Specification of AIM Crypto Engines

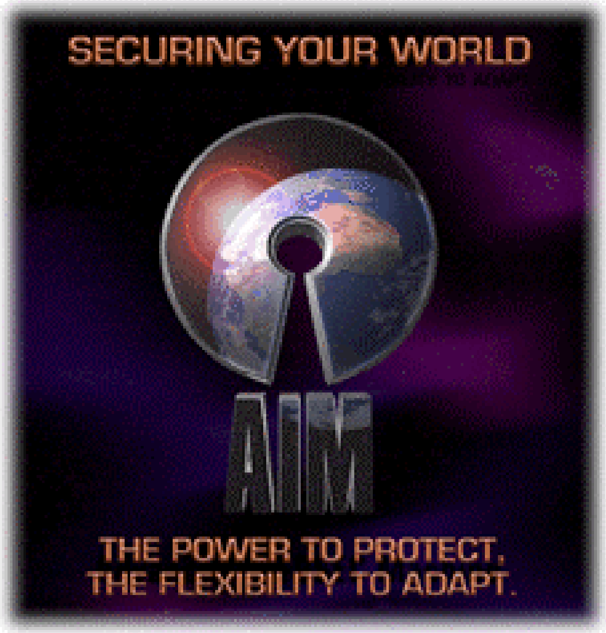
Mark Tullsen
John Launchbury
Thomas Nordin

Oregon Graduate Institute

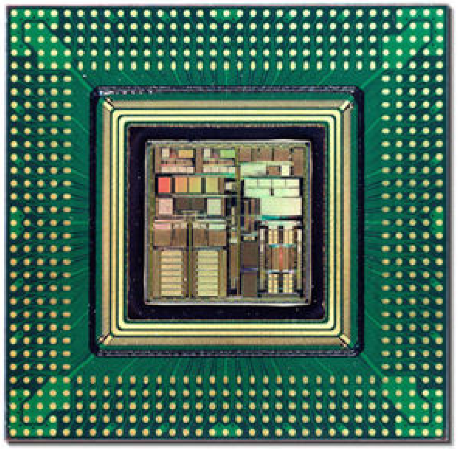
Road Map

- AIM Overview
- Specifying Cryptographic Algorithms
  - Block Ciphers on the PCE
  - Stream Ciphers on the CCE
- Verification
- Summary

AIM



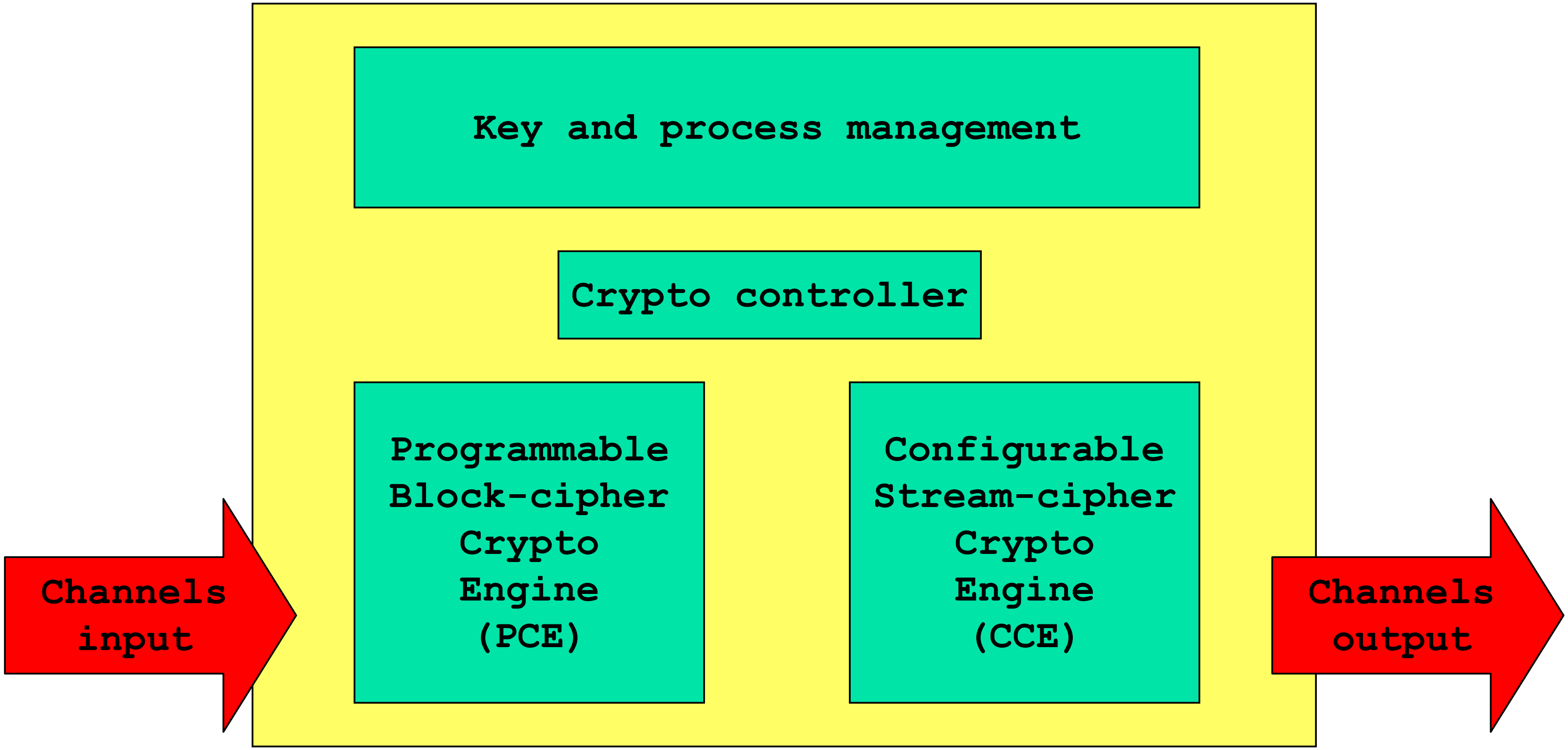
- Motorola AIM
  (Advanced INFOSEC Machine)



- On-board encryption engines
- MASK technology
  (Mathematically Assured Separation Kernel)
- Physically tamper-proof

www.motorola.com/GSS/SSTG/ISSPD/Embedded/AIM/

AIM Architecture

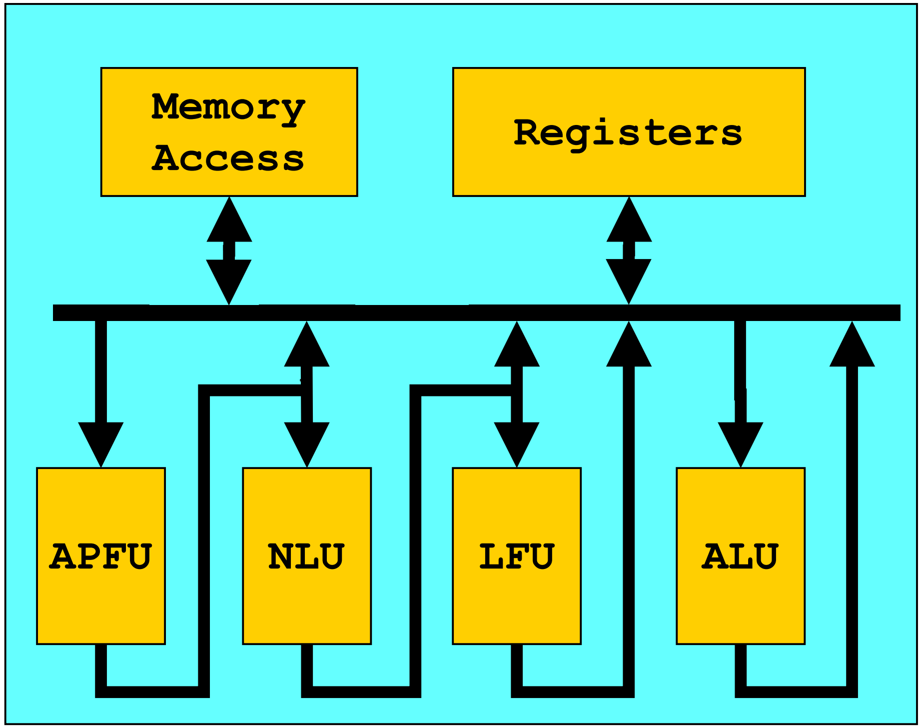


Road Map

- AIM Overview
- Specifying Cryptographic Algorithms
  - Block Ciphers on the PCE (previous work)
    - A DSL[1] for permutations and S-boxes
  - Stream Ciphers on the CCE
    - A DSL for bit-functions and feedback shift registers
- Verification
- Summary

[1] DSL – Domain Specific Language

PCE Architecture (Simplified)



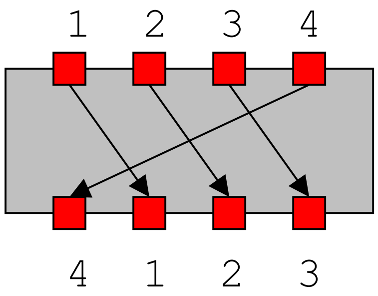
- Execution components
  - APFU (Permutation Function Unit)
    - 16 predefined permutations
  - NLU (Non-Linear Unit)
    - 16 one-bit memories
    - Independently addressable
  - LFU (Linear Function Unit)
    - XOR unit
  - ALU

