Kompaktierungsmaßnahmen

- Konstantenauswertung (constant propagation)
  - Ein Ausdruck, der nur aus Konstanten besteht, wird zur Übersetzungszeit vorausberechnet und durch die resultierende Konstante ersetzt.

```
constant size: integer := 8;
variable i: integer;
...
i := 2 * size;
```

```
constant size: integer := 8;
variable i: integer;
...
i := 16;
```

- Codeverschiebung (code motion)
  - Ausdrücke, die unabhängig von der Anzahl der Schleifendurchläufe immer den gleichen Wert ergeben, werden aus dem Schleifenrumpf herausgezogen und vor den Schleifenkopf gesetzt.

```
while (i <= limit*n) loop
. . .
end loop;
```

```
t := limit*n;
while (i <= t) loop
. . .
end loop;
```

Kompaktierungsmaßnahmen

- Verringerung der Operationskosten
  - Eine Multiplikation mit einem konstanten Wert bzw. eine Division durch einen konstanten Wert lassen sich oft in eine Folge von Shift-Operationen zerlegen.

```
x * 8 = x << 3
```

$x * 7 = x * (2^2 + 2^1 + 2^0) = x << 2 + x << 1 + x$

- Faktorisierung gemeinsamer Teilausdrücke (common subexpression elimination)
  - Mehrfach auftretende Teilausdrücke lassen sich zusammenfassen, indem ein Ausdruck nur einmal berechnet und das Ergebnis ggf. in einer Hilfsvariable bereitgestellt wird.

```
u := u - 5*x*u*dx + 3*y*dx;
y := y + x*dx;
```

```
t1 := x*dx;
u := u - 5*u*t1 + 3*y*dx;
y := y + t1;
```

Kompaktierungsmaßnahmen

- Entfernen unnötigen Codes (dead code elimination)
  - Anweisungen in einem Programm, deren Ausführung aufgrund des Steuerflusses nie zustande kommt, können eliminiert werden.

- Algebraische Transformationen
  - Unter diesen Transformationen sind solche interessant, die Ausdrücke vereinfachen bzw. kostspielige Operationen durch effizientere Operationen ersetzen. Beispielweise können die folgenden Ausdrücke y=x+0, y=x*1, y=x-0, y=x/1 eliminiert und durch die einfache Zuweisung y=x ersetzt werden.

```
a := 1 + 8 * b + 5
= (1 + 5) + (8 * b)
= 6 + (b * 8)
= 6 + (b << 3)
```

Kompaktierungsmaßnahmen

- Anwendung einiger Kompaktierungsmaßnahmen

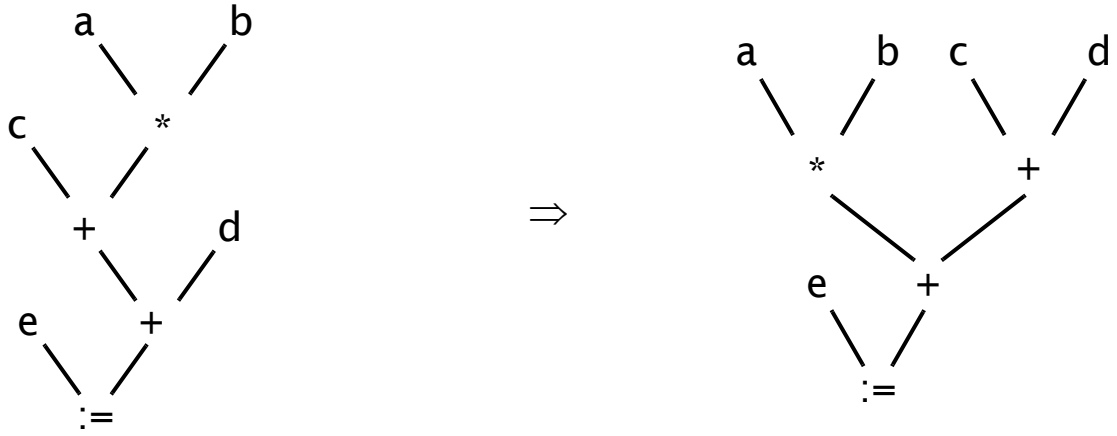
```
process
  constant size: integer := 10;
  variable x, y, w, z: integer;
  variable a, b: integer;
begin
  x := 5 * size;
  y := x + z;
  w := w * (x + z);
  if x=50 then
    a := a + 1;
    b := 0;
  else
    a := 0;
    b := b + 1;
  end if;
  ...
end process;
```

```
process
  constant size: integer := 10;
  variable x, y, w, z: integer;
  variable a, b: integer;
begin
  x := 50;
  y := x + z;
  w := w * y;

  a := a + 1;
  b := 0;




  ...
end process;
```

Maßnahmen zur Steigerung der Parallelität

- Höhenreduktion (tree high reduction).
  - Das Ziel dieser Programmtransformation ist es, unter Ausnutzung algebraischer Transformationen, wie Assoziativität und Kommutativität, einen nicht ausgeglichenen Syntaxbaum eines arithmetischen Ausdruckes in einen ausgeglichen umzuwandeln, dessen Tiefe bei n Operationen im besten Fall log2(n) beträgt. Bei paralleler Auswertung des Ausdrucks ist dann die Zeit proportional zur Tiefe des Baumes.

e := c + a * b + d => (a * b) + (c + d)



Maßnahmen zur Steigerung der Parallelität

- Schleifenabrollen (loop unrolling)
  - Eine Schleife, bestehend aus dem Schleifenkopf und Schleifenrumpf, wird durch das Hintereinanderkopieren ihres Schleifenrumpfes abgerollt.
  - Beim vollständigen Abrollen einer Schleife (Auflösung der Schleife) entfällt der Zyklus für den Wiedereintritt in die Schleife.
  - In der Schaltungssynthese entfallen dann auch die Komponenten, die zur Bestimmung des Schleifenabbruchs notwendig sind.
  - Eine Voraussetzung für ein erfolgreiches Schleifenabrollen ist die Kenntnis über die Anzahl der Schleifeniterationen zur Übersetzungszeit, was in der Regel nur bei for-Schleifen der Fall ist.

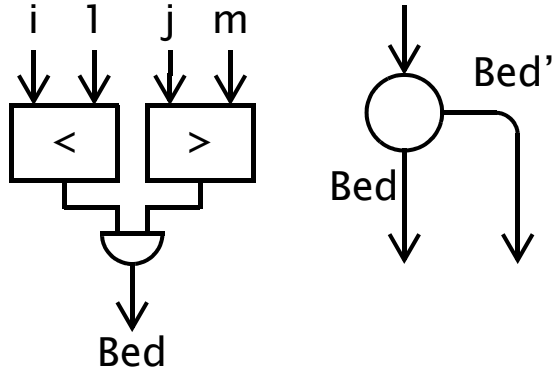
```
begin
  for i in a'range loop
    a(i) <= b(i) and not(c(i));
  end loop;
  . . .
end;
```

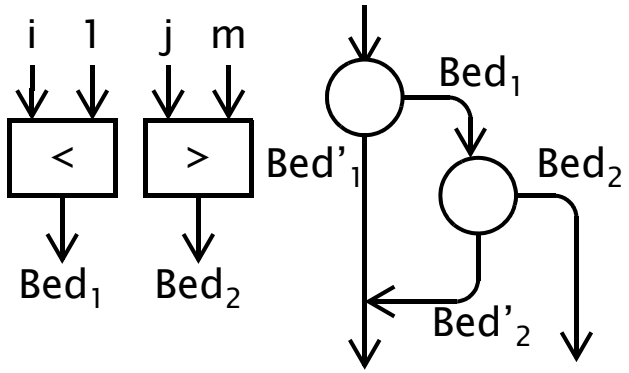
```
begin
  a(1) <= b(1) and not(c(1));
  a(2) <= b(2) and not(c(2));
  a(3) <= b(3) and not(c(3));
  a(4) <= b(4) and not(c(4));
  . . .
end;
```

