Bluespec-8
Advanced Topics

Rishiyur S. Nikhil
nikhil@sandburst.com
Thursday, January 16, 2003

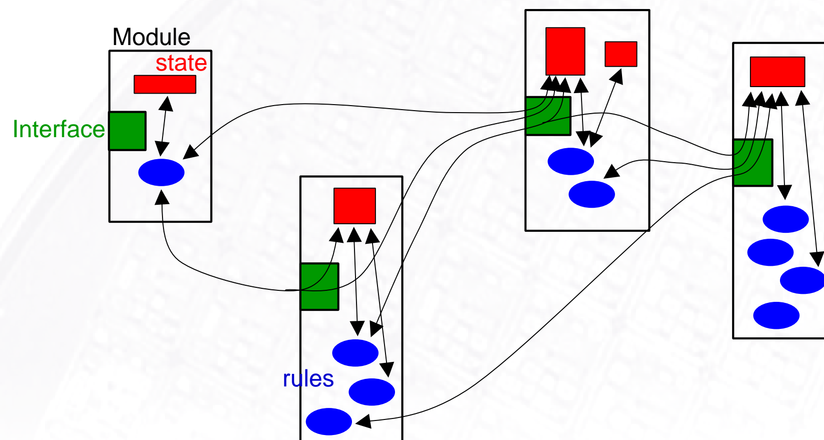
MIT IAP Course
http://www.csg.lcs.mit.edu/IAPBlue

Topics

- Connectables
  - Structured connections between modules
- Control/status registers
  - "Back door" entry into a chip
- Multiple clock domains

Connectables

- Modules communicate via their *interfaces*.



Connectables

- In general, Bluespec places no limits on what types you can use in an interface.
  - scalars
  - functions
  - tuples and structs
  - lists and arrays
  - …
  - even other interfaces

i.e., all types are first-class

Connectables

- In certain situations, there are practical considerations on what can go into an interface
  - Verilog boundary: Bluespec interface must have a translation into a Verilog interface (wires)
  - Timing analysis and timing closure:
    - Interface input and output wires must be registered immediately at the module boundary
    - Interface protocol should be "pipelinable", allowing free insertion of pipeline registers in long wires
  - Readability and maintainability:
    - "standard" interfaces
    - standard "adapters" between similar standard interfaces
- "Connectables" are a class of interfaces motivated by such considerations
  - (there can be more such classes)

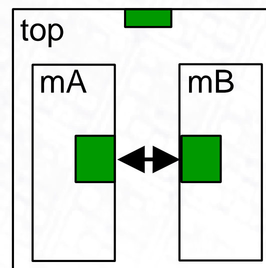
Class Connectable

- Indicates that two related types can be "connected". Does not specify the nature of the connection.

```
class Connectable a b
  (<->) :: a -> b -> Module Empty
```

- Example use (for connectable types A and B):

```
mkTop :: Module Empty
mkTop = module
           mA :: A <- mkA
           mB :: B <- mkB
           mA <-> mB
```



Class Connectable

- The simplest types in the Connectable class are Get and Put

```
interface Get  a =
    get :: ActionValue a

interface Put  a =
    put :: a -> Action

instance Connectable (Get a) (Put a)
  where g <-> p = module
                    rules
                      when True ==> action
                                      x <- g.get
                                      put x
```

Class Connectable

- FIFO interfaces can be converted into Get and Put (and therefore become connectable)

```
fifoToGet :: FIFO a -> Get a
fifoToGet  f = interface Get
                 get = do  f.deq
                           return f.first

fifoToPut :: FIFO a -> Put a
fifoToPut f = interface Put
                 put  x = f.enq x
```

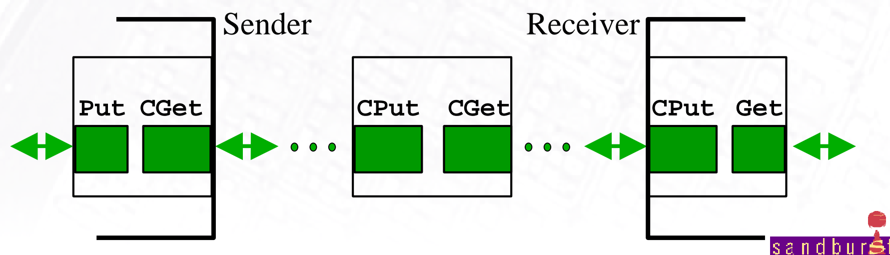
Library CGetPut

- The CGetPut library provides a fully registered credit-based "FIFO":

```
mkCGetPut :: ... Module (CGet n a, Put a)
mkGetCPut :: ... Module (Get a, CPut n a)

mkCGetCPut :: ... Module (CGet n a, CPut n a)

instance Connectable (CGet n a) (CPut n a)
```



