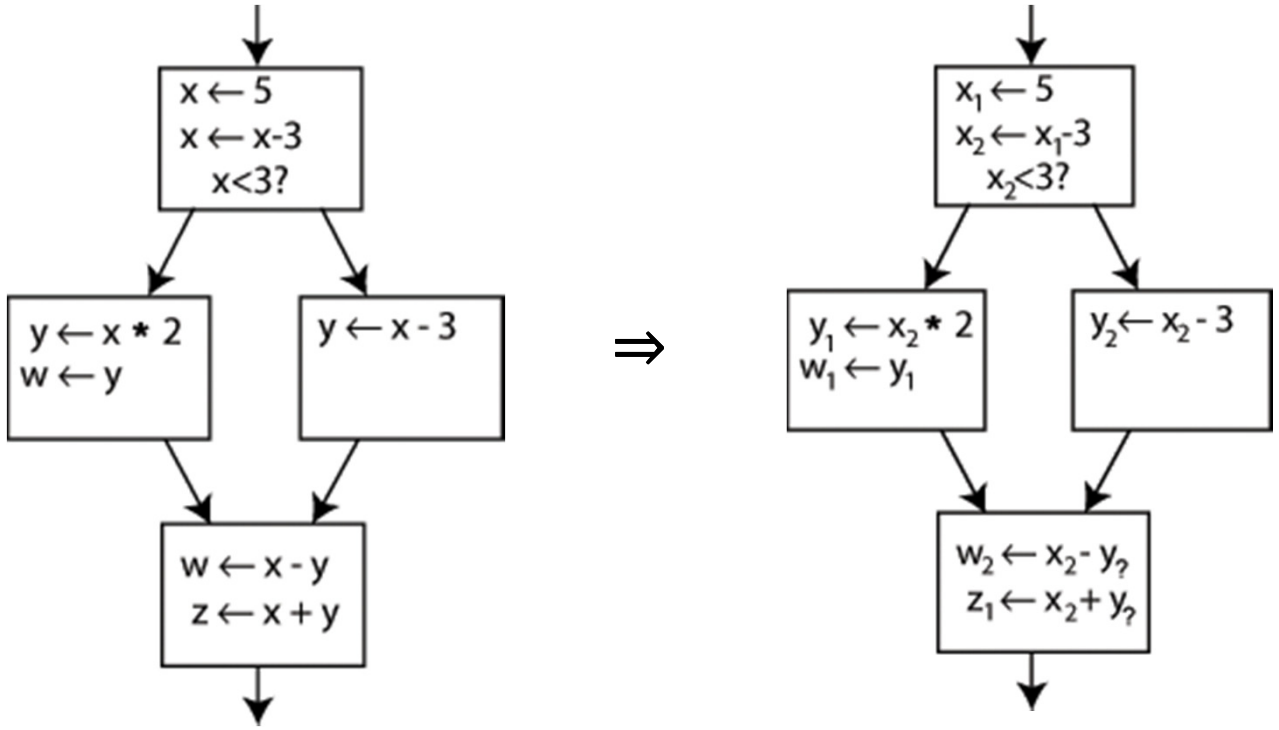
Static Single Assignment (SSA)

- Lösung: *Static Single Assignment*-Form (SSA)
  - Beschreibung, in der jede Variable exakt einmal zugewiesen wird
  - Existierende Variablen aufgeteilt in *Versionen*
    - Neue Variablen erhalten z.B. ursprünglichen Namen + Index
- Beispiel:
  - (Für uns) Offensichtlich: Erste Zuweisung überflüssig
  - Wert von y in Zeile 3 stammt aus Zeile 2
  - Compiler muss *Lebensdauer* der Variable ermitteln
  - Direkt ersichtlich aus SSA



