Robust Verification Tools for Improved Secure System Evaluation

David Hardin
David Greve
Matthew Wilding

Advanced Technology Center
Rockwell Collins

with contributions by

Tom Johnson and Sung Kim
Rockwell Collins

John Matthews and Lee Pike
Galois Connections

Eric Smith
Stanford University

Bill Young
University of Texas at Austin

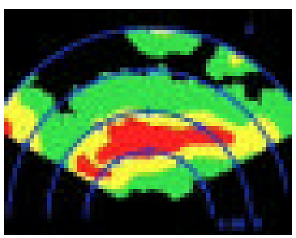
UNCLASSIFIED

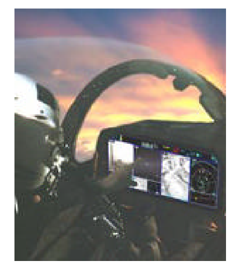
Overview

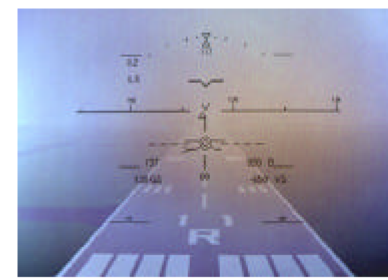
- Rockwell Collins Introduction
- AAMP7G Microprocessor
  - MILS Certification
- vFaat Program
- SHADE Program
  - AAMP7G Instruction Set Formal Model
  - AAMP7G tools
  - Microcryptol Verifying Compiler
  - Compositional Cutpoint Reasoning
- Summary

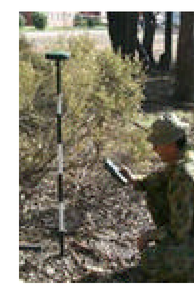
A World Leader in Aviation Electronics and Airborne/ Mobile Communications Systems for Commercial and Military Applications

- Communications
- Navigation
- Automated Flight Control
- Displays / Surveillance
- Aviation Services
- In-Flight Entertainment
- Integrated Aviation Electronics
- Information Management Systems









The Problem – High-Assurance for Security Applications

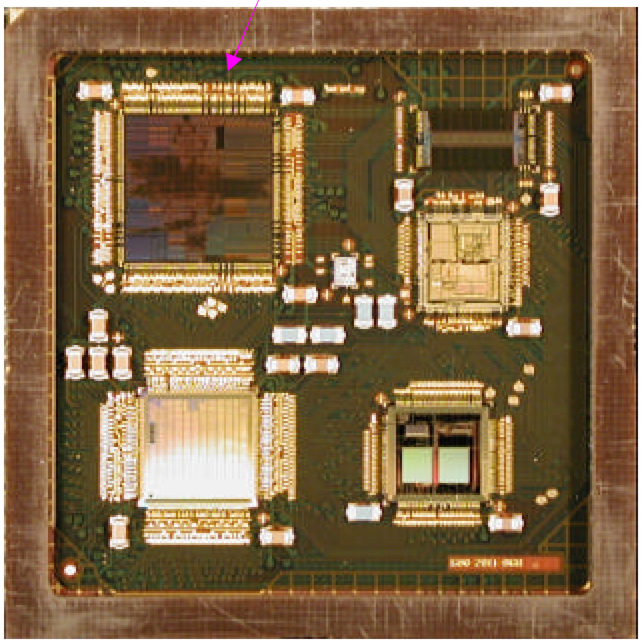
- Flawed implementations can have grave consequences
  - So NSA performs intensive evaluations of critical encryption devices
- Evaluation process is difficult
  - Increasingly numerous crypto implementations
  - Trusted experts are scarce
  - Review process is time-consuming and expensive
  - Optimized crypto algorithms are complex, easy to overlook corner cases
- Highest Evaluation Assurance Level *requires* formal proofs
  - Industry has very little practical experience in this area

Rockwell Collins AAMP7G CPU

ADVANCED COMPUTING SYSTEMS

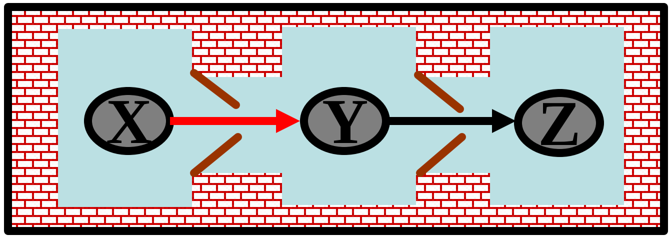
- Developed by RCI Advanced Technology Center
- Used in RCI GPS and Information Assurance products
- High Code Density
- Low Power Consumption (250 mW)
- 100 MHz operation
- Screened for full military temp range
- Implements *intrinsic partitioning*

AAMP7 in GPS SAASM MCM



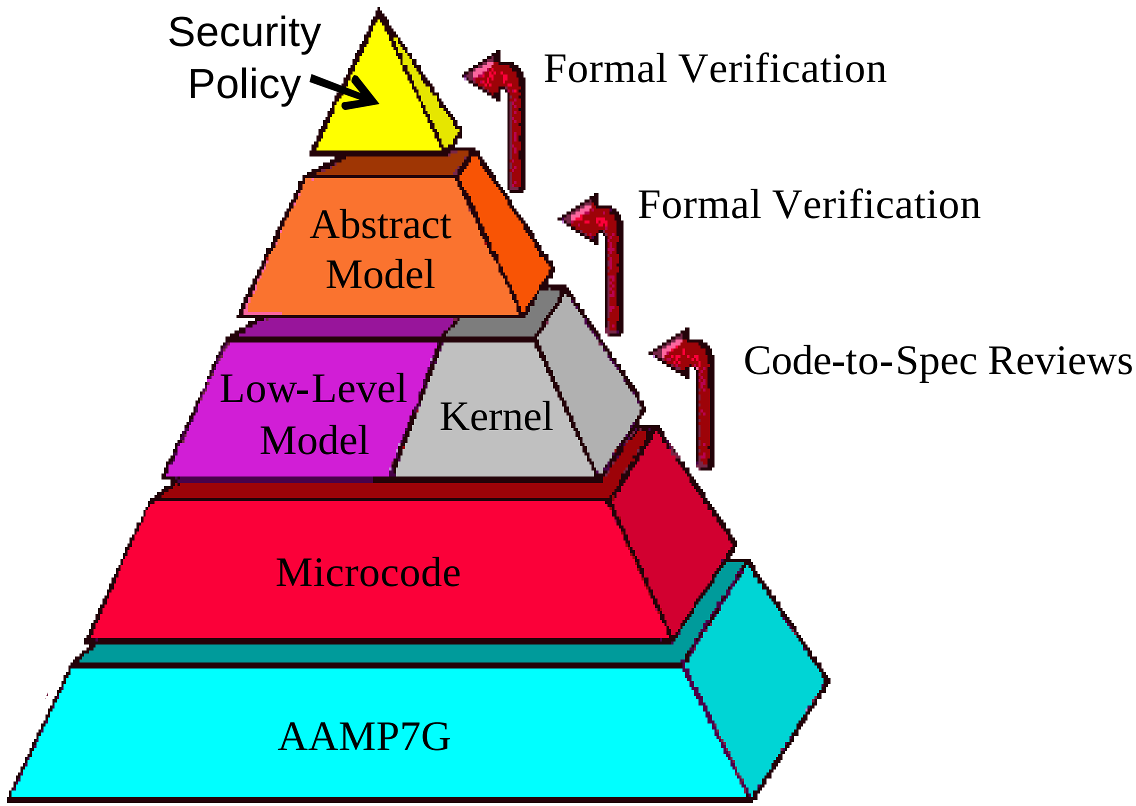
Intrinsic partitioning

- Computing Platform Enforces Data Isolation
- “Separation Kernel in Hardware”



