Kapitel 11: Hardware-Spezifikation und -Simulation

Lernziele dieses Kapitels:

1. Behandlung unendlicher Listen mit Hilfe von Laziness
2. Rückgekoppelte Schaltnetze
3. Induktionsbeweise für unendliche Listen
4. Ströme zur Hardwarebeschreibung
5. Sequenzielle Schaltungen
6. Verzögerungsmodellierung
7. Probleme des funktionalen Ansatzes
8. Netzlistengenerierung

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Einschränkungen an das Modell

- Anwendungsgebiet: rückgekoppelte Schaltnetze
- Abstraktionsebenen
  - Gate Level: logisches NOT, AND, OR; Flipflops
  - Register-Transfer Level: Register, boolesche Terme
  - Functional Level: n-bit Addierer, Multiplexer
- benutzte Eigenschaften
  - Diskretisierung der
    - Zeit: Intervalle gleicher Länge (z.B. 10 pro Takt)
    - Wertemenge: { 0, 1, hochohmig (Bus), unbekannt }
  - gerichteter Signalfluss

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Modellierung

- Signal: zeitliche Folge von Werten auf einer Leitung
  - definierter Startzeitpunkt
  - unbeschränkte Anzahl von Zeitpunkten
  - modelliert durch unendliche Listen
- Verknüpfungsglieder: elementweise Verknüpfung von Eingangssignalen zu Ausgangssignalen
- Rückkopplungen: rekursiv definierte Signale

Einschub: unendliche Listen in Haskell

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Unendliche Listen

- Bsp.: Liste der natürlichen Zahlen `[0..]`
- Laziness garantiert, dass nur benötigter Präfix erzeugt wird:
  `take 5 (map (^2) [0..])` $\leadsto$ `[0,1,4,9,16]`
- Anforderung einer gesamten unendlichen Liste führt zu Nichttermination
  Bsp.: Anwendung von `length`, `foldl`, `scanr`, `partition`, `reverse`, `sort` ...
- möglich sind
  - `map`, `zip`, `zipWith`, `concat`
  - `take`, `drop`, `takeWhile`, `span`, `filter`
  - Test auf Längenbeschränkung: `null . drop n`
  - `scanl`
  - `foldr f` mit `f` nicht strikt im zweiten Argument
  - mergen zweier geordneter unendlicher Listen

©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Unendliche Listen in List-Comprehensions

- Laziness ermöglicht Verwendung unendlicher Listen
- Programmierer muss die Reihenfolge der Generierung beachten

Beispiel: Aufzählung aller pythagoräischer Tripel

- falsch:

```
[ (x,y,z) | x<-[1..], y<-[1..], z<-[1..],
            x^2+y^2==z^2 ]
```

  es werden nie Tripel erzeugt, für die `x` oder `y` ungleich 1 ist, d.h., keine

- funktioniert (ineffizient):

```
[ (a,b,z) | x<-[1..], y<-[1..x-1], z<-[1..(x+y)^2],
            x^2+y^2==z^2, (a,b)<-[(x,y),(y,x)] ]
```

  unendliche Liste nur am Anfang der Generatoren

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Erzeugung unendlicher Listen

Beispiel: Fibonacci-Zahlen `fibNumbers :: [Integer]`

- erster Ansatz

```
fibNumbers = [ fib n | n<-[0..] ]
             where fib n = if n<2 then n
                                  else fib (n-1) + fib (n-2)
```

- mit Akkumulator

```
fibNumbers = f (0,1) where  f (a,b) = a : f (a+b,a)
```

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Bezug auf erzeugte Liste

```
fibNumbers = 0 : 1 : zipWith (+) (tail fibNumbers)
                                        fibNumbers
```

0 1
0 1 1 2 3 : 1 1 2 3 5 : 1 2 3 5 8
+

In unserem Hardware-Modell entspricht `(x:xs)` einer Verzögerung des Signals `xs` um eine Zeiteinheit.