Maßnahmen zur Steigerung der Parallelität

- Methode der teilweisen Berechnung
  - berechnet logische Ausdrücke analog zu arithmetischen Ausdrükken, d.h. komplett durch die Bewertung aller Operanden.
  - Teilergebnisse werden mit Hilfe logischer Operatoren verknüpft.
- Methode der teilweisen Berechnung (short circuit).
  - berechnet logische Ausdrücke bis das Ergebnis feststeht, wobei die Teilausdrücke nicht umgeordnet werden.
  - Bei dieser Methode werden anstelle logischer Operatoren bedingte Zustandsübergänge verwendet.

```
if (i<1 and j>m) then
   ...
end if;
```





Datenabhängigkeiten

- High-Level-Synthese:
  - Ziel: Schaltungsbeschreibung auf der Registertransferebene (Komponenten arbeiten a priori parallel).
  - Ausgangspunkt: algorithmische Verhaltensbeschreibung (enthält i.d.R. keine Syntaxkonstrukte zur Notation paralleler Anweisungen).
  - Schlußfolgerung: Es ist zweckmäßig, vor dem HL-Syntheseprozeß die algorithmischen Spezifikationen zu parallelisieren.
- Die Möglichkeit, Anweisungen in einem Programm parallel ausführen zu können, wird durch dort existierende Steuer- und Datenabhängigkeiten beschränkt.
- Verzweigungen in Programmen, die durch Steuerflußanweisungen (z.B. while, if, for) entstehen, verändern bei deren Abarbeitung den sequentiellen Ablauf in Abhängigkeit von einer Bedingung.

Datenabhängigkeiten

- Steuerflußgraph
  - ist ein gerichteter Graph, dessen Knoten Basisblöcke sind und dessen Kanten den Transfer zwischen den Basisblöcken darstellen. Ein ausgezeichneter Knoten wird als Startknoten bezeichnet und enthält den Basisblock mit der ersten Anweisung im Programm.

- Basisblock
  - ist eine Folge fortlaufender Anweisungen, in die der Steuerfluß am Anfang eintritt und die er am Ausgang verläßt, ohne daß er dazwischen anhält oder verzweigt.

- Grundsätzlich gilt es:
  - zwei Anweisungen können parallel abgerabeitet werden, wenn es auf die Reihenfolge ihrer Bearbeitung nicht ankommt. Die parallele Ausführung von Anweisungen innerhalb eines Basisblocks wird durch dort existierende Datenabhängigkeiten beschränkt.

Datenabhängigkeiten

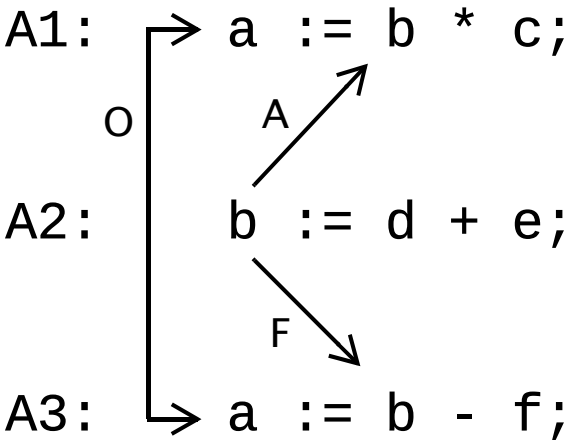
- Datenabhängigkeiten werden als Relationen zwischen zwei verschiedenen Operationen OP_i und OP_j definiert. Dabei wird vorausgesetzt, daß die beiden Operationen zu demselben Basisblock gehören und daß die Operation OP_i vor der Operation OP_j im Basisblock steht.

- Man unterscheidet zwischen drei Typen von Datenabhängigkeiten.
- *Flußabhängigkeit* (echte/essentielle Datenabhängigkeit, flow dependence, RAW - read after write dependence)
  - liegt zwischen zwei Operationen OP_i und OP_j vor, wenn die Operation OP_j das Ergebnis der Operation OP_i als Operand benötigt (notiert als OP_j flow OP_i).
    a := b * c; ... e := a + d;

Datenabhängigkeiten

- *Gegenabhängigkeit* (anti dependence, WAR - write after read dependence)
  - liegt zwischen zwei Operationen OP_i und OP_j vor, wenn die Operation OP_j eine Variable bzw. ein Register beschreibt, die bzw. das von der Operation OP_i gelesen wird (notiert als OP_j anti OP_i).
    a := b * c; ... b := d + e;
- *Schreibabhängigkeit* (output dependence, WAW - write after write dependence)
  - liegt zwischen zwei Operationen OP_i und OP_j vor, wenn beide Operationen jeweils dieselbe Variable bzw. dasselbe Register als Ziel haben (notiert als OP_j output OP_i).
    a := b * c; ... a := d + e;
  - Die Operation OP_j darf die Operation OP_i nicht „überholen", denn sonst enthält die Zielvariable bzw. das Zielregister das Ergebnis der Operation OP_i und nicht den korrekten Wert, der durch die Operation OP_j berechnet wird.

Datenabhängigkeiten



- Wodurch ergeben sich Gegen- und Schreibabhängigkeiten?
  - speicherplatzsparende Wiederverwendung von Variablen (bei höheren Programmiersprachen) bzw. Registern (in der Assembler-programmierung).
- Wie lassen sich Gegen- und Schreibabhängigkeiten eliminie-ren?
  - durch das Prinzip *der einmaligen Zuweisung* (single assignment) $\Rightarrow$ durch die Verwendung neuer Variablennamen

Datenabhängigkeiten

- Funktion OUT(OP) bestimmt die Ausgangsmenge der Variablen, die durch die Operation OP aktualisiert werden,

- Funktion IN(OP) bestimmt die Eingangsmenge der Variablen, die für die Auswertung der Operation OP benötigt werden.

- Bernsteinsche Bedingungen:

  $OUT(OP_i) \cap IN(OP_j) \neq \emptyset \Leftrightarrow OP_j$ *flow* OP_i

  $IN(OP_i) \cap OUT(OP_j) \neq \emptyset \Leftrightarrow OP_j$ *anti* OP_i

  $OUT(OP_i) \cap OUT(OP_j) \neq \emptyset \Leftrightarrow OP_j$ *output* OP_i

Datenabhängigkeiten

- Beispiel

```
A1:  a := b * c;
A2:  b := d + e;
A3:  a := b - f;
```

OUT(A1) ∩ IN(A2) =
OUT(A1) ∩ IN(A3) =
OUT(A2) ∩ IN(A3) =

IN(A1) ∩ OUT(A2) =
IN(A1) ∩ OUT(A3) =
IN(A2) ∩ OUT(A3) =

OUT(A1) ∩ OUT(A2) =
OUT(A1) ∩ OUT(A3) =
OUT(A2) ∩ OUT(A3) =

Datenabhängigkeiten

- Beispiel

```
A1:  a = b + c;
A2:  d = a * e;
A3:  f = d - g;
```

OUT(A1) $\cap$ IN(A2) =

OUT(A2) $\cap$ IN(A3) =

OUT(A1) $\cap$ IN(A3) =

- implizite Flußabhängigkeit

$$OP_j \; \textit{iflow} \; OP_i = \begin{cases} ((OP_j \; \textit{flow} \; OP_k) \wedge (OP_k \; \textit{flow} \; OP_i)) \text{ oder} \\ ((OP_j \; \textit{flow} \; OP_k) \wedge (OP_k \; \textit{iflow} \; OP_i)) \end{cases}$$