AAMP7G Formal Verification

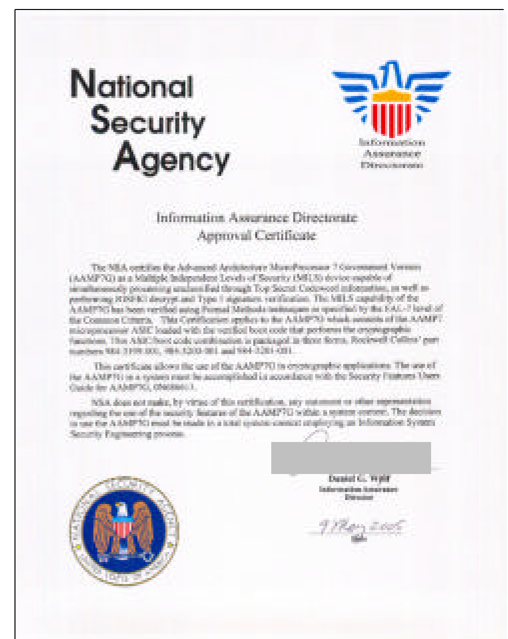
Common Criteria
EAL7 Proof Obligations



AAMP7G Intrinsic Partitioning Formal Verification

Program Accomplishments

- **Developed formal description of separation for uniprocessor, multipartition system**
- **Modeled trusted AAMP7G microcode**
- **Constructed machine-checked proof that separation holds of AAMP7G model, using ACL2**
- **Model subject of intensive code-to-spec review**
- **Satisfies NSA MILS formal methods evaluation requirements patterned after Common Criteria EAL7+ with respect to ADV**
- ***NSA MILS certificate granted in May 2005***
    - AAMP7G can concurrently process Unclassified through Top Secret Codeword information



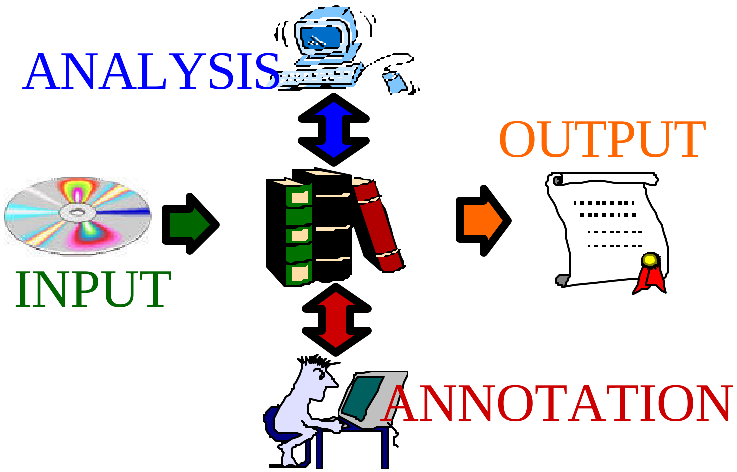
- **RCI IR&D funded**
- **Capability developed in multiyear RCI formal methods research program**

vFaat: von Neumann Formal Analysis and Annotation Tool

ADVANCED COMPUTING SYSTEMS

Program Objectives

- Extend imperative code analysis techniques to push the state-of-the-art in formal analysis
- Increase automated analysis integration into standard development practices
- Demonstrate these techniques on RC-relevant examples that provide assurance required in current evaluation efforts and help identify future certification standards



A COMPUTATIONAL LOGIC
ACL2
APPLICATIVE COMMON LISP

Philosophy

- **Build on Experience**
  - **Codify Successful Techniques**

- **Focus on Proof Structure**
  - **Driven by Control/Data Flow**
  - **Encourage Hierarchy and Abstraction**
  - **Emphasizes Compositional Reasoning**

- **Target Independence**
  - **State Machines and Data Paths**
  - **Assembly/Object Code**
  - **Microcode**
  - **Software**

- **Theorem Prover Independence**
  - **Definitional Principle**
  - **Conditional Rewrite Rules**

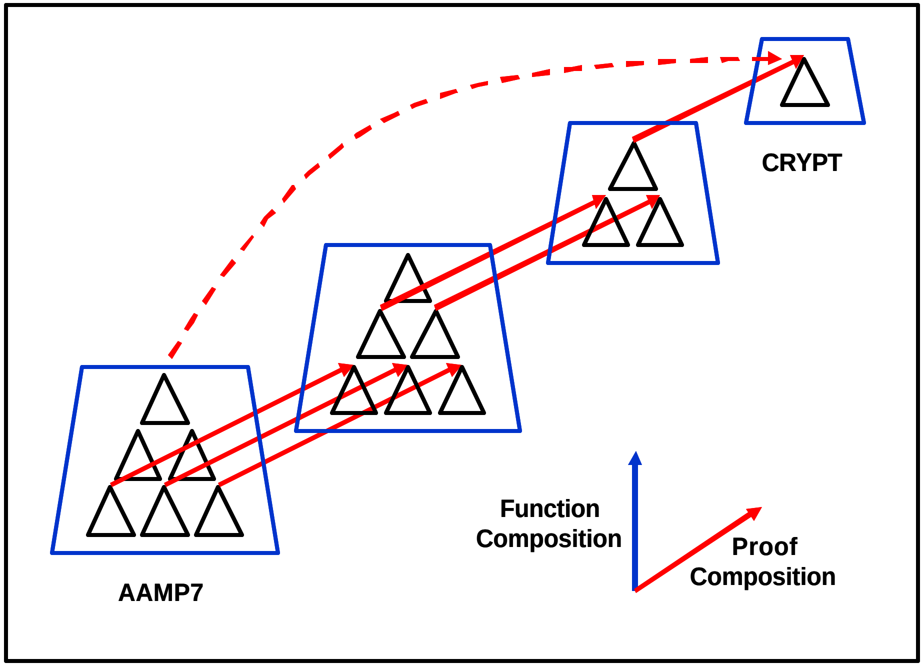
Eclipse Overview

- A multi-language IDE
  - World-class Java IDE (JDT)
  - Also C and C++ (CDT), Perl, Ada, etc.
- A tool development platform
  - Stand-alone Java tools using the JDT
  - Plug-in development using the PDE and existing Eclipse components
- A tool integration platform
  - Forms the basis for a highly integrated engineering environment
  - Use a variety of integration methods (invocation, GUI only, full)
  - Support integration of both legacy and new tools

> “an open source software development project dedicated to providing a robust, full-featured, commercial-quality, industry platform for the development of highly integrated tools.”