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Repräsentation unendlicher Listen mit Zyklus

1. als unendliche Strukturen
   - können beliebig viel Speicherplatz verbrauchen
   - werden inkrementell erzeugt
   - Bsp.: `repeat' x = x : repeat' x`
2. als zyklische Strukturen
   - benötigen nur einen beschränkten Speicherplatz
   - enthalten Referenz auf sich selbst
   - Bsp.: `repeat x = xs where xs = x:xs`
   - Anwendung: `clock = concat (repeat "00011")`

Zyklus kann durch Benutzung der Struktur nicht erkannt werden
$\Rightarrow$ keine Lösung zur Implementierung von Graphen

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Weitere Beispiele

- Folge der Fakultäten

```
facs :: [Int]
facs = 1 : zipWith (*) [2..] facs
```

- Primzahlen nach dem Sieb des Eratosthenes

```
primes :: [Int]
primes = sieve [2..] where
   sieve (x:xs) =
     x : sieve (filter ((/=0). (`mod` x)) xs)
```


©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Hamming-Folge

Teilfolge von $\mathbb{N}$ mit allen Zahlen der Form $2^k \cdot 3^l \cdot 5^m$

```
hamming :: [Int]
hamming =  1 : merge (map (*2) hamming)
                     (merge (map (*3) hamming)
                            (map (*5) hamming))

merge :: Ord a => [a] -> [a] -> [a]
merge (x:xs) (y:ys) | x<y  = x : merge xs (y:ys)
                    | x==y = x : merge xs ys
                    | x>y  = y : merge (x:xs) ys
```

```
Main> take 20 hamming
[1,2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,27,30,32,36]
```

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Induktionsbeweise auf Listen

Beweis der Gültigkeit eines Prädikats P

1. endliche Listen
   - Bsp.: `1:2:3:[]`
   - zeige $P($`[]`$)$ und $(P(xs) \Rightarrow P(x\!:\!xs))$
2. partielle Listen (Ende nicht ausgewertet)
   - Bsp.: `1:2:3:`$\perp$
   - zeige $P(\perp)$ und $(P(xs) \Rightarrow P(x\!:\!xs))$
   - falls P als Haskell-Funktion schreibbar:
     Beweis gilt auch für unendliche Listen (Stetigkeit von P)
3. beliebige Listen
   - zeige $P($`[]`$)$, $P(\perp)$ und $(P(xs) \Rightarrow P(x\!:\!xs))$

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Beispiele zu Induktionsbeweisen

(Quantoren weggelassen)

- xs`++`$ys = xs$ gilt für alle unendlichen Listen xs
  1. $\bot$`++`$ys = \bot$
  2. `(`u`:`us`)++`$ys = u$`:(`us`++`ys`)` $\overset{IV}{=} u$`:`us

  aber nicht für endliche Listen:

  `[]++`$ys = ys \neq$ `[]`
- `reverse (reverse` xs`)` $= xs$ gilt *nicht* für partielle Listen:
  - `reverse (reverse (`x`:`$\bot$`))` $=$ `reverse` $\bot = \bot \neq x$`:`$\bot$

©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Hardware-Spezifikation

Ziele:

- Simulation
- formale Analyse und Verifikation
- optimierende Transformationen
- Generierung einer Netzliste
- geometrisches Layout

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Wertemenge für die Schaltkreissimulation

- `O`: *False* bei positiver Zuordnung
- `L`: *True* bei positiver Zuordnung
- `Z`: hochohmig (Tristate-Bus ohne Schreibzugriff)
- `U`: Zwischenzustand; unsicherer oder unbekannter Wert

Verwendung von `nand2` als Basis aller booleschen Funktionen:

```
data Bit = O | L | Z | U deriving (Eq,Ord,Enum,Bounded)

nand2 :: (Bit,Bit) -> Bit
nand2 (L,L) = O
nand2 (O,_) = L
nand2 (_,O) = L
nand2 _     = U
```

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Ableitung der anderen booleschen Funktionen

```
not1 :: Bit -> Bit
not1 a = nand2 (a,a)

and2, or2, nor2, xor2, eq2 :: (Bit,Bit) -> Bit

and2       = not1 . nand2
or2  (a,b) = nand2 (not1 a, not1 b)
nor2 (a,b) = and2 (not1 a, not1 b)

xor2 (a,b) = or2 (and2 (a, not1 b), and2 (not1 a, b))
eq2  (a,b) = or2 (and2 (a,b), and2 (not1 a, not1 b))
```