Datenabhängigkeiten

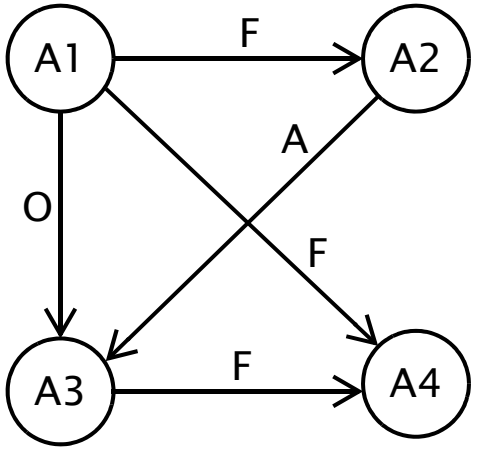
- Eliminierung der Gegen- und Schreibabhängigkeiten
  - Prinzip der einmaligen Zuweisung $\Rightarrow$ neue Variablen einführen
- Eliminierung der Gegenabhängigkeiten
  - neben einer neuen Variable auch eine neue Zuweisungen an diese Variable einführen
- Die Einführung einer neuen Variable bedeutet:
  - der Namen der Variable (welche die Gegen- bzw. Schreibabhängigkeiten verursacht hat) wird durch den Namen der neuen Variable ersetzt.
  - die Umbenennung einer Variable zieht automatisch die Umbenennungen der jeweiligen Verwendung dieser Variable im Basisblock nach sich.
  - die Umbenennungen dürfen keine Auswirkungen außerhalb eines Basisblocks haben $\Rightarrow$ die letzte Zuweisung an die Variable nicht umbenannt werden darf.

Datenabhängigkeiten

- Beispiel

```
A1:  a := b + c;
A2:  d := a * m;
A3:  a := e - f;
A4:  g := a / 5;
```

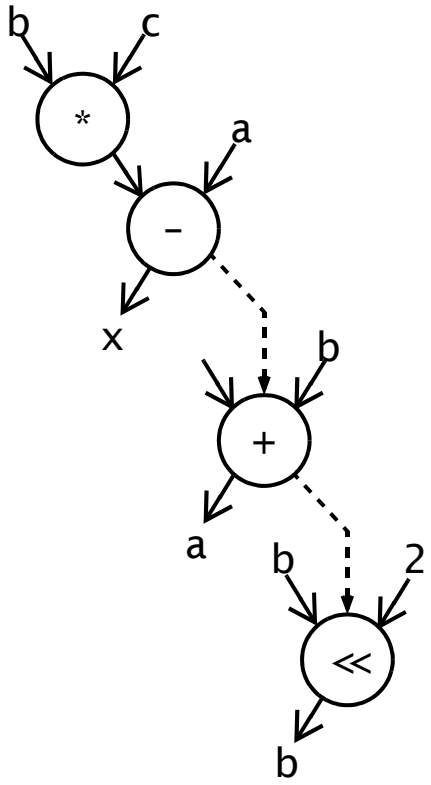
```
A1:

A2:

A3:

A4:
```

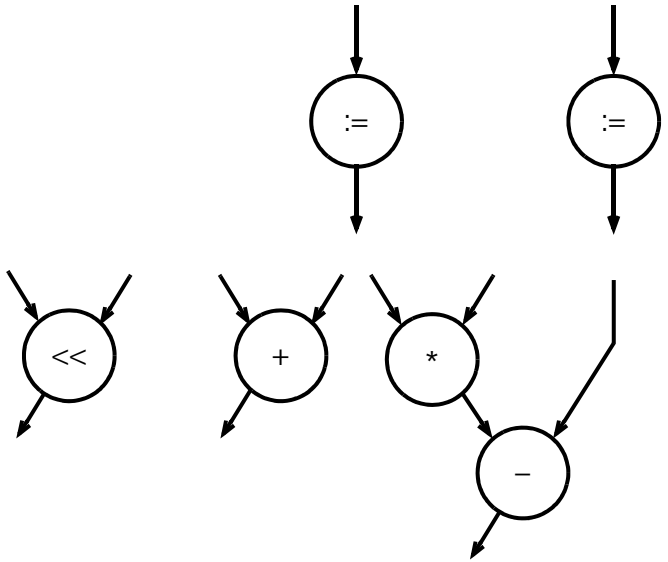


Datenabhängigkeiten

- Multizustandsoperationen

```
A1:   x = b * c - a;
A2:   a = c + b;
A3:   b = b << 2;
```





Ablaufplanung

- Einführendes Beispiel

Punkt	Aufgabe	Zeitdauer	vorhergehende Aufgaben
1	Grundstück vorbereiten	3	-
2	Baumaterial anliefern	2	-
3	Baugrube ausheben	2	1, 2
4	Fundamente gießen	2	3
5	Mauern	7	4
6	Dachstuhl bauen	3	5
7	Dach decken	1	6
8	Außeninstallationen	3	4
9	Außenputz	2	7, 8
10	Fenster einsetzen	1	7, 8
11	Decken einziehen	3	5
12	Garten anlegen	4	9, 10
13	Inneninstallationen	5	11
14	Wände isolieren	3	10, 13
15	Maler- und Tapezierarbeiten	3	14
16	Umziehen!	5	15

Ablaufplanung: Critical-Path-Methode

- Gesucht: die kürzestmögliche Gesamtdauer eines Projektes sowie Zeitpunkte, in denen einzelne Teilaufgaben begonnen werden sollen.
  - Nebenbedingung: gewisse Teilaufgaben können nicht vor der Beendigung anderer Teilaufgaben begonnen werden.

- Lösung: Critical-Path-Methode
  - Bestimmung des längsten Weges (des sog. kritischen Pfades) in einem gerichteten zyklenfreien Graphen.
  - Jeder der N Teilaufgaben eines Projektes wird ein Punkt $i \in \{1,...,N\}$ eines gerichteten Graphen zugeordnet.
  - Dabei sei (i, j) genau dann eine Kante von G, wenn die Teilaufgabe i vor dem Beginn der Teilaufgabe j beendet sein muß.
  - Jeder Kante (i, j) wird die Länge w_{ij} = Zeitdauer der Teilaufgabe i zugeordnet.

Ablaufplanung: Critical-Path-Methode

- Es wird ein neuer Punkt s (Start des Projektes) hinzugefügt, sowie Kanten (s, v) für alle Knoten v, für die $d_{in}(v)=0$ gilt, wobei $d_{in}(v)$ die Anzahl der Kanten mit Endpunkt im Knoten v bestimmt.
- Alle Kanten (s, v) erhalten die Länge $w_{sv}=0$.