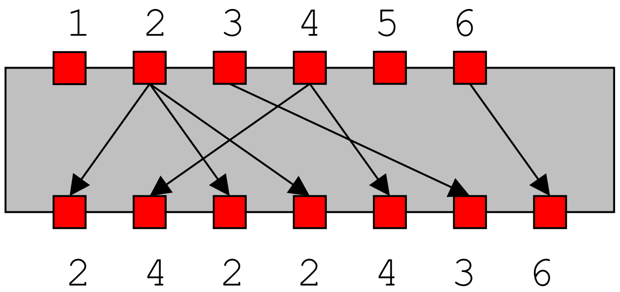
A Recipe for a DSL

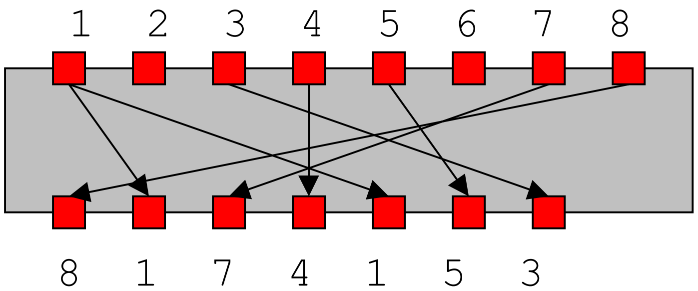
- Identify an abstraction (or Abstract Data Type)
  - Think “values” (functionally, not procedurally):
    - Yes: integers, complex numbers, polynomials, sequences, etc.
    - No: linked-list, arrays, pointers, etc.
- Develop compositional operators for it
  - Question: How can we create primitive values?
  - Question: How can we produce new values from old?
- Look for natural algebraic laws
  - Aids design of abstractions & operators
  - Provides understanding of the operators

Permutations (Abstraction No. 1)

- Sequence of numbers
  - Numbered left to right
  - Beginning at 1
- Examples
  - **[4,1,2,3]**
  - **[2,4,2,2,4,3,6]**
  - **[8,1,7,4,1,5,3]**
- Permutations can be any size
  - 16 or 32 bits is common

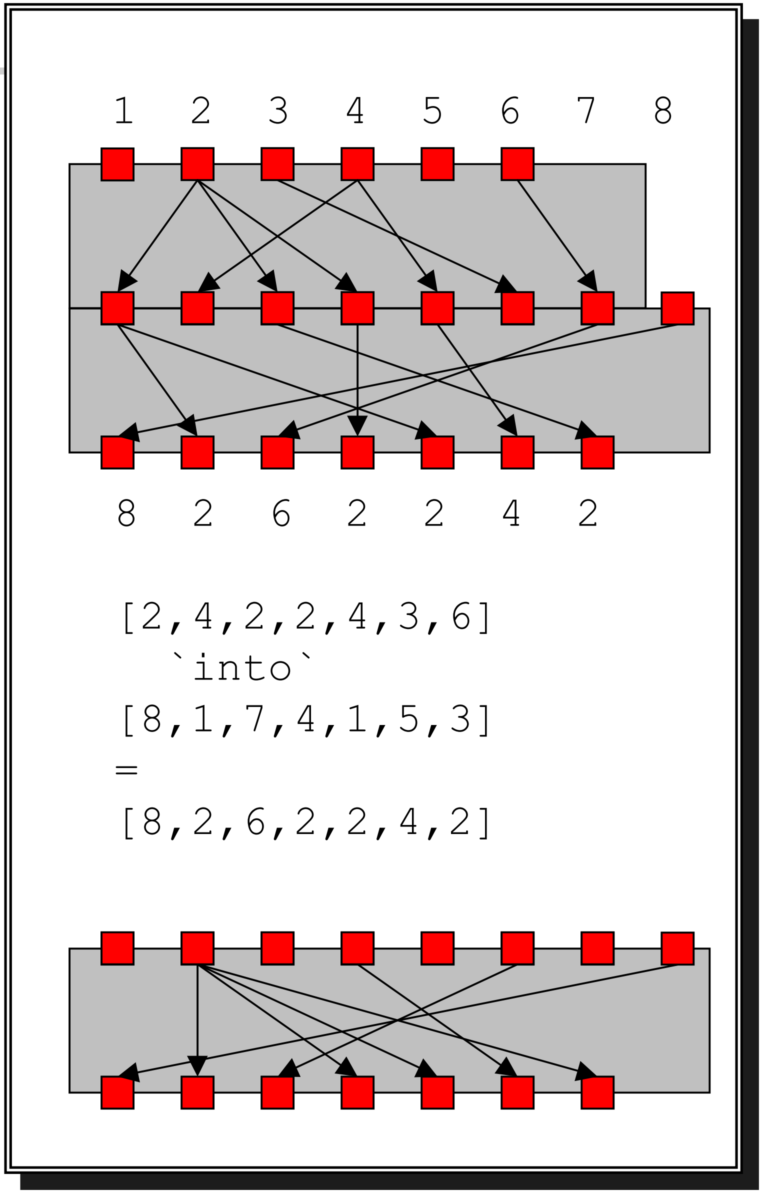






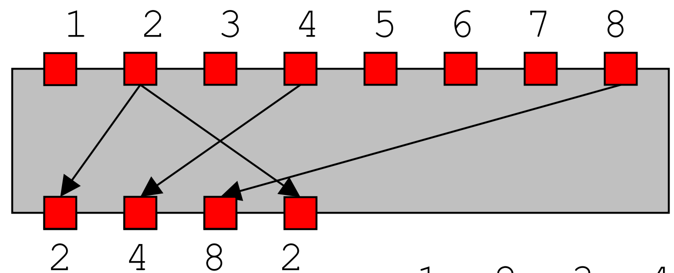
`into` Operator

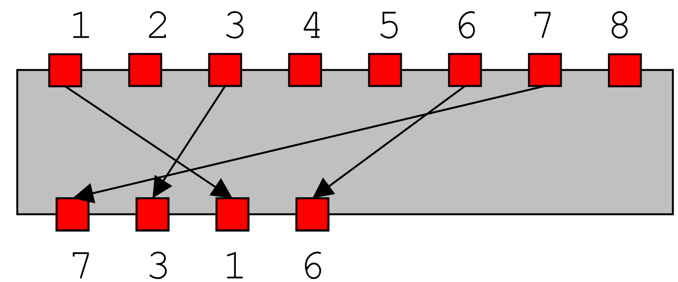
- Pipe the output of one permutation into the input of another
- Like function composition



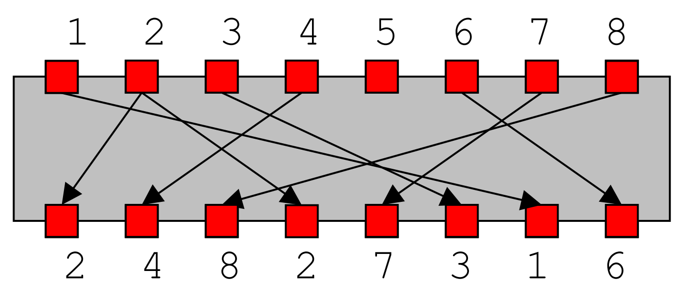
++ Operator

- Joins two permutations together, side by side
  - Each permutation draws from the same input bits
  - Obtained simply by appending the two sequences together






```
[2,4,8,2] ++ [7,3,1,6]
 = [2,4,8,2,7,3,1,6]
```



More Operations

```
xs `select` [n..m]
```

Selects bits n through m from xs

```
xs <<< n
```

Rotate xs left by n

```
xs >>> n
```

Rotate xs right by n

```
pad n xs
```

Pad xs on left to be n-bits wide

```
xs `beside` ys
```

Combine xs and ys in parallel

```
size xs
```

The number of bits output by xs (length of sequence)

Permutation Laws

- Size

```
size (xs ++ ys)         = size xs + size ys
size (xs `beside` ys) = size xs + size ys
size (xs `into` ys)   = size ys
size (pad n xs)         = n
```

- Rotating

```
(xs >>> m) >>> n  =  xs >>> m+n
(xs <<< m) <<< n  =  xs <<< m+n

 xs >>> 0 = xs
 xs <<< 0 = xs

(xs >>> m) <<< n =
          if m > n then xs >>> (m-n) else xs <<< (n-m)
```

Permutation Laws (2)

- \`into\`

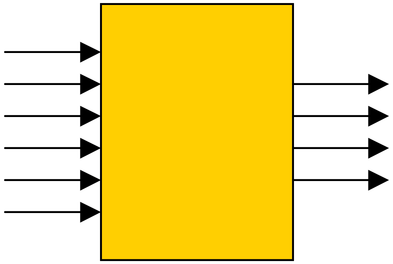
```
[1..] `into` xs = xs
xs `into` [1..size xs] = xs
xs `into` (ys ++ zs) = (xs `into` ys) ++ (xs `into` zs)
xs `into` (ys <<< n) = (xs `into` ys) <<< n
xs `into` (ys >>> n) = (xs `into` ys) >>> n
```

- Associativity

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
(xs `beside` ys) `beside` zs
                          = xs `beside` (ys `beside` zs)
(xs `select` ys) `select` zs
                          = xs `select` (ys `select` zs)
```

S-boxes (Abstraction No. 2)

- Every crypto-algorithm needs non-linear components
  - Multiplication (RC6)
  - Galois field inversion (Rijndael)
  - DES has 8 separate S-boxes; each 6-bit in, 4-bit out



- An S-box is an arbitrary function combined with a "addressing permutation"

S-box Operations & Laws

- Creating S-boxes:

```
sbox :: Perm -> Int -> [Integer] -> Sbox
```

- Combining S-boxes:

```
pack   :: Perm -> [Sbox] -> Sbox
extend :: [Sbox] -> Sbox
intoS  :: Perm -> Sbox -> Sbox
```

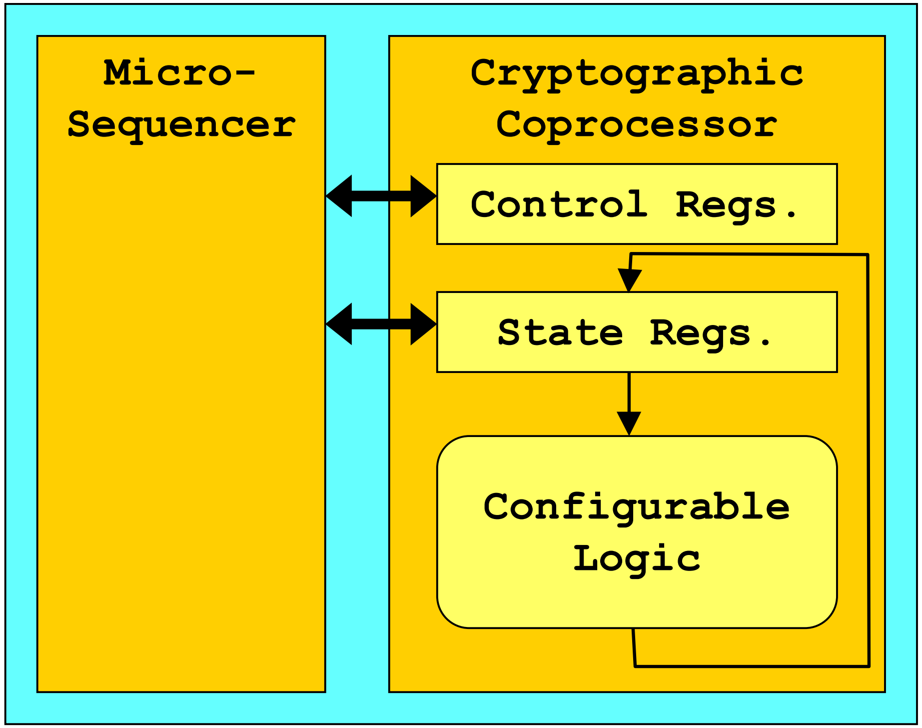
- Laws:

```
p `intoS` (sbox q n xs) = sbox (p `into` q) n xs
```

Road Map

- AIM Overview
- Specifying Cryptographic Algorithms
  - Block Ciphers on the PCE
    - A DSL for permutations and S-boxes
  - Stream Ciphers on the CCE
    - A DSL for bit-functions and feedback shift registers
- Verification
- Summary

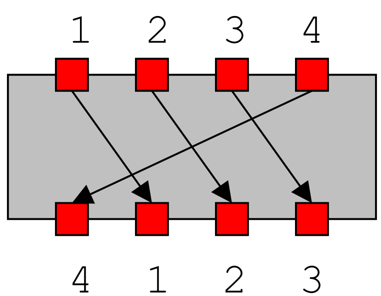
CCE Architecture (Simplified)



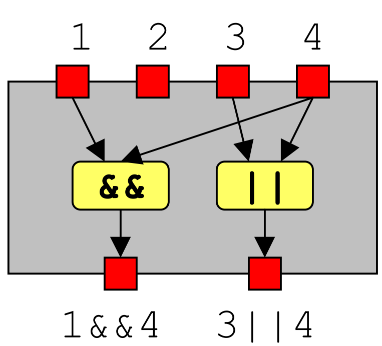
- Micro-sequencer
  - Simple RISC architecture
  - Interfaces with Crypto Controller
  - Controls Cryptographic Coprocessor
- Cryptographic Coprocessor
  - Control Registers
  - State Registers
  - Configurable Logic
    - The difficulty of programming the CCE lies in specifying this

Bit-Functions (Abstraction No. 3)

- Permutations allow for moving bits around



- Bit-Functions allow for Boolean functions



Bit-Function Examples

- Rotate (4 to 4 Bit-Function)

```
[4,1,2,3]
```

  - Note: All permutations are Bit-Functions!

- Odd Parity (4 to 1 Bit-Function)

```
[1 `xor` 2 `xor` 3 `xor` 4]
```

- Two Bit Adder (4 to 2 Bit-Function)

```
[ 1 `xor` 3
, 2 `xor` 4 `xor` (1 && 3)
]
```

Bit-Function Operations

- Permutation operations extend to Bit-Functions:
  - \`into\`
  - ++
  - \`select\`
  - <<<, >>>
  - pad
  - \`beside\`
  - size
  - …

Bit-Function Operations

- Operations on “Input Bits”:
  - Standard Boolean operators (overloaded):
    1 && 2, 1 || 2, …
  - Additional operators:
    true, false, ite 1 2 3, 1 `xor` 2, …

- Bit-Function Operations:
  ites b [x1,x2,…] [y1,y2,…] = [ite b x1 y1, ite b x2 y2, …]

Bit-Function Laws

- Permutation laws extend to Bit-Functions

```
(xs >>> m) >>> n  =  xs >>> m+n
```

- Boolean laws apply to each "bit"

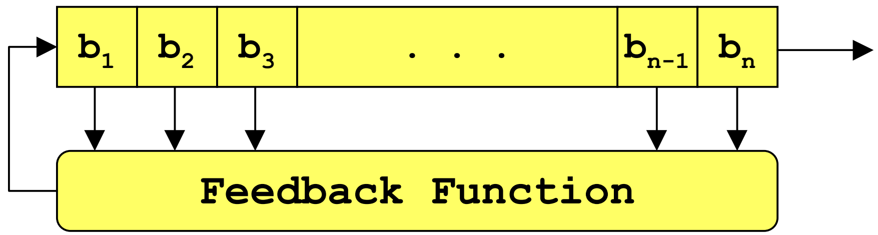
```
[1 && true] = [1]
```

- Bit-Function Laws

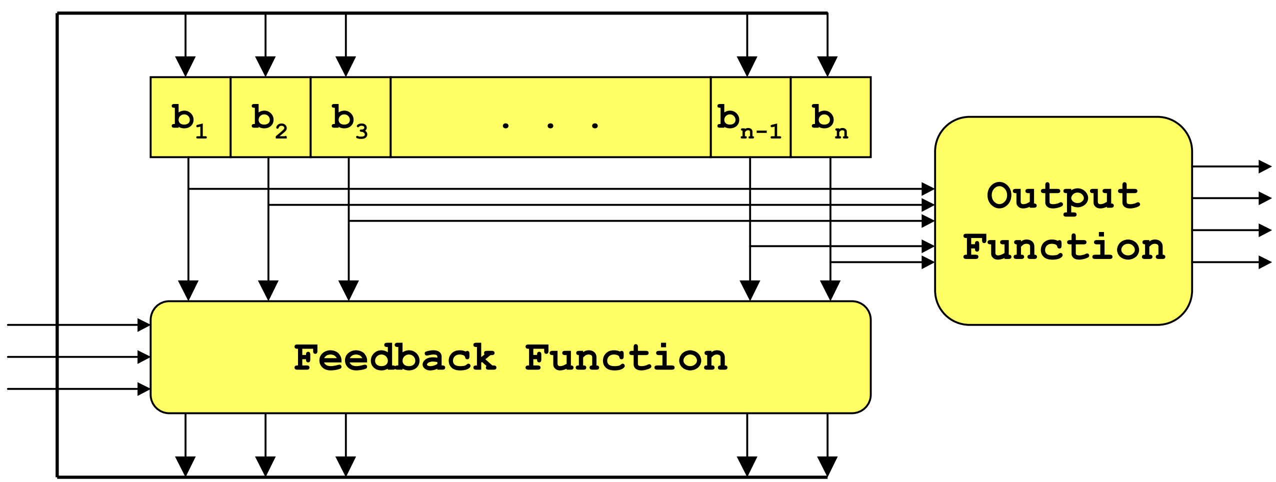
```
ites a (ites b xs ys) zs =
           ites b (ites a xs zs) (ites a ys zs)
```

A Common Structure in Stream Ciphers

- Feedback Shift Register (FSR)



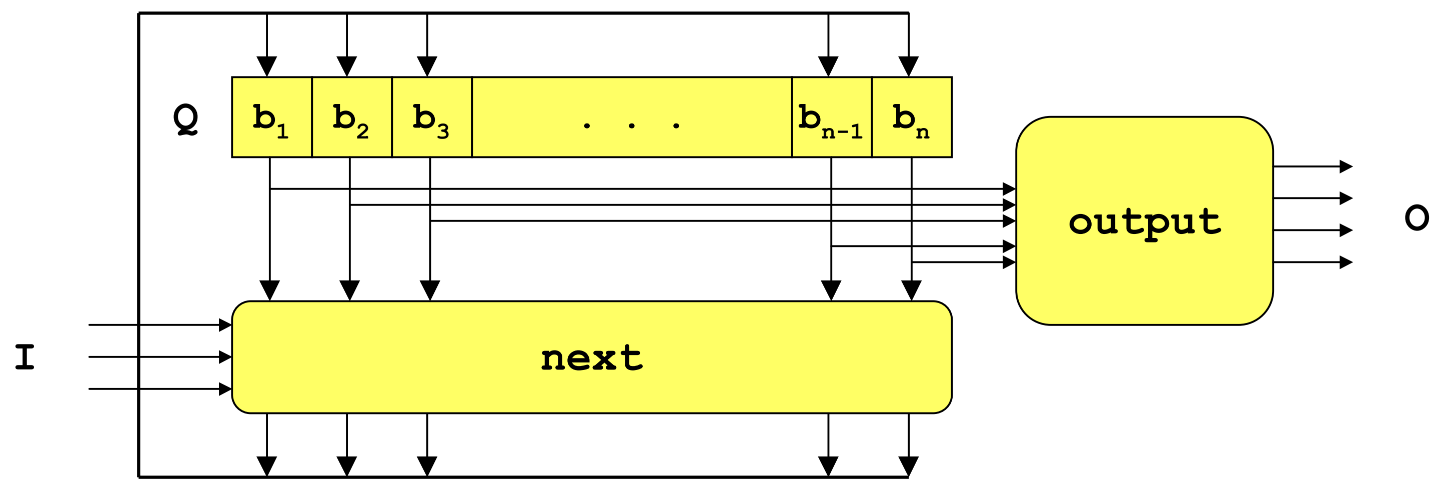
- Generalized FSR



Generalized FSR (Abstraction No. 4)

- FSR = (next,output,inputWidth)