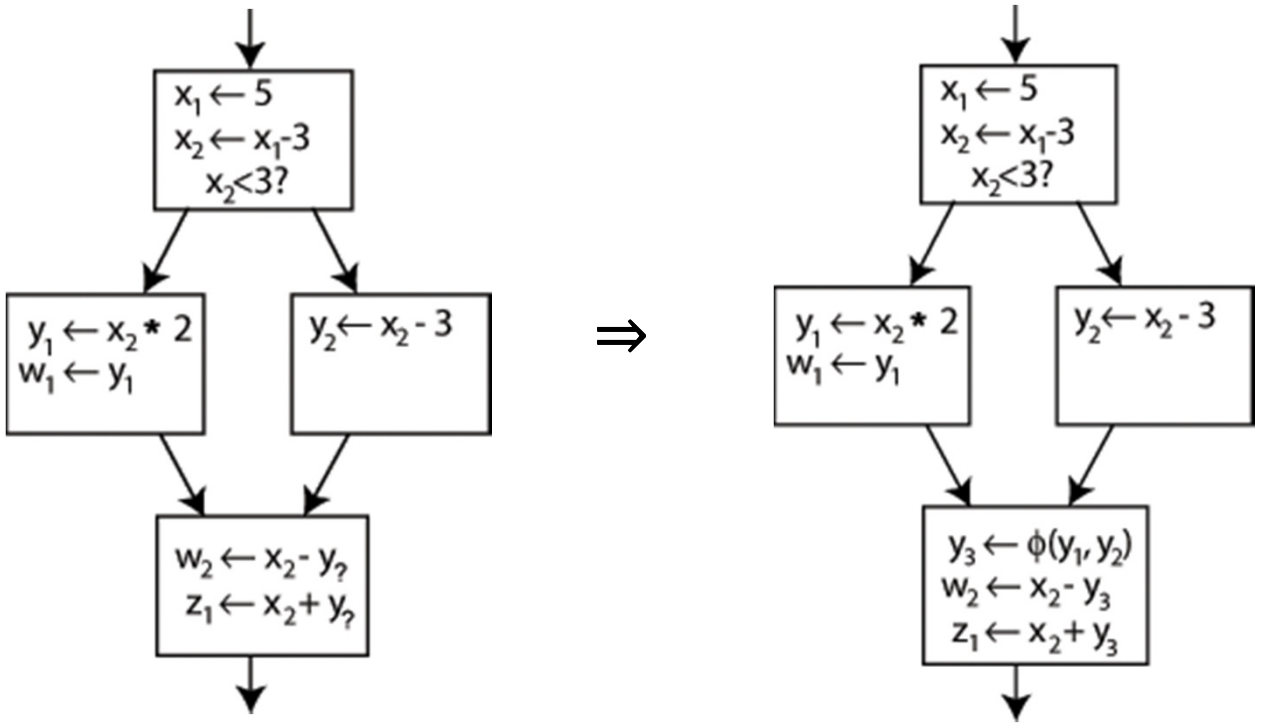
Wandlung zu SSA

- Ersetzen des Ziels jeder Zuweisung durch neue Variable
- Ersetzung der Variablenverwendung durch die Version der Variablen, die den entsprechenden Punkt erreicht



Wandlung zu SSA (2)

- *Problem:* Welches y ist in unterem Block gültig?
- *Lösung:* Einführung der Φ-(Phi-)Funktion
  - Erzeugt neue Definition von y: $\mathbf{y_3}$
    - Entweder y_1 oder y_2, abhängig von vorherigem Kontrollfluss



SSA: Dominanzgrenzen

- Φ-Funktion wird nicht tatsächlich implementiert
  - Hinweis für Compiler, alle Variablen der Φ-Funktion dem selbenSpeicherelement zuzuweisen

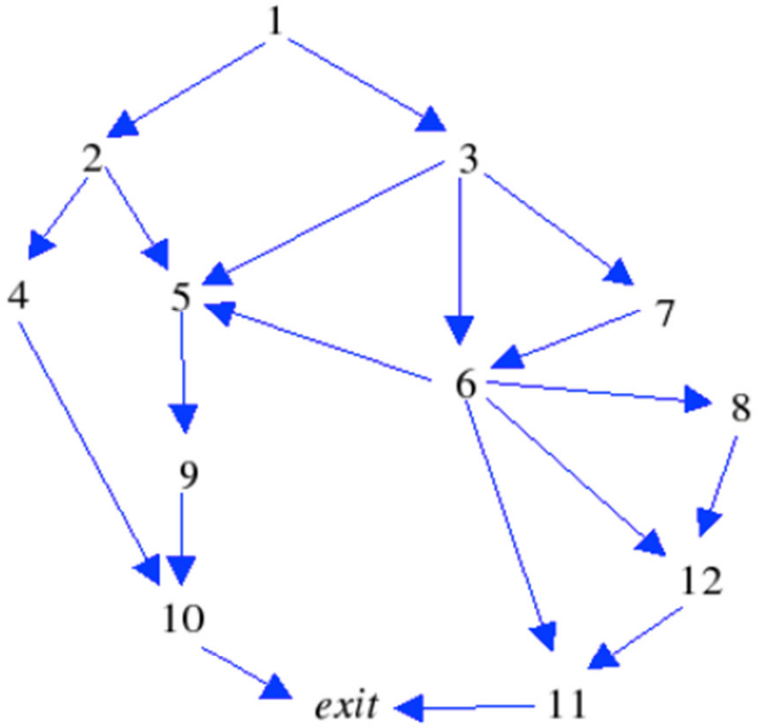
- *Frage:* Wie herausfinden, wo und für welche Variablen Φ-Funktion notwendig ist?
  - Schwierigeres Problem
  - Lösung durch Dominanzgrenzen *(dominance frontiers)*