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Addierer

- Halbaddierer: `ha(x,y)` $\widehat{=}$ `((x+y)‘div‘2, (x+y)‘mod‘2)`

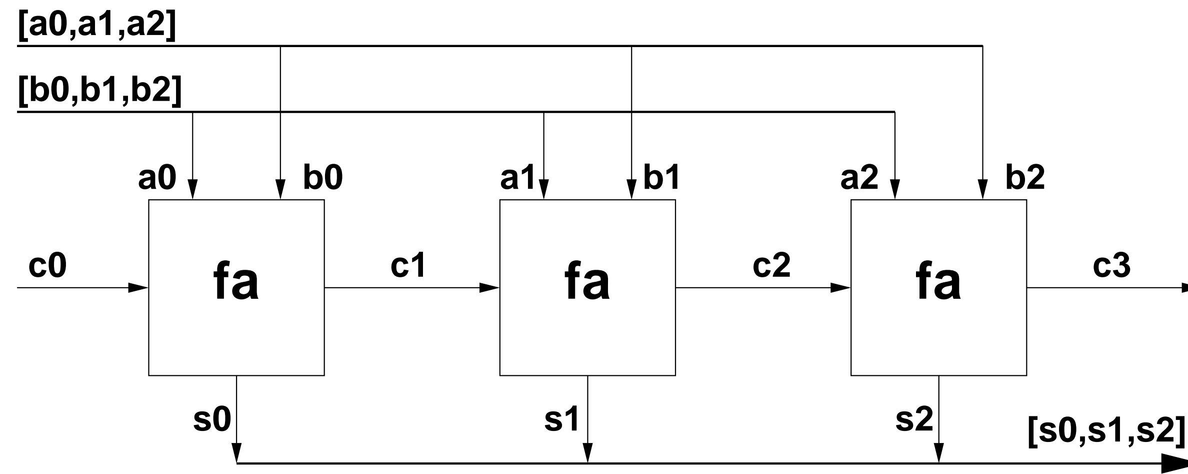
```
ha :: (Bit,Bit) -> (Bit,Bit)
ha inp = ( and2 inp, xor2 inp )
```

- Volladdierer: `fa(x,y,z)` $\widehat{=}$ `((x+y+z)‘div‘2, (x+y+z)‘mod‘2)`

```
fa :: (Bit,Bit,Bit) -> (Bit,Bit)
fa (x,y,z) = let (c0,s0) = ha (x,y)
                 (c1,s1) = ha (s0,z)
             in (or2 (c0,c1), s1)
```


©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Addierer für Binärwörter



- aufgebaut aus einer Folge von Volladdierern (`fa`)
- jeder `fa` ist für eine Binärstelle zuständig
- verkettet mit `mapAccumL`

```
nbitadder :: (Bit,[Bit],[Bit]) -> (Bit,[Bit])
nbitadder (c0,as,bs) =
  mapAccumL (\ ci (ai,bi) -> fa (ai,bi,ci)) c0 (zip as bs)
```

Konversionsfunktionen für Binärwörter

- berechne die Integer-Zahl zum Binärwort `xs`

```
nbit2int :: [Bit] -> Integer
nbit2int xs =
 if all (`elem` [O,L]) xs
   then foldr (\ x acc -> 2*acc+(if x==O then 0 else 1)) 0 xs
   else error "nbit2int on Z or U"
```

- erzeuge die untersten `n` Bit des Binärwortes der Zahl `z`

```
int2nbit :: Int -> Integer -> [Bit]
int2nbit n z | n>0 && z>= 0
    = snd $ mapAccumL
              (\ z _ -> (z`div`2,q (z`mod`2))) z [1..n]
                where q 0 = O
                      q 1 = L
```

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Test des Wortaddierers

```
test_nbitadder :: Int -> Integer -> Integer -> Integer
test_nbitadder n x y =
  nbit2int (snd (nbitadder (0,int2nbit n x,int2nbit n y)))


{-
> map (test_nbitadder 100 456) [100..110]
[556,557,558,559,560,561,562,563,564,565,566]
-}
```

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Beschreibung von Bitfolgen durch Ströme

- eine Leitung trägt zeitabhängig eine Folge von Bits
- Signalverzögerungen modellieren Trägheiten (durch Kapazitäten/Widerstände)
- Verzögerung um eine Zeiteinheit

```
delayOne :: [Bit] -> [Bit]
delayOne xs = U:xs
```

- Test auf Wertänderung (Ergebnis=U, falls einer der Werte $\notin$ {O, L})

