| galois |

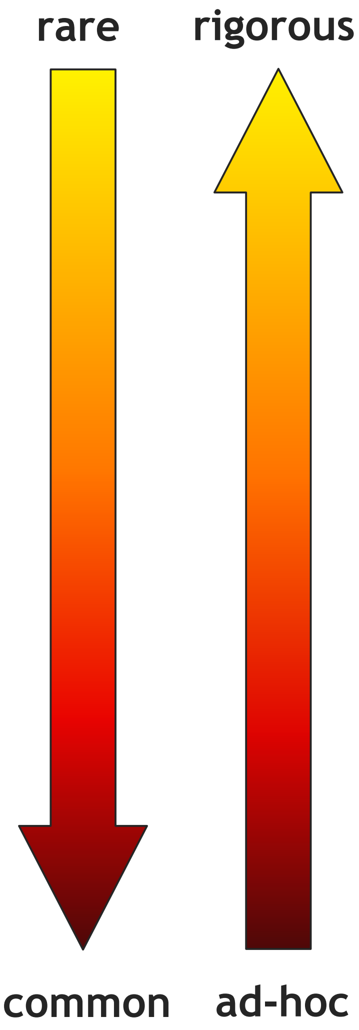
Cryptol Verification Technology

1 Mar 2005

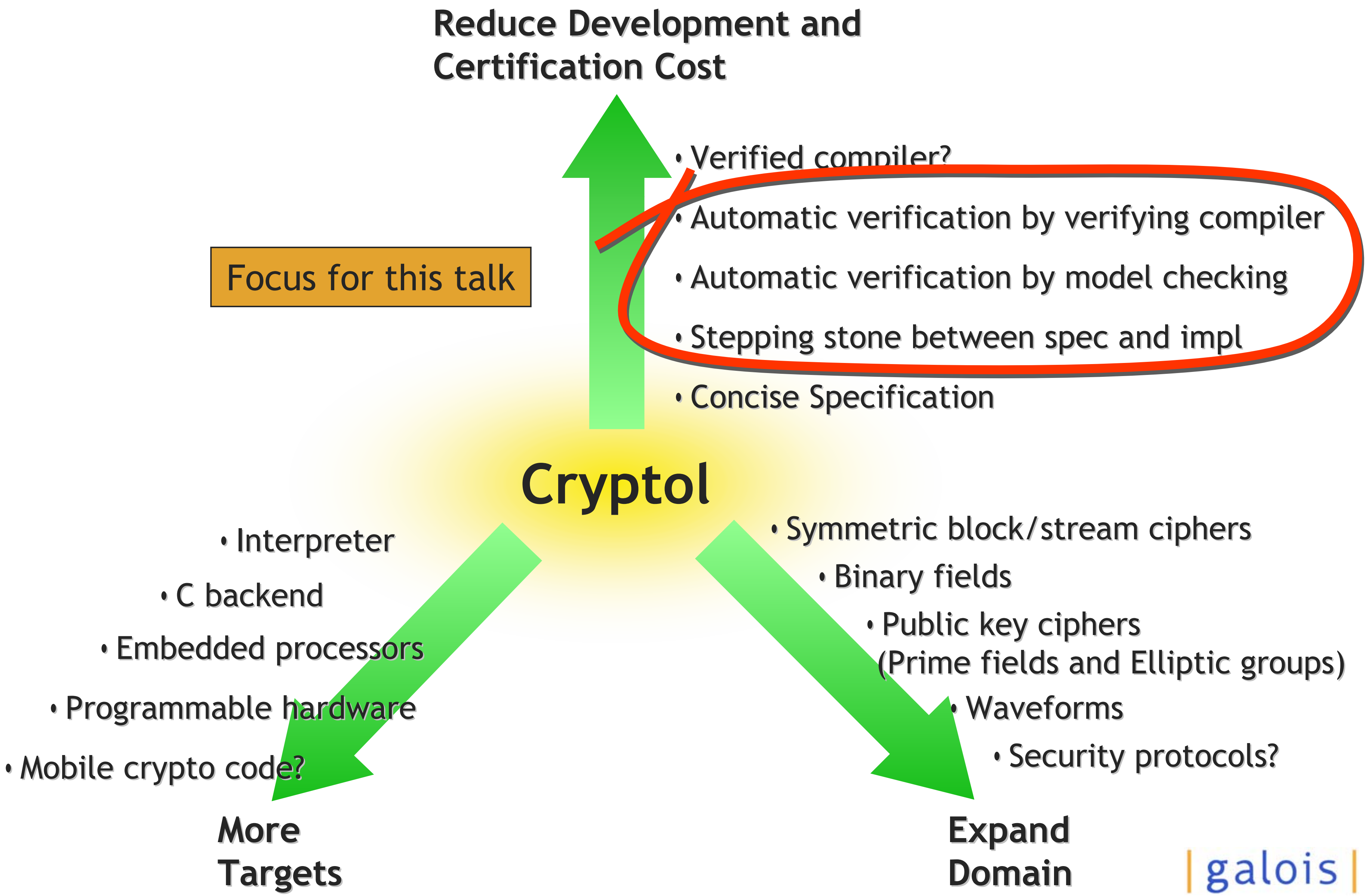
Mark Shields
Galois Connections
mbs@galois.com

Threats to cryptographic systems

- Failure in algorithm design
  - Eg: SHA-1 not cryptographically secure
- Failure in algorithm implementation
  - Examples in commercial sector?
- Failure in algorithm use
  - Eg: Microsoft's use of RC4 in Office Documents
- Side-channel attacks
  - Eg: SPA, DPA, timing, error messages, glitch
- Failure in surrounding glue and protocol
  - Eg: ASN.1 parsing, buffer overflow, non-zeroed keys/plaintext
- Failure due to outdated or no crypto at all
  - Cost of device devel. and certification very high
  - Very long delay from specification to deployment



Cryptol directions



Reduce Development and Certification Cost

- Verified compiler?
- Automatic verification by verifying compiler
- Automatic verification by model checking
- Stepping stone between spec and impl
- Concise Specification

Focus for this talk

Cryptol

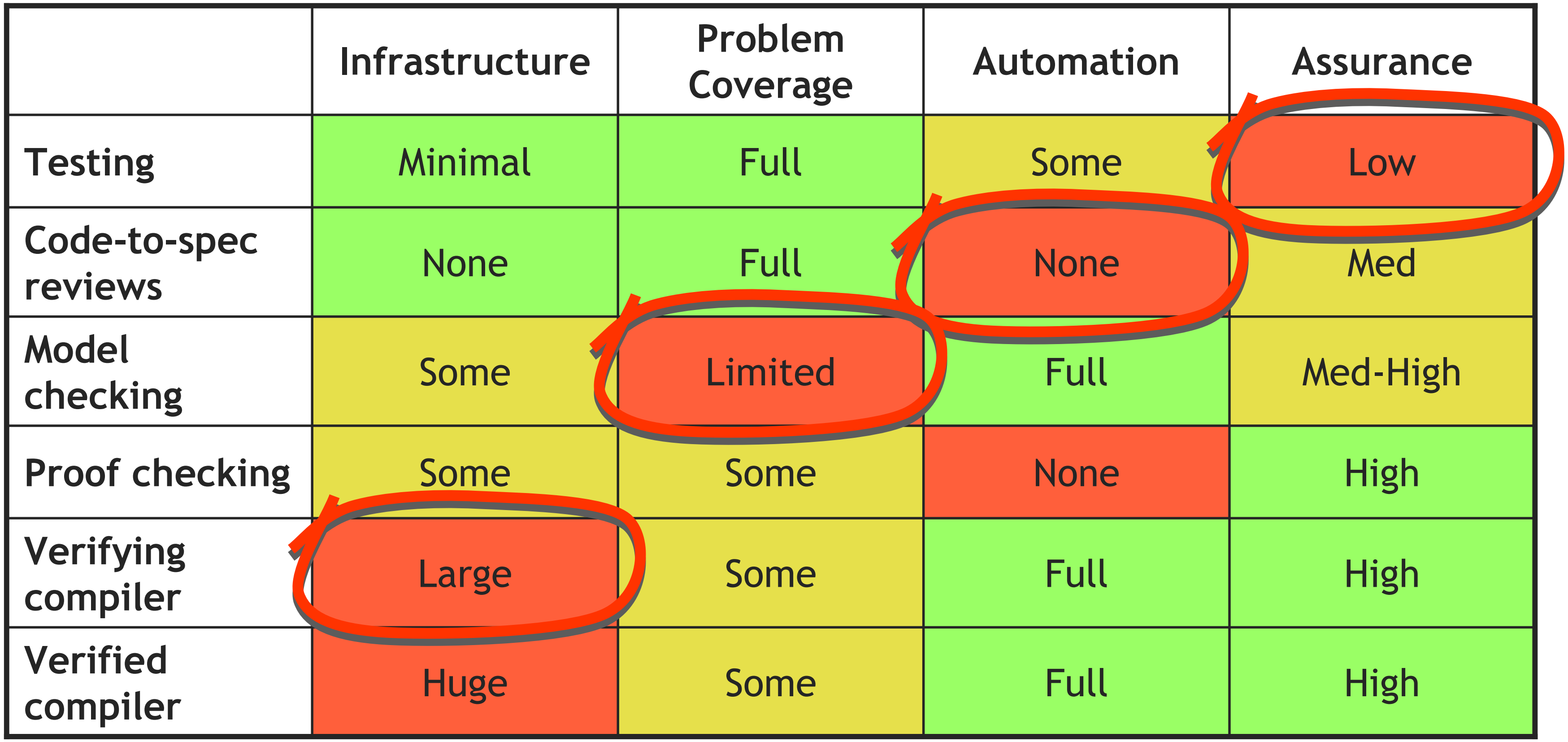
- Interpreter
- C backend
- Embedded processors
- Programmable hardware
- Mobile crypto code?

More Targets

- Symmetric block/stream ciphers
- Binary fields
- Public key ciphers (Prime fields and Elliptic groups)
- Waveforms
- Security protocols?

Expand Domain

Verification spectrum



	Infrastructure	Problem Coverage	Automation	Assurance
Testing	Minimal	Full	Some	Low
Code-to-spec reviews	None	Full	None	Med
Model checking	Some	Limited	Full	Med-High
Proof checking	Some	Some	None	High
Verifying compiler	Large	Some	Full	High
Verified compiler	Huge	Some	Full	High

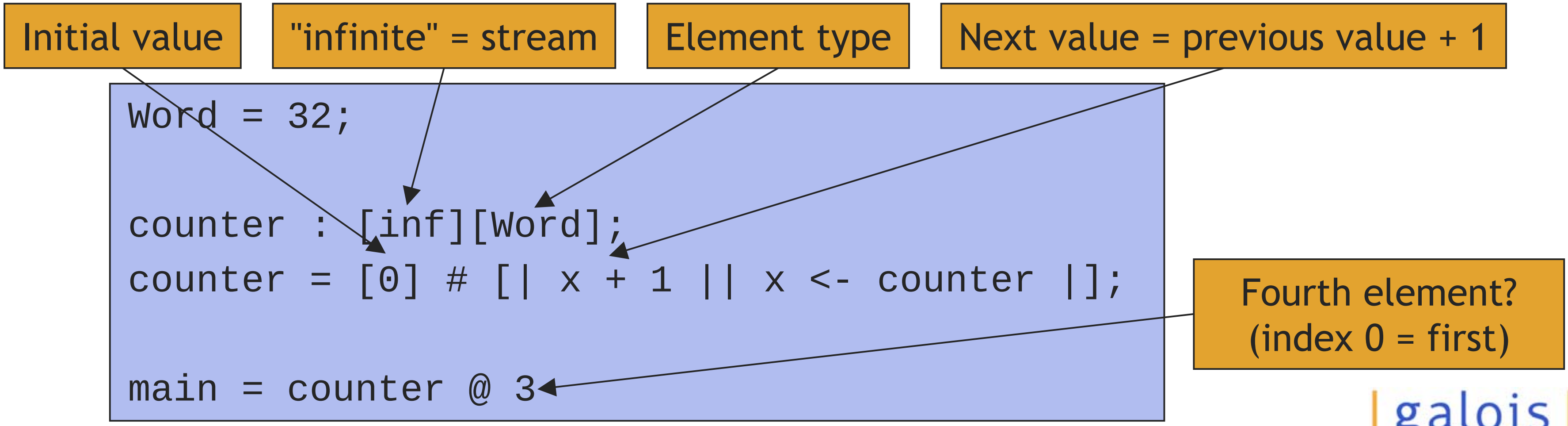
Topic 1: Increasing the precision of testing and ease of code-to-spec using Cryptol