SSA: Dominanzgrenzen (2)

- Def: *Dominator*
  - Ein Knoten A *dominiert* einen anderen Knoten B im CFG *strikt*, wenn es nicht möglich ist, B zu erreichen, ohne vorher A auszuführen

  ⇒ Wenn B erreicht wird, wurde A vorher ausgeführt
- Def: *Dominanzgrenze*
  - Knoten B ist Dominanzgrenze von Knoten A, wenn Kn. A den Kn. B *nicht* strikt dominiert, aber einen Vorgängerknoten von B dominiert
  - A kann dabei direkter Vorgänger von B sein
- Dominanzgrenze beschreibt exakt die Stellen, an denen Φ-Funktionen notwendig sind

SSA: Beispiel Dominanzgrenze



Dominanzgrenzen:

v	$DF(v)$	v	DF(v)
1	$\emptyset$	8	12
2	5, 10	9	10
3	5, exit	10	exit
4	10	11	exit
5	10	12	11
6	5, exit	exit	$\emptyset$
7	6		

SSA: Dominanzgrenzenberechnung

- Algorithmus zur Berechnung der Menge der Dominanzgrenzen:

```
for each node b
    if the number of immediate predecessors of b ≥ 2
        for each p in immediate predecessors of b
            runner := p
            while runner ≠ doms(b)
                add b to runner's dominance frontier set
                runner := doms(runner)
```

- Ein direkter Vorgänger von Knoten n ist jeder Knoten, von dem aus der Kontrollfluss an Knoten n übergeht
- `doms(b)` ist der Knoten, der b direkt dominiert

Falsche Abhängigkeiten bei Speichern

- Falsche Abhängigkeiten betreffen auch Speicher
- Annahme: Speicherstellen haben Bedeutung für Programmierer, die Register nicht haben
  - Allerdings können erforderliche Annahmen zu falschen Abhängigkeiten bei Speichern führen
- Beispiel: In der Instruktionsfolge

```
r1      = 0[r0]
4[r16] = r3
```

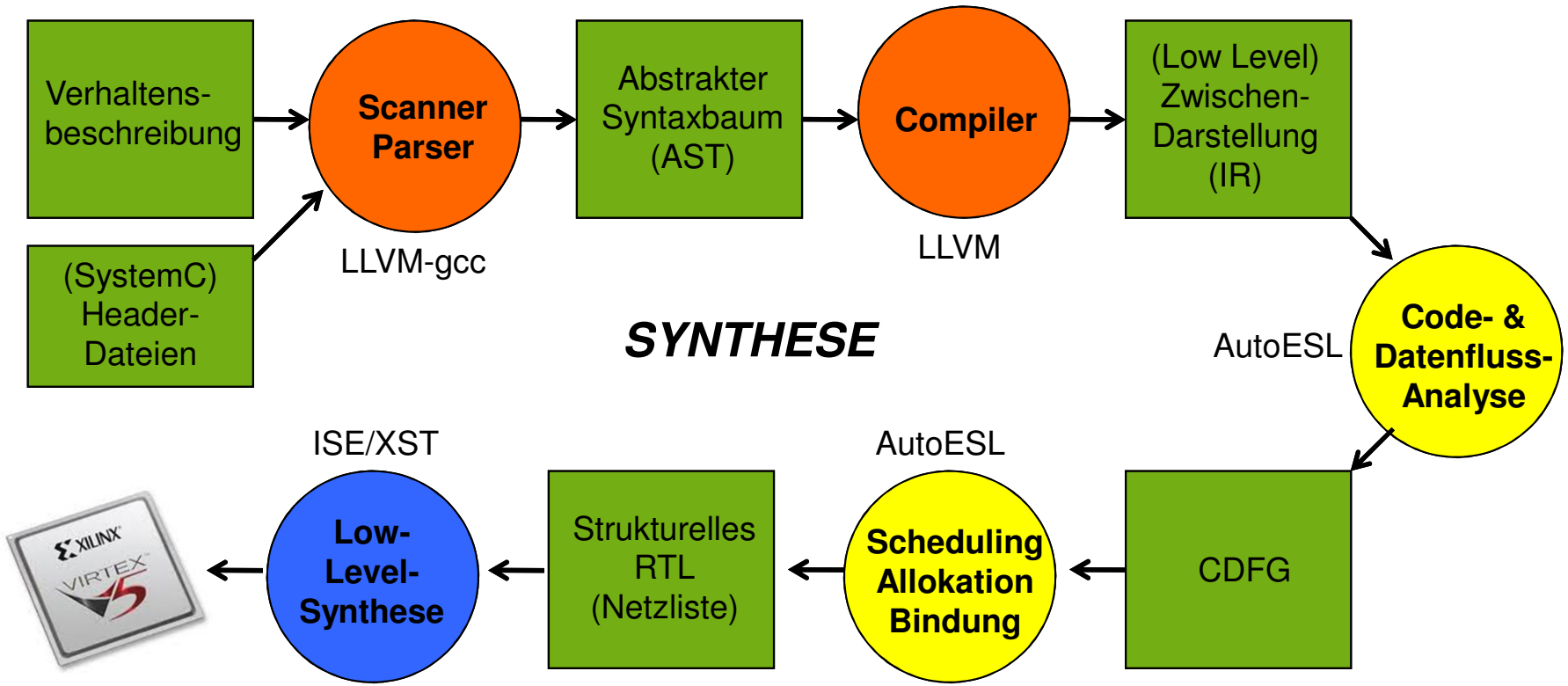
- wissen wir nicht, ob die Speicherstellen 0[r0] und 4[r16] die identische Speicherstelle referenzieren
  - Wenn dies der Fall ist, existiert eine (reale) Abhängigkeit zwischen den beiden Instruktionen
    - Erst alten Wert laden, dann neuen Wert speichern
  - Wenn nicht, existiert keine Abhängigkeit