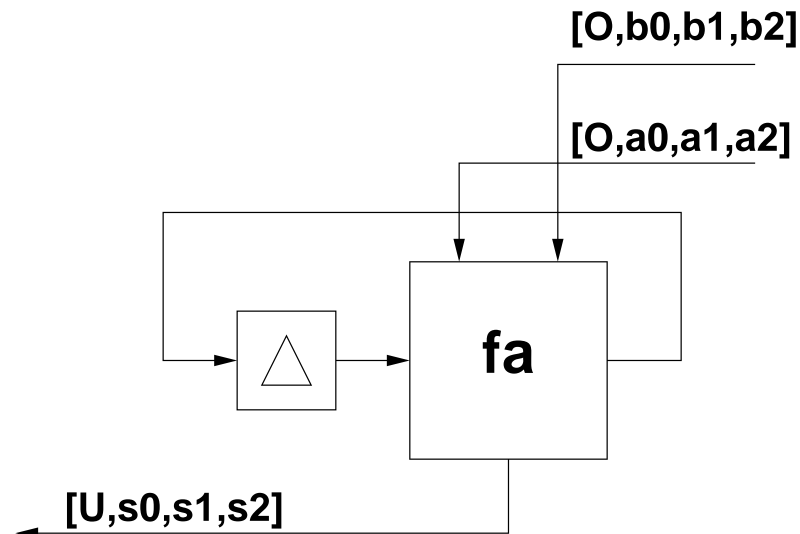
```
testchange :: [Bit] -> [Bit]
testchange xs = zipWith (curry xor2) xs (delayOne xs)

        Zeit: 0 1 2 3 4 5 6 7 8
> testchange [O,O,L,L,L,O,L,L,L]
        -->  [U,O,L,O,O,L,L,O,O]
```

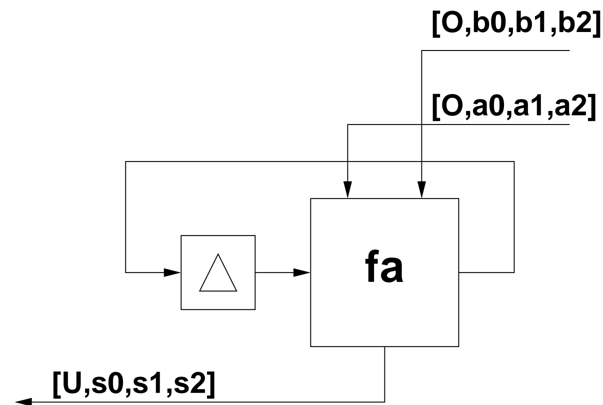

©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Sequenzieller Addierer

- nur ein Volladdiererelement
- zusätzliches Verzögerungselement Δ
  - in diesem Beispiel: eine Zeiteinheit
  - ansonsten: eine Taktperiode (Flipflop)
- Operanden, Ergebnis als Strom von Bitwerten



 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.




```
seqadder :: ([Bit],[Bit]) -> [Bit]
seqadder (x,y)   = z
 where fa' a b c = fa (a,b,c)
       (cout,z)  = unzip (zipWith3 fa' x y cin)
       cin       = delayOne cout

> take 6 $ seqadder ([O,L,L,O,L,O],
                     [O,O,L,L,L,O])
                -->  [U,L,O,O,L,L]
```

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Unterscheidung zwischen Bitströmen und Bitvektoren

- Bitvektor (räumlich): als Liste repräsentiert
- Bitstrom (zeitlich): ab jetzt Liste plus Konstruktor `S`

```
newtype Stream a = S { unS::[a] } deriving Show
type    Signal   = Stream Bit
```

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Modellierung von Verzögerungen

- Functional Level: eine Zeiteinheit pro Takt
- Register-Transfer Level: zwei Zeiteinheit(en) pro Takt
- Gate Level: feinere Aufteilung,
  Ziel : sind die Signale stabil, bevor die Werte gespeichert werden?

für Gate-Level:

```
class Delayable a where
   delay   :: Int -> Stream a -> Stream a

instance Delayable Bit where
   delay n (S s) | n >= 0 = S (replicate n U ++ s)
```

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Verfeinertes Zeitmodell: Min/Max-Delay

- Berücksichtigung von Unsicherheitsintervallen
- Änderung ist sichtbar frühestens ab `dmin`, spätestens nach `dmax` Zeiteinheiten

```
class Delayable a => MinMaxDelay a where
   mmdelay :: (Int,Int) -> a -> Stream a -> Stream a

instance MinMaxDelay Bit where
   mmdelay (dmin,dmax) undef sig
    | dmin>=0 && dmax>=dmin
      = allcompare undef [ delay i sig | i<-[dmin..dmax] ]
```

