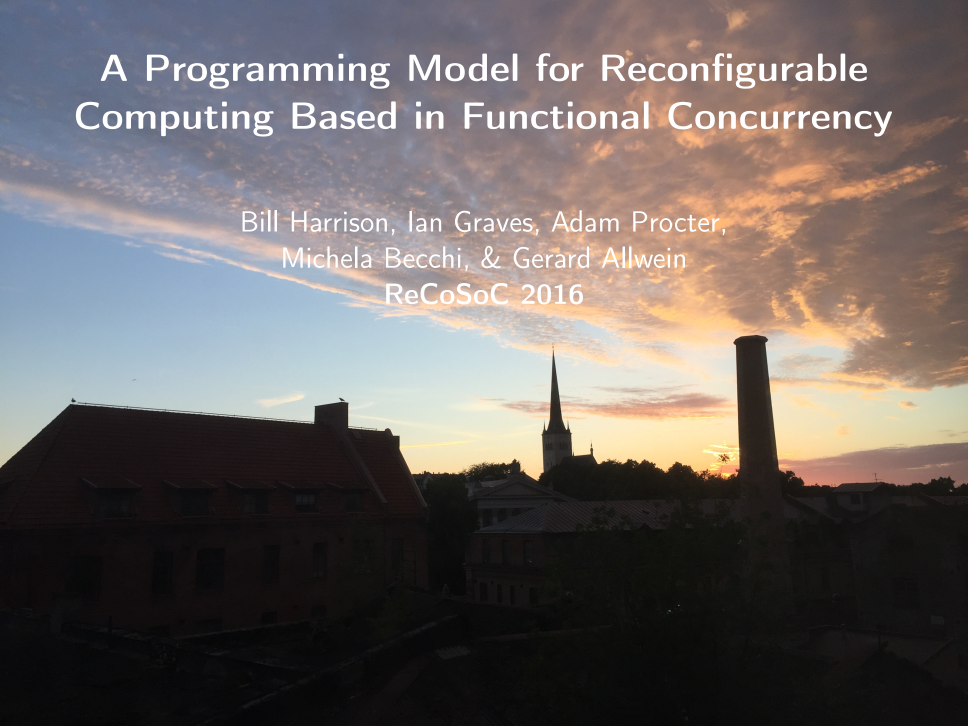


A Programming Model for Reconfigurable Computing Based in Functional Concurrency

Bill Harrison, Ian Graves, Adam Procter,
Michela Becchi, & Gerard Allwein

ReCoSoC 2016

Mission/Safety-critical, ∗**Reconfigurable**∗ Systems

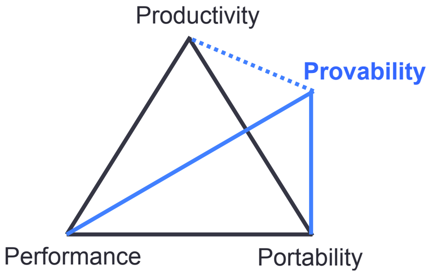
- Highly (Re)configurable Architectures/FPGAs
  - Many Specially Tailored, "One Off" Components
  - Reuse of Off-the-shelf components
  - "Mix and Match" comes to Hardware
- Challenge: High Assurance in this environment
  - Want the flexibility and speed of development
  - ...but also **need** formal guarantees of security & safety for critical systems

Mission/Safety-critical, ∗**Reconfigurable**∗ Systems

- Highly (Re)configurable Architectures/FPGAs
  - Many Specially Tailored, “One Off” Components
  - Reuse of Off-the-shelf components
  - “Mix and Match” comes to Hardware
- Challenge: High Assurance in this environment
  - Want the flexibility and speed of development
  - . . .but also **need** formal guarantees of security & safety for critical systems
- Unpleasant Reality: Traditional HW Verification cannot cope with “Mix & Match”
  - Too slow & expensive for “one off” components
  - Why? Time spent “formalizing” hardware design