Alias-Analyse

- *Frage:* Wie können wir Speicher-Abhängigkeiten feststellen?
- *Lösung:* Alias-Analyse (auch points-to-Analyse)
  - Bestimmung, ob in einem Programm auf eine Speicherstelle auf mehr als eine Art zugegriffen werden kann
- Zwei Zeiger sind “aliased”, wenn sie auf die selbe Speicherstelle verweisen
- Aliasanalyse kann bei der Entscheidung helfen, ob zwei Speicherzugriffe unabhängig sind
  - Dies ist aber ungenau – im Zweifel muss der *worst case* angenommen werden
- Später dazu mehr…

Übersetzungs- und Synthesefluss

- Synthesewerkzeuge verwenden oft “normalen” C++-Compiler + Backend
  - AutoESL: LLVM + LLVM-gcc



Übersetzung: Beispiel RTL-Beschreibung

- Programmiersprachensemantik
  - Sequentielle Ausführung
  - Minimierung des Programmieraufwandes

⇓

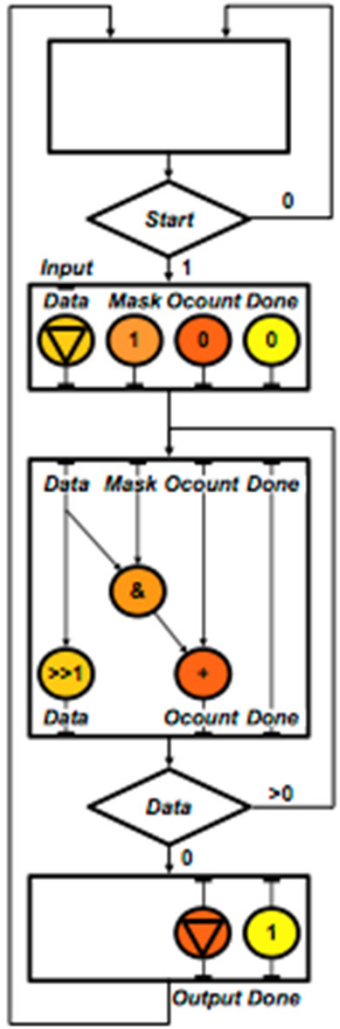
- Hardware-Entwurf
  - Parallele Ausführung
  - Kommunikation über Signale

```
01:  int OnesCounter(int Data){
02:   int Ocount = 0;
03:   int Temp, Mask = 1;
04:   while (Data > 0) {
05:    Temp = Data & Mask;
06     Ocount = Data + Temp;
07:    Data >>= 1;
08:   }
09:   return Ocount;
10:  }
```

```
01:  while(1) {
02:    while (Start == 0);
03:    Done = 0;
04:    Data = Input;
05:    Ocount = 0;
06:    Mask = 1;
07:    while (Data>0) {
08:      Temp = Data & Mask;
09:      Ocount = Ocount + Temp;
10:      Data >>= 1;
11:    }
12:    Output = Ocount;
13:    Done = 1;
14:  }
```

