Bluespec-3
Introduction to programming in Bluespec

Arvind
Laboratory for Computer Science
M.I.T.

January 14, 2003

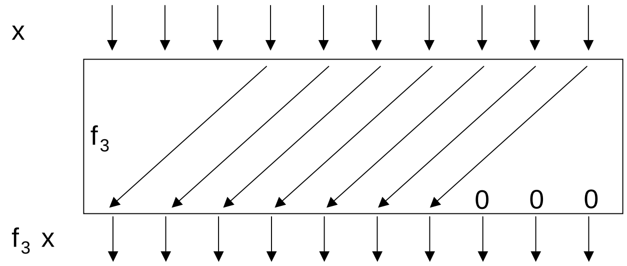
http://www.csg.lcs.mit.edu/IAPBlue

Outline

- Shifting by a variable amount (0-7) ⇐
  - using the Tabulate function
- Cascaded shifter
  - parameterized by the number of stages
  - The fold function
- Pipelined barrel shifter
  - Monadic fold
- Flattening of modules
  - module instantiation or monadic substitution

L3-3
Arvind

Left-shifting a value by 3



In Bluespec:

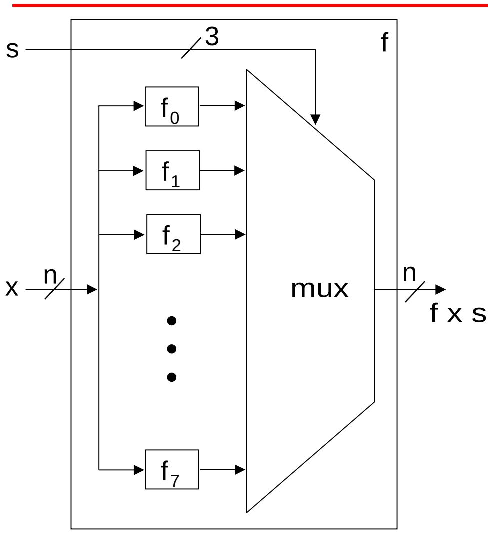
```
f3 :: (Bit 10) -> (Bit 10)
f3 x = x << 3
```

More generally:

```
f3 :: (Bit n) -> (Bit n)
```


L3-4
Arvind

Shifting by a variable amount (0-7)

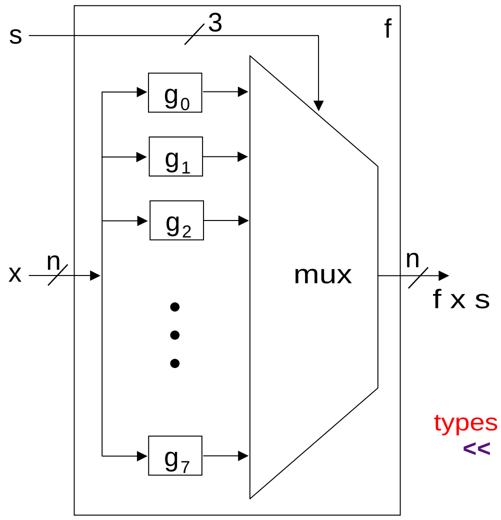



```
f :: (Bit n) -> (Bit 3)
             -> (Bit n)

f0 x = x << 0
 ...
f7 x = x << 7

f x s =
    case s of
         0 -> f0 x
         1 -> f1 x
         2 -> f2 x
         ...
         7 -> f7 x
```

Tabulate: *Automatic generation of Case*



"tabulate f x" generates a version of f for each value in the domain of x

```
f :: (Bit n) -> (Bit 3)
             -> (Bit n)

g x s = x << s toNat s
f x s =
     (tabulate (g x)) s
```

types issue

```
<< :: (Bit n) -> Nat -> (Bit n)
```


Shifting by a variable amount
a better solution

- Given three shifters to shift 1, 2 and 4 places, respectively, any shift between 0 and 7 can be expressed as a cascaded shifting through these 3 shifters

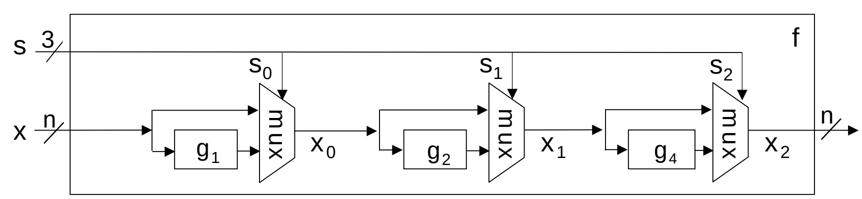
  Example: shift 5 is the same as shift 1 followed by shift 4.