Language-based Approach to High Assurance Hardware

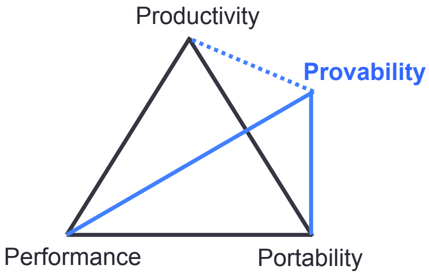
- "The Three P's"
  - DSLs & Language Virtualization
  - Delite [Olukoton,Ienne]
- ReWire
  - Fourth P: Provability
  - Rigorous Semantics supports High Assurance
    - Security & Safety Properties
    - Formal Methods Productivity



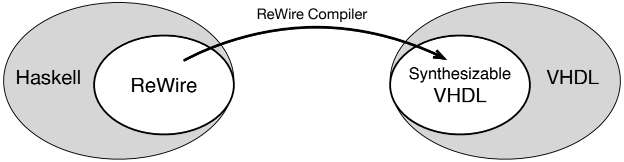
Focus on Productivity

A Programming Model for Reconfigurable Computing Based in Functional Concurrency

- Recent Work:
  - Provability [FPT15]
  - Performance [ARC15]
  - Portability [LCTES15]
- Software Engineering "Virtues"
  - Abstraction, Modularity, Program Comprehension, etc.
  - ReWire
    - Functional Language supporting Concurrency
    - ...thereby common concurrency templates



ReWire Functional Hardware Description Language



- Inherits Haskell's good qualities
  - Pure functions & types, monads, equational reasoning, etc.
  - Formal denotational semantics [HarrisonKieburtz05,Harrison05]
- Language design identifies HW representable programs
  - Mainly restrictions on recursion in functions and data
  - Built-in abstractions for clocked/parallel computations
  - "Connect Logic": Types & operators for HW abstractions.

Reasoning about ReWire Programs

Ordinary Equational Reasoning on Functional Programs:

$$e_1 = e_2 = \ldots = e_n$$

replaces "equals for equals", uses induction/coinduction, etc.

Reasoning about ReWire Programs

Ordinary Equational Reasoning on Functional Programs:

$$\mathtt{e}_1 = \mathtt{e}_2 = \ldots = \mathtt{e}_n$$

replaces "equals for equals", uses induction/coinduction, etc.

Ex: Hardware Verification from [FPT15]

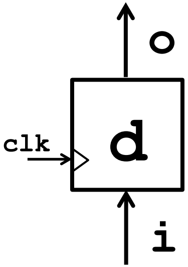
Theorem (Correctness of Iterative Salsa20)

For all nonces $\mathtt{n}, \mathtt{n}_0, \ldots, \mathtt{n}_9 :: \mathtt{W}128$ *and input streams* `is` *of the form* $[(\mathtt{High}, \mathtt{n}), (\mathtt{Low}, \mathtt{n}_0), \cdots, (\mathtt{Low}, \mathtt{n}_9), \ldots]$*, then:*

$$\textbf{\textit{salsa20}}\, \mathtt{n} = \mathtt{nth}\, 10\ (\mathtt{feed}\, \mathtt{is}\, \textbf{\textit{sls20dev}})$$

Abstract Types for Devices

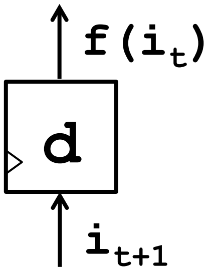
- Built-in Type **`Dev i o`**
  - Parameterized by input and output types, **`i`** and **`o`**
- Construct devices by building **`Dev i o`** values with **constructors**
- ReWire compiler translates **`Dev i o`** into synthesizable VHDL
- **`Dev i o`** is a “reactive resumption monad”
  - Algebraic structure for clocked, synchronous parallelism
  - Useful for specifying secure systems [LCTES15,JCS09]