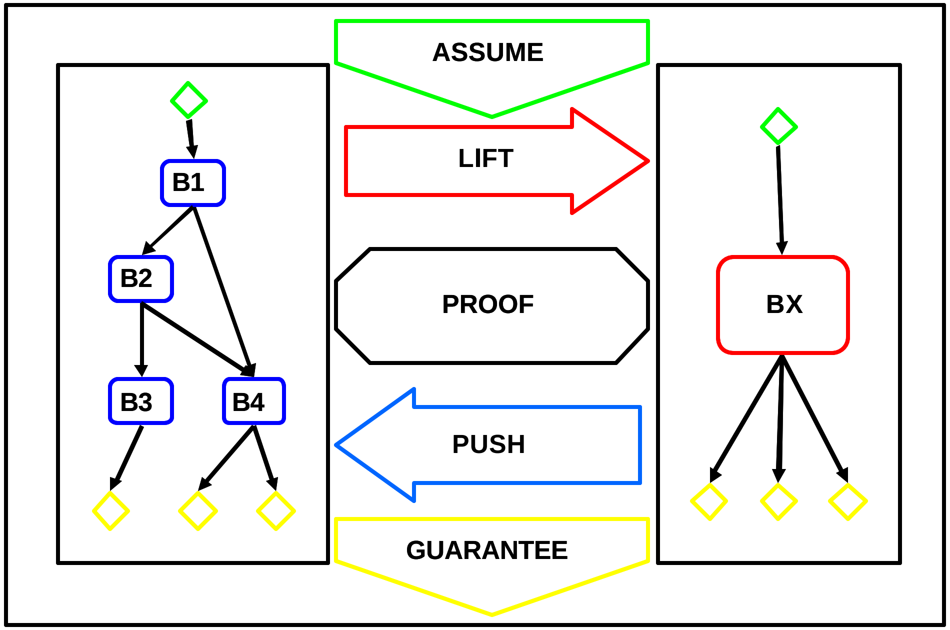
Composition

- Proof
  - Two Axis Structure
    - Function Composition
    - Proof Composition
  - vFaat Exploits This Duality
- Function Composition
  - Big Functions from Smaller Functions
  - Managed by Views and CANs
  - Encourages Good Library Development
- Proof Composition
  - Big Proofs from Smaller Proofs
  - Managed by Strata and Links
  - Encourages Generic Proof Development (Reusable Specifications)

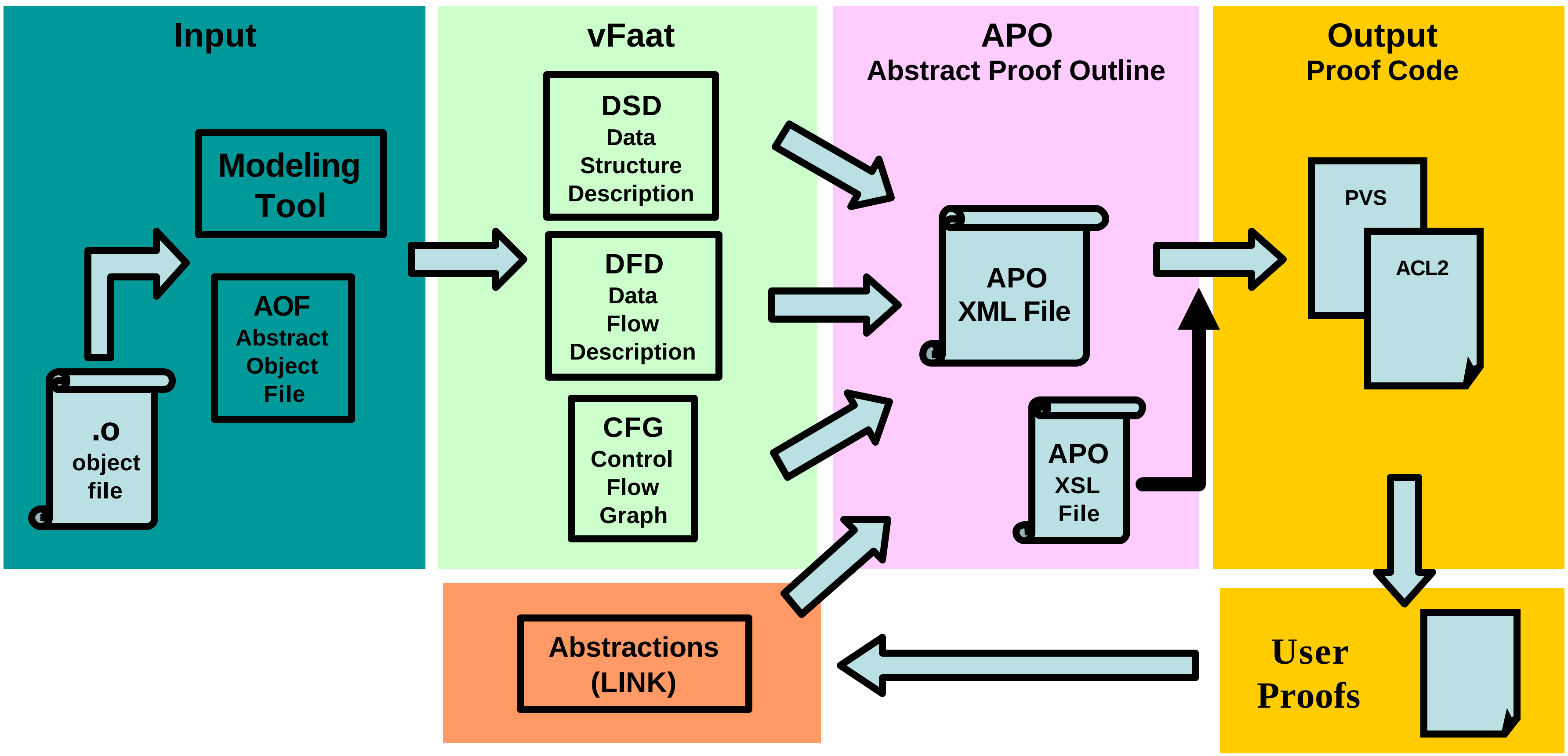


vFaat CFN

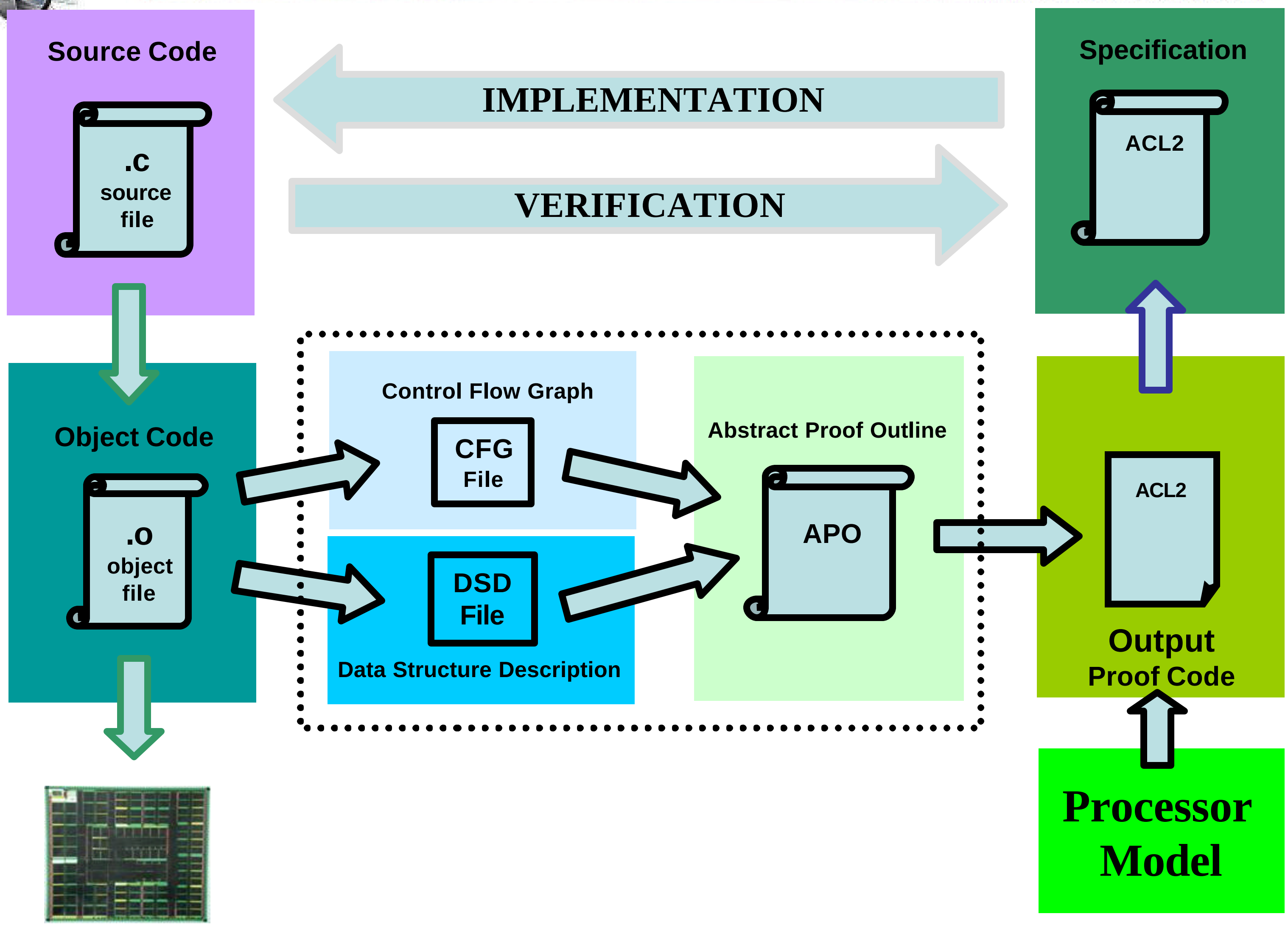
- Imperative Code Emphasis
  - Typical of Low Level Models
- Control Flow Node
  - Temporal Abstraction of Imperative Execution
  - Hierarchical
  - Defines Proof Structure
- Purpose
  - Model of Execution
  - Provides Template for
    - Proof Structure
    - Clock Function
    - Branch Function
    - Assumptions
    - Functional Composition