Übersetzung: CDFG

- CDFG für Beispiel
  - Ähnlich zu Programmiersprache
    - Schleifen, ifs, Basisblöcke (BB)
  - Explizite Abhängigkeiten
    - Kontrollflussabhängigkeiten zwischen BBs
    - Datenabhängigkeiten innerhalb von BBs
  - Fehlende Abhängigkeiten zwischen BBs



Übersetzung: Zwischendarstellung

- Darstellung des Programmablaufs in vereinfachter Form
  - Sprach- und maschinenunabhängige Version des ursprünglichen Quellcodes
  - Zwischendarstellung
    - Intermediate Representation: IR
  - Ermöglicht viele Optimierungen und Analysen
    - z.B. Schleifenoptimierungen
  - Verschiedene Detaillierungsgrade möglich:

Original
```
float a[10][20];
a[i][j+2];
```

High IR
```
t1 = a[i, j+2]
```

Mid IR
```
t1 = j + 2
t2 = i * 20
t3 = t1 + t2
t4 = 4 * t3
t5 = addr a
t6 = t5 + t4
t7 = *t6
```

Low IR
```
r1 = [fp - 4]
r2 = [r1 + 2]
r3 = [fp - 8]
r4 = r3 * 20
r5 = r4 + r2
r6 = 4 * r5
r7 = fp - 216
f1 = [r7 + r6]
```

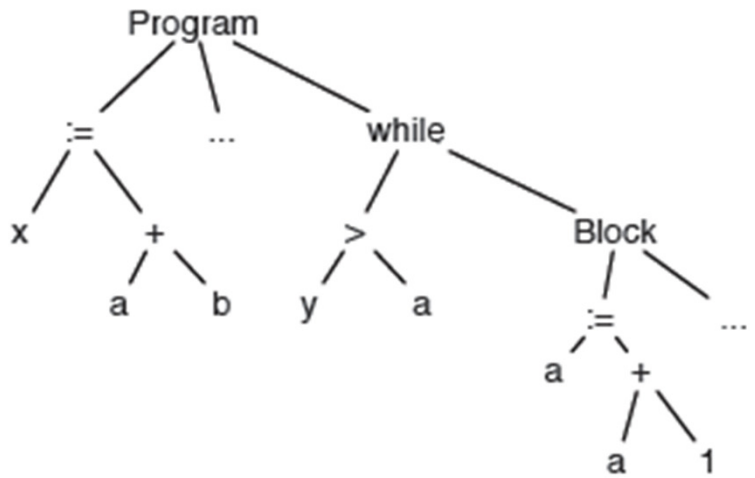
Übersetzung: Zwischendarstellung

- High-level IRs erhalten meist Informationen über Schleifenstrukturen und if-then-else-Kontrollstrukturen
  - Näher an der Quellsprache als IRs auf niedrigeren Ebenen
- Medium-level IRs versuchen oft, sowohl von Quellsprache als auch von der Zielmaschine unabhängig zu sein
- Low-level IRs ähneln meist sehr einer konkreten Zielarchitektur und sind daher oft maschinenabhängig

Zwischendarstellung: AST

- High-Level-Zwischendarstellung
- Beschreibung der Syntaxstruktur eines Programms
  - Ausgangsbasis für High-Level-Transformationen
  - Hierarchische Darstellung
- “Abstrakt”
  - Nicht alle Syntaxelemente des Quellcodes enthalten
  - z.B. keine Klammerung von Ausdrücken und Blöcken

```
x := a + b;
y := a * b;
while (y > a) {
   a := a + 1;
   x := a + b
}
```



Zwischendarstellung: LLVM-IR

- LLVM: “Low-Level Virtual Machine”
  - `http://llvm.org`
  - Moderne, modulare Compilerinfrastruktur
  - Open Source, (größtenteils) in C++ geschrieben
  - Basis für viele High-Level-Synthesewerkzeuge
- LLVM-IR: virtueller Befehlssatz
  - In allen Übersetzungsschritten verwendbar
  - Unabhängig v. Programmiersprache und Zielplattform
- Name “LL” für low-level, aber eigentlich eine Medium Level IR

Zwischendarstellung: LLVM-IR (2)

- LLVM-IR: virtueller Befehlssatz
  - Modelliert Registermaschine
    - Aber mit unendlicher Anzahl an Registern
    - Load/Store-Architektur
  - Operationen sind maschinenähnliche Befehle
    - z.B. add, load, store, icmp, call
- Verwendet Static Single Assignment
- Im Gegensatz zu Assembler:
  - Typgewahr
  - Rigorose Typüberprüfung
  - Explizite Darstellung von Zugriffen auf Arrays und structs
    - Ermöglicht Alias- und Abhängigkeitsanalyse
  - Unterstützung für Vektor- und SIMD-Datentypen