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Definition der Verknüpfungsglieder auf Signalen (1)

```
nand2d :: (Signal,Signal) -> Signal
nand2d (x,y) = S $ zipWith (curry nand2)
                           (unS $ mmdelay (1,2) U x)
                           (unS $ mmdelay (1,2) U y)
   where nand2 (L,L) = O
         nand2 (O,_) = L
         nand2 (_,O) = L
         nand2 _     = U

not1d :: Signal -> Signal
not1d a = nand2d (a,a)
```

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Definition der Verknüpfungsglieder auf Signalen (2)

```
and2d, or2d, nor2d, xor2d, eq2d :: (Signal,Signal) -> Signal
and2d       = not1d . nand2d
or2d  (a,b) = nand2d (not1d a, not1d b)
nor2d (a,b) = and2d (not1d a, not1d b)
xor2d (a,b) = or2d (and2d (a, not1d b), and2d (not1d a, b))
eq2d  (a,b) = or2d (and2d (a,b), and2d (not1d a, not1d b))
-- Halbaddierer
had :: (Signal,Signal) -> (Signal,Signal)
had inp = (and2d inp, xor2d inp)
-- Volladdierer
fad :: (Signal,Signal,Signal) -> (Signal,Signal)
fad (a,b,c) = let (c1,s1) = had (a,b)
                  (c2,s2) = had (s1,c)
              in (or2d (c1,c2), s2)
```

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

```
> let xs = replicate 15 O ++ repeat L
> take 30 xs
[O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,L,L,L,L,L,L,L,L,L,L,L,L,L,L,L]
> take 30 $ unS $ (\x -> and2d (S x,S x)) $ xs
[U,U,U,U,O,O,O,O,O,O,O,O,O,O,O,O,U,U,L,L,L,L,L,L,L,L,L,L,L]
> take 30 $ unS $ (\x -> xor2d (S x,S x)) $ xs
[U,U,U,U,U,U,U,U,O,O,O,O,O,O,O,O,O,O,O,U,U,U,U,U,U,O,O,O,O,O]
> take 40 $ unS $ fst $ (\x -> fad (S x,S x,S x)) $ xs
[U,U,U,U,U,U,U,U,O,O,O,O,O,O,O,O,O,O,O,U,U,U,U,L,L,L,L,L,L,L,
                                               L,L,L,L,L,L,L,L,L,L]
> take 40 $ unS $ snd $ (\x -> fad (S x,S x,S x)) $ xs
[U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,O,O,O,U,U,U,U,L,U,U,U,U,U,U,
                                               U,U,U,U,U,L,L,L,L,L]
```

Man benötigt auf diese Art viele Zeitpunkte pro Takt. Abhilfe:

- `nand2d` ohne Verzögerung verwenden
- `mmdelay` explizit auf bestimmte Signale anwenden

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Gezielte Zuordnung von Verzögerungen

- alle einzelnen Verküpfungsglieder inkl. `nand2d` beiben verzögerungsfrei.

```
nand2d :: (Signal,Signal) -> Signal
nand2d (S x,S y) = S $ zipWith (curry nand2) x y
```

- Benutzung einer Gruppe von Gates wird mit Verzögerung simuliert
  - minimale Verzögerung: 1 Zeiteinheit
  - maximale Verzögerung: 2 Zeiteinheiten

```
had :: (Signal,Signal) -> (Signal,Signal)
had inp = (mmdelay (1,2) U (and2d inp),
           mmdelay (1,2) U (xor2d inp))

fad :: (Signal,Signal,Signal) -> (Signal,Signal)
fad (a,b,c) = let (c1,s1) = had (a,b)
                  (c2,s2) = had (s1,c)
              in (mmdelay (1,2) U (or2d (c1,c2)), s2)
```

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Testfunktion für 10-bit Addierer