```
next        :: BitFunction    (Q × I → Q)
output      :: BitFunction    (Q → O)
inputWidth  :: Int
```

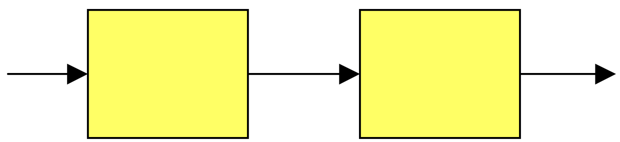


FSR Compared to Moore Machine

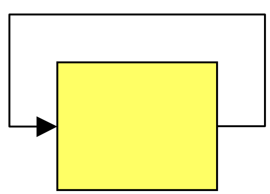
- Moore Machine:
  - Q = set of states
  - I = set of inputs
  - O = set of outputs
  - q0 :: Q = initial state
  - next :: $Q \times I \rightarrow Q$ = next state function
  - output :: $Q \rightarrow O$ = output function
- FSR Differences:
  - FSR has no initial state
  - State (Q) represented as a bit-vector, not arbitrary set
  - Input and output (I and O) are bit-vectors, not sets

FSR Operators: Basic Three

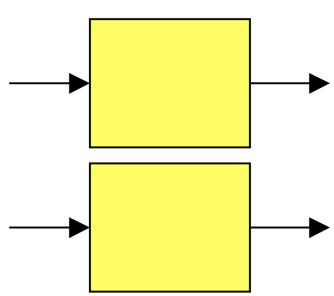
- compose :: FSR -> FSR -> FSR (ab)



- cycle :: FSR -> FSR (a*)

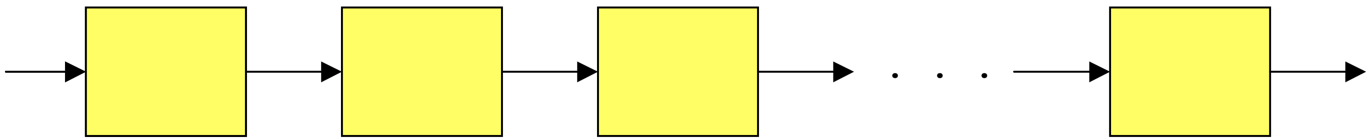


- parallel :: FSR -> FSR -> FSR (a|b)

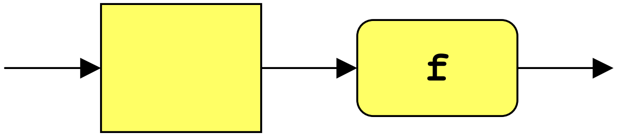


More FSR Operators

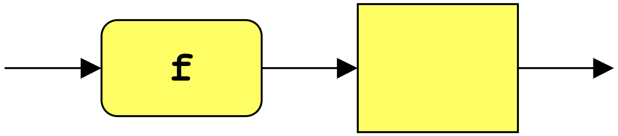
- cascade :: [FSR] -> FSR



- outputInto :: FSR -> BitFunction -> FSR

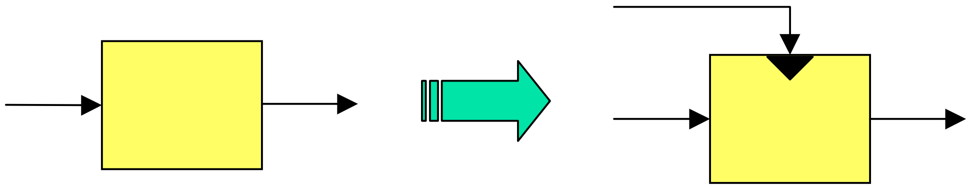


- intoInput :: BitFunction -> FSR -> FSR

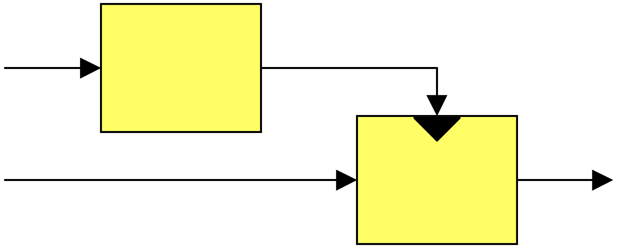


And More FSR Operators

- clocked :: FSR -> FSR



- clocks :: FSR -> FSR -> FSR



- N.B.: A FSR does not have a clock.

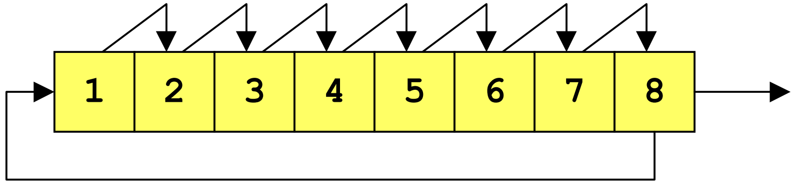
Example: Simple Shift Register

shift :: Int -> FSR

shift n = ([1..n] >>> 1, [n], 0)

Example:

shift 8 = ([8,1,2,3,4,5,6,7], [8], 0)



Note:

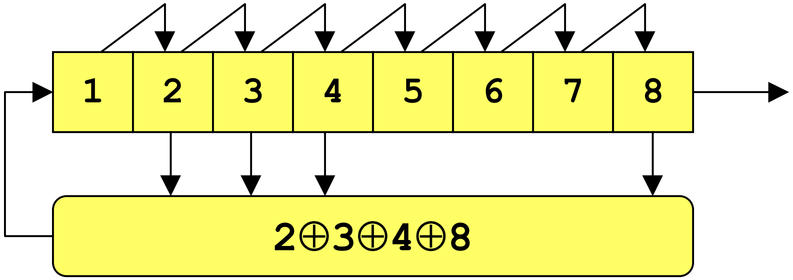
FSR = (BitFunction,BitFunction,Int)

Example:
Linear Feedback Shift Register

lfsr :: [Int] -> FSR

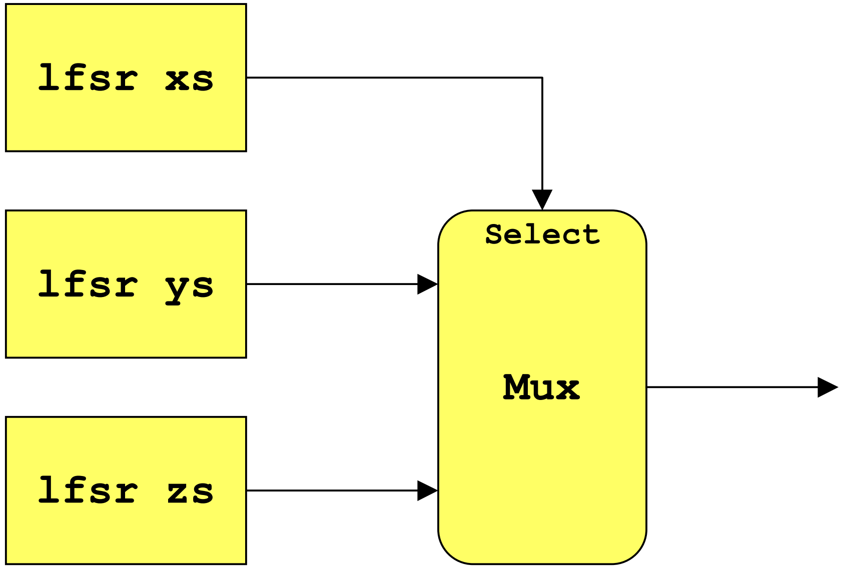
Example:

lfsr [2,3,4,8] =
  ([(2 `xor` 3 `xor` 4 `xor` 8), 1, 2, 3, 4, 5, 6, 7]
  ,[8]
  ,0)

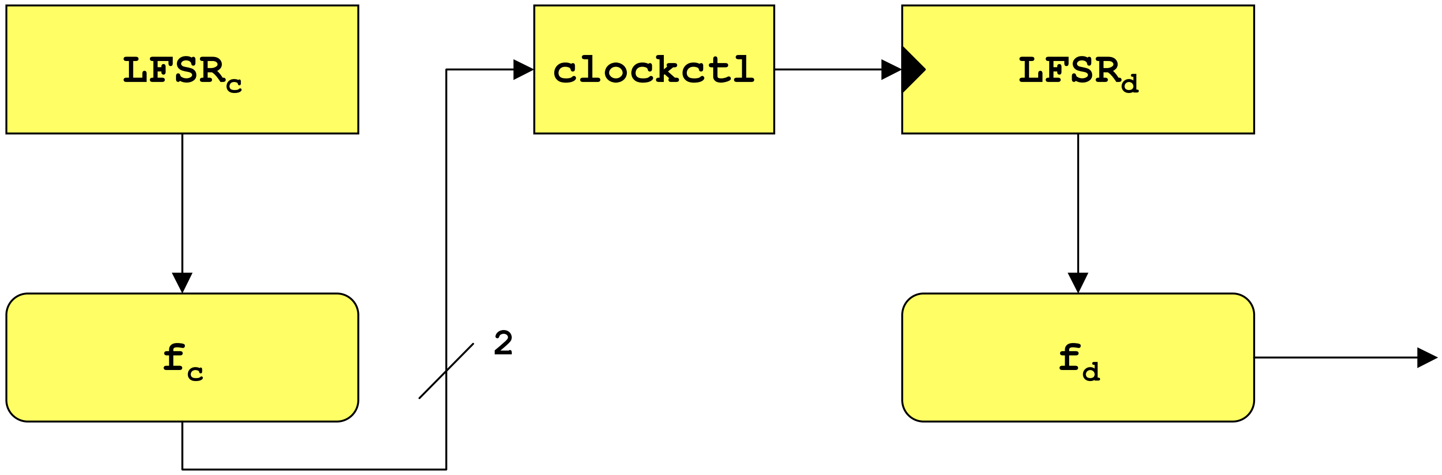


Example: Geffe Generator