- Es wird ein neuer Punkt z (Ende des Projektes) hinzugefügt, sowie Kanten (v, z) für alle Knoten v, für die $d_{out}(v)=0$ gilt, wobei $d_{out}(v)$ die Anzahl der Kanten mit Anfangspunkt im Knoten v bestimmt.
- Alle Kanten (v, z) erhalten die Länge w_{vz}.
- Mit t_i wird der früheste Zeitpunkt bezeichnet, in dem die Aufgabe i begonnen werden kann.
- Da alle der Teilaufgabe i direkt vorhergehenden Teilaufgaben bereits beendet sein müssen, läßt sich die CPM mit folgendem Gleichungssystem beschreiben:
- $t_s = 0$ und $t_i = \max \{t_k + w_{ki}: (k, i) \text{ eine Kante in } G\}$

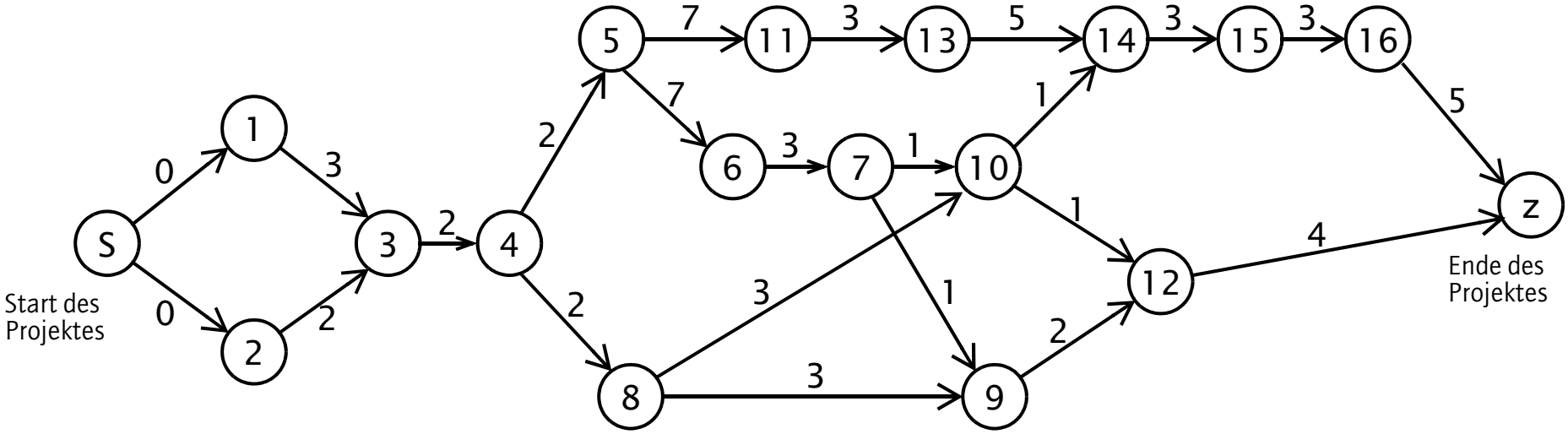
Ablaufplanung: Critical-Path-Methode

- Die Minimaldauer T des ganzen Projektes ist also die Länge t_z des längsten Weges von s nach z.
- Soll das Projekt tatsächlich im Zeitpunkt T beendet sein, so läßt sich der späteste Zeitpunkt T_i, in dem die Teilaufgabe i begonnen werden kann, mit folgendem Gleichungssystem beschreiben
- $T_z = T$ und $T_i = \min \{T_j - w_{ij}: (i, j)$ eine Kante in G$\}$

- Der Spielraum (Pufferzeit, Zeitrahmen) mi, der für den Beginn einer Teilaufgabe i vorhanden ist, wird definiert als $m_i = T_i - t_i$.

- Alle Teilaufgaben i mit $m_i=0$ werden als *kritisch* bezeichnet, da sie exakt im Zeitpunkt $T_i=t_i$ begonnen werden müssen, wenn nicht die gesamte Projektdauer verlängert werden soll.

Ablaufplanung: Critical-Path-Methode



	s	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	z
t_i																		
T_i																		
m_i																		

Anwendung in der High-Level-Synthese

- Ausgangspunkt der HL-Synthese:
  - algorithmische Verhaltensbeschreibung (Steuer-/Datenflußgraphen),
  - Syntheseparameter: Dauer der Taktperiode (Zustandsdauer), maximale Schaltungskomplexität,
  - Modulenbibliothek mit technischen Parametern von Registertransferkomponenten:
    - Register (Verarbeitungsbreite in Bits, Vorbereitungszeit (setup time), Haltezeit (hold time), Verzögerung (clock-to-output time), Komplexität),
    - Multiplexer (Verarbeitungsbreite in Bits, Anzahl der selektierbaren Eingänge, Verzögerung, Komplexität),
    - Funktionseinheiten (Verarbeitungsbreit in Bits, Verzögerung, Verknüpfungsvorrat, Vorzeichenkennung, Organisationsform: Schaltnetze, Module in Fließbandorganistion – Anzahl der Fließbandstufen, Module mit eigenem Steuerwerk – Synchronisation mit Handshake-Signalen: Start/Ready),

Anwendung in der High-Level-Synthese

- Resultat der HL-Synthese
  - strukturelle Schaltungsbeschreibung auf der Registertransferebene:
    - Steuerwerk in der Form eines endlichen Zustandsautomaten,
    - Rechenwerk bestehend aus Registern, multiplexerbasierenden Verbindungen sowie arithmetischen und logischen Funktionseinheiten.

- Gemeinsame Nutzung von Ressourcen:
  - Hardwareoptimierungstechnik, in der bestimmte Funktionseinheiten einer Schaltung für verschiedene Zwecke genutzt werden, um dadurch den Chipflächenbedarf (Schaltungskomplexität) zu reduzieren.
  - Vor allem bei komplexen arithmetischen Funktionseinheiten, wie z.B. Dividierer, Multiplizierer oder Bussteuerwerk, kann die gemeinsame Nutzung von Ressourcen durch Zeitmultiplexbetrieb erhebliche Einsparungen der Chipfläche bringen.

Anwendung in der High-Level-Synthese

- Die gemeinsame Nutzung von Ressourcen ergibt sich aus zwei Aspekten:
  - Eine algorithmische Spezifikation enthält normalerweise Steuerflußanweisungen, welche die Reihenfolge in der Ausführung von Operationen festlegen. Bedingte Verzweigungen in Programmen lassen den Steuerfluß in zwei Pfade verzweigen, die nie gleichzeitig aktiv sind.
  - Stehen für die HL-Synthese weniger arithmetische Funktionseinheiten zur Verfügung als gleichzeitig ausführbare Operationen im Datenflußgraphen, so müssen Ressourcen einer Schaltung ohnehin gemeinsam genutzt werden.

Anwendung in der High-Level-Synthese

- Gemeinsame Nutzung von Ressourcen impliziert den Einsatz von Multiplexern, die vor die gemeinsam genutzten Ressourcen einzubauen sind
  $\Rightarrow$ Änderungen in der Struktur des Rechenwerks

- Multiplexer sind keine verzögerungsfreien Registertransferkomponenten
  $\Rightarrow$ verlängerte Signallaufzeiten in Datenpfaden um die Verzögerungen der Multiplexer
  $\Rightarrow$ Einfluß auf das Steuerprogramm
  $\Rightarrow$ der Synthesevorgang ist ggf. mit der Berücksichtigung der eingebauten Multiplexern zu wiederholen.

Anwendung in der High-Level-Synthese

- Die gemeinsame Nutzung von Schaltnetzen, die für die booleschen Operationen AND, OR, XOR und NOT sowie für die arithmetischen ganzzahligen Operationen Inkrementieren und Dekrementieren, bereit gestellt werden, ist nicht sinnvoll, denn die Komplexität eines Multiplexers übersteigt die Komplexität eines Schaltnetzes.
- Die gemeinsame Nutzung von Schaltnetzen für die arithmetischen ganzzahligen Operationen Addition und Subtraktion sowie für ganzzahligen Vergleichsoperationen ist unter bestimmten Umständen nicht sinnvoll, denn die Komplexität eines Multiplexers ist mit der Komplexität eines Addierer- bzw. Subtrahierer-Schaltnetzes (Carry-Ripple-Technik) vergleichbar.