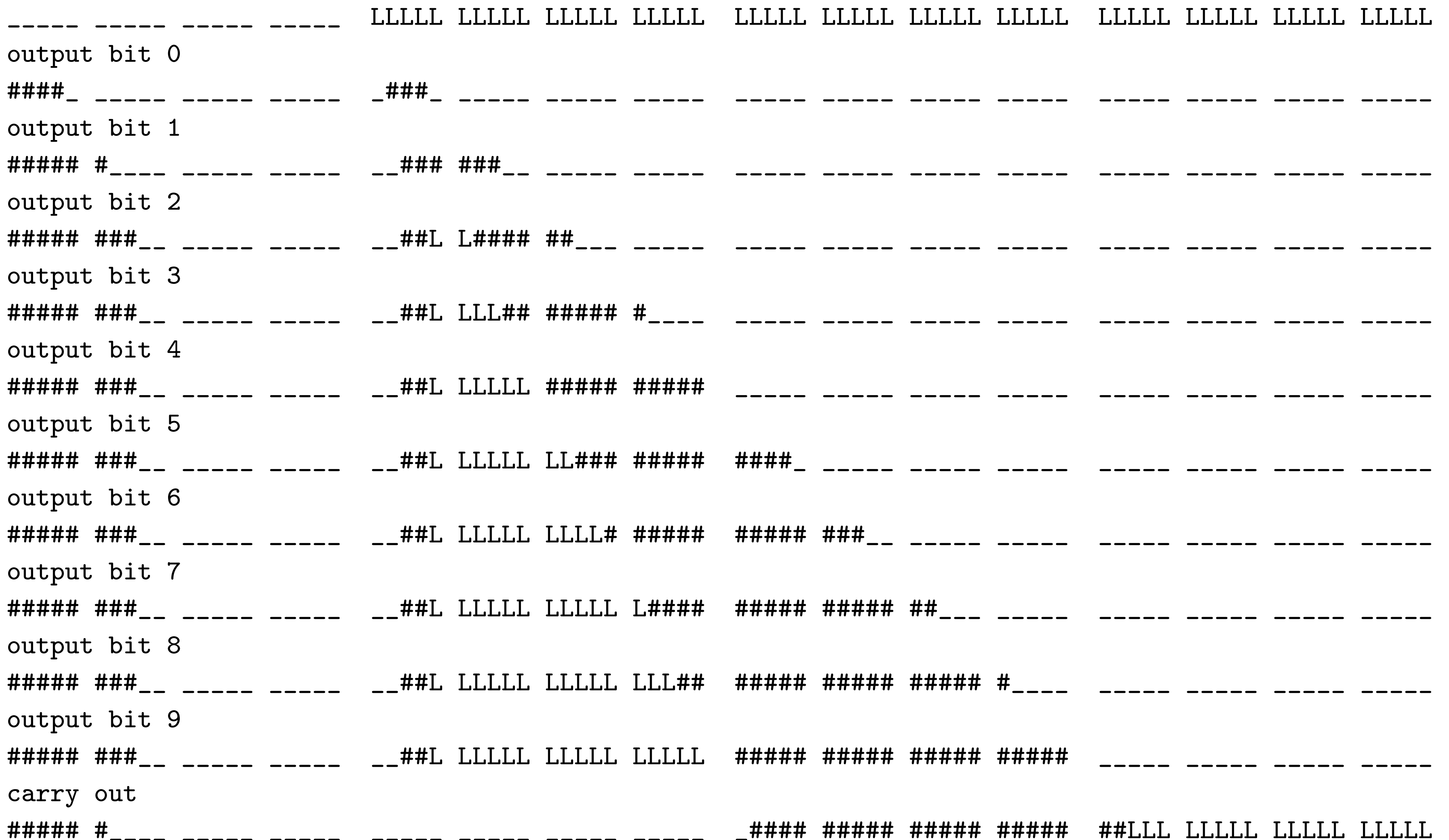
```
showBit O = '_'
showBit L = 'L'
showBit U = '#'
showBit Z = 'Z'

showSig :: Int -> Signal -> String
showSig n (S sig) = block 24 . block 5 $  map showBit (take n sig)
```

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

```
main = do
        let xs = S (replicate 20 O ++ repeat L)
            yss = replicate 10 $ S (repeat O)
            xss = replicate 10 xs
            (cout,zss) = nbitadder (xs,xss,yss)
        putStrLn "input bits of one operand and carry in"
        putStrLn (showSig 80 xs)
        mapM (\(zs,n) -> do
                            putStrLn $ "output bit "++show n
                            putStrLn $ showSig 80 zs)
             (zip zss [0..])
        putStrLn "carry out"
        putStrLn (showSig 80 cout)
        return ()
```