```
geffe :: [Int] -> [Int] -> [Int] -> FSR
geffe xs ys zs =
  (lfsr xs `parallel` lfsr ys `parallel` lfsr zs)
  `outputInto` [ite 1 2 3]
```



Example: LILI-128



clockctl

Input	Output sequence
0	0,0,0,1
1	0,0,1,1
2	0,1,1,1
3	1,1,1,1

Example: LILI-128

```
lili128  =
  cascade [ shift 4 `clocks`
                lfsr′ [2,14,15,17,31,33,35,39] [12,20]
             , clockctl `clocks`
                lfsr′ [1,39,42,53,55,80,83,89] fd
             ]
fd         = [1,2,4,8,13,21,31,45,66,81] `into` [fd']
clockctl =
  ([4,1,2,3] ++ ites 1 [i1 && i2, i2, i1 || i2]
                        [false, 5, 6]
  ,[1 || 7]
  ,2)
```

FSR Laws

- Associative Laws

```
(x `parallel` y) `parallel` z = x `parallel` (y `parallel` z)

(x `compose` y) `compose` z = x `compose` (y `compose` z)
```

- Moving computation between FSRs

```
(x `outputInto` f ) `compose` y = x `compose` (f `inputInto` y)
```

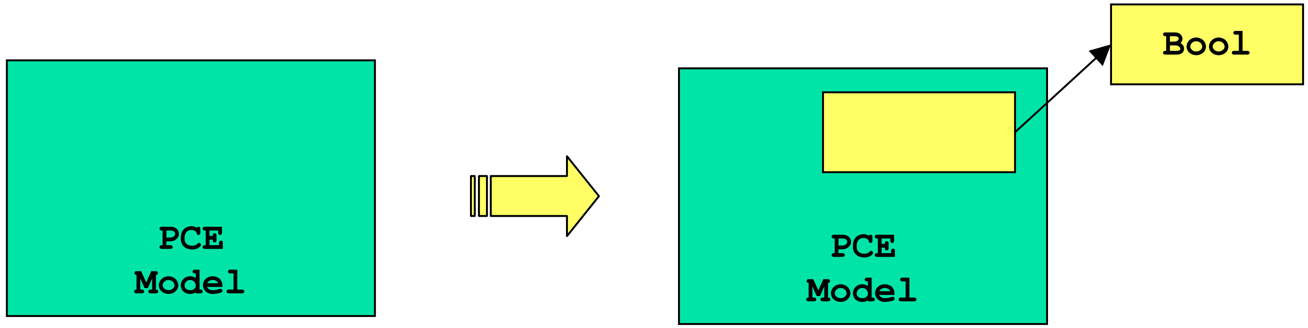
Road Map

- AIM Overview
- Specifying Cryptographic Algorithms
  - Block Ciphers on the PCE
  - Stream Ciphers on the CCE
- Verification
  - Is an implementation (micro-code and configuration) equivalent to the specification?
- Summary

Verification: Three Steps

- Parameterize model w.r.t. bit-operations on registers
- Instantiate to three implementations of "Booleans"
  (Giving us three related models)
- Do testing and verification using these models

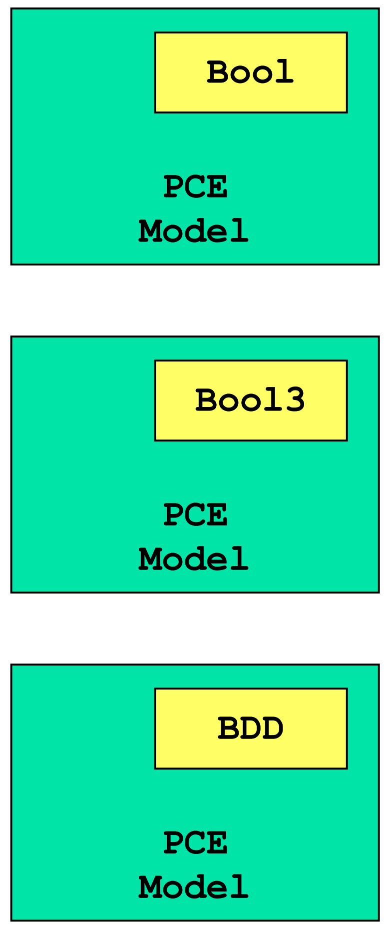
Step 1: Parameterize Model



- Transform PCE Model:
  - Parameterize over Boolean operators on machine registers and flags
    - Achieved with Haskell’s type classes

Step 2: Instantiate Model Thrice

- Apply parameterized model to three implementations of Boolean operators

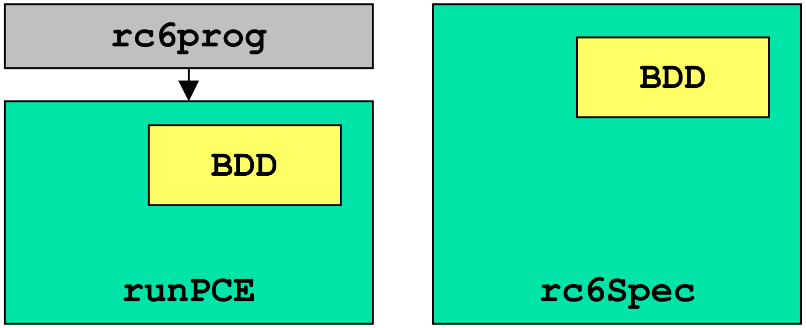


Equivalent to original model

More abstract than original model

Symbolic execution of original model

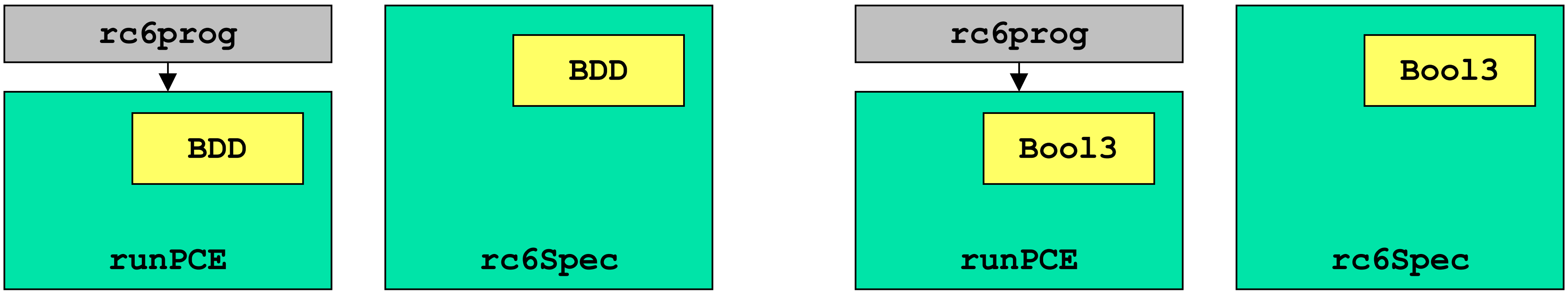
Step 3: Use BDD Model to Verify




```
hugs> runPCE rc6i' i `isEqual` rc6s' i
True
```

- "i" a symbolic value
- rc6i' and rc6s' – program segments.
- What if verification doesn't succeed?

Step 3: Use Bool3 Model to Test




```
hugs> runPCE rc6i' i `isEqual` rc6s' i
False
```

- Verification is complemented by testing:

- Debug specification:

```
rc6Spec input1 == output1
rc6Spec input2 == output2
. . .
```

- Debug "**runPCE**" and "**rc6prog**":

```
runPCE rc6prog input1 == output1
runPCE rc6prog input2 == output2
. . .
```

Step 1: Parameterize Model

```
data Bool = True | False

True  && x = x
False && x = False

False || x = x
True  || x = True

...
```

```
class Boolean b where
  true  :: b
  false :: b
  (&&)  :: b -> b -> b
  (||)  :: b -> b -> b
  not   :: b -> b
  ite   :: b -> b -> b -> b
  nor   :: b -> b -> b
  xor   :: b -> b -> b

  ite c a b =
    c && a  ||  not c && b
  nor a b = not (a || b)
  xor a b =
    a && not b  ||  not a && b
```

Step 1: Parameterize Model

- Generalizing PCE model to use Boolean
  - Sometimes automatic:
    - a && b
  - Sometimes easy:
    - if a then b else c => ite a b c
  - Sometimes harder:
    - lookup table (toInt bs) => ???

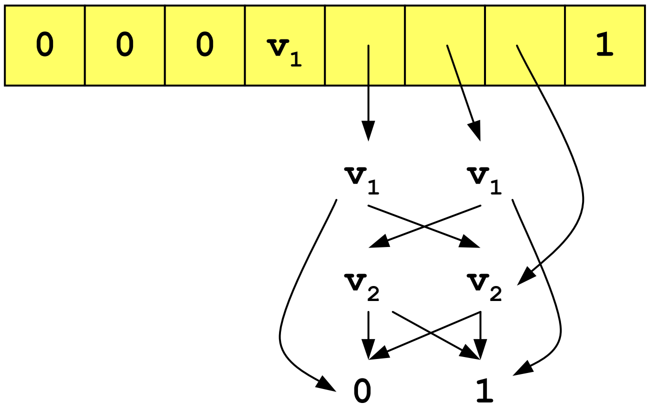
Step 2: Instantiate Model Thrice

0	0	0	1	1	0	1	1

```
instance Boolean Bool  where
  …
```

0	0	1	?	?	0	1	?