vFaat Tool Chain



Design and Verification Flow

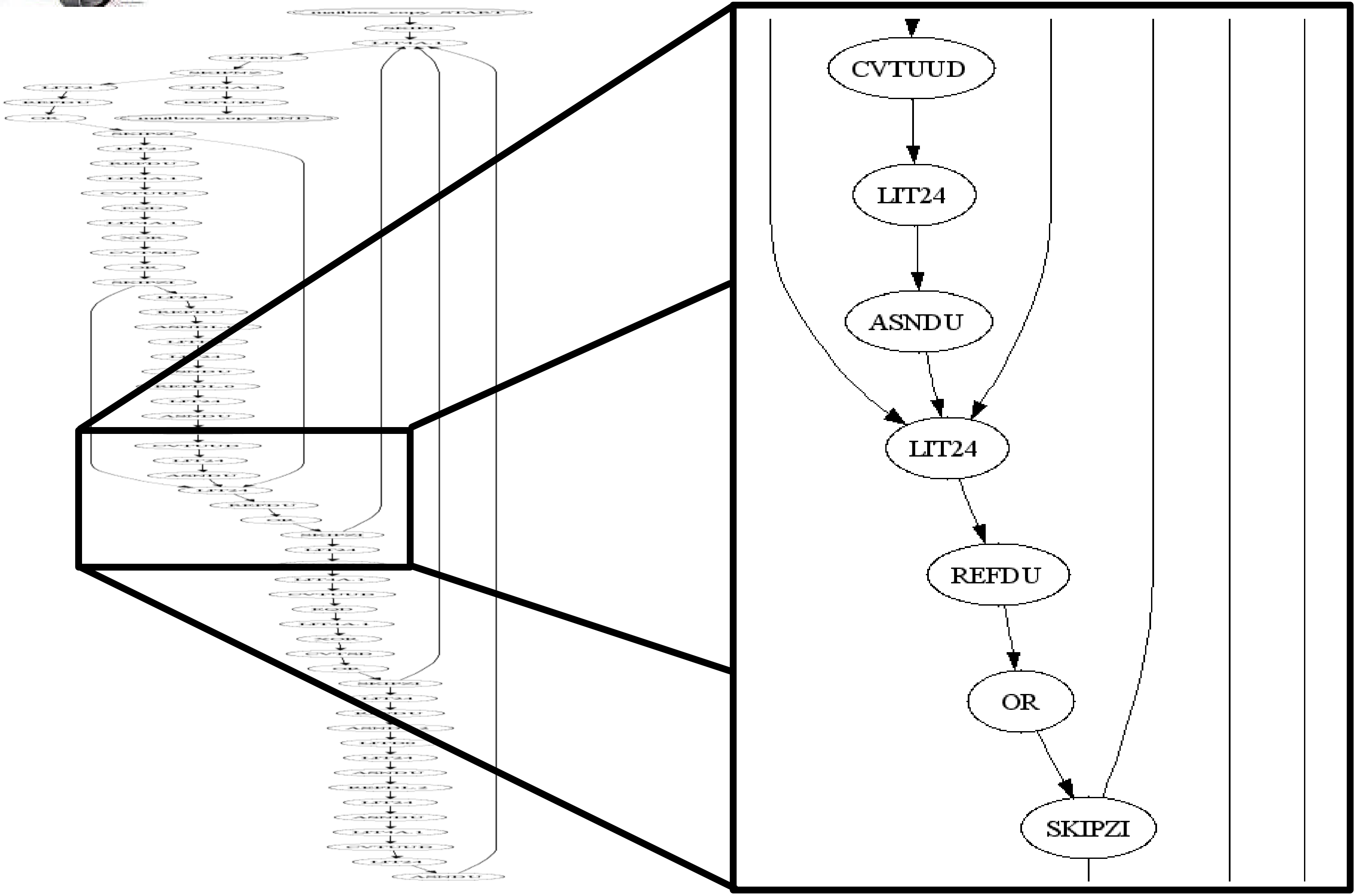


C Source Code Implementation


```
mailbox_copy() {

 uword32_t input_data;
 uword32_t output_data;

 while (1) { // forever

  if (CHARACTER_INPUT_0_READY && OUTPUT_1_CONSUMED) {

   // Consume the character and allow the producer to continue
   input_data = READ_INPUT_0;
   NOTIFY_INPUT_0_CONSUMED;

   WRITE_OUTPUT_1(input_data);

   // Notify the consumer that there is output
   NOTIFY_OUTPUT_1_READY;
  }

  if (CHARACTER_INPUT_1_READY && OUTPUT_0_CONSUMED) {

   // Consume the character and allow the producer to continue
   output_data = READ_INPUT_1;
   NOTIFY_INPUT_1_CONSUMED;

   WRITE_OUTPUT_0(output_data);

   // Notify the consumer that there is output
   NOTIFY_OUTPUT_0_READY;
  }

 } // forever
```