Anwendung in der High-Level-Synthese

- Unter der Ablaufplanung versteht man in der High-Level-Synthese die Zuordnung von Operationen einer algorithmischen Spezifikation zu Steuerschritten.
- Die Ablaufplanung erfolgt unter Berücksichtigung der Datenabhängigkeiten in der algorithmischen Spezifikation und unter Einhaltung von Randbedingungen:
  - die Anzahl der benötigten Ressourcen wird bei einer vorgegebenen Anzahl der Steuerschritte minimiert, wobei die Anzahl der Steuerschritte nicht überschritten werden darf,
  - die Anzahl der Steuerschritte wird bei ausreichend vorhandenen Ressourcen minimiert.
- Voraussetzung für die HL-Synthese: die Art und die Anzahl der zur Verfügung stehenden Module muß zur Beginn der Ablaufplanung bekannt sein.

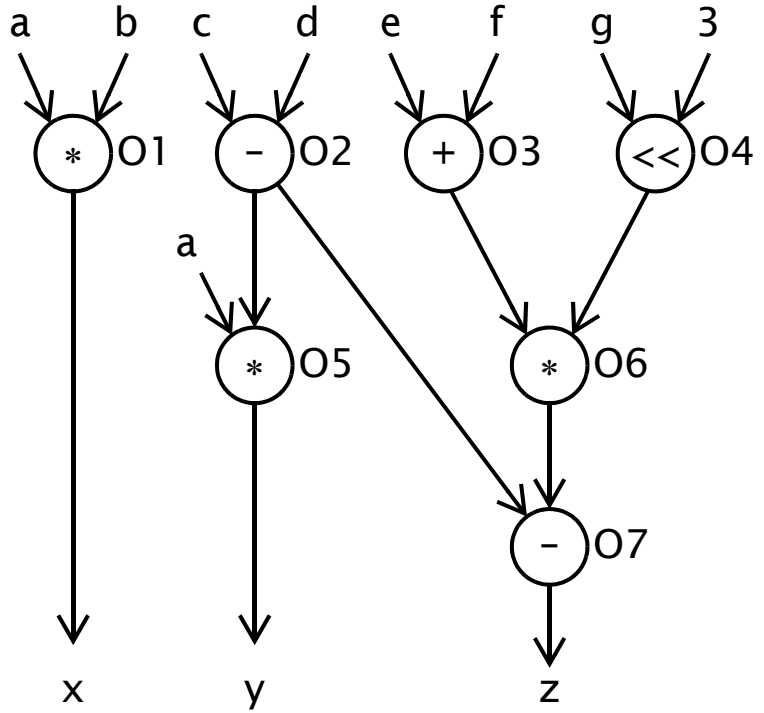
Anwendung in der High-Level-Synthese

- Annahme: eine Operation wird in einem Steuerschritt ausgeführt.
  - Für die Dauer eines Steuerschrittes = Taktperiode wird oft die Verzögerung der langsamsten Funktionseinheit inklusive eventueller zusätzlicher Verzögerungen zugrunde gelegt.
  - Bei Funktionseinheiten mit kürzerer Bearbeitungszeit bleibt dadurch ein Teil der Taktperiode ungenutzt.
  - Dieses Problem kann durch *Multizustandsoperationen* (multicycle-operations) und/oder durch die *Verkettung* (chaining) von Operationen gelöst werden.

- Beispiel

```
x := a * b;
y := a * (c - d);
z := (c - f) - (e + f) * (g << 3);
```

- Zur Verfügung stehen:
  - MUL mit Operation: *
  - ALU mit Operationen: +, – und <<



Ablaufplanungsalgorithmen

- ASAP-/ALAP-Algorithmen
  - ASAP-Zuordnungskriterium (as soon as possible):
    Operationen eines Datenflußgraphen *ohne Vorgänger* haben die höchste Priorität. Sie werden zum frühestmöglichen Zeitpunkt ausgeführt.
  - ALAP-Zuordnungskriterium (as late as possible):
    Operationen eines Datenflußgraphen *ohne Nachfolger* haben die höchste Priorität. Sie werden zum spätestmöglichen Zeitpunkt ausgeführt
  - Die Zustandszuordnung der Operationen erfolgt in der Reihenfolge ihrer Positionen in der Prioritätenliste.
  - Operationen mit gleicher werden willkürlich priorisiert
    z.B. durch die räumliche Stellung im Graphen, indem weiter links stehende Operationen bevorzugt werden.

Ablaufplanungsalgorithmen

- ASAP-/ALAP-Algorithmen
  - Die Operation an der obersten Position wird genau dann dem aktuellen Zustand zugeordnet, wenn alle ihre direkten Vorgänger zugeordnet sind und noch ein unbenutztes Modul für ihre Ausführung zur Verfügung steht.
  - Die zugeordnete Operation wird aus der Prioritätenliste entfernt.
  - Sind alle Module im betrachteten Zustand ausgelastet, fährt der Algorithmus mit dem nächsten Zustand fort.

Ablaufplanungsalgorithmen

- Zustandsgenerierung nach dem ASAP-Scheduling-Algorithmus

Prioritätenliste vor der Zuordnung	MUL	ALU	Zustand

Ablaufplanungsalgorithmen

- Zustandsgenerierung nach dem ALAP-Scheduling-Algorithmus

Prioritätenliste vor der Zuordnung	MUL	ALU	Zustand

Ablaufplanungsalgorithmen

- Nachteil der ASAP-/ALAP-Algorithmen:
  - Priorisierung von Operationen, die auf den zeitlich längsten Ketten im Graphen (auf kritischen Pfaden) liegen, bleibt unberücksichtigt.

- List-Scheduling-Algorithmus:
  - Weiterentwicklung der ASAP-/ALAP-Verfahren mit beschränkten Ressourcen und mit einem globalen Prioritätskriterium.
  - Zu Beginn des Verfahrens werden Operationen eines Datenflußgraphen mit Hilfe der Algorithmen ASAP und ALAP topologisch sortiert. Auf diese Wiese werden Ausführungszeitpunkte jeder Operation festgelegt.

Ablaufplanungsalgorithmen

- List-Scheduling-Algorithmus:
  - Für jede Operation wird eine Priorität nach einem der folgenden Kriterien berechnet:
    - die Beweglichkeit (mobility) gibt an, innerhalb welchen Zeitintervalls eine Operation ausgeführt werden muß, ohne die Gesamtzahl der Verarbeitungsschritte zu verändern.
    - die Länge des kritischen Pfades oder
    - die Anzahl der Nachfolgeroperationen.

Operationen	O1	O2	O3	O4	O5	O6	O7
Mobilität	2	1	0	0	1	0	0
Pfadlänge	1	2	3	3	1	2	1
Nachfolger	0	2	1	1	0	1	0

Ablaufplanungsalgorithmen

- List-Scheduling-Algorithmus:
  - In jedem Iterationsschritt wird für den aktuellen Steuerschritt sowohl eine Menge der freien Komponenten als auch eine Menge der Operationen (sog. Kandidatenmenge) bestimmt, für deren Vorgänger eine Zuordnung zu Steuerschritten bereits vorgenommen ist.
  - Aus der Kandidatenmenge werden Operationen mit der maximalen Priorität (nach einem der Kriterien) ausgewählt und den freien, verfügbaren Komponenten zugeordnet.
  - Dabei werden nur maximal so viel Operationen zugeteilt, wie es die Ressourcenbeschränkungen zulassen.
  - Eine Operation, für deren Ausführung im betrachteten Steuerschritt (Zustand) keine Ressource mehr verfügbar ist, wird in die Liste des Folgezustands verschoben.

