



Bluespec-2

Bluespec Compilation Model & Introduction to programming

Arvind
Laboratory for Computer Science
M.I.T.

November 18, 2002

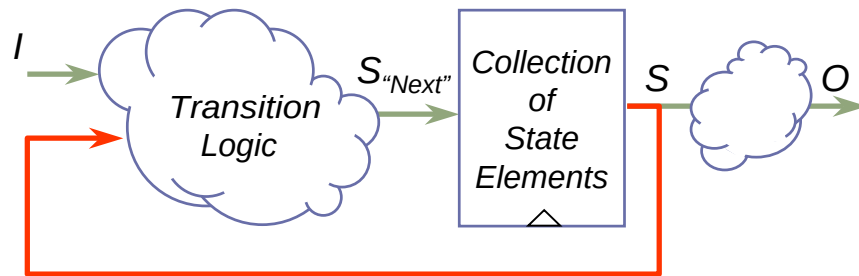
<http://www.csg.lcs.mit.edu/6.827>

Outline

- Bluespec compilation ←
- Bluespec programming
 - Example: Barrel shifter



From TRS to Synchronous CFMS



November 18, 2002

<http://www.csg.lcs.mit.edu/6.827>



TRS Execution Semantics

L18-4
Arvind

Given a set of rules and an initial term s

While (some rules are applicable to s)

- ◆ choose an applicable rule
(*non-deterministic*)
- ◆ apply the rule atomically to s

The trick to generating good hardware is to schedule as many rules in parallel as possible without violating the sequential semantics given above

November 18, 2002

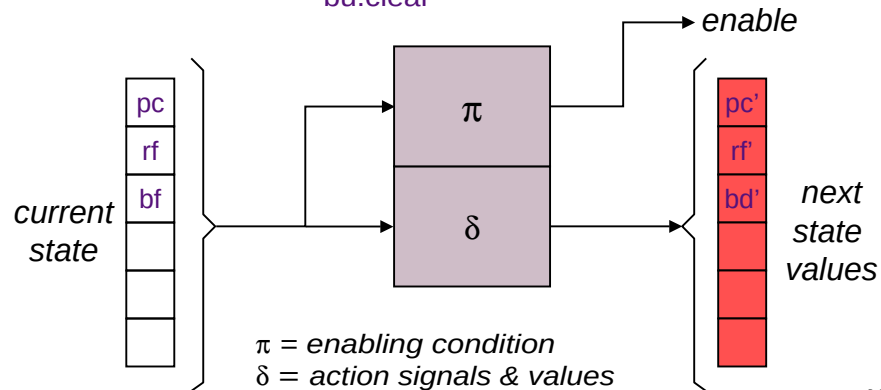
<http://www.csg.lcs.mit.edu/6.827>



Compiling a Rule

“Bz Taken”:

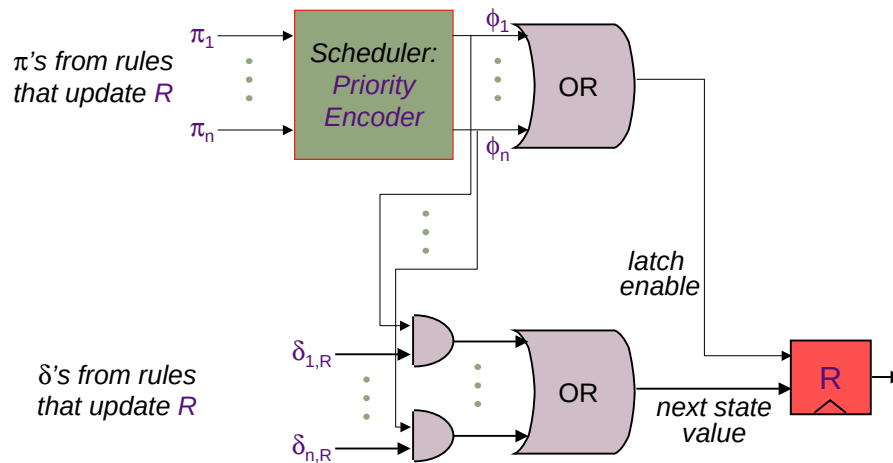
when (Bz rc ra) <- bu.first, rf!rc == 0
 $\implies$ action pc := rf!ra
 bu.clear



November 18, 2002

<http://www.csg.lcs.mit.edu/6.827>

Combining State Updates



Scheduler ensures that at most one ϕ_i is true

November 18, 2002

<http://www.csg.lcs.mit.edu/6.827>

Executing Multiple Rules Per Cycle

“Fetch”:
 when True
 $\Rightarrow$ action $pc := pc+1$
 $bu.enq(imem.read\ pc)$

“Add”:
 when (Add rd rs rt) <- bu.first
 $\Rightarrow$ action $rf!rd := rf!rs + rf!rt$
 $bu.deq$

Can these rules be executed simultaneously?