Iteration Constructor

```
iter :: (i -> o) ->
        o        ->
        Dev i o
```

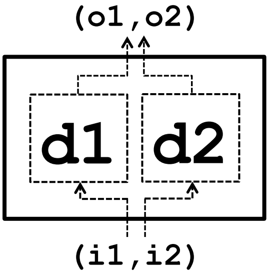
d = iter f o



Parallelism Constructor

d1 <&> d2

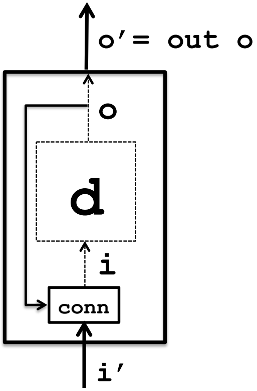
```
<&> :: Dev i1 o1 ->
       Dev i2 o2 ->
       Dev (i1,i2) (o1,o2)
```



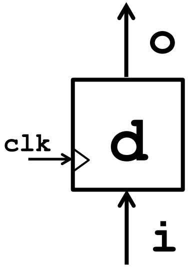
Feedback Constructor

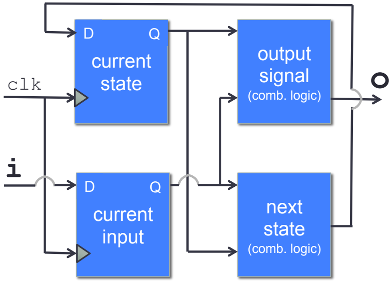
```
refold :: (o1 -> o2)          ->
          (o1 -> i2 -> i1) ->
          Dev i1 o1           ->
          Dev i2 o2
```

refold out conn d



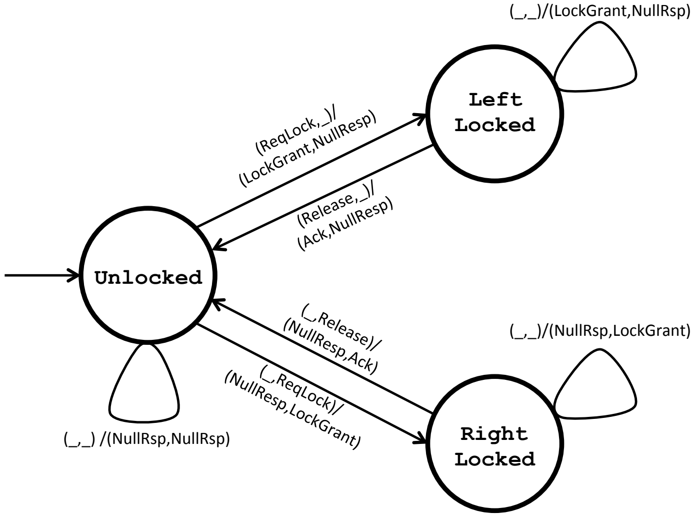
Representing **`Dev i o`** as a circuit



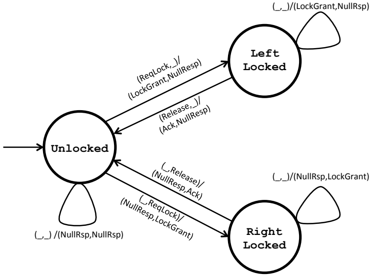


Mealy Machines

Ex: Mealy Machine for Mutex



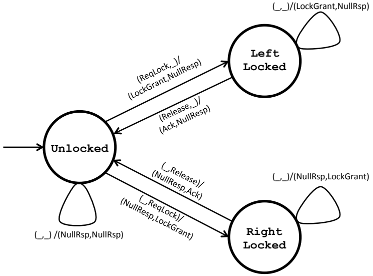
Implementing Mealy Machines in Connect Logic