Object Code CFG

ADVANCED COMPUTING SYSTEMS



Verification Hypotheses

vFaat Generated

```
(defun data-structures (offset aamp::st)
  (let ((g1 (byte-stream-pointer-block (CHARACTER_INPUT_0_DATA)  offset aamp::st))
        (g2 (byte-stream-pointer-block (CHARACTER_INPUT_0_READY) offset aamp::st))
        (g3 (byte-stream-pointer-block (CHARACTER_INPUT_1_DATA)  offset aamp::st))
        (g4 (byte-stream-pointer-block (CHARACTER_INPUT_1_READY) offset aamp::st))
        (g5 (byte-stream-pointer-block (CHARACTER_OUTPUT_0_DATA)  offset aamp::st))
        (g6 (byte-stream-pointer-block (CHARACTER_OUTPUT_0_READY) offset aamp::st))
        (g7 (byte-stream-pointer-block (CHARACTER_OUTPUT_1_DATA)  offset aamp::st))
        (g8 (byte-stream-pointer-block (CHARACTER_OUTPUT_1_READY) offset aamp::st))
        )
        (and
         (check-program-image (mailbox_copy_image) offset aamp::st)
         (aamp::no-code-data-clash cenv denv)
         (bag::unique
          (append
           (image-footprint footprint cenv pc offset)
           (gacc::addresses-of-data-words-univ2 g1)
           (gacc::addresses-of-data-words-univ2 g2)
           (gacc::addresses-of-data-words-univ2 g3)
           (gacc::addresses-of-data-words-univ2 g4)
           (gacc::addresses-of-data-words-univ2 g5)
           (gacc::addresses-of-data-words-univ2 g6)
           (gacc::addresses-of-data-words-univ2 g7)
           (gacc::addresses-of-data-words-univ2 g8)
           (gacc::addresses-of-data-words 5 denv (+ tos -5))
           (gacc::addresses-of-data-words 5 denv lenv)
           )))))
```

Executable Image

Code Location

Data/Code Separation

Global Pointers

Stack Space

Local Variables

Correspondence Proof

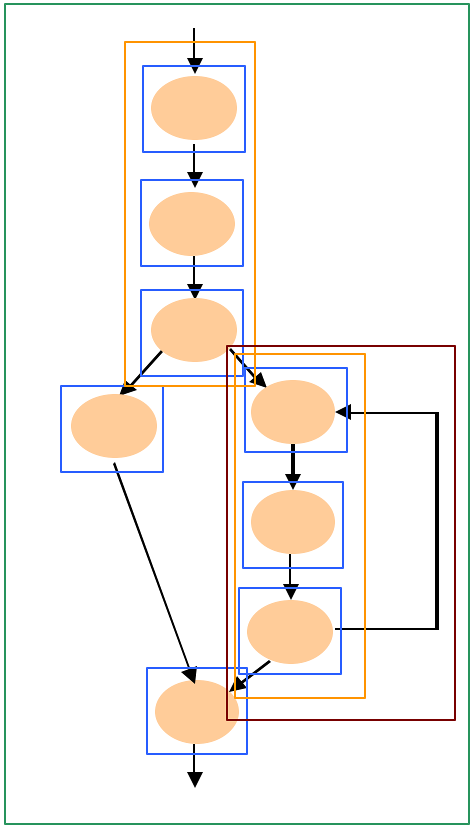
- Statement of Correspondence
  - Proof is Driven by CFG
    - Much Like Single Stepping a Debugger

```
(defthm mailbox_copy_21_22_implements_app-io-spec
  (implies
   (and
    (data-structures -102 aamp::st)
    (aamp::st-p aamp::st))
   (equal (lift (mailbox_copy_21_22_comp kst))
          (app::app-io-spec (lift kst)))))
```

Hypotheses

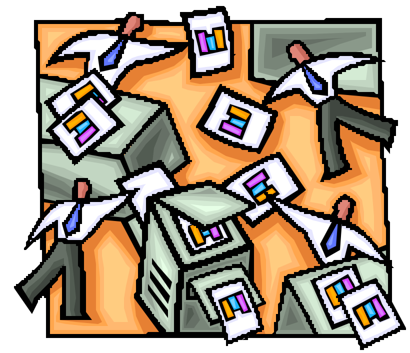
C Function (Implementation) vFaat Generated

Low Level Specification



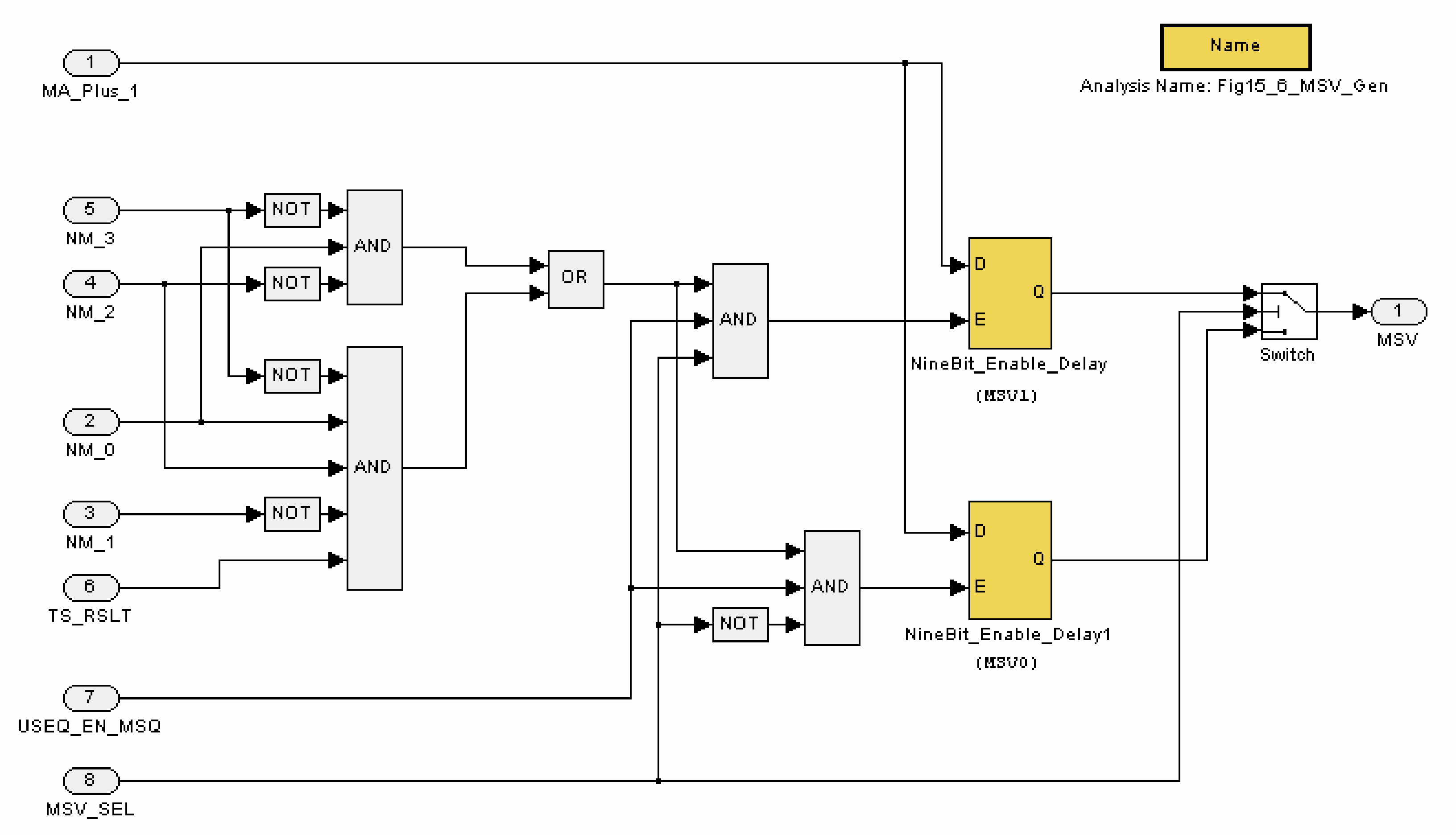
Translating HW Specifications to vFaat

- **Leveraging Existing Translation Platform**
  - Supports Model Based Development Tools
    - Simulink/Scade
  - Primary focus is Model Checking



- **vFaat Translation outputs:**
  - Control Flow Graph (CFG) for evaluation order
  - Data Structure Description (DSD)
  - Data Flow Description (DFD)