Topic 2: Improving coverage for SAT-based verification of C/Cryptol against Cryptol

Topic 3: Improved approach to assertional verification of programs

A flavor of Cryptol

- Basics: numbers, vectors, tuples, rich set of primitives
- Key ingredient: recurrence relations
  - Block ciphers must "mix" key and block bits
  - Typically this requires repeated applications of substitutions and other transformations
- "Repeated" in hardware ⇒ latches and feedback
- "Repeated" in C ⇒ arrays and loops
- "Repeated" in Cryptol ⇒ streams



Initial value

"infinite" = stream

Element type

Next value = previous value + 1

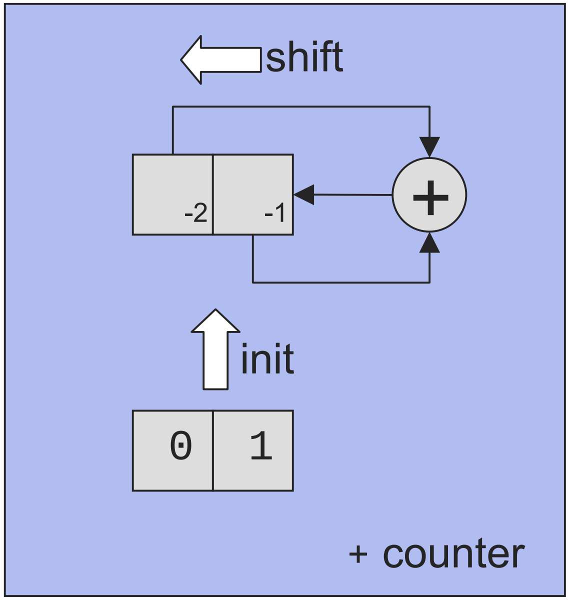
```
Word = 32;


counter : [inf][Word];
counter = [0] # [| x + 1 || x <- counter |];


main = counter @ 3
```

Fourth element?
(index 0 = first)

Eg: Fibonacci numbers



```
word fib(int n) {
  word h[2];
  h[0] = 1; h[1] = 1;
  for (i = 2; i <= n; i++)
    h[i % 2] = h[(i - 1)%2]
             + h[(i - 2)%2];
  return h[n % 2];
}
```

```
fib : Word -> Word;
fib n = fibs @ n where {
  fibs : [inf][Word];
  fibs = [1 1] # [| x + y || x <- drop(1, fibs)
                          || y <- fibs |];
};
```

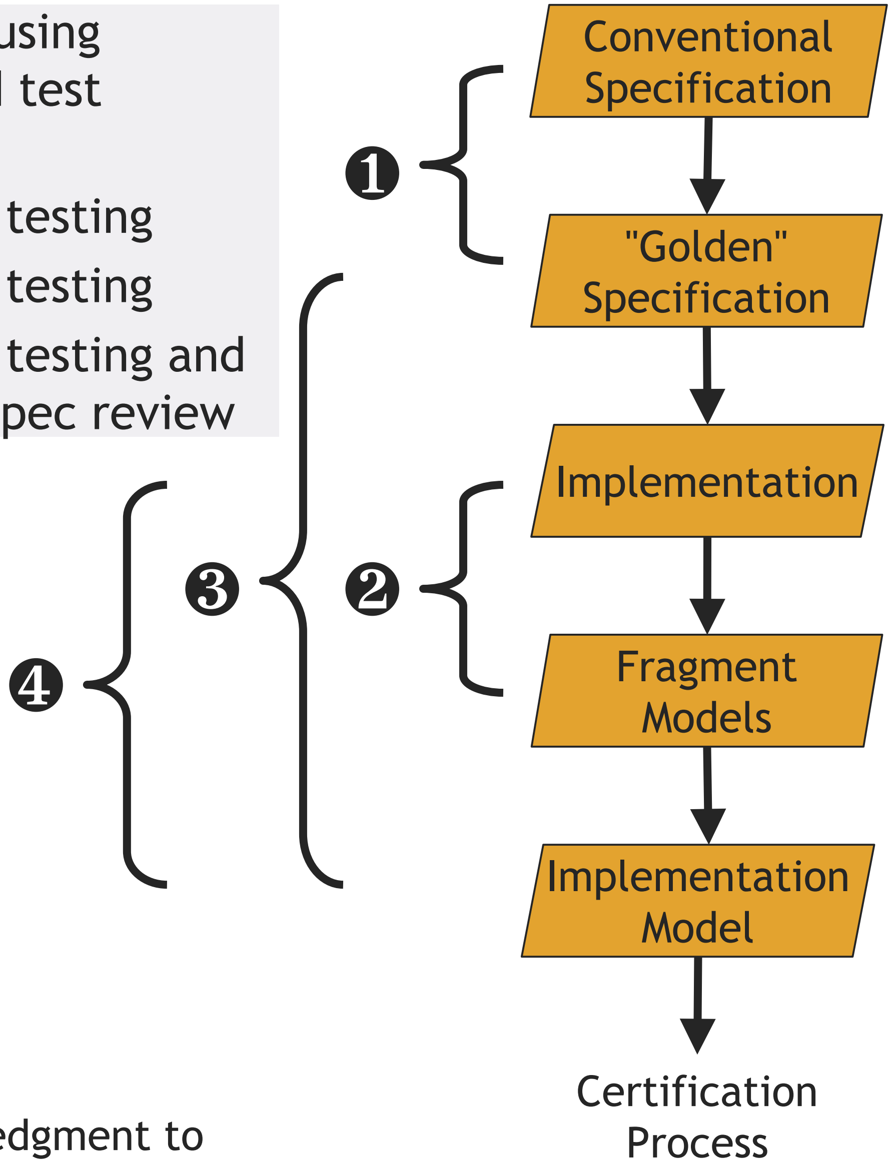
Cryptol as an aid to implementation and certification

From specification to implementation

- On conventional μP, conceptual gap from specification to implementation is small enough to be bridged "on-the-fly" by the programmer
- On specialized hardware, gap can be very much wider
  - Parallelization on VLIW architectures
  - Deep pipelining
  - Monolithic operators with many configuration parameters
- Cryptol can be a stepping-stone between specification and implementation
  - Can use the Cryptol interpreter to produce test vectors
  - Can embed Cryptol program fragments within comments to capture intended semantics of complex instructions

Cryptol in the development process

- ❶ Validate using published test vectors
- ❷ Verify by testing
- ❸ Verify by testing
- ❹ Verify by testing and code-to-spec review



❶ ❷ ❸ ❹

Conventional Specification

(Pseudo-code/Mathematics)
Eg: NIST/NSA spec, paper

"Golden" Specification

(Cryptol)
Test understanding of specification

Implementation

(Eg: Microcode)

Fragment Models

(Cryptol)
Capture intended semantics of code fragments

Implementation Model

(Cryptol)
Capture structure of implementation

Certification Process

With acknowledgment to
Alan Newman of General Dynamics

Status

- General Dynamics has multiple crypto devices under certification for which Cryptol programs form part of the supporting documentation
- We could go much further:
  - Support Cryptol assertions within microcode/assembly/C programs
  - Support automatic test case generation based on Cryptol fragments
  - Support reasoning about equivalence of Cryptol programs
  - Support reasoning about equivalence of implementation and Cryptol programs

Equivalence verification by SAT-solving

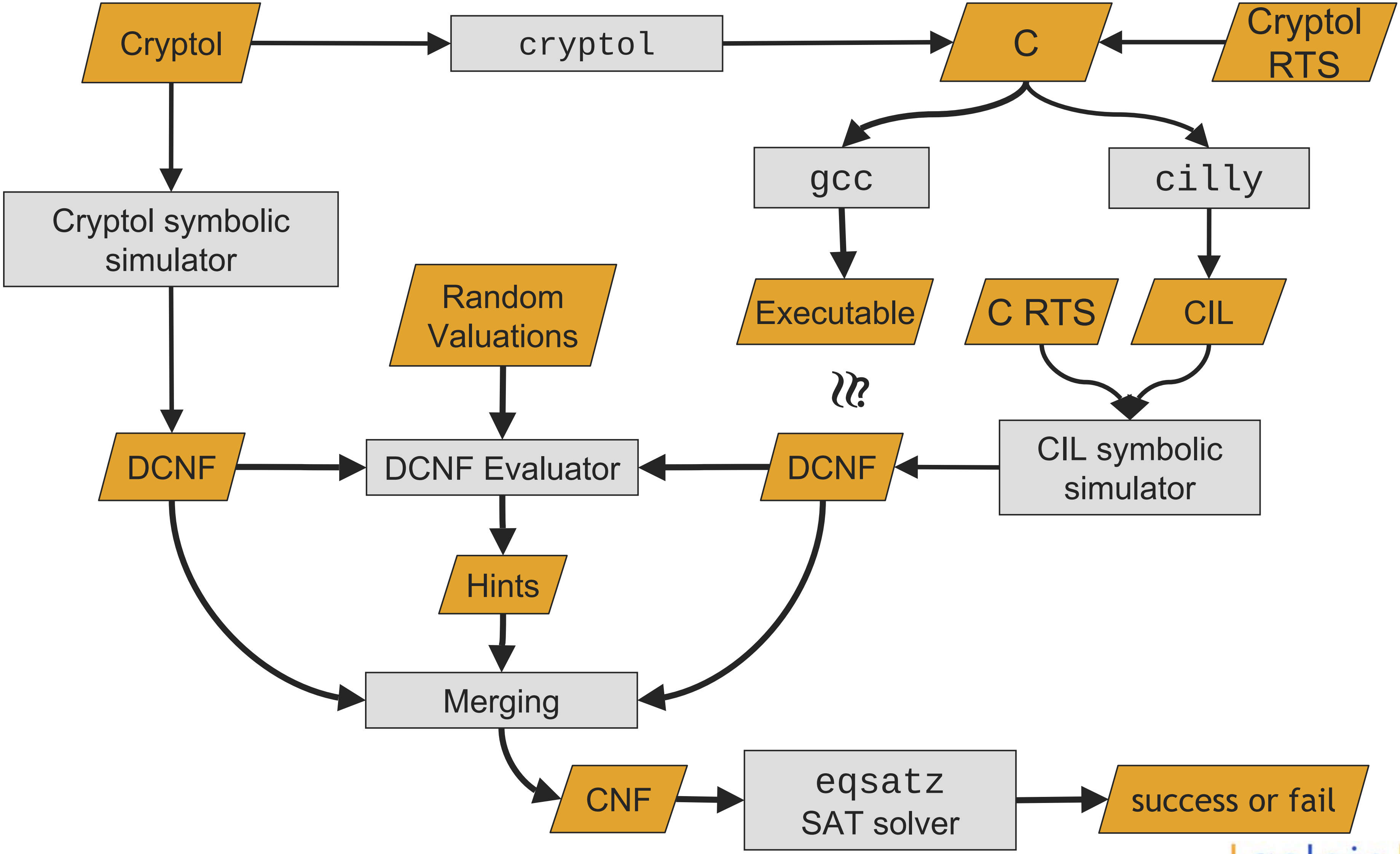
Symbolic simulation

- A symbolic simulator computes each output bit of a program as boolean expressions in terms of symbolic variables representing each input bit

```
main = encrypt (var "key", var "pt")
```

- Cryptol symbolic simulator is easy to implement
- C symbolic simulator not so easy!
  - Luckily `cilly` (developed by George Necula et al) can translate C to CIL, an intermediate language simpler than C
  - We may then compile CIL to a simple stack machine
  - We then model every bit of the stack and heap symbolically
  - Each machine transition induces a relation between states
  - Machine will print its output as a series of bits
- A boolean expression may be represented as a directed acyclic graph of AND nodes (with possibly inverted inputs)