Zwischendarstellung: Beispiel LLVM-IR

Beispielprogramm in C

```
 1 #include <stdio.h>
 2
 3 int main(int argc, char *argv[]) {
 4   if (argc < 2) {
 5         puts("Usage: hello <string>");
 6         return 0;
 7   }
 8
 9   printf("Hello %s\n", argv[1]);
10   return 0;
11 }
```

Zwischendarstellung: Beispiel LLVM-IR (2)

LLVM-IR-Darstellung: Deklarationen

```
1  ; ModuleID='helloWorld.bc'
2
3  …
4
5  @.str=internal constant[22 x i8]
6    c"Usage: hello <string>\00"; <[22 x i8]*> [#uses=1]
7  @.str=internal constant[10 x i8]
8    c"Hello %s\0A\00"; <[10 x i8]*> [#uses=1]
9
10 declare i32 @puts(i8*)
11 declare i32 @prints(i8*, …)
```

Typgewahr

Zwischendarstellung: Beispiel LLVM-IR (3)

Kontrollfluss: if (i1 = bool)

LLVM-IR-Darstellung: Code

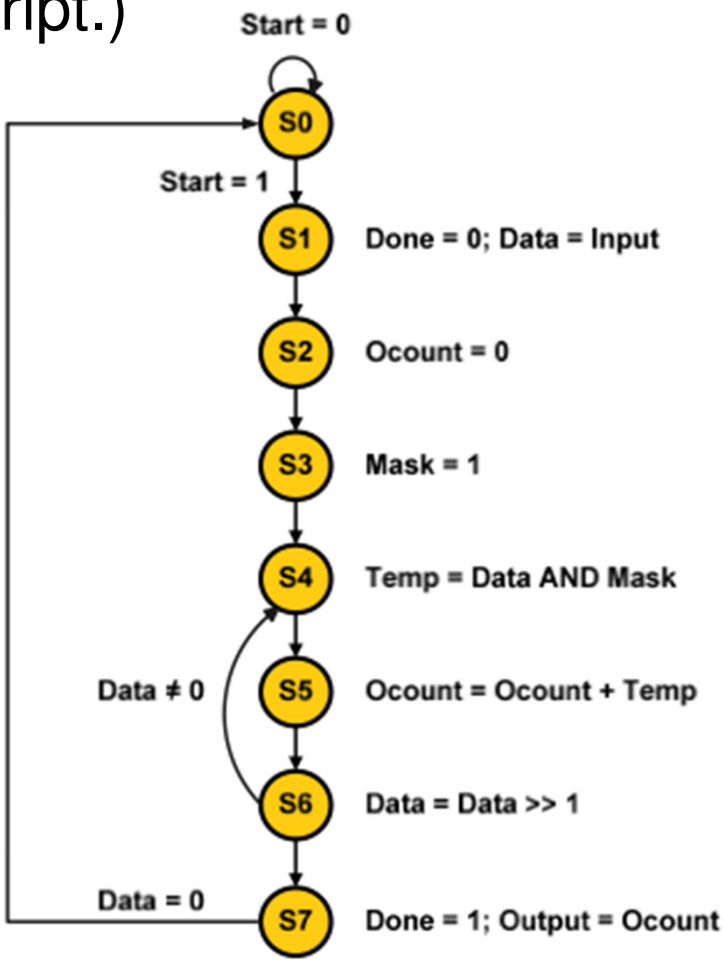
```
 1 define i32 @main(i32 %argc, i8** %argv) {
 2 entry:
 3   %tmp2 = icmp slt i32 %argc, 2;<i1> [#uses=1]
 4   br i1 %tmp2, label %cond_true, label %cond_next
 5
 6 cond_true:  ; preds = %entry
 7   %tmp5 = call i32 @puts(i8* getelementptr
 8                  ([22 x i8]* @.str, i32 0, i32 i)); <i32> [#uses=0]
 9   ret i32 0
10
11 cond_next:  ; preds = %entry
12   %tmp8 = getelementptr i8 ** %argv, i32 1; <i8**> [#uses=1]
13   %tmp9 = load i8** %tmp8; <i8*> [#uses=1]
14   %tmp11 = call i32 (i8*, …)* @printf
15                (i8* getelementptr([10 x i8]* @.str1, i32 0, i32 0),
16                 i8* &tmp9); <i32> [#uses=0]
17   ret i32 0
18 }
```