Ablaufplanungsalgorithmen

- Zustandsgenerierung nach dem List-Scheduling-Algorithmus mit der maximalen Pfadlänge der Operationen als Prioritätskriterium

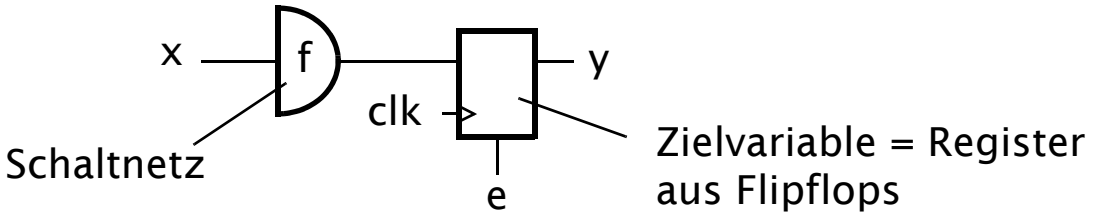
Kandidatenliste vor der Zuordnung	MUL	ALU	Zustand

Hardware-Compilation

- direkte Umsetztung einer Hardwarebeschreibung
  - synchrone Schaltwerke (alle Speicherelemente werden mit demselben Signal getaktet)
  - ausgelegt auf maximale Parallelität (sequentieller Ablauf wird explizit angegeben)
  - das Semikolon (;) bekommt eine semantische Bedeutung – Generierung eines Zustandsflipflops im Steuepfad (in herkömmlichen Programmier-/Beschreibungssprachen hingegen lexikalische/syntaktische Bedeutung – als Trenn-/Abschlußzeichen)
  - kommt ohne Scheduling-Algorithmen aus
  - Zustandskodierung im Steuerpfad mit one-hot-encoding
  - einfaches Datentypkonzept: ein Bit = ein Flipflop,
  - Variablendeklaration: der Name der Variable wird mit dem Ausgangssignal eines Speicherelements (Flipflop, Register) assoziert

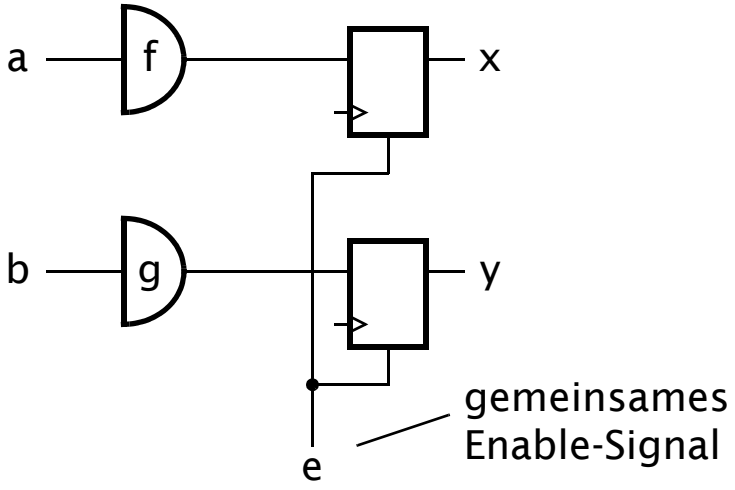
- elementare Zuweisung

  y := f(x)



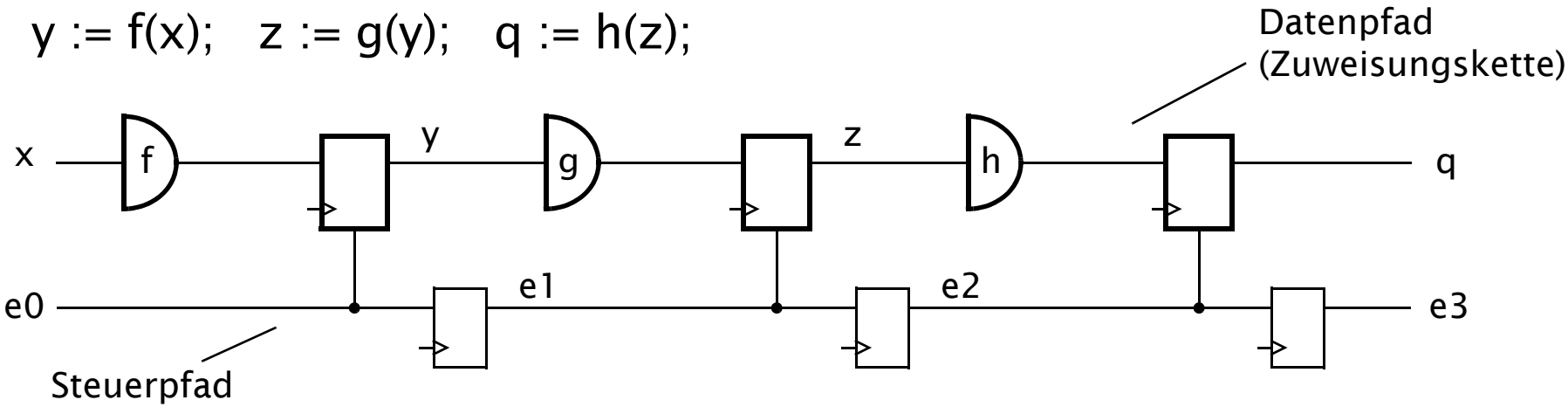
- parallele Zuweisungen

  x := f(a), y := g(b)



- sequentielle Zuweisungen

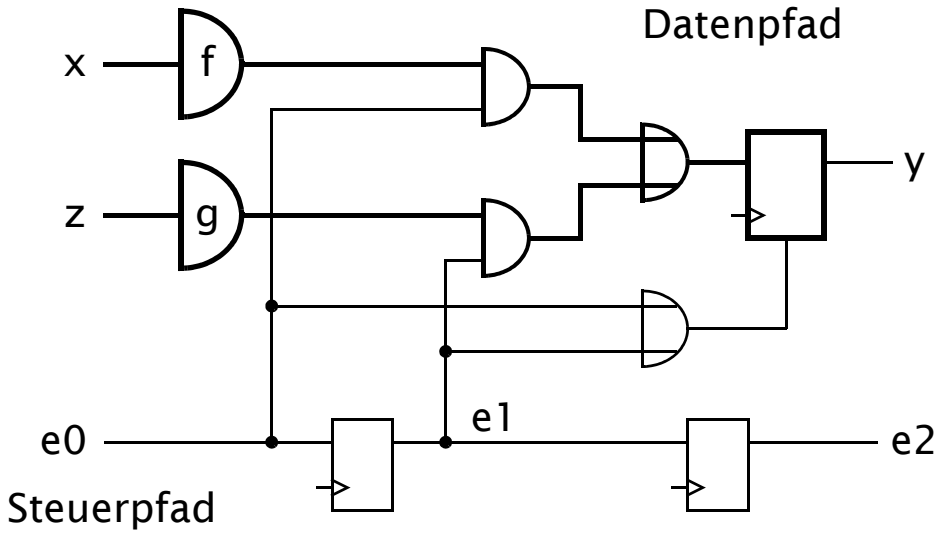
  y := f(x); z := g(y); q := h(z);



e0	e1	e2	e3	x	y	z	q
1	0	0	0	x0	?	?	?
0	1	0	0	x0	f(x0)	?	?
0	0	1	0	x0	f(x0)	g(f(x0))	?
0	0	0	1	x0	f(x0)	g(f(x0))	h(g(f(x0)))