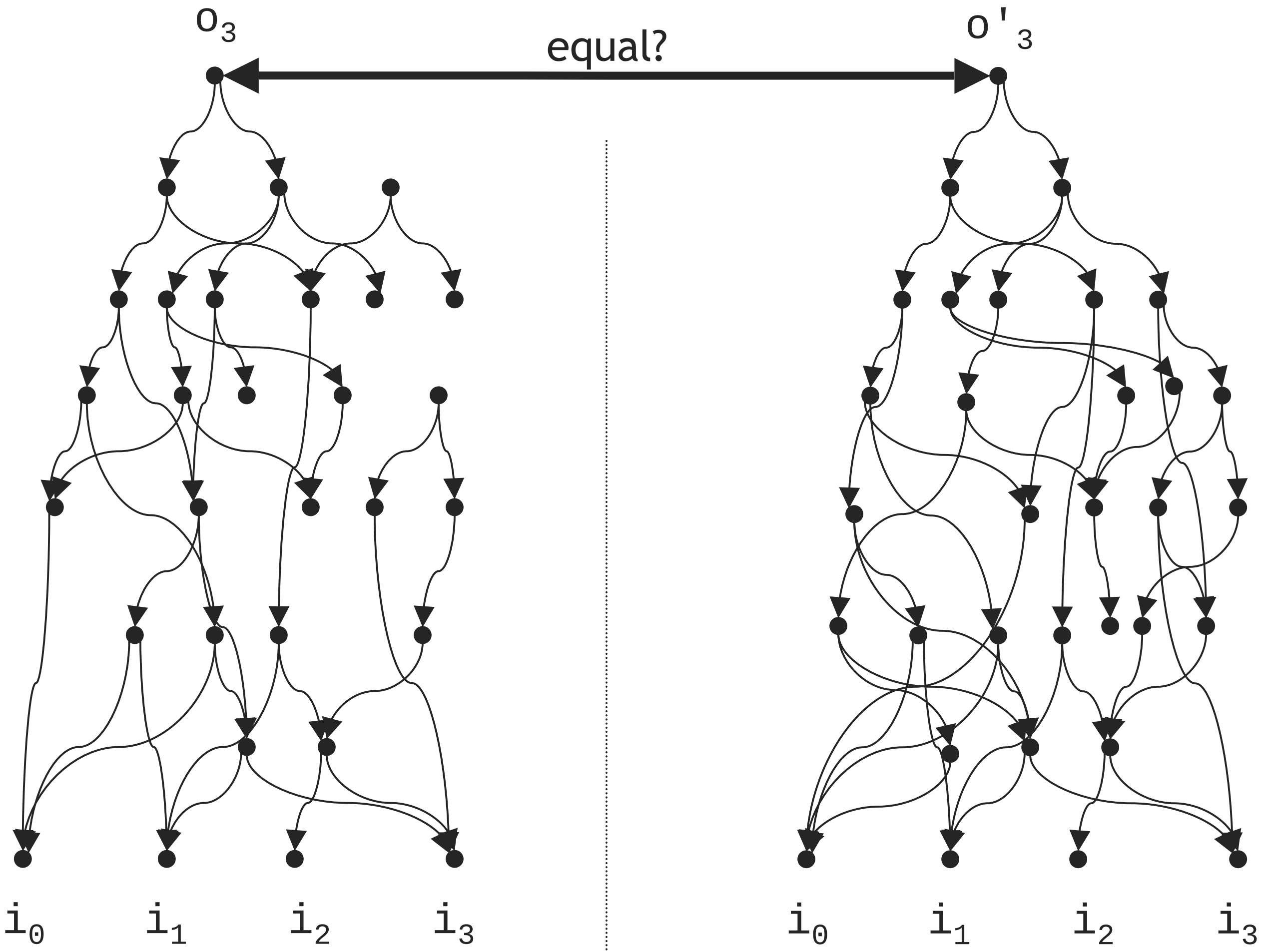
Verification approach



Equivalence checking

- For each output bit, we now have two DCNF graphs in terms of a common set of symbolic variables
- We now need to show
  - For every valuation of symbolic variables, output values of two DCNF graphs are equal
- `eqsatz` (by Chu Min Li) is SAT solver using the Davis-Putnam procedure with built-in support for equality
  - Given a CNF, it answers whether the formula is a tautology
- Now we must encode the above problem as one CNF

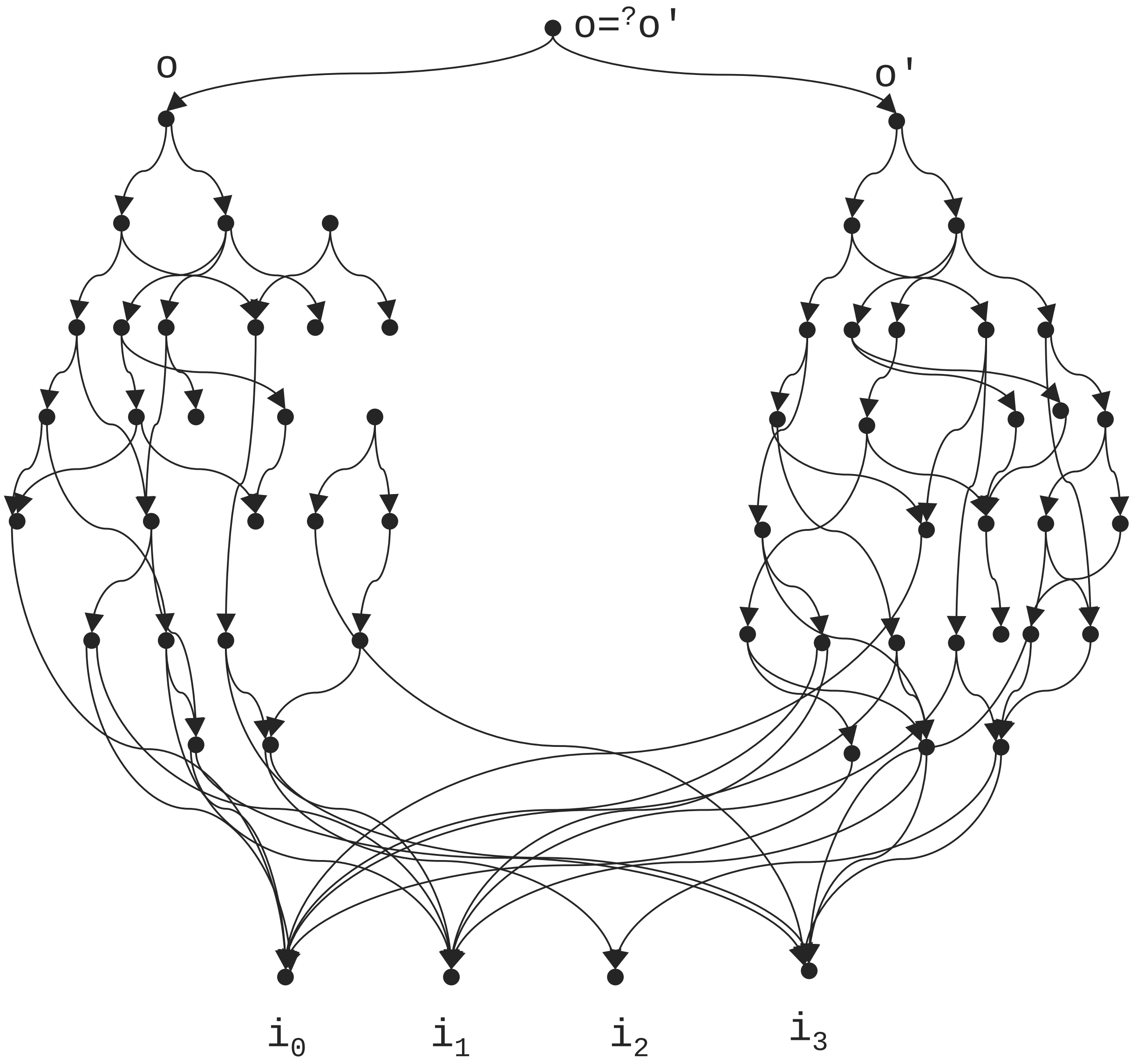
Equivalence checking problem



DCNF directly from Cryptol

DCNF from mini-C from C from Cryptol

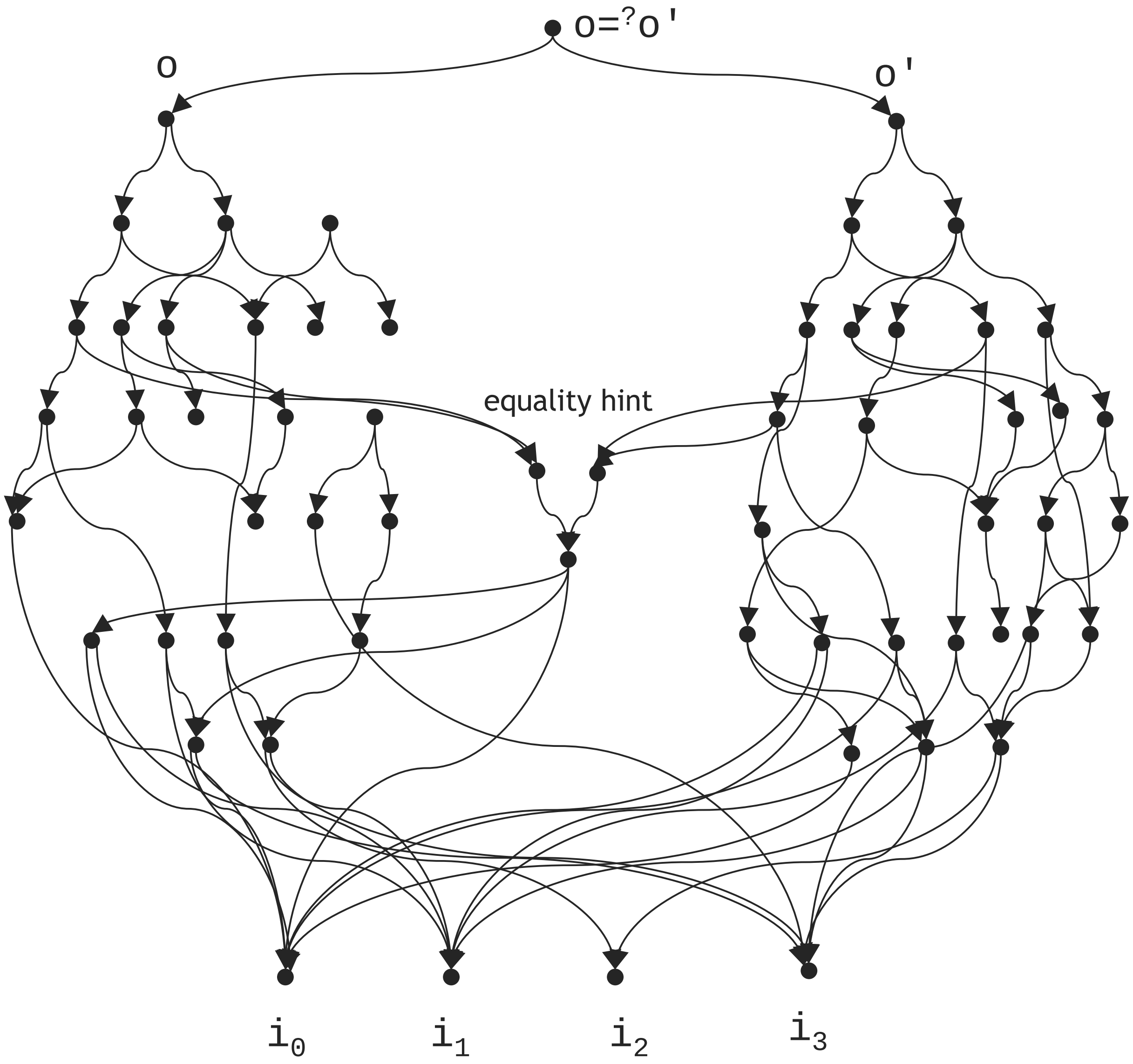
Equivalence checking after merging



Equivalence checking with hints

- However, even small cryptographic algorithms are too complicated to be directly verified this way by `eqsatz`
  - We need to merge more aggressively
- Remarkably, simply "hash-consing" during the "bottom-up" construction of the merged CNF does quite a good job
- We could also give the SAT solver "hints" as to which interior CNF nodes are *probably* equivalent
  - If hint is unsound, equivalence will fail
  - If hint is sound, equivalence holds even without hint
- One approach: use concrete simulation on random inputs to eliminate nodes which are definitely not equal
  - Run multiple times to eliminate more nodes
  - Remainder are likely to be equal for all inputs
  - Effective because cryptographic algorithms are very good at dispersing input bits to interior nodes!