SSA

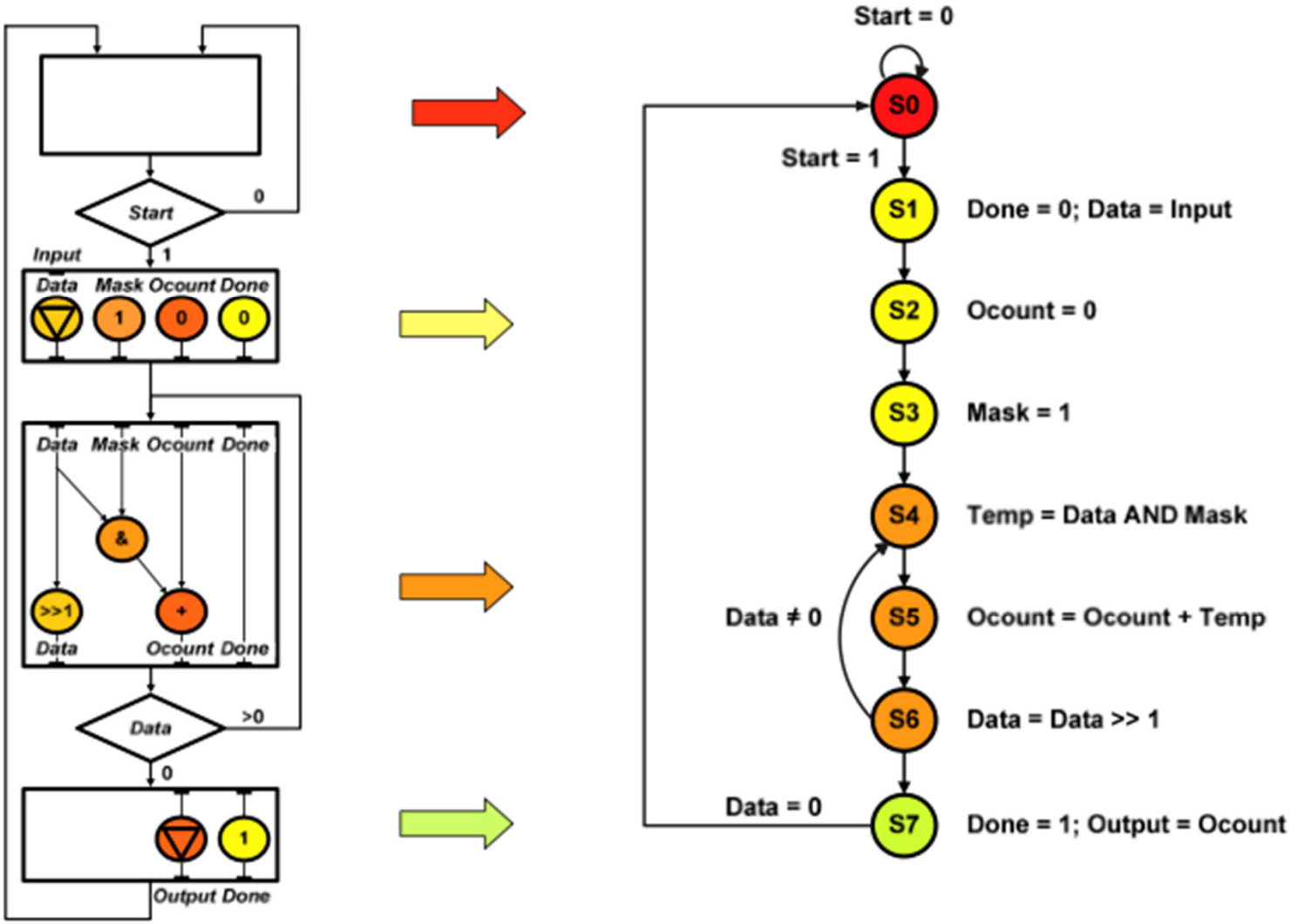
Datenfluss

Übersetzung: Endlicher Automat

- FSMD (Finite State Machine Descript.)
  - Detaillierter als CDFG
  - Zustände können Takt-zyklen darstellen
  - Bedingungen und Befehle: gleichzeitig ausgeführt
  - Alle Befehle in einem Zustand: parallel ausgeführt
  - Steuersignal- und Variablen-zuweisungen: parallel ausg.
- FSMD *beinhaltet Scheduling*
- FSMD gibt *weder Bindung noch Verbindungen* an