- Solution: Three cascaded steps such that the j^{th} step shifts by 0 or 2^j depending on the j^{th} bit of s

Outline

- Shifting by a variable amount (0-7) √
  - using the Tabulate function
- Cascaded shifter ⇐
  - parameterized by the number of stages
  - The fold function
- Pipelined barrel shifter
  - Monadic fold
- Flattening of modules
  - module instantiation or monadic substitution

Cascaded Barrel Shifter

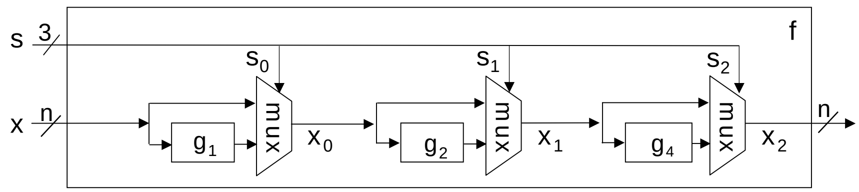


- The j^{th} step shifts by 0 or 2^j depending on the j^{th} bit of s

```
f x s = let x0 = if s[0:0] == 0 then x
                 else (g x 1)
            x1 = if s[1:1] == 0 then x0    ?
                 else (g x0 2)
            x2 = if s[2:2] == 0 then x1
                 else (g x1 4)
        in
            x2
```

L3-9
Arvind

Barrel Shifter: *generalization*



m

`f :: (Bit n) -> (Bit ~~3~~) -> (Bit n)`
`f x s = ...`

generalize to m stages?

- In the j^{th} step shift by 0 or 2^j depending on the j^{th} bit of s

```
step s x j = if s[j:j]==0 then x
                    else (g x (1 << j))
```

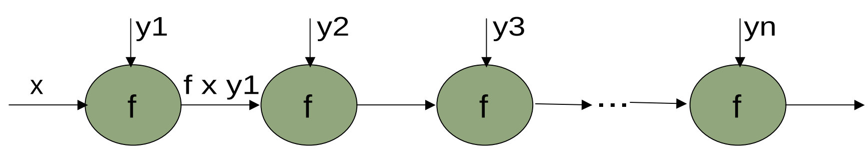
2^j

- Apply this step m times to the initial value of x

```
f x s = foldl (step s) x (upto 0 (m – 1))
```


L3-10
Arvind

The Fold Function

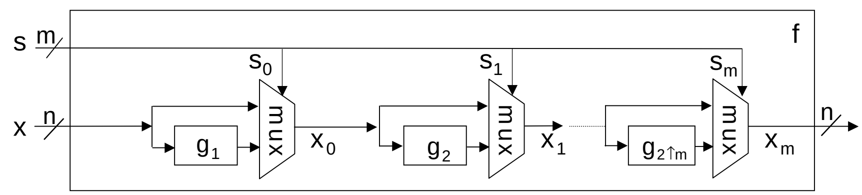



```
foldl :: (tx -> ty -> tx) -> tx ->
                   (List ty) -> tx
foldl f x  Nil            = x
foldl f x (Cons y ys)     = foldl f (f x y) ys
```

Barrel shifter: The folding function f is “step s” and y1, y2,... yn are 0, 1, ... (m-1)

```
foldl (step s) x (upto 0 (m – 1))
```

Barrel Shifter: *a "types" issue*



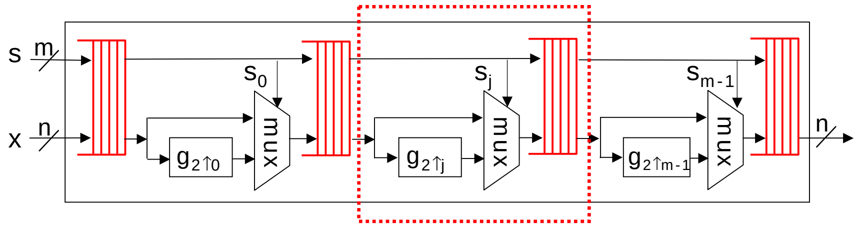

```
f :: (Bit n) -> (Bit m) -> (Bit n)
f x s = let
            step s x j = if s[j:j]==0 then x
                                      else (g x (1 << j))
        in
            foldl (step s) x (upto 0 (m - 1))
                                valueOf(m)
```

m in (Bit m) has something to do with types. We need to use valueOf(m) for m in expressions.

Outline

- Shifting by a variable amount (0-7) √
  - using the Tabulate function
- Cascaded shifter √
  - parameterized by the number of stages
  - The fold function
- Pipelined barrel shifter ⇐
  - Monadic fold
- Flattening of modules
  - module instantiation or monadic substitution

Pipelined shifter



- In the j^{th} step
  - shift by 0 or 2^j depending on the j^{th} bit of s

```
step s x j = if s[j:j]==0 then x
                           else (g x (1 << j))
```

  - given the input FIFO fIn, produce the circuit and the FIFO fOut

Pipelined shifter *continued*

```
mkLsStep:: FIFO (Bit n,Bit m) -> (Bit m) ->
                  -> Module (FIFO (Bit n,Bit m))
mkLsStep fIn j =
      module
        fOut :: FIFO (Bit n,Bit m) <- mkFIFO
        rules
          when (x,s) <- fIn.first
             ==> action fIn.deq
                        fOut.enq (step s x j, s)  ?
        return fOut
```

Pipelined shifter *continued*

```
mkLsStep:: FIFO (Bit n,Bit m) -> (Bit m) ->
                  -> Module (FIFO (Bit n,Bit m))
```

```
module
    fifo1 <- mkLsStep fifo0 0
    fifo2 <- mkLsStep fifo1 1
   ...
    fifom <- mkLsStep fifo2 m
    return
           fifom
```

works only for a fixed m !