Equivalence checking with hints



Status

- Currently verified 32-round TEA in around 3.5 minutes with hash-consed merging, but without hints

Equivalence verification by theorem proving

Context

- The SHADE project (joint work with Rockwell Collins) is building a verifying compiler from µCryptol to the AAMP7 microprocessor
  - µCryptol is a variation of the Cryptol language intended to support embedded applications
  - The Rockwell Collins AAMP7 is an embedded µP supporting very high-assurance process partitioning
- "Verifying" means that, for a given µCryptol program, the complier emits:
  - An AAMP7 binary image
  - A proof script which demonstrates behavioral equivalence of the µCryptol program with the final AAMP7 program

How to verify equivalent behavior?

- We must know the intended meaning of every μCryptol program:
  - Galois have developed the semantics of μCryptol, written in conventionally accepted mathematical notation
  - The semantics will be validated against a conventional interpretation of μCryptol:
    - Semantics of each feature inspected to see if it corresponds with expectations
      - Eg: `reverse (reverse [0,1,2]) == [0,1,2]`
    - Common cryptographic algorithms will be implemented in μCryptol, and tested against published test vectors
      - Using the semantics, not the compiler!

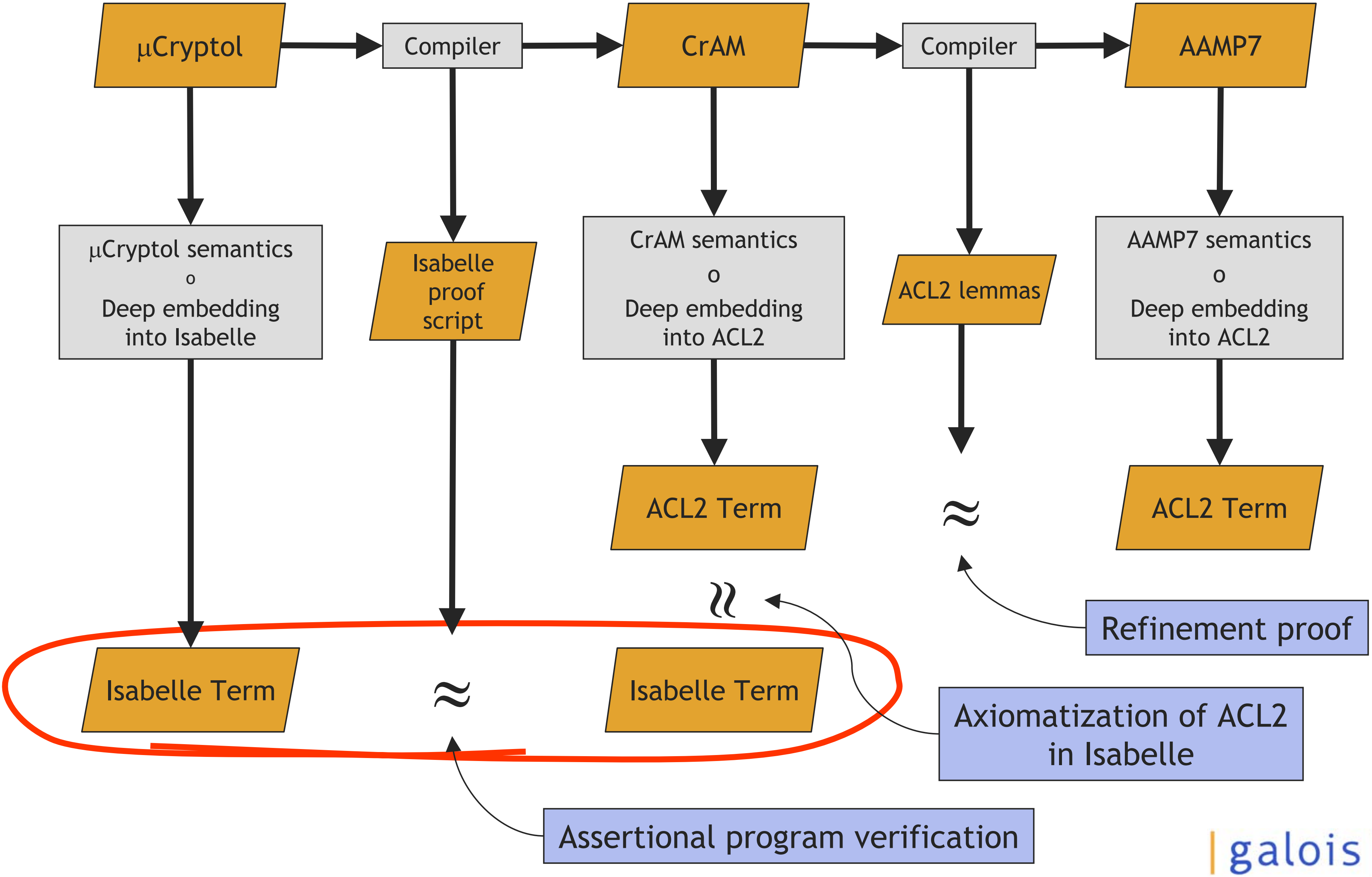
How to verify equivalent behavior?

- We must know the intended meaning of every AAMP7 program:
  - Rockwell Collins have developed a simulator for AAMP7 binaries
  - The simulator will be validated against the actual AAMP7 hardware
    - By inspection of each opcode transition
    - By test vectors run in parallel on simulator and hardware
- We must decide what behavior we are interested in:
  - Input/output correspondence
  - Termination

Verification approach

- The μCryptol compiler uses a stack-machine based abstract machine ("CrAM") language as an intermediate form
- We exploit this to break the verification problem into two halves:
  - Using Isabelle/HOL: Verify CrAM program implements μCryptol program using assertional reasoning
  - Using ACL2: Verify AAMP7 programs implements CrAM program using state-machine refinement

Verification approach



μCryptol

Compiler

CrAM

Compiler

AAMP7

μCryptol semantics
o
Deep embedding
into Isabelle

Isabelle
proof
script

CrAM semantics
o
Deep embedding
into ACL2

ACL2 lemmas

AAMP7 semantics
o
Deep embedding
into ACL2

ACL2 Term

≈

ACL2 Term

≀

Refinement proof

Isabelle Term

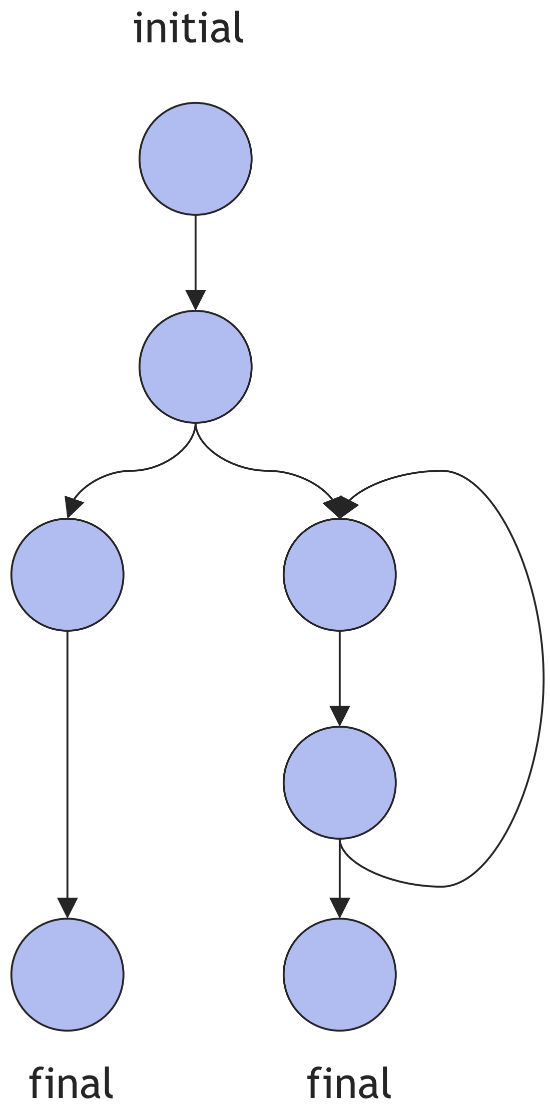
≈

Isabelle Term

Axiomatization of ACL2
in Isabelle

Assertional program verification

CrAM verification problem



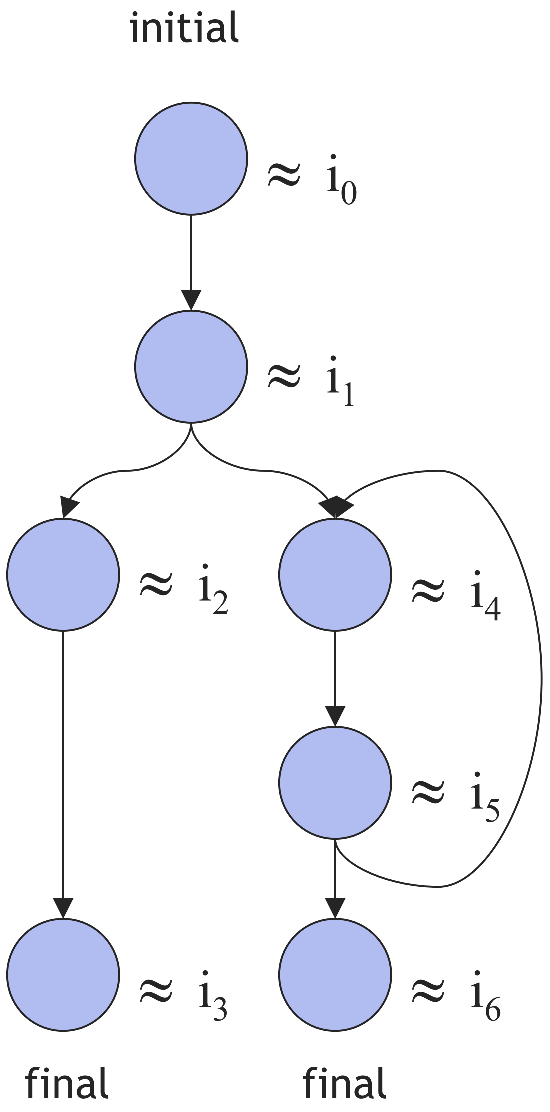
- We wish to verify that

  if the initial CrAM state corresponds to symbolic inputs of μCryptol program

  then each final CrAM state corresponds to expected output of μCryptol program

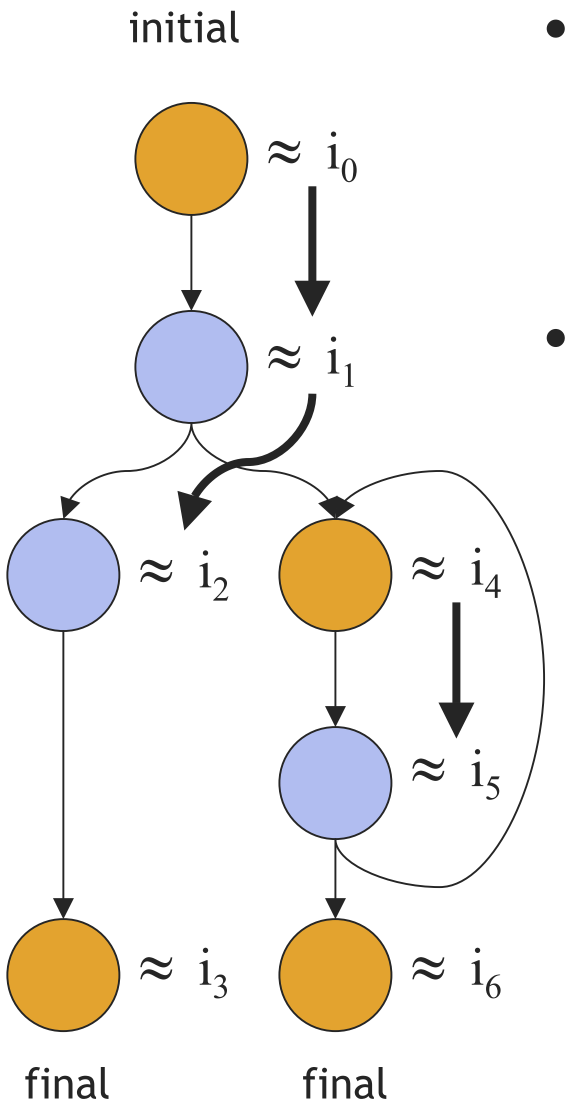
  and every execution trace reaches a final state