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.



```
input bits of one operand and carry in
_____ _____ _____ _____  LLLLL LLLLL LLLLL LLLLL  LLLLL LLLLL LLLLL LLLLL  LLLLL LLLLL LLLLL LLLLL
output bit 0
####_ _____ _____ _____  _###_ _____ _____ _____  _____ _____ _____ _____  _____ _____ _____ _____
output bit 1
##### #____ _____ _____  __### ###__ _____ _____  _____ _____ _____ _____  _____ _____ _____ _____
output bit 2
##### ###__ _____ _____  __##L L#### ##___ _____  _____ _____ _____ _____  _____ _____ _____ _____
output bit 3
##### ###__ _____ _____  __##L LLL## ##### #____  _____ _____ _____ _____  _____ _____ _____ _____
output bit 4
##### ###__ _____ _____  __##L LLLLL ##### #####  _____ _____ _____ _____  _____ _____ _____ _____
output bit 5
##### ###__ _____ _____  __##L LLLLL LL### #####  ####_ _____ _____ _____  _____ _____ _____ _____
output bit 6
##### ###__ _____ _____  __##L LLLLL LLLL# #####  ##### ###__ _____ _____  _____ _____ _____ _____
output bit 7
##### ###__ _____ _____  __##L LLLLL LLLLL L####  ##### ##### ##___ _____  _____ _____ _____ _____
output bit 8
##### ###__ _____ _____  __##L LLLLL LLLLL LLL##  ##### ##### ##### #____  _____ _____ _____ _____
output bit 9
##### ###__ _____ _____  __##L LLLLL LLLLL LLLLL  ##### ##### ##### #####  _____ _____ _____ _____
carry out
##### #____ _____ _____  _____ _____ _____ _____  _#### ##### ##### #####  ##LLL LLLLL LLLLL LLLLL
```

Problem 1: Analyse der Schaltung

- Der Graph der Schaltung kann bisher nicht erkannt werden
- Lösung: `Signal` als abstrakter Datentyp inkl. `nand2d` etc.
  1. Simulation: `data Signal = Stream Bit`
     Elementfunktionen wie oben
  2. Analyse: Funktionen bauen Baum der Schaltung auf

```
data Signal = Input String
                        -- Name eines Eingangssignals
            | Nand2d (Signal,Signal)
                        -- abstraktes nand2d


nand2d = Nand2d
```

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Problem 2: gemeinsame Teilausdrücke

```
   and2d (x,y)
⇝ not1d (nand2d (x,y))
⇝ nand2d (nand2d (x,y), nand2d (x,y))
⇝ Nand2d (Nand2d (x,y), Nand2d (x,y))
```

- Es werden drei anstelle zweier `Nand2d` benutzt.
- Lösung: struktureller Vergleich, neue Konstruktoren: `LetSig` definiert, `RefSig` verwendet Signal

```
LetSig ("a",Nand2d (x,y))
       (Nand2d (RefSig "a",RefSig "a"))
```


©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Problem 3: zyklische Abhängigkeiten

- Es bestehen keine zyklische Abhängigkeiten von *Werten*, aber:
  1. Leitungen entsprechen Signalen, nicht Werten
  2. Es kann zyklische Abhängigkeiten von Signalen geben
- Beispiel: sequenzieller Addierer

  ```
  cout = Fst (FaSignal (x,y,(DelayOne ...(Fst (FaSignal
  ```

  Grund: referenzielle Transparenz erlaubt es nicht, eine unendliche von einer zyklischen Struktur zu unterscheiden.
- Abhilfemöglichkeiten:
  1. Nummerierung von Elementen in der Spezifikation, z.B.
     `faSignal 42 (x,y,z)` (unpraktisch ohne Präprozessor)
  2. Monadische Definition (nicht so elegant)
  3. Methode des *Observable Sharing* (Änderung von Haskell)

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Monadische Spezifikation von Logikschaltungen

Modifikation von `StateTransformer.hs`:

```
module StateTransformer(ST,runST,readState,writeState)
       where
...

runST :: ST s a -> a
runST m = fst (unST m undefined)

readState :: ST s s
readState = ST (\state -> (state,state))

writeState :: s -> ST s ()
writeState state = ST (const ((),state))
```


©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Typdefinitionen und Monadenoperationen

```
newtype Wire = Wire { unWire::Int }

data Component = Gate {name::String, inputs::[Wire], outputs::[Wire]}
type State = (Int,[Component])

fresh :: ST State Wire
fresh = do (i,xs) <- readState
           writeState (i+1,xs)
           return (Wire i)

addNetList :: Component -> ST State ()
addNetList c = do (i,xs) <- readState
                  writeState (i,c:xs)
```