- sequentielle Zuweisungen mit demselben Ziel

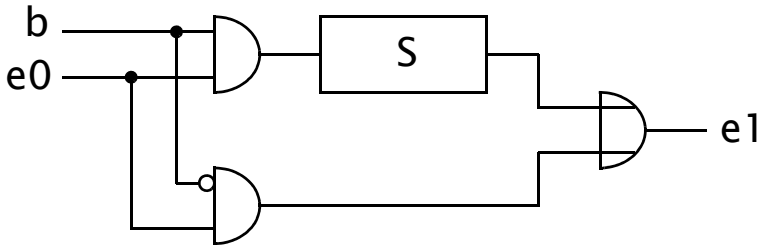
```
y := f(x);
y := g(z);
```



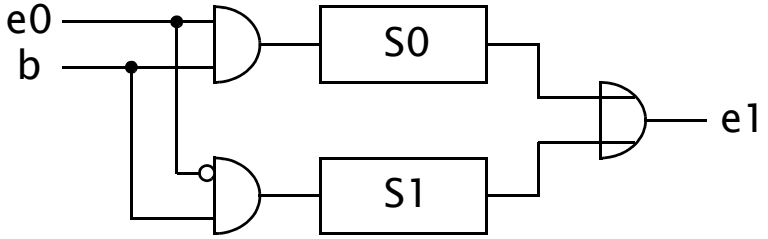
e0	e1	e2	y
1	0	0	?
0	1	0	f(x0)
0	0	1	z(x0)

- bedingte Anweisungen

IF b THEN S END

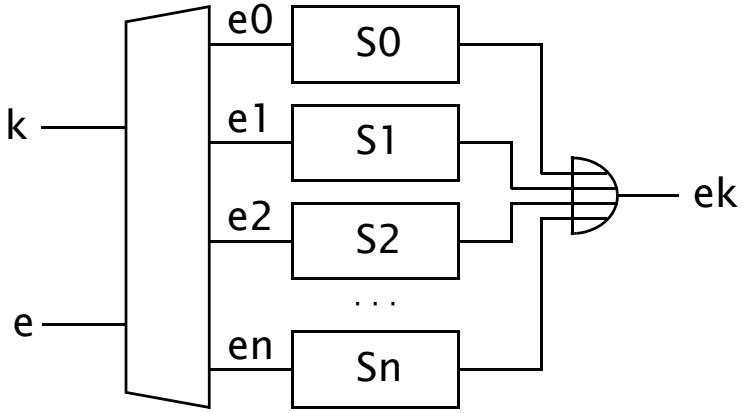


IF b THEN S0 ELSE S1 END



- CASE-Anweisung

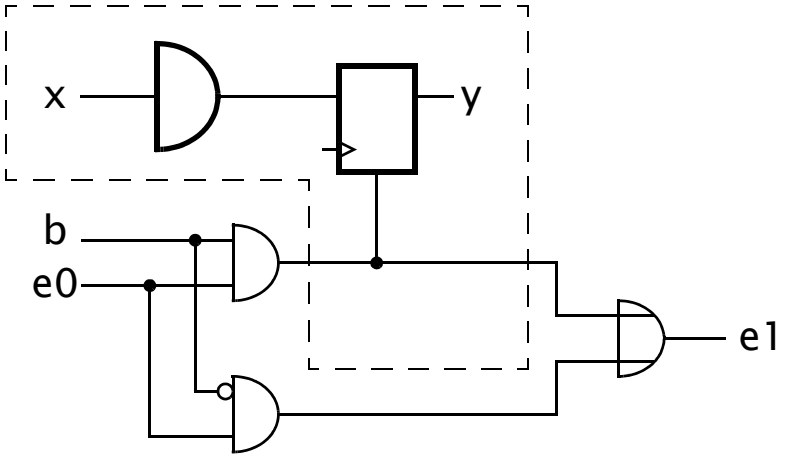
```
CASE k OF
  0: S0
  1: S1
  . . .
  n: Sn
END
```