(_,_)/(LockGrant,NullRsp)
Left Locked
(ReqLock,_)/ (LockGrant,NullResp)
(Release,_)/ (Ack,NullResp)
Unlocked
(_,Release)/ (NullResp,Ack)
(_,_)/(NullRsp,LockGrant)
(_,ReqLock)/ (NullResp,LockGrant)
(_,_) /(NullRsp,NullRsp)
Right Locked

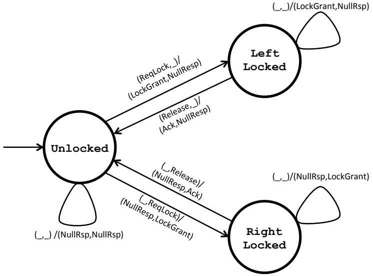
Implementing Mealy Machines in Connect Logic

States

```
data State = Unlocked | LeftLocked | RightLocked
data Req   = ReqLock | Release  | NullReq
data Rsp   = LockGrant | Ack | NullRsp
```


(_,_)/(LockGrant,NullRsp)
Left Locked
(ReqLock,_)/ (LockGrant,NullResp)
(Release,_)/ (Ack,NullResp)
Unlocked
(_,Release)/ (NullResp,Ack)
(_,_)/(NullRsp,LockGrant)
(_,ReqLock)/ (NullResp,LockGrant)
Right Locked
(_,_) /(NullRsp,NullRsp)

Implementing Mealy Machines in Connect Logic



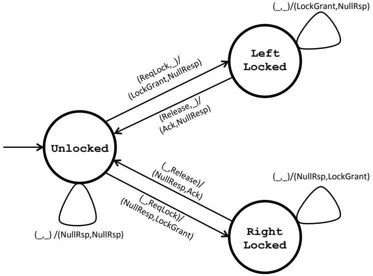
States

```
data State = Unlocked | LeftLocked | RightLocked
data Req   = ReqLock | Release   | NullReq
data Rsp   = LockGrant | Ack | NullRsp
```

Transition Function

```
delta ::  State -> (Req,Req) -> (State,(Rsp,Rsp))
delta Unlocked (ReqLock,_)
           = (LeftLocked, (LockGrant,NullRsp))
delta Unlocked (_,ReqLock)
           = (RightLocked, (NullRsp,LockGrant))
delta Unlocked (_,_)
           = (Unlocked, (NullRsp,NullRsp))
delta LeftLocked (Release,_)
           = (Unlocked, (Ack,NullRsp))
delta LeftLocked (_,_)
           = (LeftLocked, (LockGrant,NullRsp))
delta RightLocked (_,Release)
           = (Unlocked, (NullRsp,Ack))
delta RightLocked (_,_)
           = (RightLocked, (NullRsp,LockGrant))
```

Implementing Mealy Machines in Connect Logic


(_,_)/(LockGrant,NullRsp)
Left Locked
(ReqLock,_)/ (LockGrant,NullResp)
(Release,_)/ (Ack,NullResp)
Unlocked
(_,Release)/ (NullResp,Ack)
(_,ReqLock)/ (NullResp,LockGrant)
(_,_)/(NullRsp,LockGrant)
Right Locked
(_,_) /(NullRsp,NullRsp)

States

```
data State = Unlocked | LeftLocked | RightLocked
data Req   = ReqLock | Release   | NullReq
data Rsp   = LockGrant | Ack | NullRsp
```