State invariants



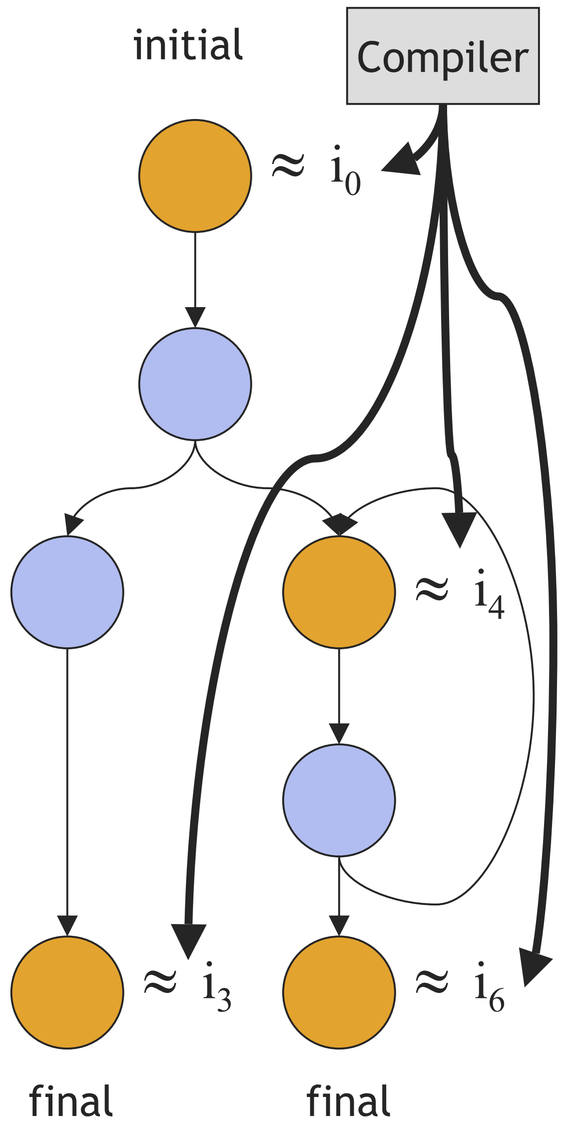
- We tie states to inputs and expected outputs by adding invariants
  - Invariant on initial state ties operand stack to symbolic values for program inputs
  - Invariant on final states tie operand stack to (the meaning of) μCryptol expression describing output in terms of symbolic inputs
- What about all the interior states?
  - At first blush, need to find invariant for every state, perhaps using a verification condition generator
- Luckily, J Moore presented a beautiful short-cut at HCSS 2004

State invariants: Insight 1



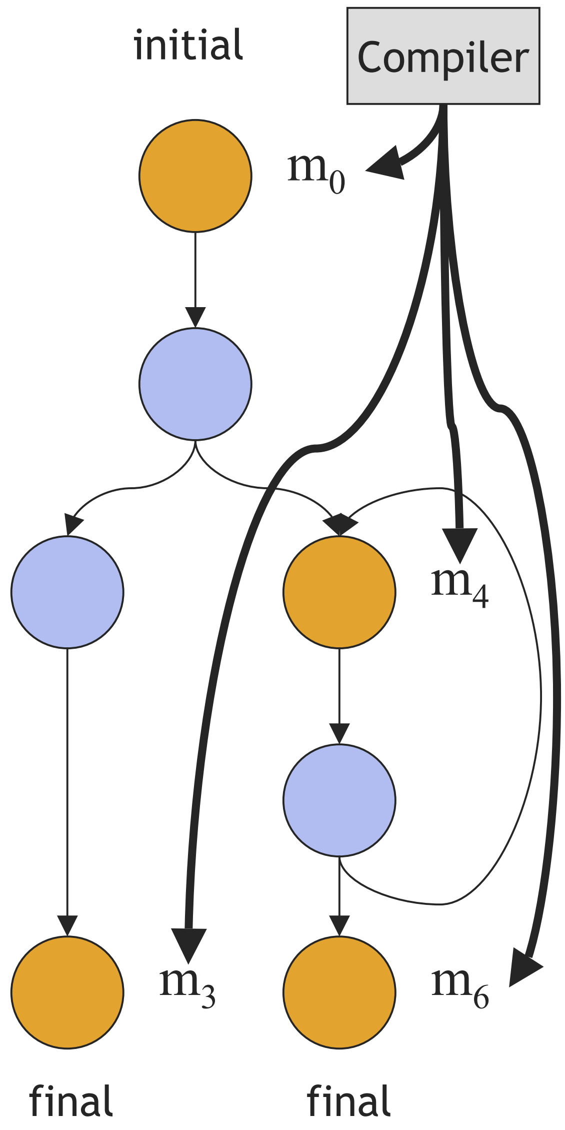
- **We only need invariants on cutpoint states**
  - Ie those which are either initial, final, or break a loop
- Once we have a small-step semantics for the machine, we may use it to propagate invariants from cutpoint states to all other states

State invariants: Insight 2



- **The μCryptol compiler already knows these invariants**
  - Frame and non-interference axioms
  - Input/output correspondence with source term
  - Stack, locals and heap locations of all relevant source variables
  - Purpose and indexes for all loops
- Remember: we are not demonstrating correctness w.r.t. an absolute property, but equivalence with an existing program
- Hence we do not have to deal with inferring or supplying complicated loop invariants

State invariants: Insight 3



- To show termination, we associate a well-founded measure value to each state, and show
  - Each state transition strictly decreases the measure
- **Compiler also knows these measures**
  - They may be derived from the control structure of the μCryptol source program

Status

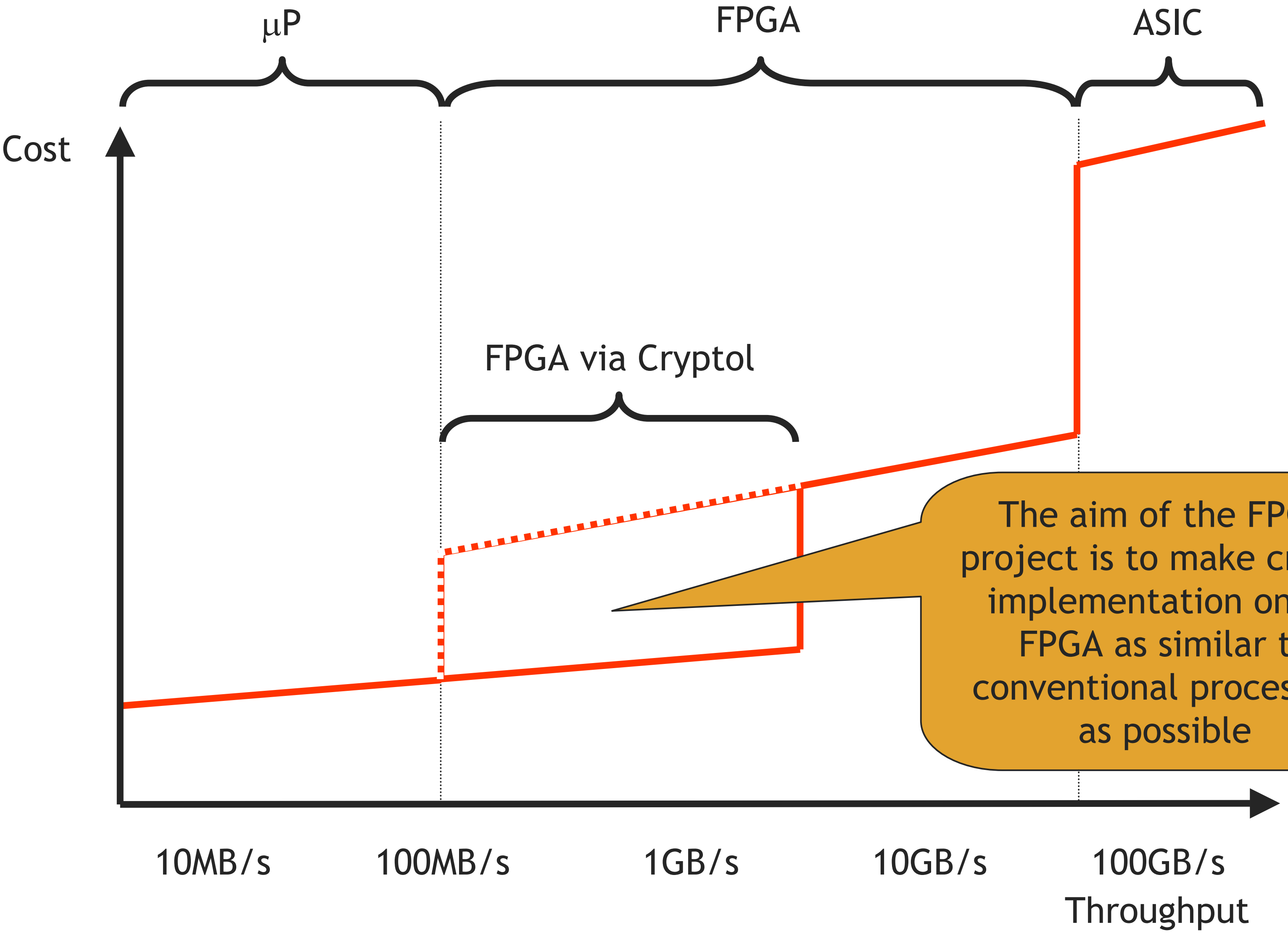
- See:
  - *A Symbolic Simulation Approach to Assertional Program Verification*
    John Matthews, J S. Moore, Sandip Ray and Daron Vroon.
    (Submitted for publication)
  - *Partial Clock Functions in ACL2*
    John Matthews and Daron Vroon.
    Appeared in the Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2004), Austin, Texas, Nov 2004.

- Compiler currently generating AAMP7 binaries, which may be executed on both real hardware and ACL2 model
- Currently developing μCryptol semantics in Isabelle

Other ongoing work

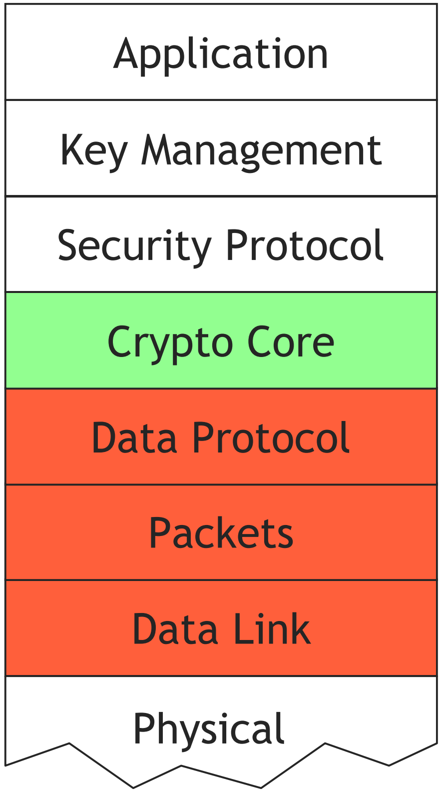
- Cryptol Embedded
  - Refined language and type system to support static memory allocation for embedded devices
- Cryptol to FPGA
  - Compile Cryptol directly to VHDL, which may be realized on an FPGA using existing toolchain
- Public Key Algorithms
  - Additional primitives to support prime field and elliptic curve arithmetic with run-time field/group parameters
- Waveforms
  - Extend Cryptol's applicability to describing the "waveform" or "glue" code which surrounds cryptographic algorithms in actual devices

Cryptol FPGA: Cost vs Throughput



The aim of the FPGA project is to make crypto implementation on an FPGA as similar to conventional processors as possible

Typical cryptographic device layering



- The entire device must be certified
- The actual cryptographic core is a small fraction of overall code
- A great deal of tedious and error-prone engineering must go into the lower level "waveform" layers:
    - padding and packet boundaries
    - cryptographic modes, initialization, keying
    - error detection and correction
    - packet parsing and encoding
    - packet protocol: start, data, end, ack, timeout, resend
    - parsing and encoding highly structured data (eg certificate in ASN.1)

Tackling the waveform problem

- Much lower-layer code is bit-twiddling
  - With use of error-correction primitives
- Bit-twiddling is Cryptol's bread and butter
- Possible approach
  - Allow packet layout to be declared as a new Cryptol type
  - Allow packet protocols to be declared
  - Allow packet recognition to be declared
  - Compile all of above down to vanilla Cryptol
- Generated code may be subject to verification by same methods we have already discussed

Cryptol team and partners

- Core
  - Jeff Lewis, Sigbjorn Finne
- Cryptol development methodology
  - General Dynamics
- FPGA
  - Andy Gill, Fergus Henderson
  - Xilinx
- SHADE
  - John Matthews, Mark Shields
  - Rockwell Collins
- SAT Verifier
  - Thomas Nordin
- Public Key
  - Thomas Nordin, Frank Taylor

Questions?