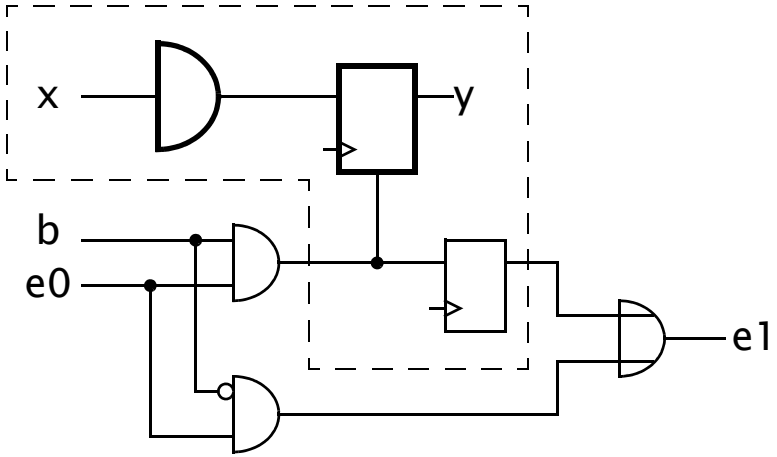


- bedingte Anweisungen – Beispiele

```
IF b THEN
  y := f(x)
END
```

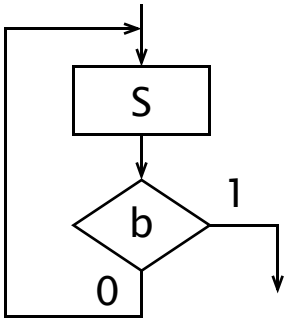
```
IF b THEN
  y := f(x);
END
```

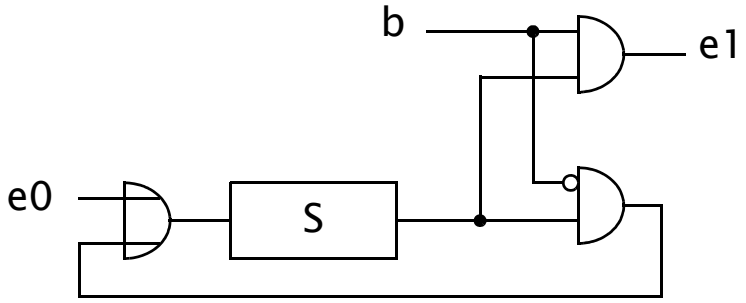




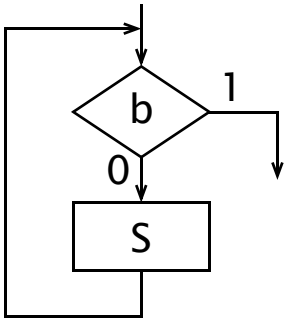
- iterative Anweisungen

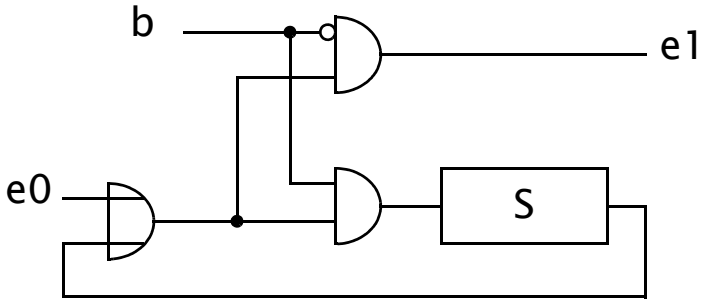
REPEAT S UNTIL b





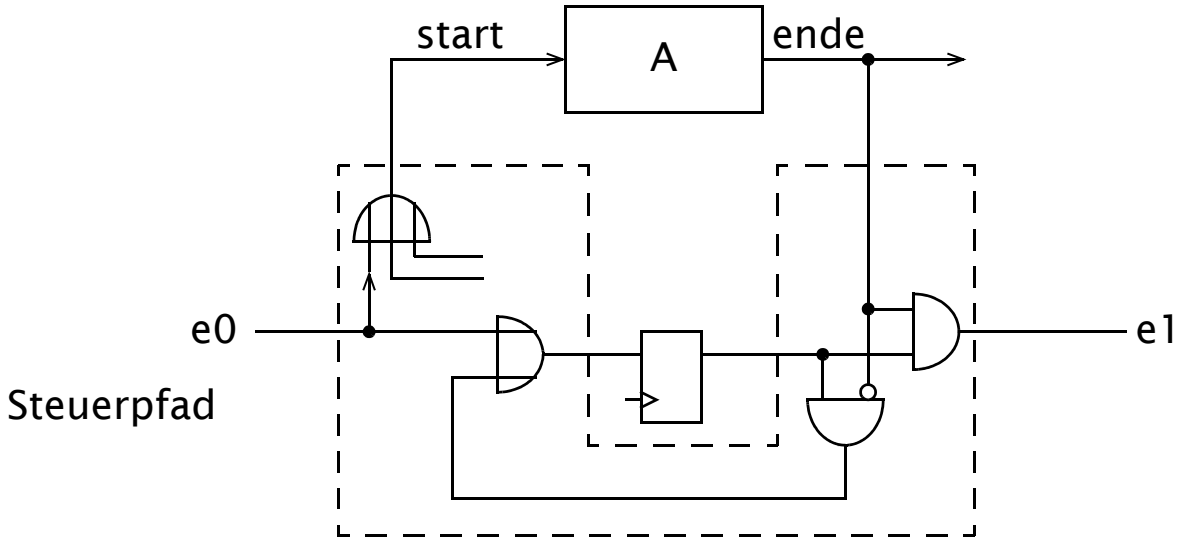
WHILE b DO S END



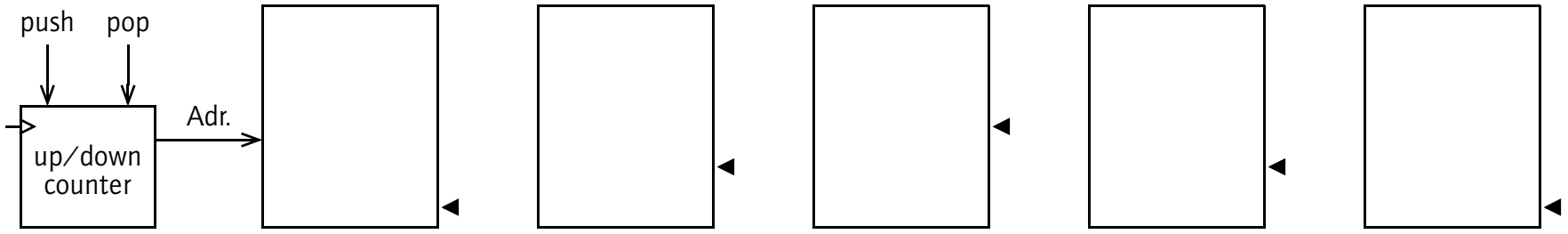


- direkte Unterprogrammaufrufe

  A; B; A; C; B; A;

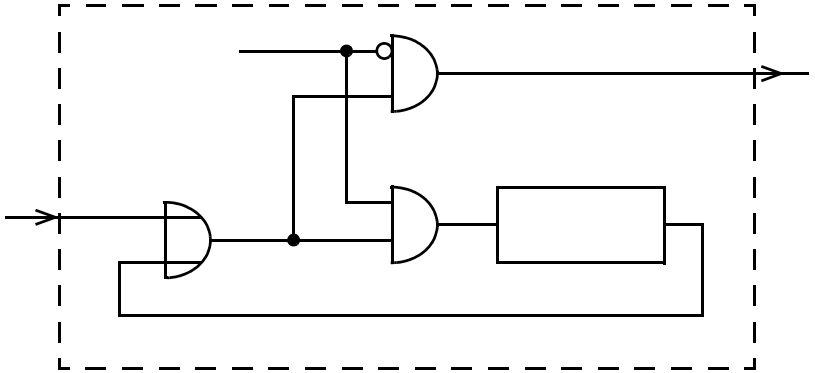


- Unterprogrammaufrufe über Stack
  - Eine Applikation wird in ein Hauptmodul (Hauptprogramm) und in n Submodule (Unterprogramme) aufgeteilt.
  - Jedes Modul bekommt ein Zustandsflipflop zugeordet, die zusammen ein n+1-Bit-Zustandsregister bilden. Ein Zustandsflipflop liefert ein Enable-Signal, mit dem sämtliche Speicherelemente eines Moduls aktiviert werden können.
  - Es werden ein Zähler sowie ein Speicher mit m Worten und mit einer Wortbreite von n+1 Bits bereit gestellt.
  - Der Zähler wird incrementiert, wenn ein Push-Signal aktiviert wird (Unterprogrammaufruf) und dekrementiert, wenn ein Pop-Signal aktiviert wird (Rückkehr aus dem Unterprogramm).



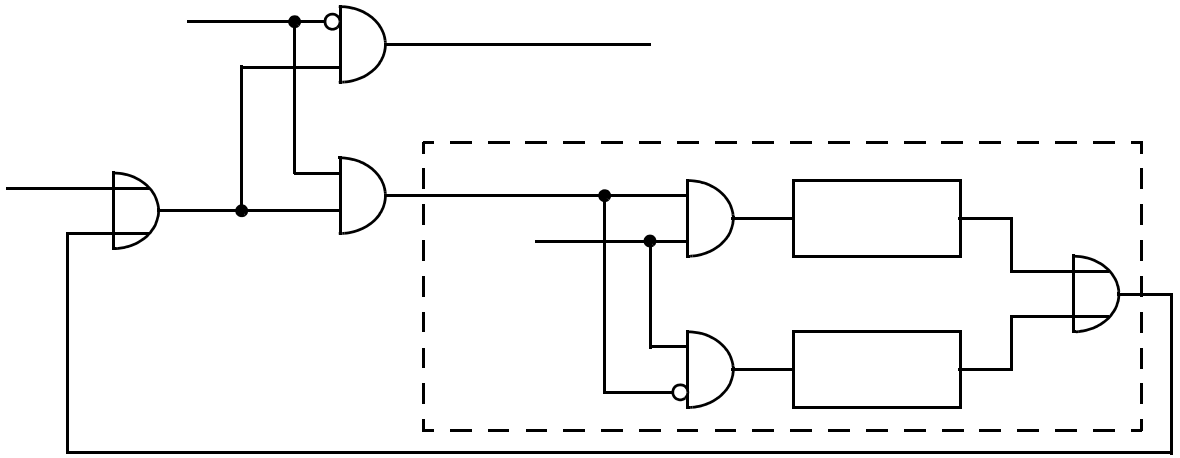
Hardware-Compilation – Beispiel

```
VAR x, y: INTEGER;

WHILE x /= y DO
  IF x < y THEN
    y := y - x;
  ELSE
    x := x - y;
  END
END
```



Hardware-Compilation – Beispiel

```
VAR x, y: INTEGER;

WHILE x /= y DO
   IF x < y THEN
      y := y - x;
   ELSE
      x := x - y;
   END
END
```



Hardware-Compilation – Beispiel