- Iterate mkLsStep m times:
  start by supplying the leftmost FIFO

```
mkLs fifo0 =
  foldlM mkLsStep fifo0 (upto 0 (valueOf m – 1))
```


Monadic Fold

```
foldl :: (tz -> tx -> tz) -> tz ->
                        (List tx) -> tz
foldl f z Nil           = z
foldl f z (Cons x xs)   = let
                              z’ = (f z x)
                            in  foldl f z’ xs
```

```
foldlM :: (tz -> tx -> Module tz) -> tz ->
                    (List tx) -> (Module tz)

foldlM f z Nil          = return z
foldlM f z (Nil x xs)   = module
                              z’ <- f z x
                              foldlM f z’ xs
```

Pipelined shifter *remarks*

- The program to generate the circuit is parametric
  - n bits represent the datawidth in the FIFO
  - m represents the number of bits needed to specify the shift (= log n)
- The language scaffolding needed to express, for example, iteration disappears after the first phase of compilation
  - no "circuit" penalty for using high-level language constructs

Outline

- Shifting by a variable amount (0-7) √
  - using the Tabulate function
- Cascaded shifter √
  - parameterized by the number of stages
  - The fold function
- Pipelined barrel shifter √
  - Monadic fold
- Flattening of modules ⇐
  - module instantiation or monadic substitution

Unfolding during Compilation

```
foldlM f z Nil            = return z
foldlM f z (Cons x xs)    = module
                                z' <- (f z x)
                                foldlM f z' xs
```

Suppose the list is [x0,x1,x2]. The compiler will unfold foldlM as follows:

```
module
    z1 <- f z x0
    module
        z2 <- f z1 x1
        module
            z3 <- f z2 x2
            return
               z3
```

➔

```
module
    z1 <- f z x0
    z2 <- f z1 x1
    z3 <- f z2 x2
    return
           z3
```


Unfolding of Pipelined shifter

```
mkLs fifo0 =
  foldlM mkLsStep fifo0 (upto 0 (valueOf m – 1))
```

Suppose m is 3. The compiler will unfold foldlM as follows:

```
module
    fifo1 <- mkLsStep fifo0 0
    fifo2 <- mkLsStep fifo1 1
    fifo3 <- mkLsStep fifo2 2
    return
           fifo3
```

next step: *inline* **`mkLsStep`**

Pipelined shifter: *Inlining* mkLsStep

```
module
    fifo1 <-
     module
       fOut <- mkFIFO
       rules
         when (x,s) <- fifo0.first
            ==> action fifo0.deq
                       fOut.enq (step s x 0, s)
       return fOut
    fifo2 <- mkLsStep fifo1 1
    fifo3 <- mkLsStep fifo2 2
    return
           fifo3
```

next step: instantiate the module to produce fifo1

Pipelined shifter: *module instantiation*

```
module
    fOut1 <- mkFIFO
    rules
          when (x,s) <- fifo0.first
             ==> action fifo0.deq
                        fOut1.enq (step s x 0, s)
    let fifo1 = fOut1
    fifo2 <- mkLsStep fifo1 1
    fifo3 <- mkLsStep fifo2 2
    return
           fifo3
```

next step: instantiate modules to produce fifo2 and fifo3

Flattened Pipelined shifter

```
module
    fOut1 <- mkFIFO
    rules
          when (x,s) <- fifo0.first
             ==> action fifo0.deq
                        fOut1.enq (step s x 0, s)
    fOut2 <- mkFIFO
    rules
          when (x,s) <- fOut1.first
             ==> action fOut1.deq
                        fOut2.enq (step s x 1, s)
    fOut3 <- mkFIFO
    rules
          when (x,s) <- fOut2.first
             ==> action fOut2.deq
                        fOut3.enq (step s x 2, s)
   return   fOut3
```


Flattened Pipelined shifter

```
module
    fOut1 <- mkFIFO
    fOut2 <- mkFIFO
    fOut3 <- mkFIFO
    rules
          when (x,s) <- fifo0.first
             ==> action fifo0.deq
                        fOut1.enq (step s x 0, s)
          when (x,s) <- fOut1.first
             ==> action fOut1.deq
                        fOut2.enq (step s x 1, s)
          when (x,s) <- fOut2.first
             ==> action fOut2.deq
                        fOut3.enq (step s x 2, s)
   return   fOut3
```