```
instance Boolean Bool3 where
  …
```




```
instance Boolean BDD   where
  …
```

Step 2: Instantiate Model Thrice

```
data Bool3 = B3True | B3False | B3Unk

instance Boolean Bool3 where
  true  = B3True
  false = B3False

  B3True  && x  = x
  B3False && x  = B3False
  B3Unk   && _  = B3Unk

  not B3True  = B3False
  not B3False = B3True
  not B3Unk   = B3Unk
  . . .
```

Step 2: Instantiate Model Thrice

```
instance Boolean BDD where
  true  = bddTrue
  false = bddFalse

  (&&)  = bddAnd
  (||)  = bddOr
  not   = bddNot
```

- BDD primitives implemented by foreign calls to Buddy BDD library

Step 3: Use Models to Verify/Test

```
Hugs[AIM]> load "square.aim"
R0 = 00000000000000000000000000000000  R1 = 00000000000000000000000000000000
R2 = 00000000000000000000000000000000  R3 = 00000000000000000000000000000000
R4 = 00000000000000000000000000000000  R5 = 00000000000000000000000000000000
R6 = 00000000000000000000000000000000  R7 = 00000000000000000000000000000000

->0: R7 = 00000000000000000000000000001000;
  1: Shift_Count = 00000000000000000000000000001000;
  2: PERMUTE(APFU10, R31, R31, R0, R7) | R1 = P1 | R2 = P2 | R3 = P3;
  3: PERMUTE(APFU2, R31, R31, R0, R31);
  4: PERMUTE(APFU4, R31, R31, R0, R31) | R5 = NL | R3 = SUB(R1, R3);
  5: PERMUTE(APFU1, R31, R31, R0, R31) | R2 = SUB(R1, R2);
  6: PERMUTE(APFU3, R31, R31, R0, R31);
```

Step 3: Use Models to Verify/Test

```
Hugs[AIM]> setReg R0 newVars16
R0 = 0000000000000000################  R1 = 00000000000000000000000000000000
R2 = 00000000000000000000000000000000  R3 = 00000000000000000000000000000000
R4 = 00000000000000000000000000000000  R5 = 00000000000000000000000000000000
R6 = 00000000000000000000000000000000  R7 = 00000000000000000000000000000000

->0: R7 = 00000000000000000000000000001000;
  1: Shift_Count = 00000000000000000000000000001000;
  2: PERMUTE(APFU10, R31, R31, R0, R7) | R1 = P1 | R2 = P2 | R3 = P3;
  3: PERMUTE(APFU2, R31, R31, R0, R31);
  4: PERMUTE(APFU4, R31, R31, R0, R31) | R5 = NL | R3 = SUB(R1, R3);
```

Step 3: Use Models to Verify/Test

```
Hugs[AIM]> step 4
R0 = 0000000000000000################  R1 = 000100000000000000001000########
R2 = 00000000########0000000000000000  R3 = 00000000########00000000########
R4 = 00000000000000000000000000000000  R5 = 00000000000000000000000000000000
R6 = 00000000000000000000000000000000  R7 = 00000000000000000000000000001000

  2: PERMUTE(APFU10, R31, R31, R0, R7) | R1 = P1 | R2 = P2 | R3 = P3;
  3: PERMUTE(APFU2, R31, R31, R0, R31);
->4: PERMUTE(APFU4, R31, R31, R0, R31) | R5 = NL | R3 = SUB(R1, R3);
  5: PERMUTE(APFU1, R31, R31, R0, R31) | R2 = SUB(R1, R2);
  6: PERMUTE(APFU3, R31, R31, R0, R31);
```

Step 3: Use Models to Verify/Test

```
Hugs[AIM]> step 4
R0 = 0000000000000000################  R1 = 00010000000000000000001000########
R2 = 0000############00001000########  R3 = 0000############0000############
R4 = 00000000000000000000000000000000  R5 = 0000000000000000##############0#
R6 = 00000000000000000000000000000000  R7 = 00000000000000000000000000001000


  6: PERMUTE(APFU3, R31, R31, R0, R31);
  7: PERMUTE(APFU11, R31, R31, R3, R31) | R4 = P2 | R3 = LINEAR(P2_P3) | R1 =
   ADD(R5, NL);
->8: R6 = ADD(A, A, LSL);
  9: PERMUTE(APFU2, R31, R31, R2, R31) | R3 = SUB(R3, R4);
 10: PERMUTE(APFU2, R31, R31, A, R31) | R6 = SUB(R6, NL, LSL);
```

Step 3: Use Models to Verify/Test

```
Hugs[AIM]> step 8
R0 = ##############################0#  R1 = 000000000000000#################
R2 = 0000############00001000########  R3 = 0000###0########0000###0########
R4 = 00000000########00000000########  R5 = ##############################0#
R6 = ##############################0#  R7 = 00000000000000000000000000001000

  12: PERMUTE(APFU4, R31, R31, R3, R31) | R5 = ADD(R5, R1, LSL);
  13: PERMUTE(APFU12, R31, R31, R6, R31) | R5 = SUB(A, NL, LSL);
  14: R0 = ADD(P1, A);
->15: JMP(15);