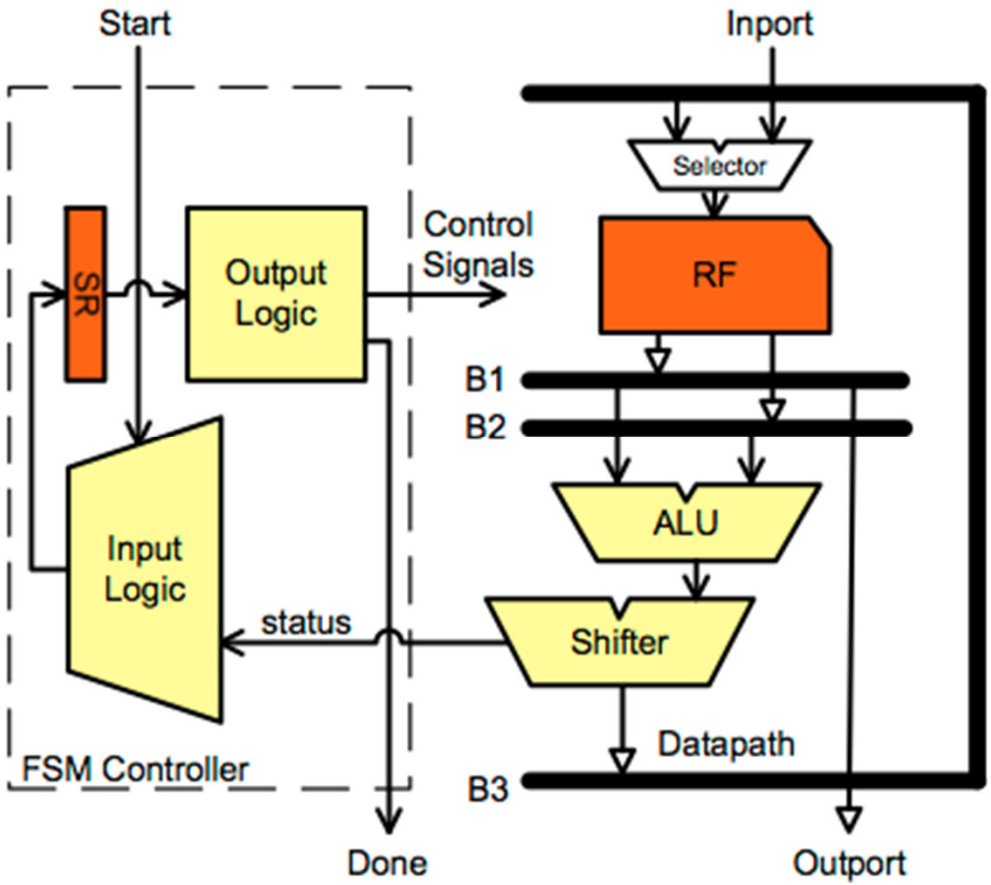


Übersetzung: CDFG und FSMD



Übersetzung: RTL-Spezifikation (1)

- Netzlisten für Steuerung und Datenpfad



Übersetzung: RTL-Spezifikation (2)

- Eingabe-/Ausgabetabellen für Logiksynthese
- RTL-Bibliothek für Netzliste benötigt

Input logic table

Present State	Inputs: Start	Inputs: Data = 0	Next State	Output: Done
S0	0	X	S0	X
S0	1	X	S1	X
S1	X	X	S2	0
S2	X	X	S3	0
S3	X	X	S4	0
S4	X	X	S5	0
S5	X	X	S6	0
S6	X	0	S4	0
S6	X	1	S7	0
S7	X	X	S0	1

Output logic table

State	RF Read Port A	RF Read Port B	ALU	Shifter	RF selector	RF Write	Outport
S0	X	X	X	X	X	X	Z
S1	X	X	X	X	Inport	RF[0]	Z
S2	RF[2]	RF[2]	subtract	pass	B3	RF[2]	Z
S3	RF[2]	X	increment	pass	B3	RF[1]	Z
S4	RF[0]	RF[1]	AND	pass	B3	RF[3]	Z
S5	RF[2]	RF[3]	add	pass	B3	RF[2]	Z
S6	RF[0]	X	pass	shift right	B3	RF[0]	Z
S7	RF[2]	X	X	X	X	disable	enable

Bindung (Speicher):
RF[0] = Data
RF[1] = Mask
RF[2] = Ocount
RF[3] = Temp

Zusammenfassung

- Übersetzung
  - AST
  - CDFG und FSMD
- Parallelisierung
  - reale und falsche Abhängigkeiten
  - SSA
- Übersetzungs- und Synthesefluss
  - AST
  - Zwischendarstellung

Literatur

① Giovanni De Micheli:
Synthesis and Optimization of Digital Circuits
McGraw-Hill 1994

② Stephen A. Edwards:
The Challenges of Synthesizing Hardware from C-Like Languages
IEEE Design & Test Volume 23 Issue 5, September 2006

③ David W. Wall:
Limits of instruction-level parallelism
Proceedings of ASPLOS-IV, 1991