Additional Material

A flavor of Cryptol

Cryptol values and operators

- Values:
  - Bits: `True, False` : `Bit`
  - Vectors of bits: `[True False True], 5` : `[3]`
  - Tuples of any type: `(3 True [True])` : `([2], Bit, [1])`
  - Vectors of any type: `[(3, 2) (2, 1)]` : `(B^2, B^2)^2`
- Built in operators:
  - Modular arithmetic: `(3:[3]) +7` `== 2`
  - Comparison: `7 < 8` `== True`
  - Logical: `7 < 8 && (3:[3]) == 1+2` `== True`
  - Bitwise logical: `6 || 1` `== 7`
  - Shift and rotate: `[7 9 11] <<< 2` `== [11 7 9]`
  - Indexing: `[7 9 11]@0` `== 7`
  - Polynomials: `pmult 3 4` `== 12`

Cryptol values and operators

- More advanced operations on vectors:
  - Append: `[1 2] # [3 4] == [1 2 3 4]`
  - Reverse: `reverse [(1, 2) (3, 4)] == [(3,4) (1,2)]`
  - Join: `join [[1 2] [3 4]] == [1 2 3 4]`
  - Split: `split [1 2 3 4 5 6] : [2][3][8]`
`== [[1 2 3] [4 5 6]]`
  - Drop: `drop [1 2 3 4] : [3][8] == [2 3 4]`
  - Take: `take [1 2 3 4] : [3][8] == [1 2 3]`
  - Transpose: `transpose [[1 2] [3 4]] == [[1 3] [2 4]]`
- Note that:
  - The type checker knows the width of every vector at compile time
    - Type checker performs arithmetic at compile time
  - All the vector operators work on vectors of anything
    - We say they are "polymorphic" on their element type and width

Cryptol constructs

- Enumerations (shorthand for sequences of numbers):

```
[3, 5 .. 11] == [3 5 7 9 11]
```

- Local definitions:

```
x + y where { x = 7; y = 8; }
```

- Functions:

```
f : [8] -> [8];
f x = g (x + 1) * 3
  where { g : [8] -> [8]; g y = y + x; }
```

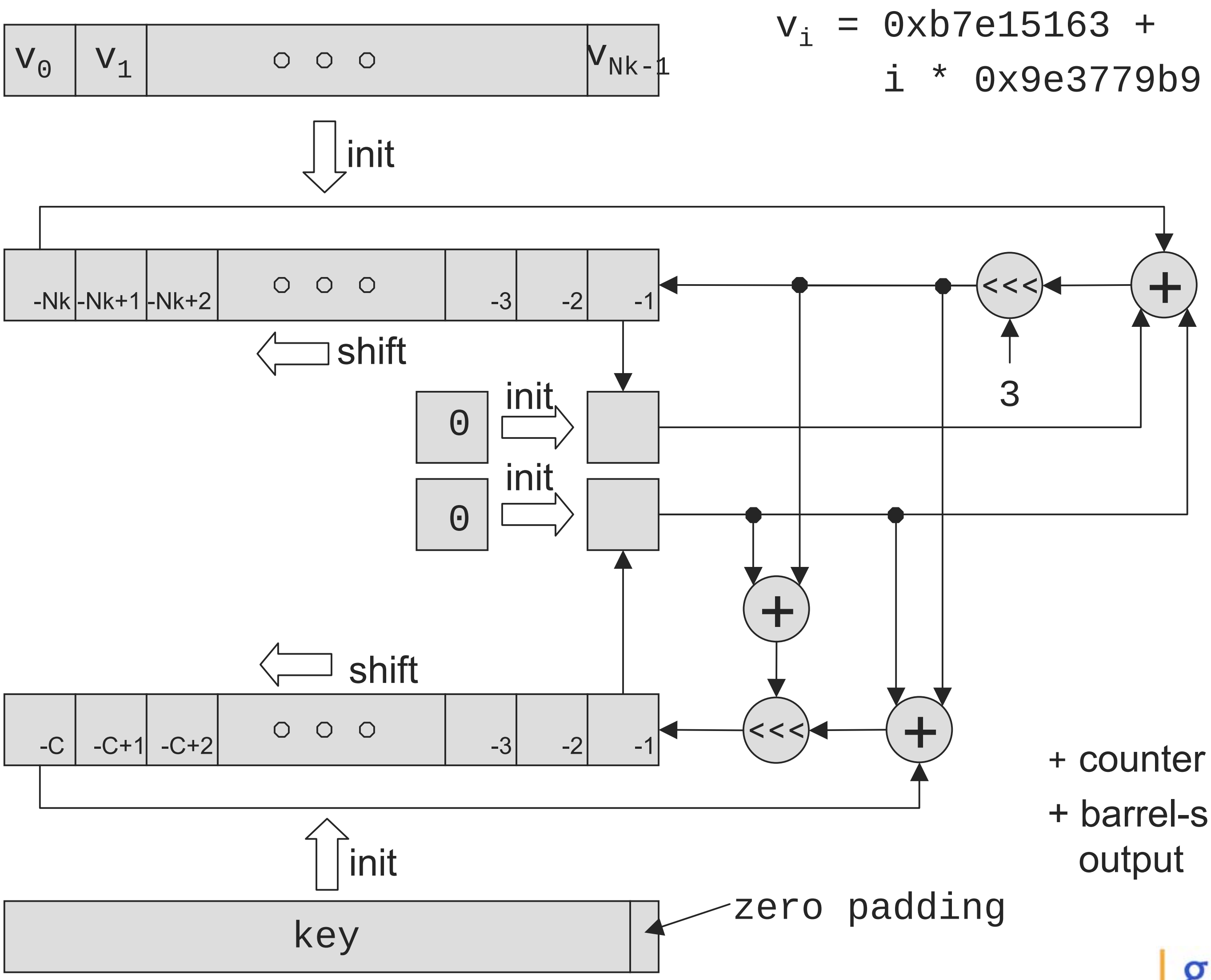
- Branching:

```
if x > 3 then x - 1 else x + 1
```

- Comprehensions ("calculate for each element of..."):

```
[| x + 1 || x <- [0{8}..3]]               == [1 2 3 4]
[| x + y || x <- [0 1], y <- [2 3]]       == [2 3 3 4]
[| x + y || x <- [0 1] || y <- [2 3]]     == [2 4]
```

Eg: RC6 Key Expansion - Hardware



+ counter
+ barrel-shift output

Eg: RC6 Key Expansion - C

```
#define A ...
#define Nk 44
#define C (max(1, (A + 3) / 4))
#define V (3 * max(C, Nk))

void rc6exp(byte key[A], byte s[Nk]) {
  word l[C]; int i, j, s; word a, b;
  l[C - 1] = 0; memcpy(l, key, A);
  l[0] = 0xb7e15163;
  for (i = 1; i < Nk; i++)
    s[i] = s[i - 1] + 0x9e3779b9;
  a = b = 0; i = j = 0;
  for (s = 0; s < V; s++) {
    a = s[i] = (s[i] + a + b) <<< 3;
    b = l[j] = (l[j] + a + b) <<< (a + b);
    i = (i + 1) % Nk;
    j = (j + 1) % C;
  }
}
```

Eg: RC6 Key Expansion - Cryptol

```
A = ...;
Nk = 44;
C = max(1, (A + 3) / 4);
V = 3 * max(C, Nk);

rc6exp : [A][Byte] -> [Nk][Word];
rc6exp key = segment(V-Nk, s) >>> (V - 3 * Nk)
  where {
    consts : [inf][Word];
    consts = [0xb7e15163] # [| x + 0x9e3779b9 || x <- consts |];
    inits : [Nk][Word];
    inits = segment(0, consts);
    initl : [C][Word];
    initl = split (join ((key # zero) : [4*C][Byte]));
    s : [inf][Word];
    s = [| (x+a+b) <<< 3
        || x <- inits # s || a <- [0] # s || b <- [0] # l |];
    l : [inf][Word];
    l = [| (x+a+b) <<< (a+b)
        || x <- initl # l || a <- s || b <- [0] # l |]; };
```

μCryptol

Cryptol as an implementation language

- Implementations have many concerns which may be conveniently ignored in a specification:
  - Efficient and bounded use of memory
  - Efficient use of available hardware primitives
  - Timing and power analysis attacks
  - Zeroing sensitive memory after use
- Many implementation details are device dependent
  - Eg: Software only vs custom hardware targets
- So is it realistic to push these issues up into the language?
- Our strategy:

Support as many implementation refinements within Cryptol itself.

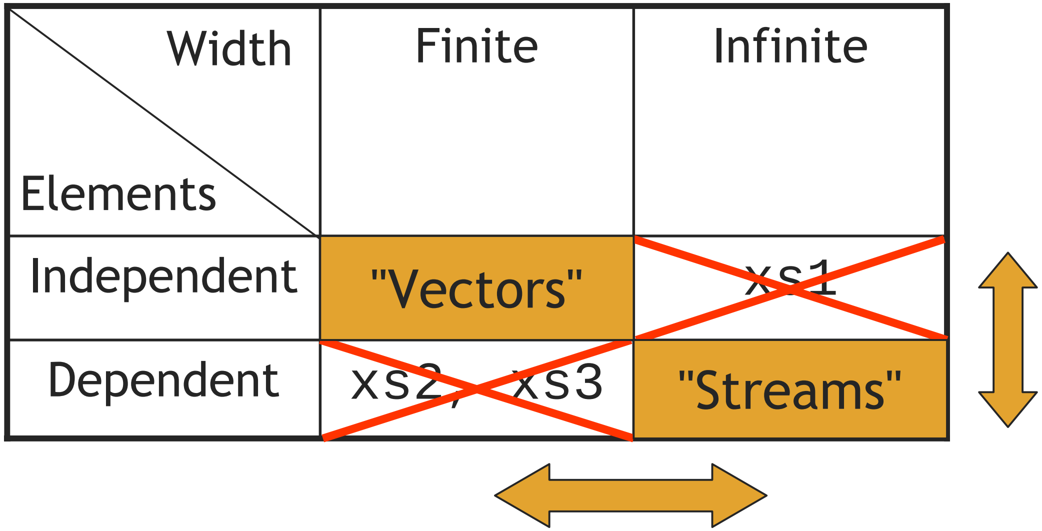
- Programmer may thus start with a reference implementation, and progressively refine it to an efficient implementation

Constraints on embedded devices

- Dynamic allocation of memory generally frowned upon
- Memory at a premium
- Don't always have access to high quality C compiler
- Alas, these all work against the implementation of a declarative language such as Cryptol
  - Existing backend targets C, and makes use of garbage collected heap allocated memory
- We have developed μCryptol, a sub-language of Cryptol intended for embedded devices
  - Current target is the Rockwell Collins AAMP7 processor
  - Complier goes directly from source to AAMP7 binary image
  - Complier intended to be verifying: AAMP7 program may be shown input/output equivalent to μCryptol source program
- Biggest challenge is dealing with streams