Transition Function

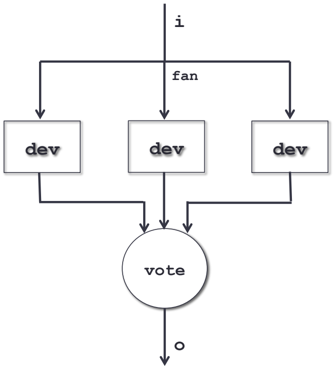
```
delta ::  State -> (Req,Req) -> (State,(Rsp,Rsp))
delta Unlocked (ReqLock,_)
          = (LeftLocked, (LockGrant,NullRsp))
delta Unlocked (_,ReqLock)
          = (RightLocked, (NullRsp,LockGrant))
delta Unlocked (_,_)
          = (Unlocked, (NullRsp,NullRsp))
delta LeftLocked (Release,_)
          = (Unlocked, (Ack,NullRsp))
delta LeftLocked (_,_)
          = (LeftLocked, (LockGrant,NullRsp))
delta RightLocked (_,Release)
          = (Unlocked, (NullRsp,Ack))
delta RightLocked (_,_)
          = (RightLocked, (NullRsp,LockGrant))
```

ReWire Device

```
mutex ::  Dev (Req, Req) (Rsp, Rsp)
mutex = iterS delta (Unlocked,(NullRsp,NullRsp))
```

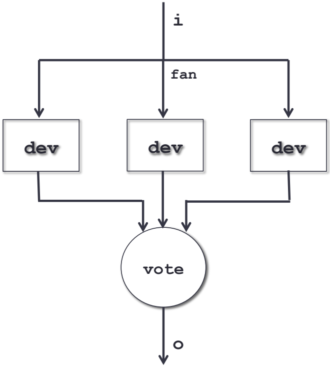
Simple Triple Modular Redundancy

The Rule of Three



Simple Triple Modular Redundancy

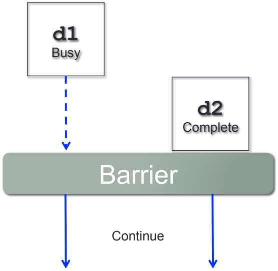
The Rule of Three



```
vote ::  (a,a,a) -> a
vote (a1,a2,a3) | a1 == a2  = a1
                | a1 == a3  = a1
                | a2 == a3  = a2
                | otherwise = a1

fan ::  a -> i -> (i,i,i)
fan _ i = (i,i,i)

tmr ::  Dev i o -> Dev i o
tmr dev = refold vote fan
               (dev <&> dev <&> dev)
```

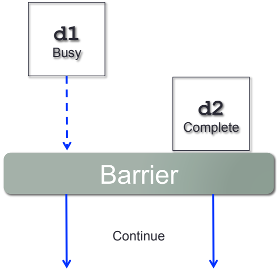
Programming Synchronization

Barriers



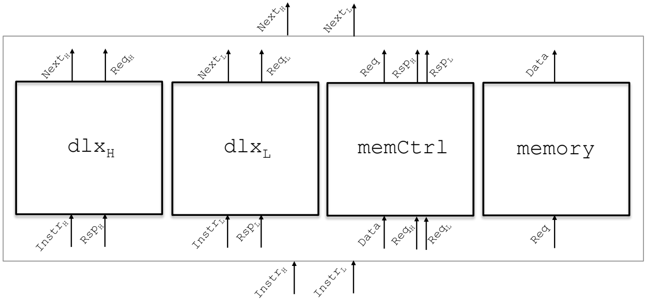
Programming Synchronization

Barriers



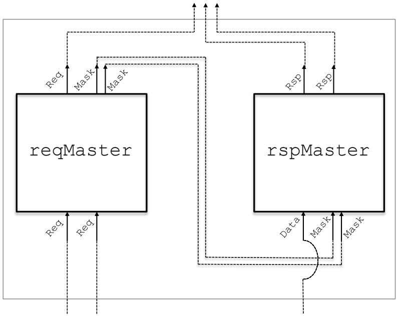
```
data Status a  = Busy | Complete a

barrier ::  Dev i1 (Status o1) ->
            Dev i2 (Status o2) ->
            Dev (i1,i2) (Status (o1,o2))
barrier d1 d2 =
    refold out inp
           (makeStall d1 <&> makeStall d2)
  where
    inp (Busy,Busy) (i1,i2)
                        = (Continue i1,Continue i2)
    inp (Complete l,Busy) (i1,i2)
                        = (Stall, Continue i2)
    inp (Busy,Complete r) (i1,i2)
                        = (Continue i1,Stall)
    inp (Complete l,Complete r) (i1,i2)
                        = (Continue i1,Continue i2)
    out (Busy,_)                  = Busy
    out (_,Busy)                  = Busy
    out (Complete a,Complete b) = Complete (a,b)
```