Hugs[AIM]> R0 `isEqual` (newVars16 * newVars16)
R0 == ##############################0# --> True
```

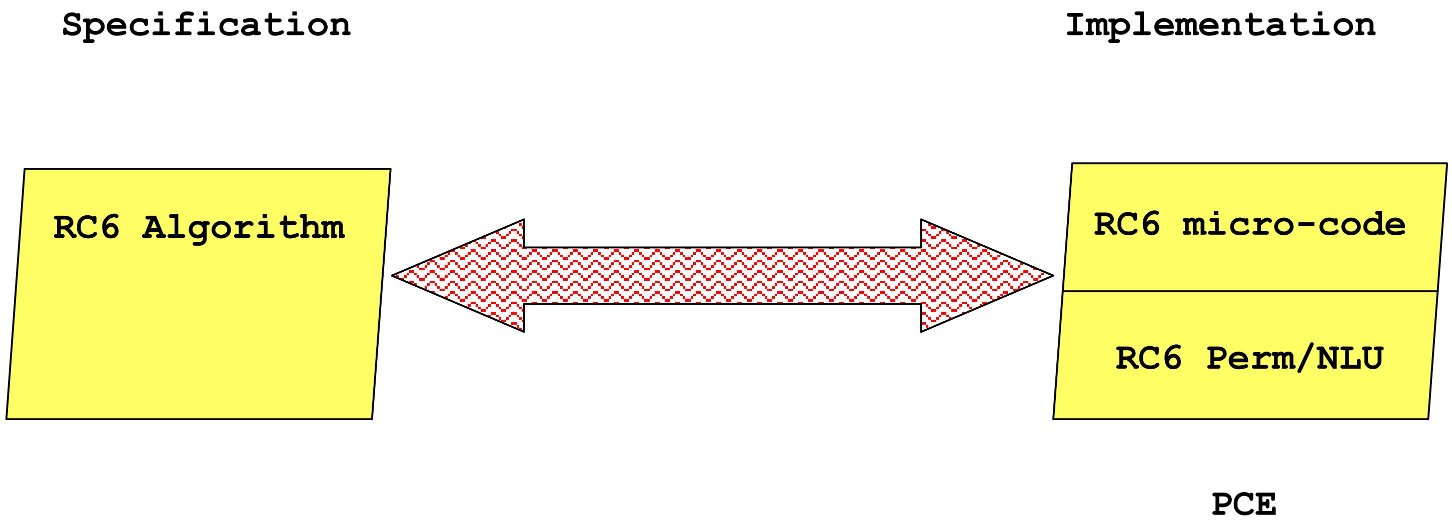
Road Map

- AIM Overview
- Specifying Cryptographic Algorithms
  - Block Ciphers on the PCE
  - Stream Ciphers on the CCE
- Verification
- Summary

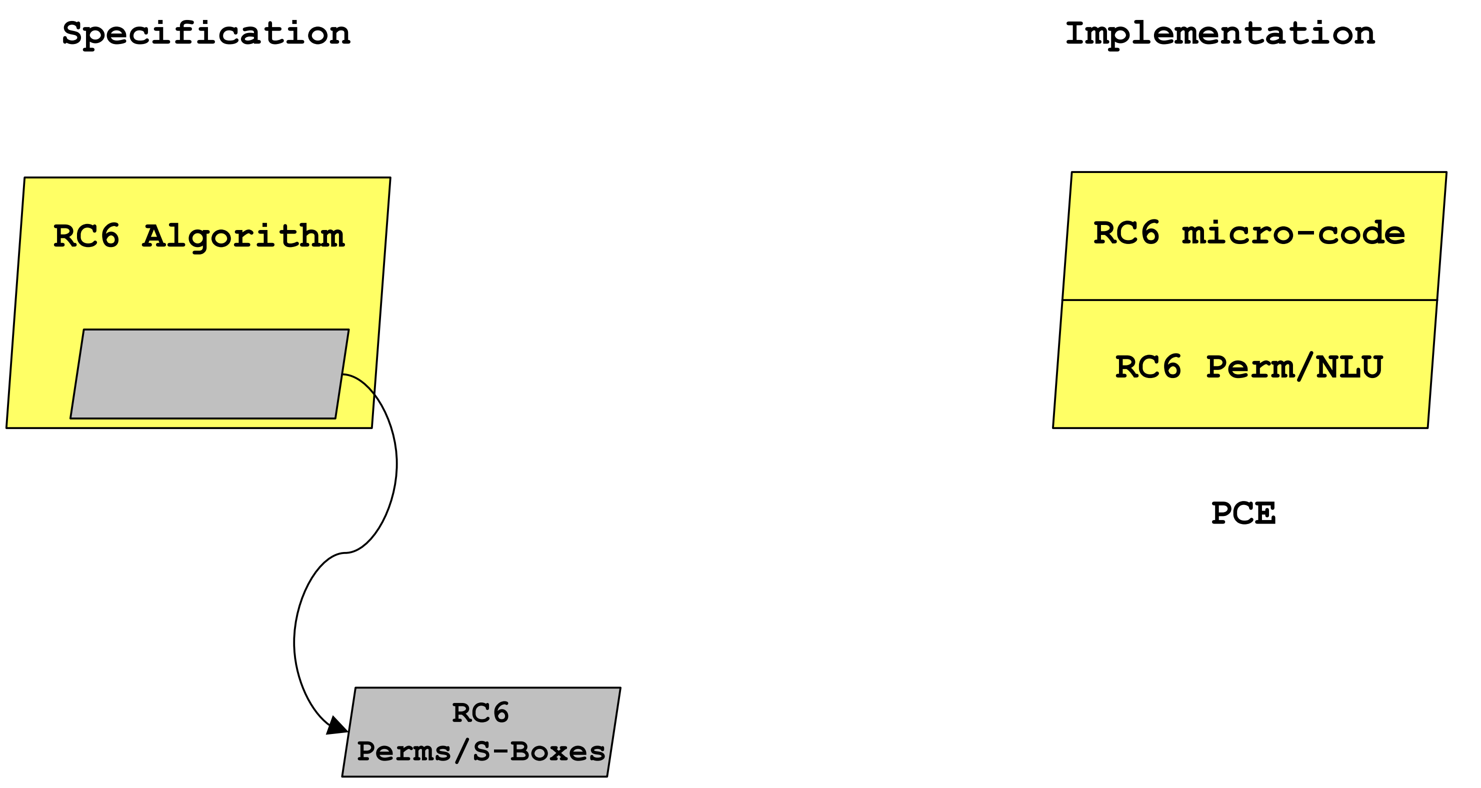
Summary

- Large gap between specification & implementation
- Multiple techniques to span the gap
  - Domain Abstractions (DSL)
  - Configuration (PNLFU or Logic) Generators
  - Machine Models
    - Parameterized Models: Standard, Symbolic
  - Executable Specifications
- Haskell is the infrastructure for it all

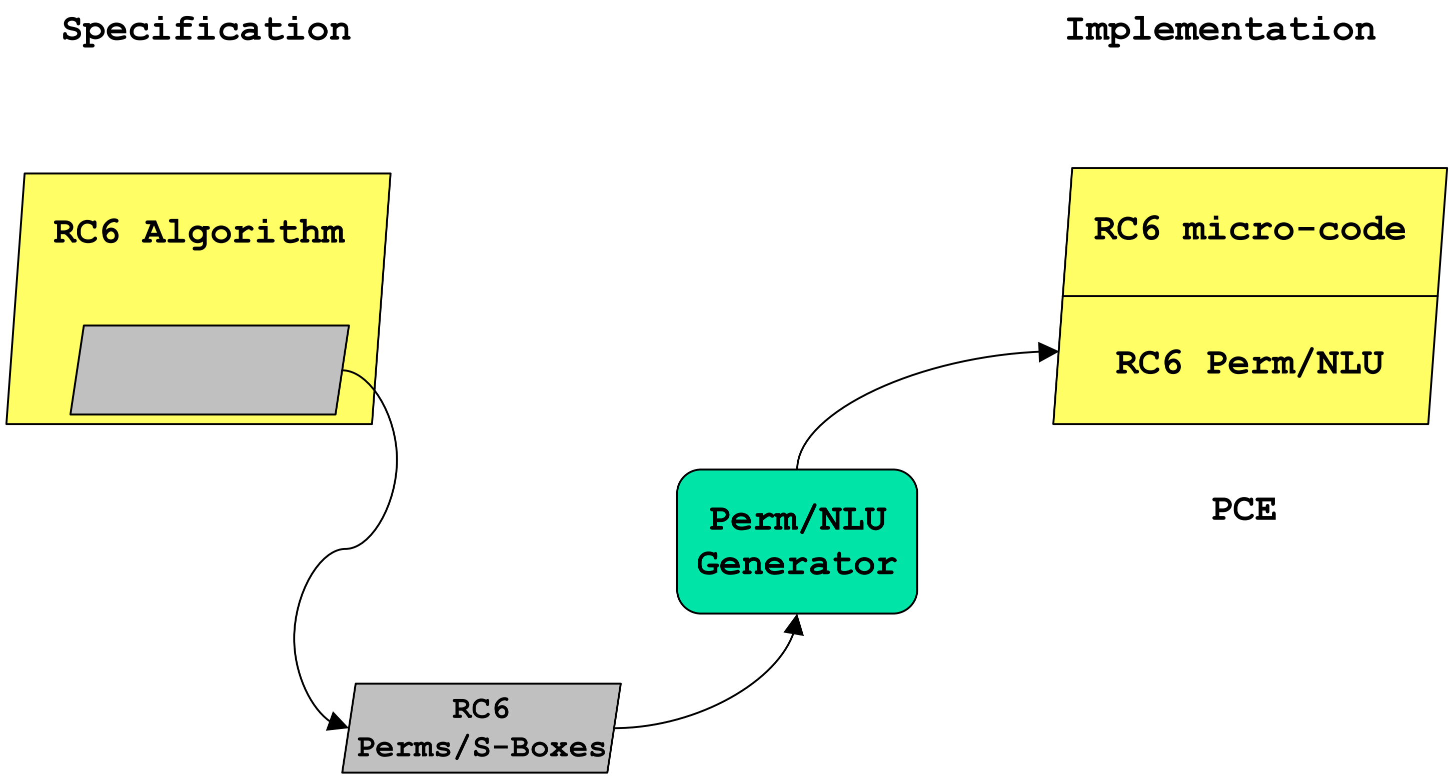
A Large Gap


Specification
Implementation
RC6 Algorithm
RC6 micro-code
RC6 Perm/NLU
PCE

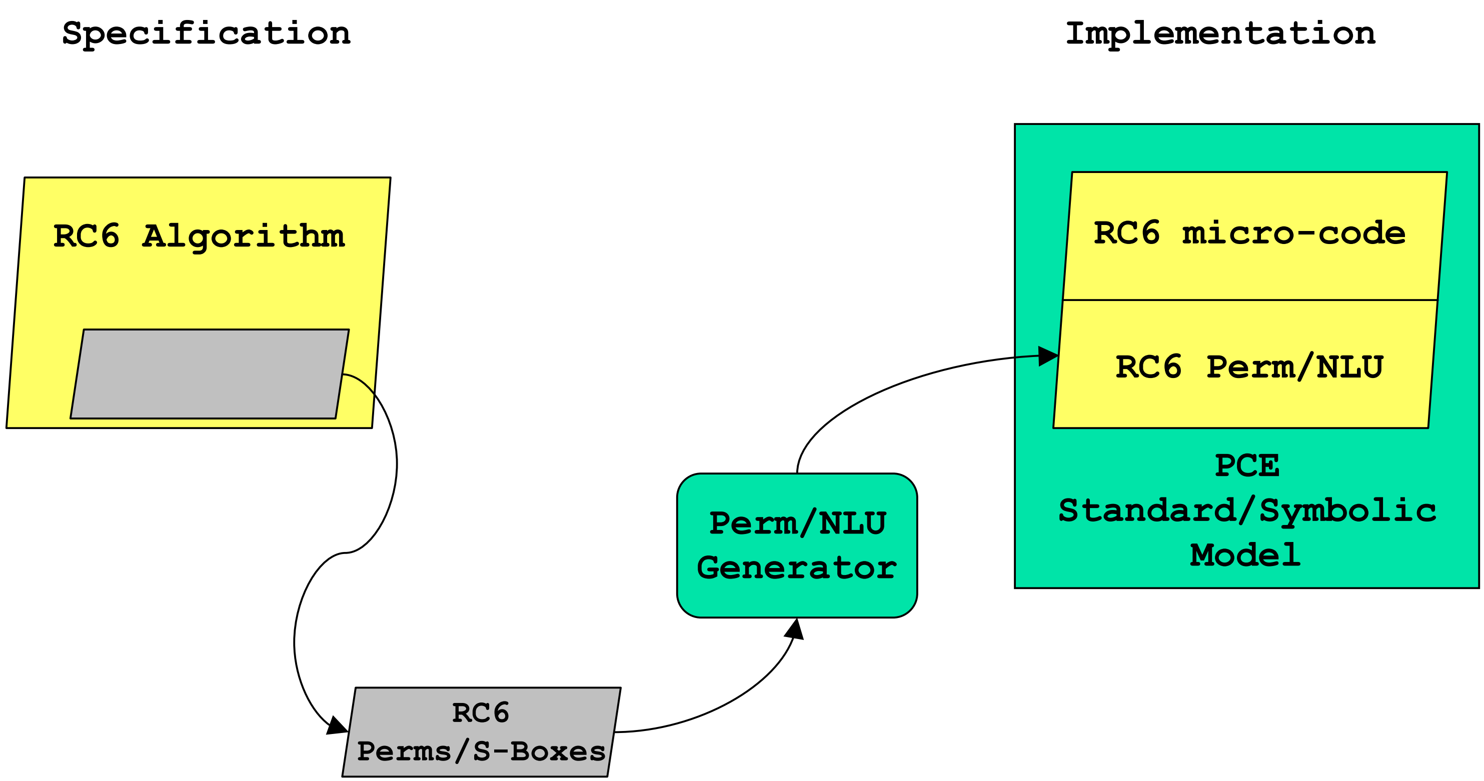
Domain Abstractions (DSL)



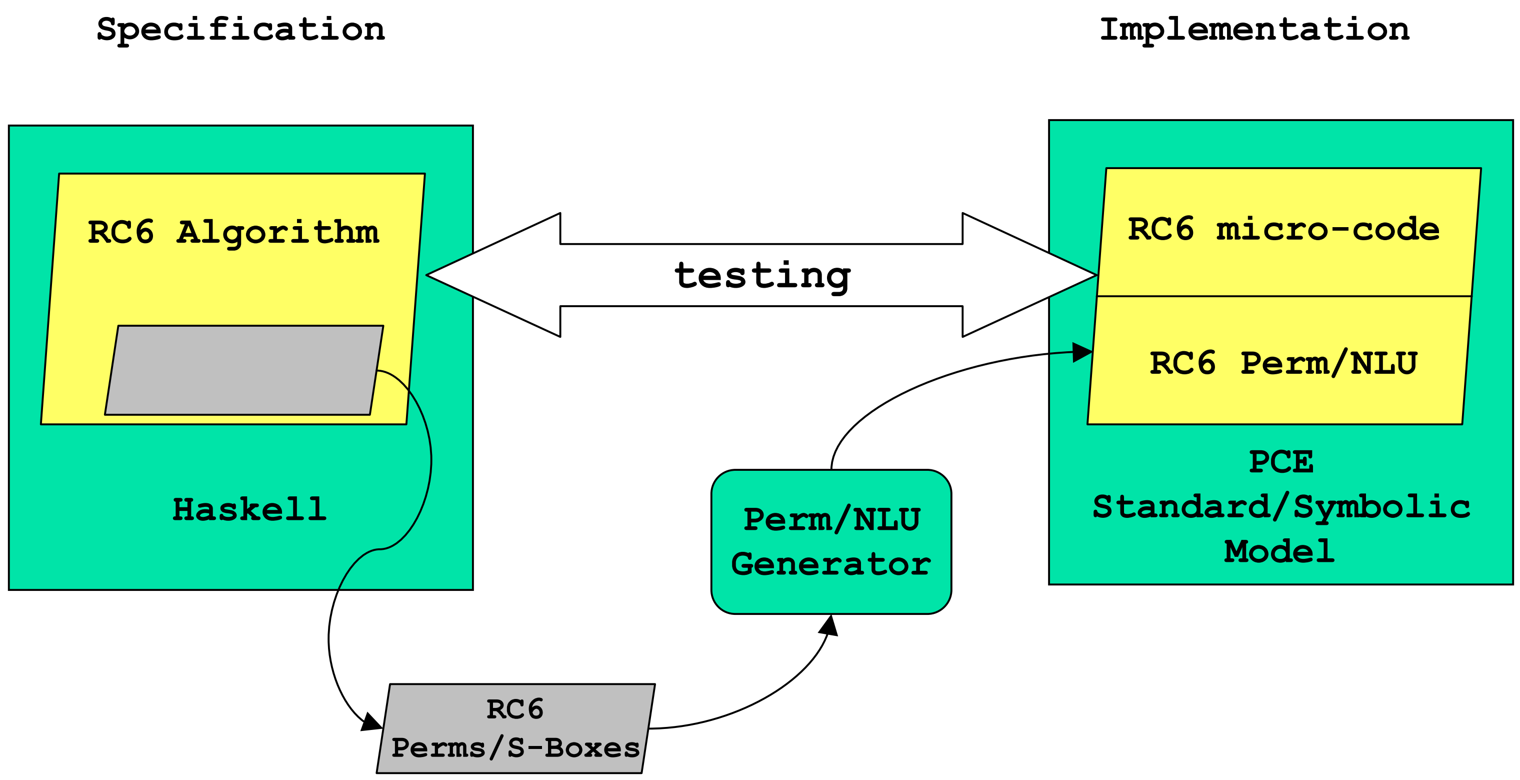
Configuration Generators



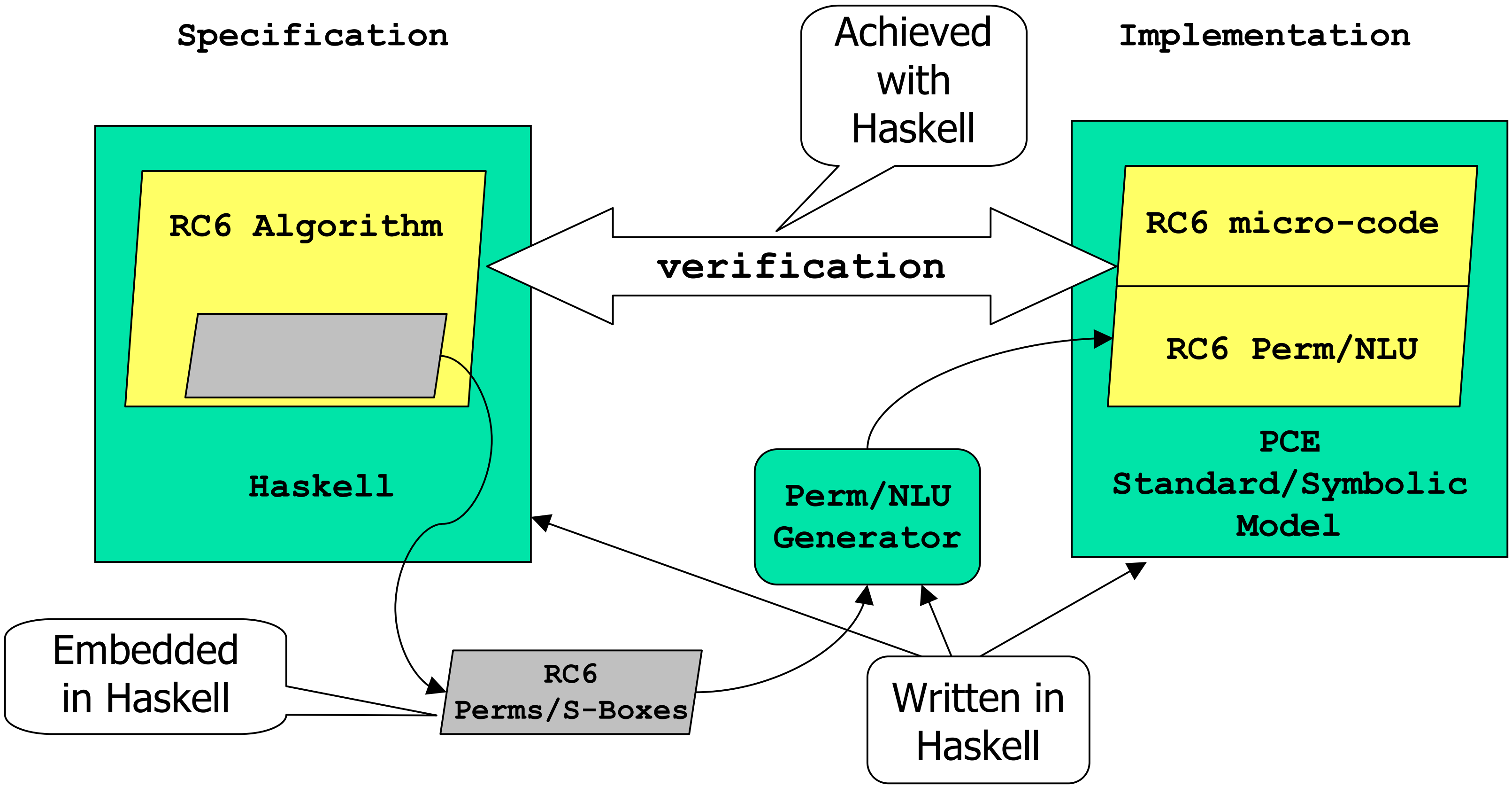
Machine Models (Std, Symbolic)



Executable Specifications



Haskell is the infrastructure



Accomplishments

- Designed DSL for Bit-Functions/Finite-Shift-Registers
  - Clean extension of previous DSL for Permutations/S-boxes
  - Formal semantics
  - Algebra
- Wrote HW models for PCE and CCE
- Developed "parameterized" model for PCE
- Developed specifications and implementations
  - RC6 (needs multiplication), Rinjdael, TEA
- Integrated BDD package into Haskell
- Verified 3 micro-code implementations of squaring

Lessons

- A single language greatly simplified our job

  Using Haskell to

  - Embed DSL
  - Model
  - Specify

  enables us to

  - Verify in Haskell
- Investment in DSL design was worthwhile
  - Can amortize over many ciphers
  - Makes specifications shorter and clearer
  - Can generate correct configurations
    - Automatically for PCE, semi-automatically for CCE.
- Haskell's overloading (type classes) greatly facilitated
  - Embedding DSL into Haskell
  - Model "parameterization"