*These rules are “**conflict free**” because they manipulate different parts of the state (i.e., pc and rf), and enq and deq on a FIFO can be done simultaneously.*

November 18, 2002

<http://www.csg.lcs.mit.edu/6.827>

Conflict-Free Rules

Rule_a and Rule_b are conflict-free if

$$\forall s. \pi_a(s) \wedge \pi_b(s) \Rightarrow$$

1. $\pi_a(\delta_b(s)) \wedge \pi_b(\delta_a(s))$
2. $\delta_a(\delta_b(s)) == \delta_b(\delta_a(s))$
3. $\delta_a(\delta_b(s)) == \delta_a(s) \oplus \delta_b(s)$

where $\oplus$ is a sort of LUB operator

Theorem: Conflict-free rules can be executed concurrently without violating TRS's sequential semantics

November 18, 2002

<http://www.csg.lcs.mit.edu/6.827>

Conflict-Free Scheduler

- Partition rules into maximum number of disjoint sets such that
 - a rule in one set may conflict with one or more rules in the same set
 - a rule in one set is conflict free with respect to all the rules in all other sets

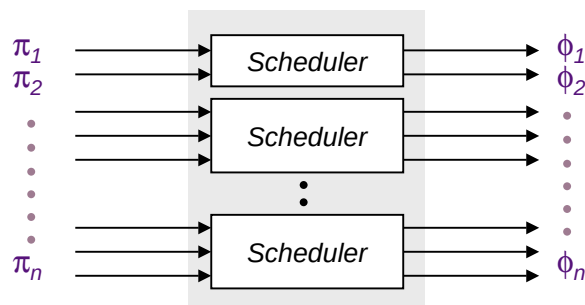
(Best case: All sets are of size 1!!)
- Schedule each set independently
 - Priority Encoder, Round-Robin Priority Encoder
 - Enumerated Encoder

The state update logic remains unchanged

November 18, 2002

<http://www.csg.lcs.mit.edu/6.827>


Multiple-Op-per-Cycle Scheduler



$$1. \phi_i \Rightarrow \pi_i$$

$$2. \pi_1 \vee \pi_2 \vee \dots \vee \pi_n \Rightarrow \phi_1 \vee \phi_2 \vee \dots \vee \phi_n$$

$$3. \text{ Multiple operations such that } \phi_i \wedge \phi_j \Rightarrow R_i \text{ and } R_j \text{ are conflict-free}$$

November 18, 2002

<http://www.csg.lcs.mit.edu/6.827>


Multiple Rewrites Per Cycle

“Fetch”:
when True
 ==> action pc := pc+1
 bu.enq (imem.read pc)

“Bz Taken”:
when (pc' , Bz rc ra) <- bu.first, rf!rc == 0
 ==> action pc := rf!ra
 bu.clear

Can these rules be executed simultaneously?

Yes, as long as the action of Bz Taken rule *dominates!*
many other possibilities for parallel execution ...

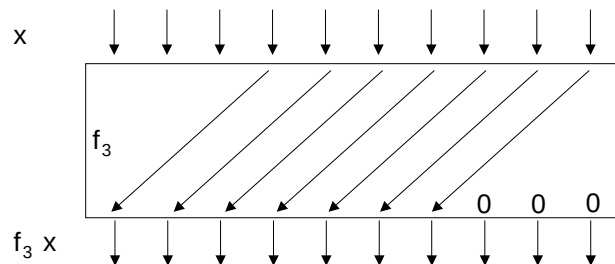


Outline

- Bluespec compilation ✓
- Bluespec programming ⇐
 - Example: Barrel shifter



Left-shifting a value by 3



In Bluespec:

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

More generally:

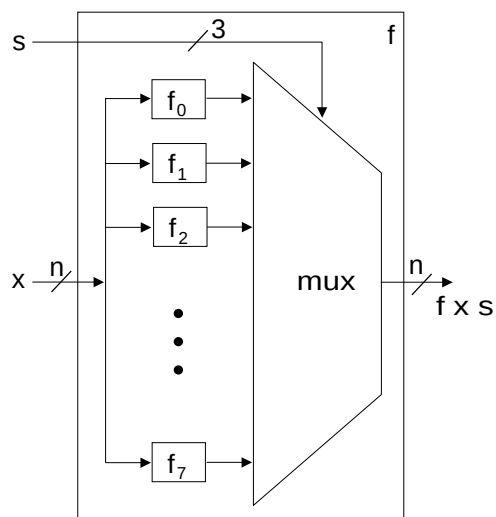
```
f3 :: (Bit n) -> (Bit n)
```

November 18, 2002

<http://www.csg.lcs.mit.edu/6.827>



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
```

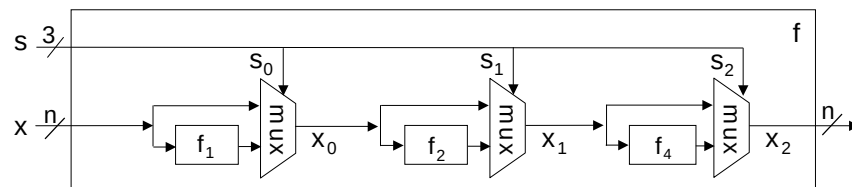
But this is an expensive solution !

November 18, 2002

<http://www.csg.lcs.mit.edu/6.827>



Shifting by a variable amount: solution 2



- three cascaded steps such that 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 (x << (1 << 0))
x1 = if s[1:1] == 0 then x0
    else (x0 << (1 << 1))
x2 = if s[2:2] == 0 then x1
    else (x1 << (1 << 2))