A Dual Core System realized in ReWire

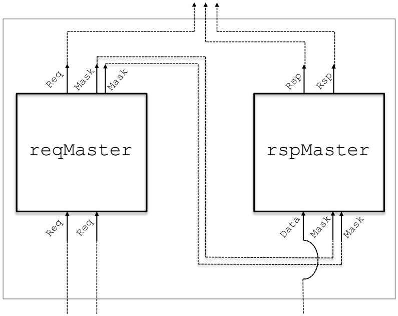


```
dlxℓ     :: Dev (Instrℓ,Rspℓ) (Nextℓ,Reqℓ)
memCtrl :: Dev (Data,ReqH,ReqL) (Req,RspH,RspL)
memory  :: Dev Req Data
system ::  Dev (InstrH,InstrL) (NextH,NextL)
system =
  refold
      systemOut
      systemIn
      (dlxH <&> dlxL <&> memCtrl <&> memory)
```

The Memory Controller Pattern



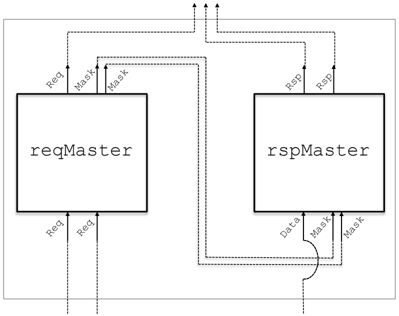
The Memory Controller Pattern



Access Policies as Functions

```
reqMaster = reqMaster_ policyH policyL
reqMaster_ ::
        Policy ->
        Policy ->
        Dev (Req,Req) (Req,(Mask,Mask))
```

The Memory Controller Pattern



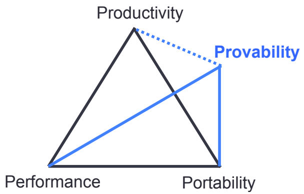
Access Policies as Functions

```
reqMaster = reqMaster_ policyH policyL
reqMaster_ ::
        Policy ->
        Policy ->
        Dev (Req,Req) (Req,(Mask,Mask))
```

Memory Controller Device

```
memCtrl :: Dev (Data,(Req,Req))
               (Req,(Rsp,Rsp))
memCtrl = refold
            outputSelect
            inputSelect
            (reqMaster <&> rspMaster)
```

Related Work



- HW Synthesis from DSLs
  - Delite [Olukotun, Ienne, et al.]
  - DSLs and Language Virtualization
  - The “Three P’s” + *Provability*
- Functional HDLs
  - Chisel, Bluespec, Lava
  - ReWire project motivated by formal methods & security
  - ReWire: functional concurrent language
- [Procter et al., 2015;2016] produce a verified secure dual-core processor in ReWire
- Cryptol

Summary, Conclusions & Future Work

- FPGA Programmability: [Andrews15] argues SE virtues precondition for wider adoption of Reconfigurable Tech
  - to enable productivity, reuse, scalability
- Encapsulated a wide variety of concurrency templates
  - Synchronization, Memory Protection, Voting
  - Each of which displays Abstraction, Modularity and Comprehensibility
    - Enabled by functional HDL ReWire
- Approach relies on semantically-faithful compiler
  - Mechanization in Coq; Compiler Verification
- Rewire is open source: `https://github.com/mu-chaco/ReWire`

THANKS!

* This research supported by the US National Science Foundation CAREER Award #0746509 and the US Naval Research Laboratory.