sandburst

Library CGetPut

- The intermediate buffers (mkCGetPut) are optional:
  - Insert them to add buffering along long wires
  - They add latency
- The latency of the transfer is *r,* in the absence of any intermediate buffers
  - $r = 4$ in our implementation
- The "credit" value is *n.*
  - Choose $n = 4$ for full bandwidth
  - Choose $n = 1$ for minimum registers (but ¼ bandwidth)

sandburst

Library BGetPut

- The BGetPut library provides a fully registered connection that makes no assumptions about setup and hold times, and so can connect different clock domains

```
mkBGetPut :: ... Module (BGet a, Put a)
mkGetBPut :: ... Module (Get a, BPut a)

instance Connectable (BGet a) (BPut a)
```

(but it's not fast)

Multiple connections

- Two pairs of (corresponding) connectable types are themselves connectable:

```
instance (Connectable a b,
          Connectable b c) => Connectable (a,c) (b,d)
where
     (<->) (a,b) (c,d) = do
                           a <-> c
                           b <-> d
```

- (see also ClientServer library, and CGetPut and BGetPut versions of ClientServer)

Multiple connections

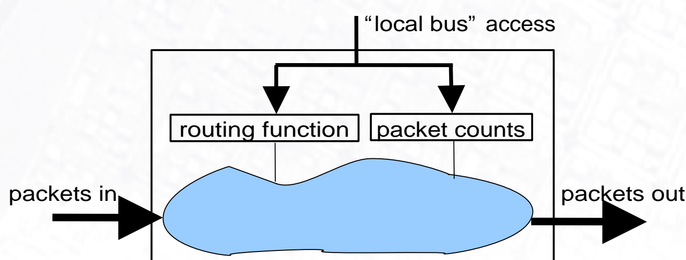
- Two lists of (corresponding) connectable types are themselves connectable:

```
instance (Connectable a b) =>
          Connectable (ListN n a) (ListN n b)
where
  xs (<->) ys = do
                  sequence (zipWith (<-> xs ys))
                  return (interface Empty)
```

"Control and Status" Registers

Control/Status registers

- Many chips have "control and status registers" for configuration, diagnostics, statistics, debugging, etc.
- These registers are read/written from a special "register access" port into the chip, such as a PCI bus



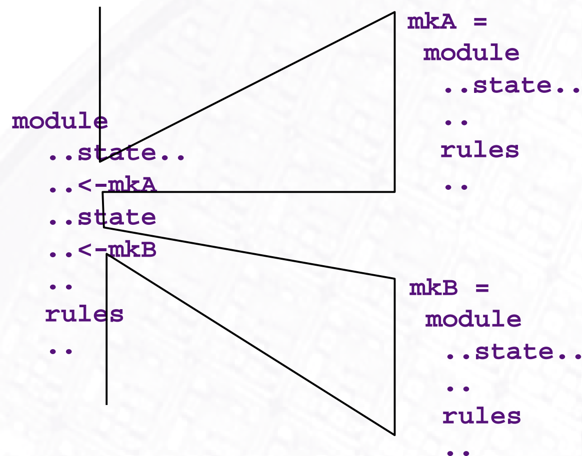
sandburst

Control/Status Registers (CSRs)

- CSRs have "two faces":
  - "Chip side": the ordinary register interface
  - "Local bus side": read/write like a memory location
    - Each CSR has its own address
- CSRs may be scattered all over the design
- The "plumbing" for local bus side of CSRs can be quite messy if done explicitly
- Bluespec has mechanisms that hide/ simplify/ automate the local bus side
  - Part of the power of monads/modules-- not built into the language!

sandburst

Modules allow "collecting" things



Normally, we collect state and rules

In general, modules can collect other things (lib ModuleCollect)

- Collect items of type *a* and return a value of type *i* (usually an interface):

```
data ModuleCollect a i
```

- Add an item to a collection

```
addToCollection :: a -> ModuleCollect a i
```

- Retrieve the collection and the regular value

```
getCollection :: ModuleCollect a i -> Module (i, List a)
```

Library ModuleCollect

- A module expression, in general, has type
  `m t`
- `Module t` is just a special case, where the only things being collected are state and rules.
- Instead, we can treat the whole module structure as a ModuleCollect structure and, at the top, extract the collected objects and restore it back to a Module type.