Sequence flavors

```
xs0 = [ x + 1 | x <- [0..3] ];
xs1 = [0..];
xs2 = take{5} ([0] # xs2);
xs3 = [0] # [ x + y | x <- xs3 | y <- [0..3] ];
xs4 = [0, 1] # [ x + y | x <- xs4 | y <- drops{1} xs4 ];
```



Width / Elements	Finite	Infinite
Independent	"Vectors"	~~xs1~~
Dependent	~~xs2, xs3~~	"Streams"

- Cryptol Classic distinguishes sequences according to width
- Semantics and compilation must distinguish according to element dependencies
- For simplicity, μCryptol allows only two combinations
- Easy to re-express others using just these two

Vectors and streams in μCryptol

- Vectors
  - Types like `B^8`, `(B, B^8)^4`
  - Must be non-recursive
  - Must be finite, with statically known width
  - May compute elements in any order
    - Eg sequential for loop, parallel hardware, etc
- Streams
  - Types like `B^inf[32,4]`, `B^5^inf[8,2]`
  - Must be recursive
  - Must be infinite (unbounded) width
  - Must compute elements in a particular order

Stream expressiveness

- How expressive a language of streams do we need?
- Choices have huge impact on time and space efficiency

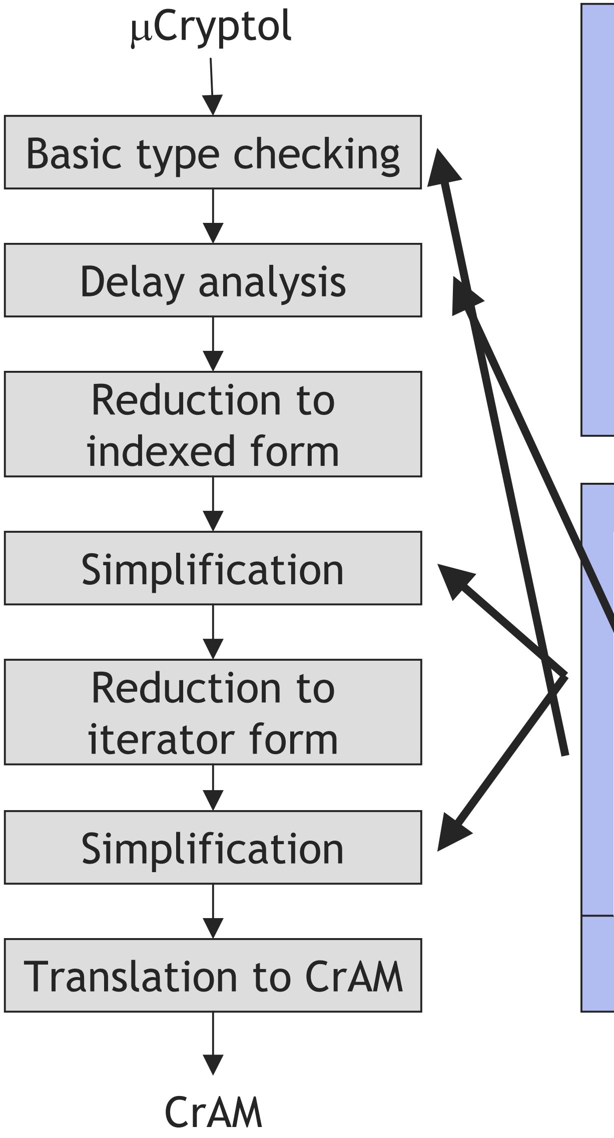
```
ys0 = [0, 1] ## [ x + y | x <- ys0 | y <- drops{1} ys0 ];
ys1 = (drops{4} ys1 ## [0..3]) ## ys1;
ys2 = [0] ## [ x + y | x <- ys2, y <- [0, 1] ];
ys3 = [0..3] ## [ (ys3 @ (3 - (x % 4))) + 1 | x <- ys3 ];
ys4 = [0] # [ x + y | x <- ys4 | y <- drops{1} ys4 ];
```

	Denotational	Operational
ys0	$S = 1+E+E^2 \to E$	Sequential, finite history ✓
~~**ys1**~~	~~$S = E^8$~~	~~Non-sequential, cyclic~~
~~**ys2**~~	~~$S = \nu\alpha.E \times \alpha$~~	~~Sequential, unbounded history~~
~~**ys3**~~	~~$S = N \to E_\perp$~~	~~Not obviously sequential or cyclic~~
~~**ys4**~~	~~$S = N \to E_\perp$~~	~~Possibly undefined elements~~

μCryptol

Cryptol Classic

Compiling streams



μCryptol

Basic type checking

Delay analysis

Reduction to indexed form

Simplification

Reduction to iterator form

Simplification

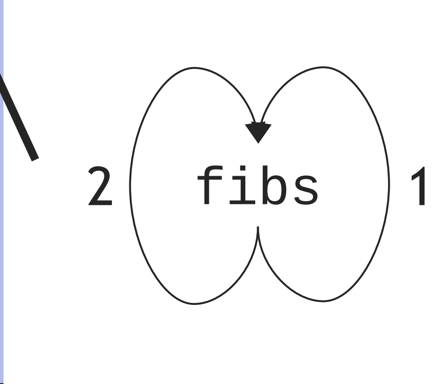
Translation to CrAM

CrAM

```
rec fibs : 2^8^inf;
    fibs = [0, 1] ##
      [ x + y | x <- fibs
              | y <- drops{1} fibs ];


fib : 2^16 -> 2^8;
fib i = fibs @@ i;
```

```
fibs : 2^16 -> 2^8^2;
```



```
                    @ i
                    (i-2 % 2) +
                    (i-1 % 2);
```

on of actual situation

```
                    i % 2);
```

```
      (a^inf, 2^b) -> a
```

Type checking streams

- We implement delay analysis within the type system
  - "External" stream types (as seen by the programmer)

    τ^inf[w,h]

  - "Internal" stream types (as used by the type checker)

    τ^inf{w,m,l}

  where

τ	stream element type
w	width of stream indexes
h	no. previous stream elements needed to compute next
m	delay from stream definition to current term context
l	recursive stream level

- Stream primitives track delays by polymorphism

```
## : forall wl, wi, t, d, l .
         t^wl, t^inf{wi, d + wl, l} -> t^inf{wi,d,l}
```

Status

- Type system implemented within the μCryptol compiler
- Work needed to integrate μCryptol and current Cryptol

Public-key Algorithms

Symmetric vs Public

- Symmetric-key algorithms typically work in:
  - $\mathbf{Z}_{2^n}$ Arithmetic on naturals modulo 2^n (where n is known at compile-time)
  - $\mathbf{F}_{2^n}$ Binary field (polynomials over $\mathbf{F}_2$) (where n is known at compile-time) *(eg AES)*
  - Vectors and tuples over the above
  - Recursive streams over the above
- Public-key algorithms typically work in:
  - $\mathbf{F}_p$ Prime field on prime p (where p may only be known at run-time) *(eg RSA)*
  - $\mathbf{E}(p,a,b,P,n,h)$ Group of points on elliptic curve over $\mathbf{F}_p$ defined by $y^2 = x^3 + ax + b$ with base point P of order prime n, and group order nh (where above may only be known at run-time) *(eg ECC)*

Key design decisions

- Cryptol already has built-in support $\mathbf{Z}_{2^n}$ and $\mathbf{F}_{2^n}$
- Extending to $\mathbf{F}_p$ and $\mathbf{E}(\ldots)$ presents many challenges:
  - How to handle the run-time field or elliptic curve parameters?

    $\Rightarrow$ Specially named variable
  - Is an element of (eg) $\mathbf{F}_{29}$ incompatible with an element of $\mathbf{F}_{31}$?

    $\Rightarrow$ No, the programmer must keep them separate
  - Is an element of (eg) $\mathbf{F}_{31}$ incompatible with an element of $\mathbf{Z}_{2^5}$?

    $\Rightarrow$ No, the programmer may switch between these two views
  - Should the new operators be implemented as built-in primitives, or supplied as a library?

    $\Rightarrow$ For prime fields, implemented within interpreter using GMP

    $\Rightarrow$ For elliptic groups, implemented as a Cryptol library

Public-key in Cryptol

- The type system remains unchanged. Eg:
  - An element of $\mathbf{F}_{31}$ is represented by a 5 or greater bit word
- New operators expect a specially named variable to bind the necessary run-time parameters. Eg:
  - Move a 6-bit word into $\mathbf{F}_{31}$

```
Cryptol> @% 33 where modulus = 31
  2
```

  - Perform arithmetic in $\mathbf{F}_{31}$

```
                      **% 2 where modulus = 31
  8
```

  - Perform arithmetic on a pre-defined curve `f13`

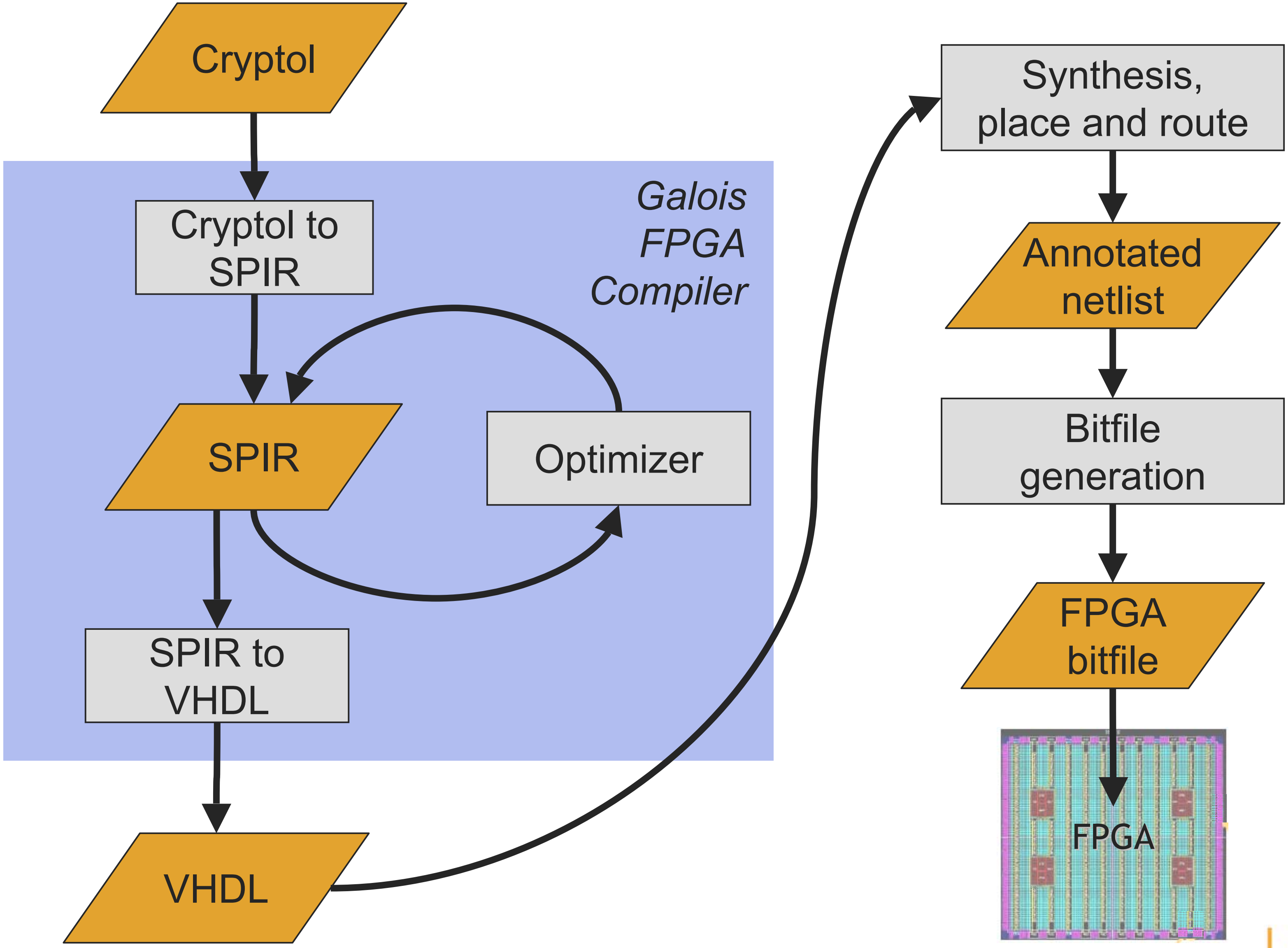
```
Cryptol> @&(1,4,1) +& @&(1,4,1) where ellipticcurve = f13
  (11, 9, 1)
```