Example Simulink Hardware Component



Generated ACL2 code

```
(defun Fig15_6_MSV_Gen_Combination_comp_step (k st)
(let ((src st)
(dest (default-destT-value )))
(if (uval? k 0)
(let ((dest (let ((src (not (gp (cons :NM_3 nil) src))))
(sp (cons :Logical_Operator nil) src dest))))
(let ((dest (let ((src (not (gp (cons :NM_2 nil) src))))
(sp (cons :Logical_Operator1 nil) src  dest))))
(let ((dest (let ((src (not (gp (cons :NM_3 nil) src))))
(sp (cons :Logical_Operator2 nil) src  dest))))
(let ((dest (let ((src (not (gp (cons :NM_1 nil) src))))
(sp (cons :Logical_Operator3 nil) src  dest))))
(let ((src  (not (gp (cons :MSV_SEL nil) src))))
(sp (cons :Logical_Operator4 nil) src  dest)))))
(if (uval? k 1)
(let ((dest (let ((src (and (and (and (and (gp (cons :Logical_Operator2 nil) src)
(gp (cons :NM_0 nil) src))
(gp (cons :NM_2 nil) src))
(gp (cons :Logical_Operator3 nil) src))
(gp (cons :TS_RSLT nil) src))))
(sp (cons :Logical_Operator5 nil) src  dest))))
(let ((src  (and (and (gp (cons :Logical_Operator nil) src)
(gp (cons :NM_0 nil) src))
(gp (cons :Logical_Operator1 nil) src))))
(sp (cons :Logical_Operator6 nil) src  dest)))
(if (uval? k 2)
…
```

"NOT" gate

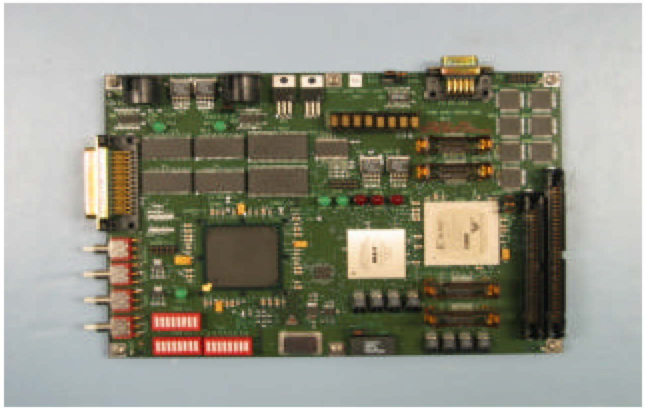
CFG steps

"AND" gate

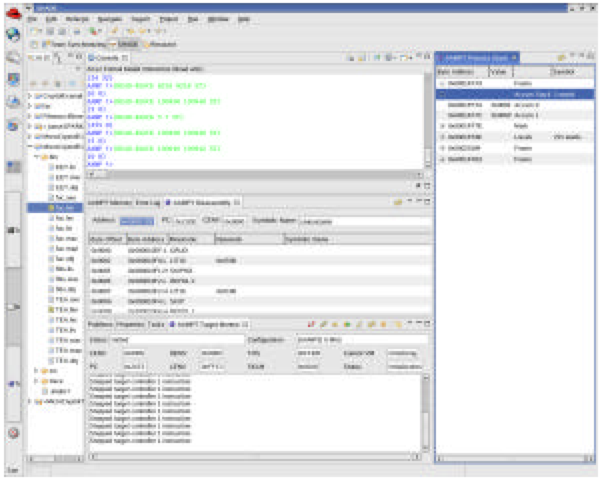
Secure, High Assurance Development Environment (SHADE)

Program Objectives

- Provide a "nuts-and-bolts" partitioned development environment.
- Develop tools and techniques to provide formal analysis at the instruction level for the AAMP7 processor
- Develop a verifying compiler for an "embeddable" subset of the Cryptol cryptographic language targeting the AAMP7
- Demonstrate a convenient, high-assured toolchain path from high-level algorithm description to load image.