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Low-Level-Funktionen der Bausteinbibliothek

```
useGate_2_1 :: String -> (Wire,Wire) -> ST State Wire
useGate_2_1 gateName (x,y)
  = do z <- fresh
       addNetList (Gate {name=gateName, inputs=[x,y], outputs=[z]})
       return z

and2, xor2, or2 :: (Wire,Wire) -> ST State Wire
and2 (x,y) = useGate_2_1 "and" (x,y)
xor2 (x,y) = useGate_2_1 "xor" (x,y)
or2  (x,y) = useGate_2_1 "or " (x,y)
```


©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Halbaddierer und Volladdierer

```
halfadder :: (Wire,Wire) -> ST State (Wire,Wire)
halfadder (x,y) = do
                  c <- and2 (x,y)
                  s <- xor2 (x,y)
                  return (c,s)

fulladder :: (Wire,Wire,Wire) -> ST State (Wire,Wire)
fulladder (x,y,z) = do
                    (c0,s0) <- halfadder (x,y)
                    (c1,s1) <- halfadder (s0,z)
                    c2      <- or2 (c0,c1)
                    return (c2,s1)
```


©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Verwendung

```
buildCircuit :: ST State ([Wire],[Wire],[Component])
buildCircuit = do
                [x,y,z] <- mapM (const fresh) [0..2]
                (cout,s) <- fulladder (x,y,z)
                (_,netlist) <- readState
                return ([x,y,z],[cout,s],netlist)

build :: String -> ST State ([Wire],[Wire],[Component]) -> IO ()
build name design    -- design=buildCircuit
 = do
    putStrLn $ "Netzliste für Design: " ++ name
    let (inports,outports,netlist) = runST design initState
    putStr $ "Eingangsknoten: " ++ show (map unWire inports) ++ "\n"
    putStr $ "Ausgangsknoten: " ++ show (map unWire outports) ++ "\n"
    mapM displayComponent netlist
```


©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Ausgabe der Netzliste

```
displayComponent :: Component -> IO ()
displayComponent (Gate name inputs outputs)
 = let displ = concatMap ((++" ").show.unWire)
   in putStr (name ++ " " ++ displ inputs
                ++ " -> " ++ displ outputs ++ "\n")
```

```
Netzliste für Design: Volladdierer
Eingangsknoten: [0,1,2]
Ausgangsknoten: [7,6]
or  3 5  -> 7
xor 4 2  -> 6
and 4 2  -> 5
xor 0 1  -> 4
and 0 1  -> 3
```

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Sequenzieller Addierer

Annahmen:

- Modelliert wird eine Zeiteinheit pro Takt
- `delay1` als Flipflop mit unsichtbarem Takteingang

```
delay1 :: Wire -> Wire -> ST State ()
delay1 x y = addNetList (Gate {name="del", inputs=[x], outputs=[y]})
```

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

```
buildCircuit :: ST State ([Wire],[Wire],[Component])
buildCircuit  = do
                 x   <- fresh     -- input 0
                 y   <- fresh     -- input 1
                 cin <- fresh     -- feedback loop
                 (cout,s) <- fulladder (x,y,cin)
                 delay1 cout cin
                 (_,netlist) <- readState
                 return ([x,y],[s],netlist)
```

```
Eingangsknoten: [0,1]
Ausgangsknoten: [6]
del 7  -> 2
or  3 5  -> 7
xor 4 2  -> 6
and 4 2  -> 5
xor 0 1  -> 4
and 0 1  -> 3
```

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.

Sprachen zur funktionalen Hardware-Spezifikation

- D^3L [Hahn, 1990]: vierwertige Logik, keine HOFs, Simulationscode für Datenflussrechner
- Ruby [Sheeran, 1990]: Retiming (Verschieben von Delay-Elementen)
- Hydra [O'Donnell, 1995]: Netzlisten aus Haskell-Funktionen
- Hawk [Launchbury, 1998]: Mikroprozessor-Simulation
- Lava [Claessen, 2001]: Observable Sharing

  Dissertation von Koen Claessen (Univ. Chalmers, April 2001): "Embedded Languages for Describing and Verifying Hardware"

 ©Universität Passau, Lehrstuhl Prof. Lengauer, Ph.D.