Status

- Current implementation:
  - 3 point multiplies (on a NIST curve) per second
- Future work:
  - Support in multiple backends (currently just interpreter)

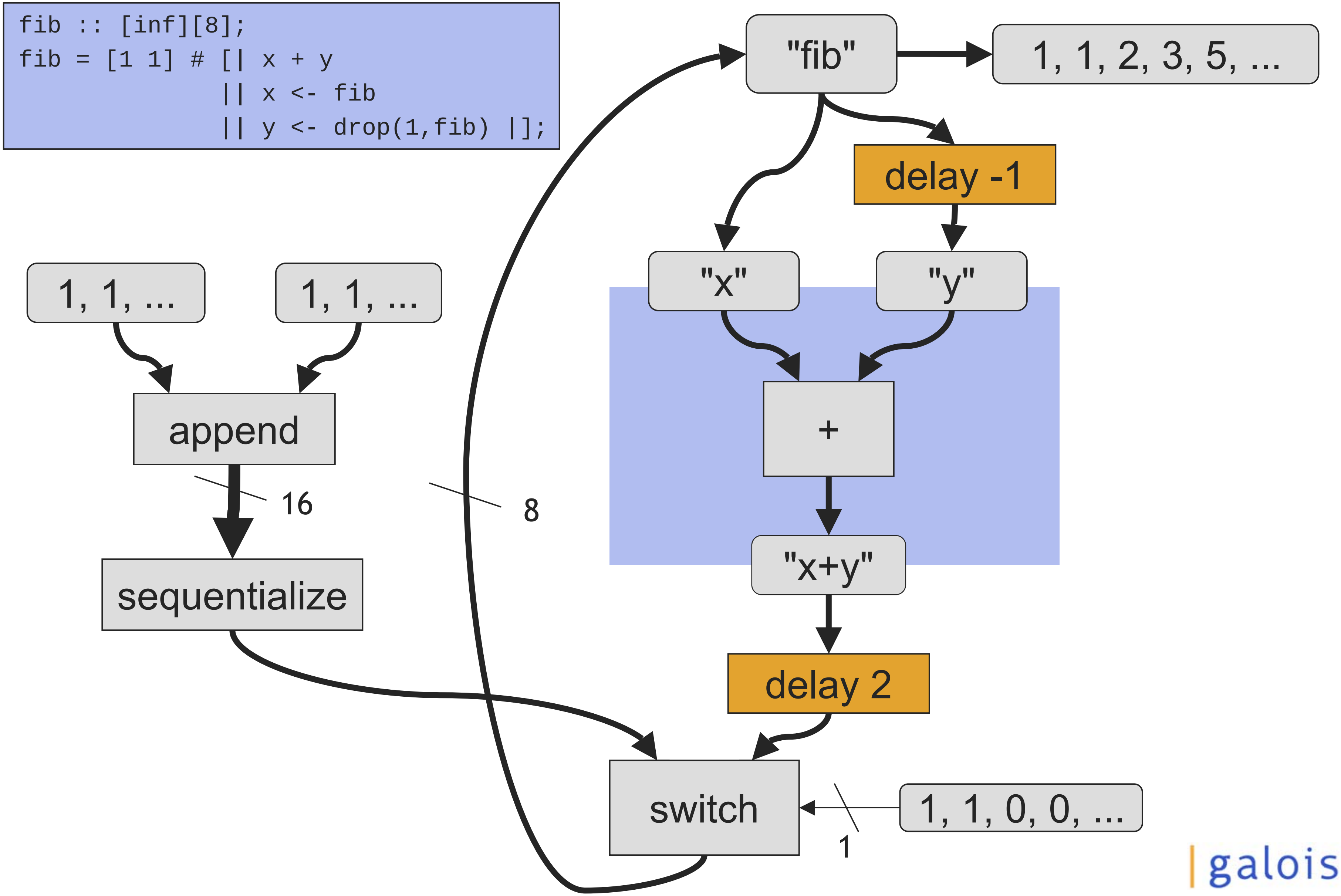
Cryptol to FPGA

Technical approach



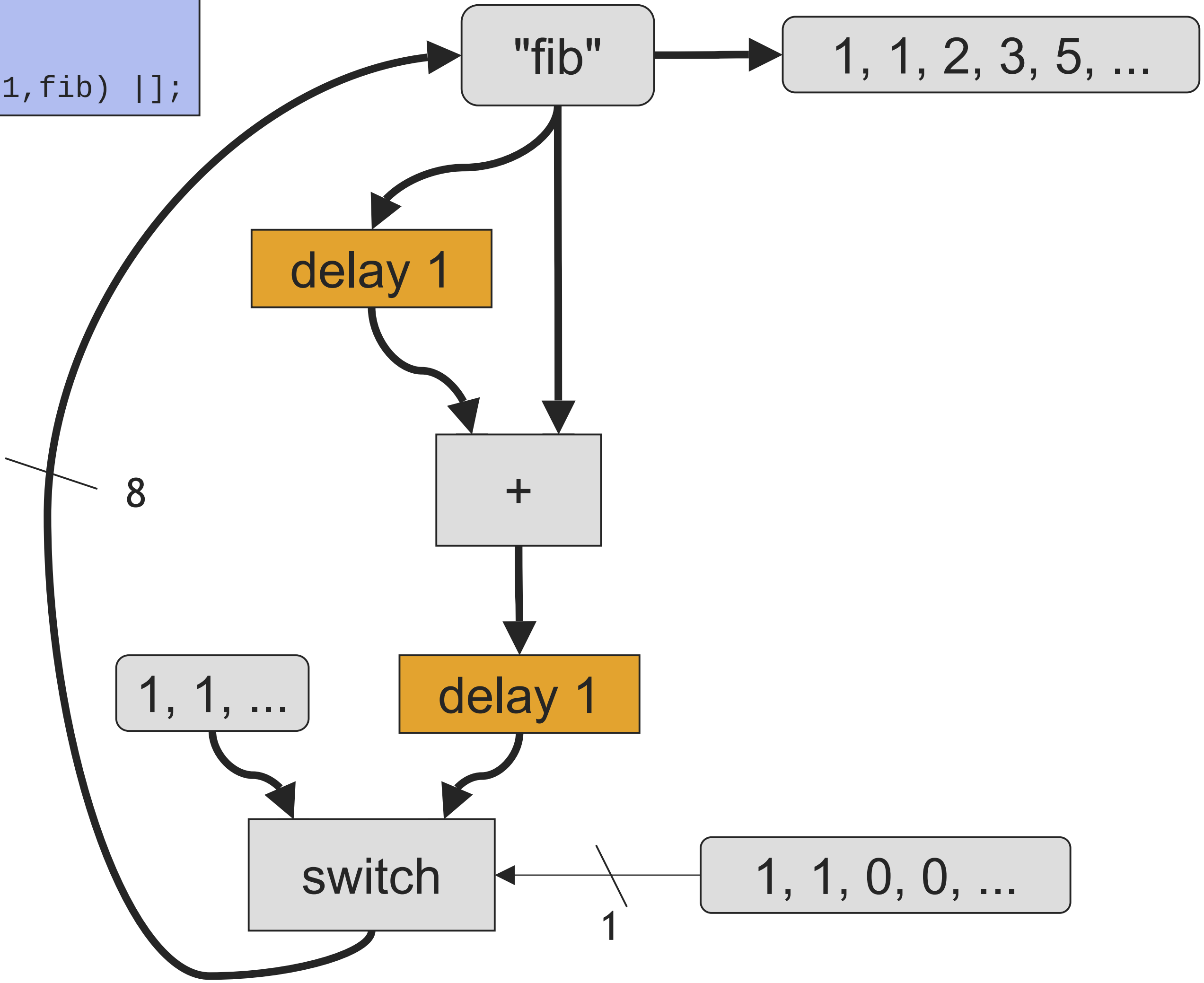
`fib`: Intermediate representation

```
fib :: [inf][8];
fib = [1 1] # [| x + y
              || x <- fib
              || y <- drop(1,fib) |];
```



`fib`: Optimized representation

```
fib :: [inf][8];
fib = [1 1] # [| x + y
              || x <- fib
              || y <- drop(1,fib) |];
```



Pipelining TEA: starting point

```
code : ([2][32],[4][32]) -> [2][32];
code ([y z], [k0 k1 k2 k3]) = [(ys @ 32) (zs @ 32)]
  where {
    sums = [0x9e3779b9] # [| x + 0x9e3779b9 || x <- sums |];
    ys = [y] # [| (y + ((z  << 4) + k0 ^ (z +sum) ^ (z  >> 5) + k1))
               || sum <- sums || y <- ys || z <- zs |];
    zs = [z] # [| (z + ((y' << 4) + k2 ^ (y'+sum) ^ (y' >> 5) + k3))
               || sum <- sums || y' <- tail ys || z  <- zs |];
  };
```

- What are the sequential dependencies?
  - 32 outer rounds, each requires result of previous
  - Expression in `zs` comprehension depends on value of `ys` at the same round
  - `sums` could be precomputed

Pipelining TEA: outer rounds

- Convert streams `ys`, `zs` and `sums` to a round function
- Then unwind outer loop 32 times

```
round : ([32], [32], [32], [4][32]) -> ([32], [32], [32], [4][32]);
round (y, z, sum, [k0 k1 k2 k3]) = (nexty, nextz, nextsum, [k0 k1 k2 k3])
  where {
    nexty = y + ((z  << 4) + k0 ^ (z +sum) ^ (z >> 5) + k1);
    nextz = z + ((nexty << 4) + k2 ^ (nexty+sum) ^ (nexty >> 5) + k3);
    nextsum = sum + delta; };

pipeline32 : [inf]([32],[32],[32],[4][32]) -> [inf]([32],[32],[32],[4][32]);
pipeline32(vs0) = drop(32,vs32) where {
    vs32 = [zero] # [| round x || x <- vs31 |];
    vs31 = [zero] # [| round x || x <- vs30 |];
    ...
    vs1  = [zero] # [| round x || x <- vs0 |]; };
```

Pipelining TEA: inner pipeline

- Pipeline round function into two parts:

```
roundA (y, z, sum, [k0 k1 k2 k3]) = (nexty, z, sum, [k0 k1 k2 k3])
  where {
    nexty = y + ((z  << 4) + k0 ^ (z +sum) ^ (z  >> 5) + k1); };
roundB (nexty, z, sum, [k0 k1 k2 k3]) = (nexty, nextz, nextsum, [k0 k1 k2 k3])
  where {
    nextz = z + ((nexty << 4) + k2 ^ (nexty+sum) ^ (nexty >> 5) + k3);
    nextsum = sum + delta;
};

pipeline64 : [inf]([32],[32],[32],[4][32]) -> [inf]([32],[32],[32],[4][32]);
pipeline64(vs0) = drop(64,vs64)
  where {
    vs64 = [zero] # [| roundB x || x <- vs63 |];
    vs63 = [zero] # [| roundA x || x <- vs62 |];
    vs62 = [zero] # [| roundB x || x <- vs61 |];
    vs61 = [zero] # [| roundA x || x <- vs60 |];
    ...
    vs2  = [zero] # [| roundB x || x <- vs1 |];
    vs1  = [zero] # [| roundA x || x <- vs0 |]; };
```

Status

- Have tested on Spartan 3 (Xilinx XC3S200, 200 Kgates) and Wildcard II (Xilinx XC2V3000, 3000 Kgates) evaluation hardware
- Pipelined DES performance comparable with hand-written VHDL using Xilinx VHDL synthesis toolchain