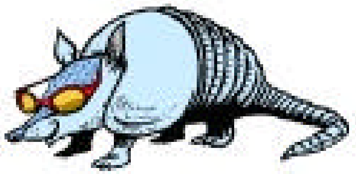


AAMP7G development board

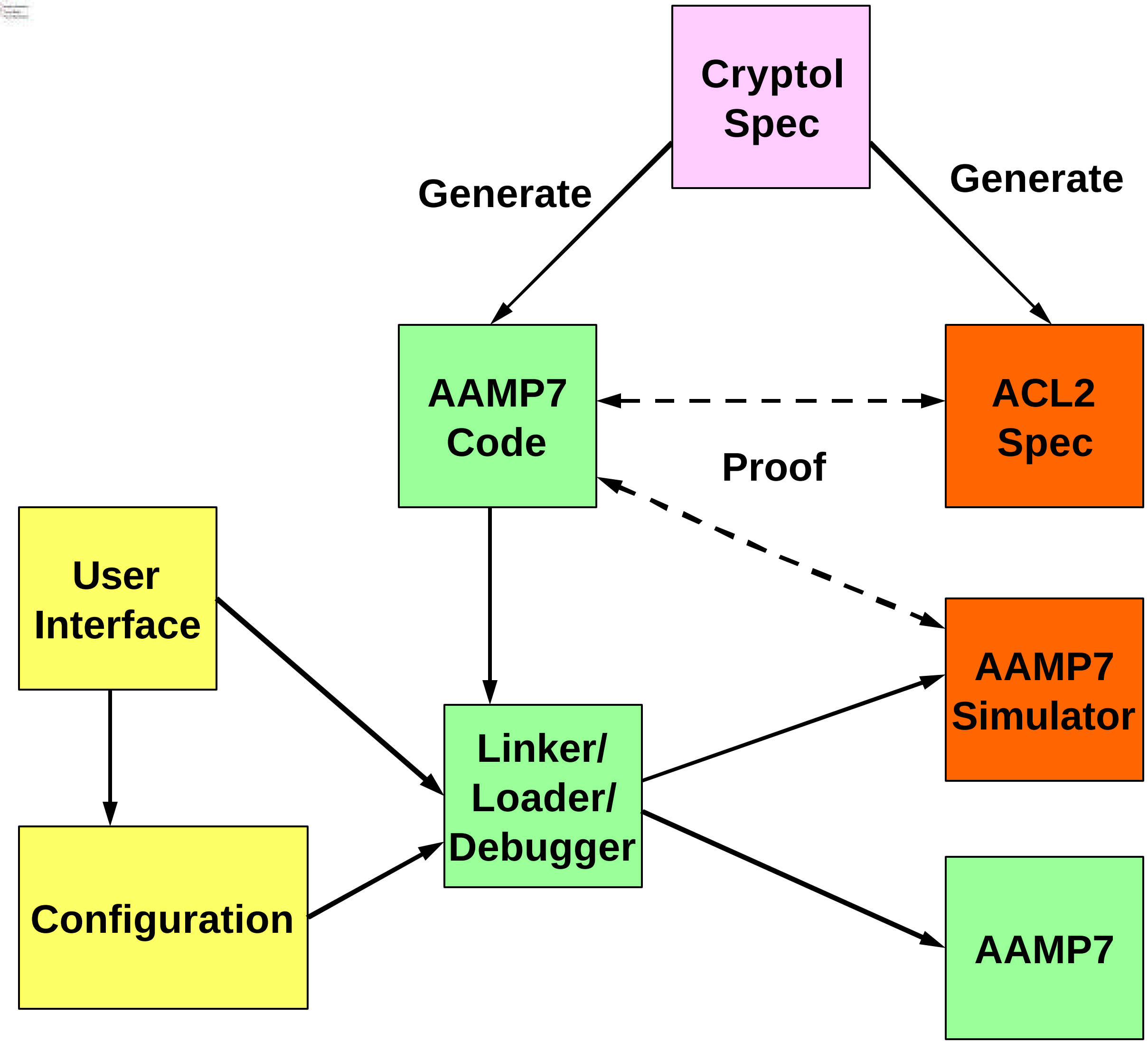


Eclipse-based AAMP7G development environment

RCI subcontractors: Galois Connections, University of Texas at Austin



SHADE Summary



SHADE Software Components

- **Eclipse-Based AAMP7G Partitioning Development Tools**
  - Target Monitor
  - Target Board Editor
  - Multipartition Builder
  - Eclipse: Very large and capable Java-based IDE construction framework
- **µCryptol -> AAMP7 verifying compiler**
  - Generates ACL2, as well as AAMP7 assembly, AAMP7 binary
  - OCaml-based
- **Instruction-level formal AAMP7G model**
  - Written in the language of the ACL2 Theorem Prover
  - Applicative subset of Common Lisp
- **AAMP Legacy Tools**
  - Compilers, Linkers, Assemblers, etc.
  - Mostly Ada

AAMP7 Instruction-Set Formal Model

- **Provides instruction-level simulator for the AAMP7**
- **Written in ACL2**
  - ***~100 KSLOC with all RCI support books***
  - ***~750 MB Lisp heap required***
- **Can be used as a processor simulator, as well as a vehicle for proof**
  - **Validated by loading AAMP processor diagnostic tests into (simulated) memory, and running the model**
- **Utilizes ACL2 single threaded object (stobj) to model CPU state; stobj updates are performed "in place", greatly reducing garbage generation at model execution time**
- **GACC (Generalized Accessor) library used to model memory, same as used in AAMP7 separation proofs**
- **New bitvector library, "super-ihs", extends ACL2 Integer Hardware Specification (IHS) library**

AAMP7G Partition Views

SHADE - Eclipse SDK

File Edit Refactor Navigate Search Project CodePro Run Window Help

AAMP7 Disassembly | AAMP7 Memory | AAMP7 Partition Schedules

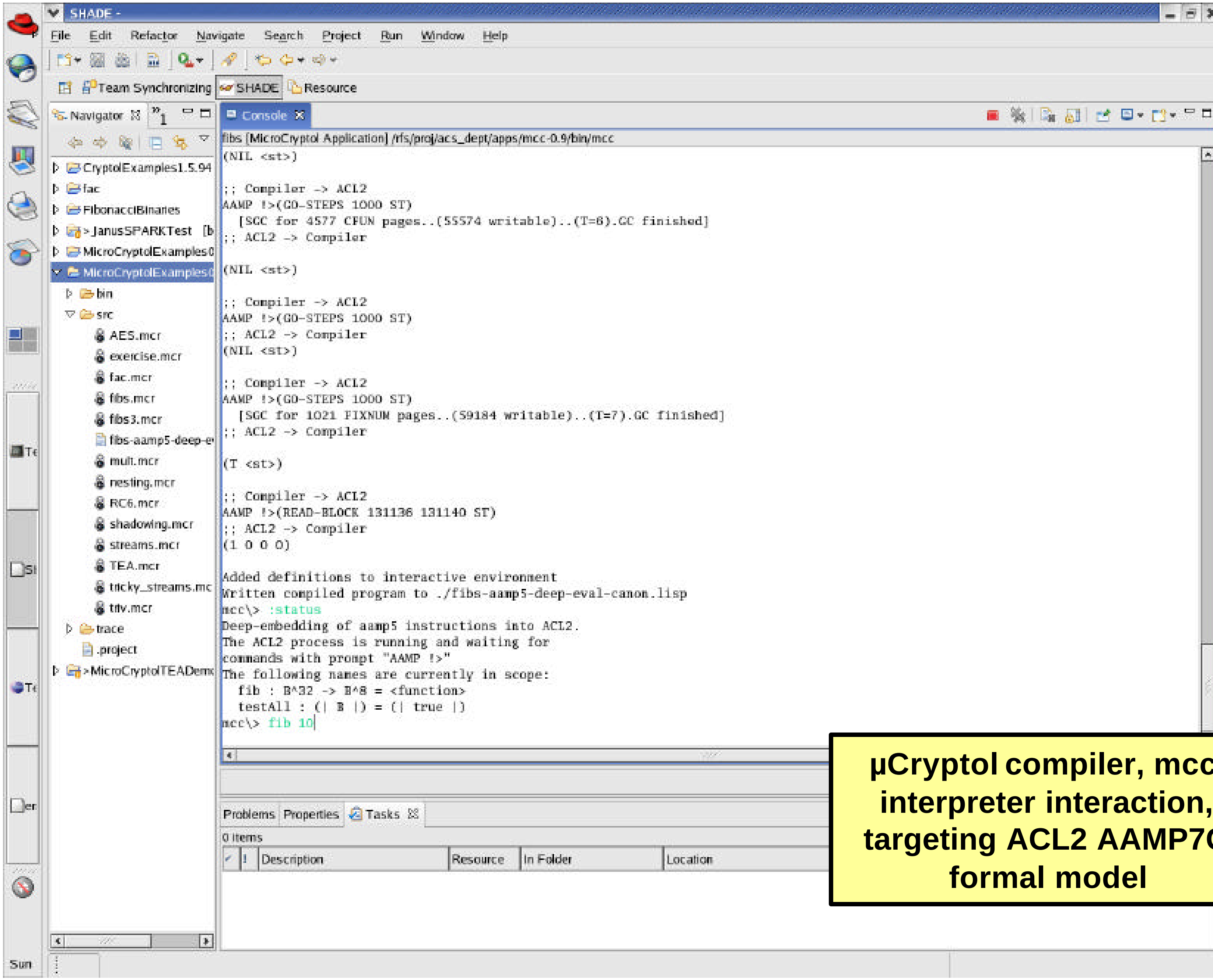
Schedule Name	VCE Address	Partition Name	Next VCE	VM #	Time Count
Cold Reset 0					
	0x00000040	mini_rte__startup	0x0000007C	0	0
	0x0000007C	mini_rte__startup	0x000000B8	1	2000
	0x000000B8	mini_rte__startup	0x000000F4	2	2000
	0x000000F4	mini_rte__startup	0x0000007C	3	2000
Cold Reset 1					
Cold Reset 2					
Warm Reset					
Power Down					