```

in

x2

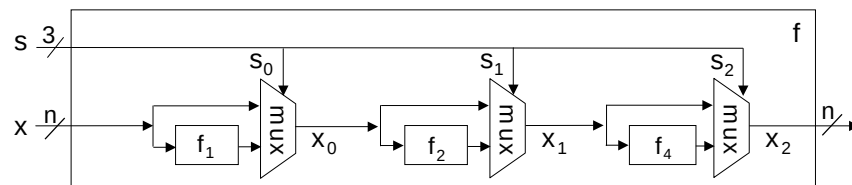
November 18, 2002

<http://www.csg.lcs.mit.edu/6.827>



2²

Shifting by a variable amount: generalization



```

f :: (Bit n) -> (Bit 3m) -> (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 (x << (1 << j))

```

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

```

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

```

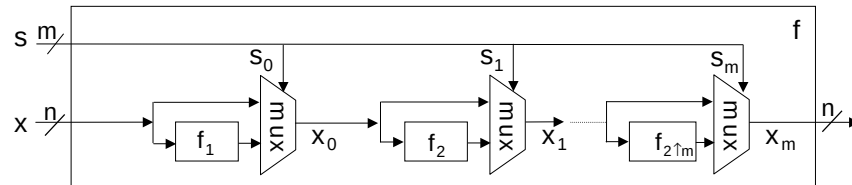
November 18, 2002

<http://www.csg.lcs.mit.edu/6.827>



2^j

Barrel Shifter: a “types” issue



```

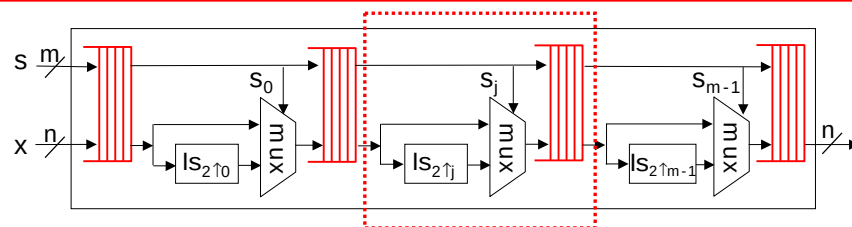
f :: (Bit n) -> (Bit m) -> (Bit n)
f x s = let
    step s x j = if s[j:j]==0 then x
                  else (x << (1 << j))
    in
    foldl (step s) x (upto 0 (m - 1))
    valueOf(m)
  
```

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

November 18, 2002

<http://www.csg.lcs.mit.edu/6.827>


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 (x << (1 << j))
  
```

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

November 18, 2002

<http://www.csg.lcs.mit.edu/6.827>


Pipelined shifter *continued*

```

mkLsStep :: FIFO (Bit n, Bit m) -> (Bit m) ->
          -> Module (FIFO (Bit n, Bit m))
mkLsStep fIn j =
  module
    State → fOut :: FIFO (Bit n, Bit m) <- mkFIFO
    Internal behavior → rules
      when (x,s) <- fIn.first
        ==> action fIn.deq
          fOut.enq (step s x j, s)
    External interface → return fOut

```

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

```

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

```

November 18, 2002

<http://www.csg.lcs.mit.edu/6.827>

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

November 18, 2002

<http://www.csg.lcs.mit.edu/6.827>

Monadic Fold

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

November 18, 2002

<http://www.csg.lcs.mit.edu/6.827>

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 {x1,x2,x3}. 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
```

November 18, 2002

<http://www.csg.lcs.mit.edu/6.827>

Compilation 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
```



Compilation of Pipelined shifter *continued*

```
module
  fifo1 <-
    module
      f0ut <- mkFIFO
      rules
        when (x,s) <- fifo0.first
          ==> action fifo0.deq
                    f0ut.enq (step s x 0, s)
      return f0ut

  fifo2 <- mkLsStep fifo1 1
  fifo3 <- mkLsStep fifo2 2
  return
    fifo3
```



Compilation of Pipelined shifter

continued-2

```
module
  let fifo1 = f0ut
  f0ut <- mkFIFO
  rules
    when (x,s) <- fifo0.first
      ==> action fifo0.deq
              f0ut.enq (step s x 0, s)

  fifo2 <- mkLsStep fifo1 1
  fifo3 <- mkLsStep fifo2 2
  return
    fifo3
```