Library LocalBus

- We use ModuleCollect to automatically gather up all the “local bus side” interfaces of Control/ Status Registers

```
interface LBSReg sa sd =
  lbsAddr :: Bit sa            -- addr of CSR
  lbsSet  :: Bit sd -> Action
  lbsGet  :: Bit sd

lbRegRW :: Bit sa -> r ->
           ModuleCollect (LBSReg sa sd) (Reg r)
```

Library LocalBus

- In a module, for a CSR, we simply use lbRegRW instead of mkReg

```
module
    ...
    r   :: Reg (Bit 32) <- mkReg 15
    csr :: Reg (Bit 32) <- lbRegRW  0x100 15
    ...
```

- 0x100 is the localbus address of the CSR
- The module's type is something like:

```
ModuleCollect (LBSReg 24 32) t
```

instead of:

```
Module t
```

Library LocalBus

- Simplifying local bus address management:
  - Add a base address to all the LBSRegs in a module

```
lbsOffset :: Bit sa ->
             ModuleCollect (LBSReg sa sd) i ->
             ModuleCollect (LBSReg sa sd) i
```

Library LocalBus

- Collecting and converting into a memory-like interface

```
lbsCollect :: ModuleCollect (LBSReg sa sd) i ->
              Module (RAM sa sd, i)
```

(and there are further facilities to collect multiple such RAM interfaces into a single RAM interface)

Library LocalBus: summary

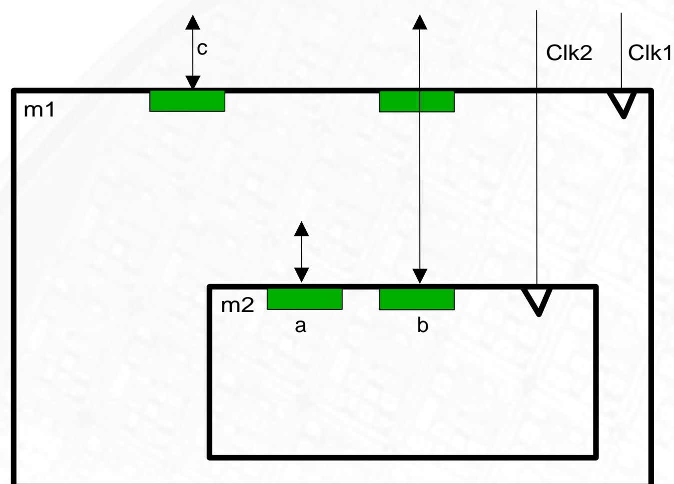
- Wherever you want a CSR, use a constructor like lbRegRW instead of mkReg
  - Give it a local bus address (can be a computed value, argument to module constructor function)
- When instantiating sub-modules, use lbsOffset to move the CSRs of each sub-module to a relatively unique local bus address region
- At the top-level, use lbsCollect to obtain the original interface to the module, and a RAM interface to all its CSRs.

L8- 25

Multiple clock domains

L8- 26

Multiple clock domains



Library ClockConv

- A type for clocks

```
data Clock
```

  - The type is abstract: you can essentially only pass it in as a value from the outside.
  - Externally represented as 2 wires, a clock and a reset.
- A class of types that are allowed to cross a clock domain boundary:

```
class ClockConv a
```

- Two useful types in this class:

```
instance ClockConv (Get a)
instance ClockConv (Put a)
```

- Any type can be "closed", making it a ClockConv type

```
instance ClockConv (Closed a)
close :: a -> Closed a
```

Library ClockConv

- Convert a module to be clocked by a given clock instead of the default clock:

```
clockConv :: (ClockConv a) =>
             Clock ->
             Module a -> Module a
```

Example

```
mkM2 :: Module (A, Closed B)
  module
    ...
    ...
    return (ia, close ib)
```

```
mkM1 clk2 =
  module
    ....
    (m2a,m2b) <- clockConv clk2 mkM2
    ...
    return (ic, m2b)
```

The End