Console | AAMP7 History | AAMP7 Partition Status

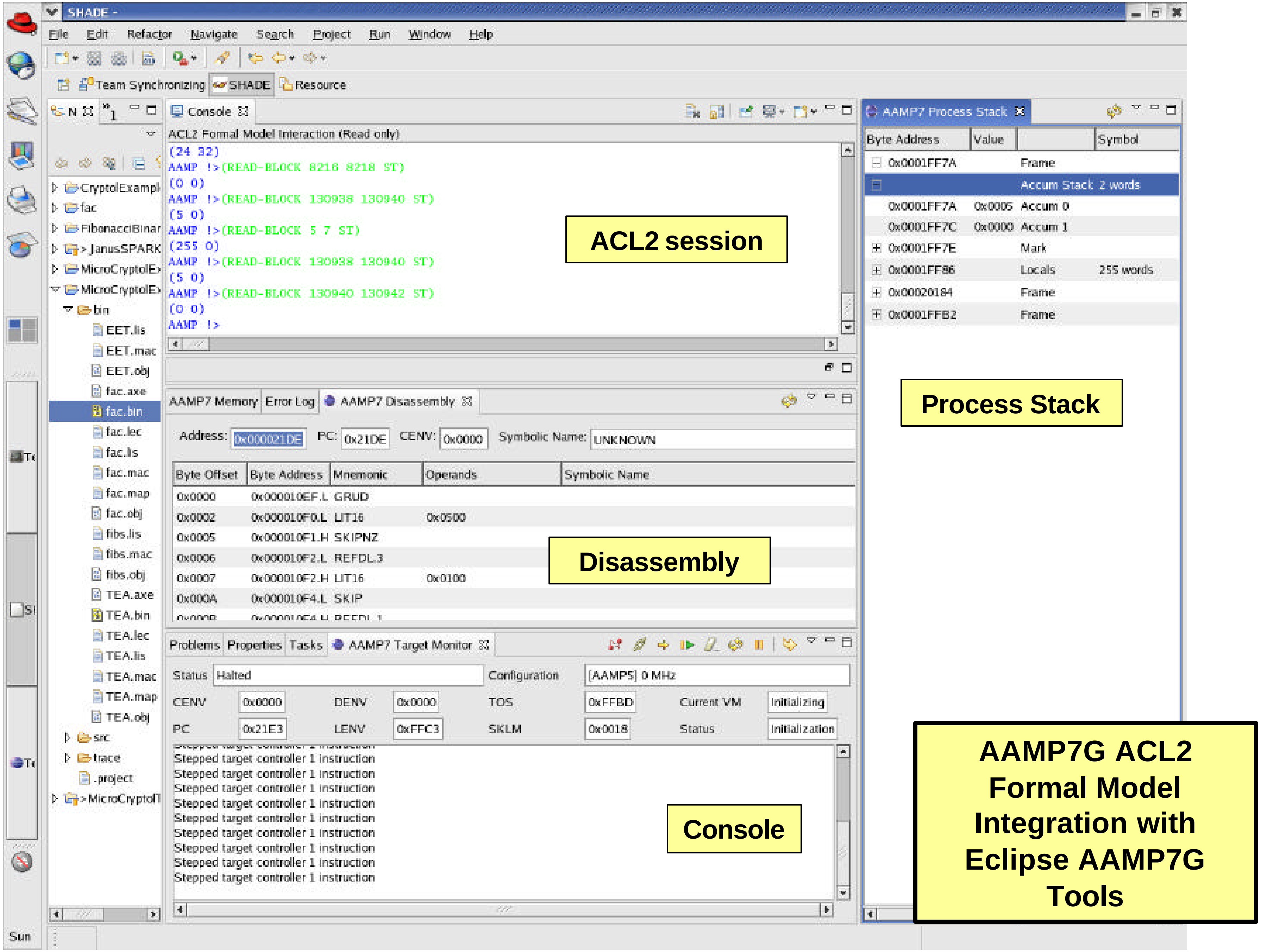
Partition Name	VM #	Status	Location	CENV	PC	DENV	LENV	Time Count	Control Block	State
mini_rte__startup	0	Continue (suspen...		0x0000	0x98CA	0x0000	0xBA50	0	0x00008000	0x00020020
			Boot:_ada_boot line 6	0x0000	0x98CA		0xBA50			
			Ada_Main:ada_main__main line 34	0x0000	0x9886		0xBA56			
			Mini_Rte:mini_rte__startup line 300	0x0000	0x9117		0xBA5E			
mini_rte__startup	1	Continue (suspen...		0x0000	0xAB7C	0x0001	0x26B3	2000	0x0000A000	0x0002004C
mini_rte__startup	2	Continue (suspen...		0x0000	0x6B80	0x0000	0xE23C	2000	0x00006000	0x00020078
mini_rte__startup	3	Continue (current)		0x0000	0xEB84	0x0001	0x6235	2000	0x0000E000	0x000200A4

Problems | Properties | Tasks | AAMP7 Partition Access Rights | AAMP7 Target Monitor

Partition Name	VM #	Region#	Low Address	High Address	Source Mode	Type Mode	Execute Mode
mini_rte__startup	0	0	0x00008000	0x000098DF	TAU/Data/Code	Read/Fetch	Err/Exec/User
mini_rte__startup	0	1	0x00013000	0x0001759B	TAU/Data	Write/Read	Err/Exec/User
mini_rte__startup	1	0	0x0000A000	0x0000BED3	TAU/Data/Code	Read/Fetch	Err/Exec/User
mini_rte__startup	1	1	0x00023000	0x0002759F	TAU/Data	Write/Read	Err/Exec/User
mini_rte__startup	2	0	0x00006000	0x00007E77	TAU/Data/Code	Read/Fetch	Err/Exec/User
mini_rte__startup	2	1	0x00018000	0x0001C59F	TAU/Data	Write/Read	Err/Exec/User
mini_rte__startup	3	0	0x0000E000	0x0000FD2B	TAU/Data/Code	Read/Fetch	Err/Exec/User
mini_rte__startup	3	1	0x00010000	0x0001051B	TAU/Data/Code	Read/Fetch	Err/Exec/User
mini_rte__startup	3	2	0x00028000	0x0002C59F	TAU/Data	Write/Read	Err/Exec/User



µCryptol compiler, mcc, interpreter interaction, targeting ACL2 AAMP7G formal model


ACL2 session
Process Stack
Disassembly
Console
AAMP7G ACL2 Formal Model Integration with Eclipse AAMP7G Tools

Cryptol

- **Galois' domain-specific language for cryptography algorithms**
  http://www.cryptol.net

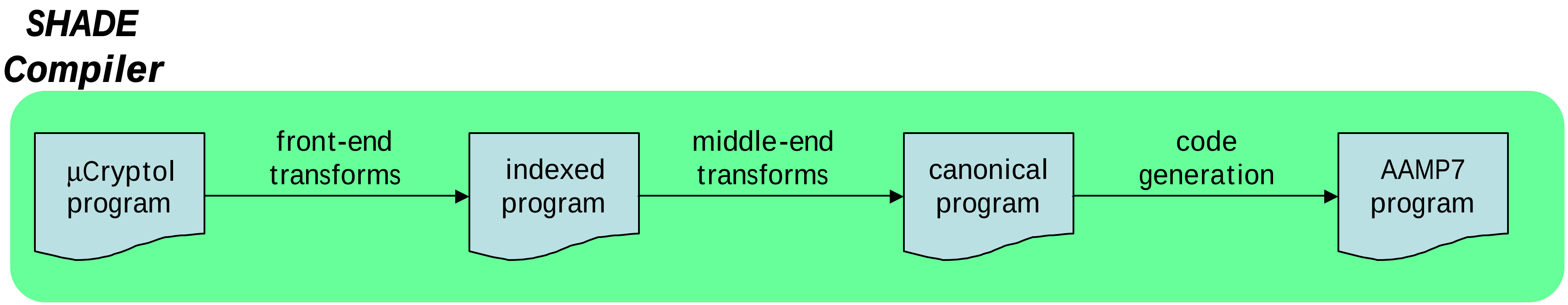
- **Cryptol features:**
  - **Purely functional**
  - **Size-indexed bitvector types, no limits on bitvector size**
  - **Lazy infinite streams**
  - ***Not* Turing-complete**

- **µCryptol**
  - **Cryptol subset, tailored for systems with constrained memory**
  - **Formal semantics**
  - **Designed for verification**
  - **Creating a verifying compiler targeting the AAMP7G**
  - **See paper in HCSS06 Proceedings**

Why a verifying compiler for µCryptol?

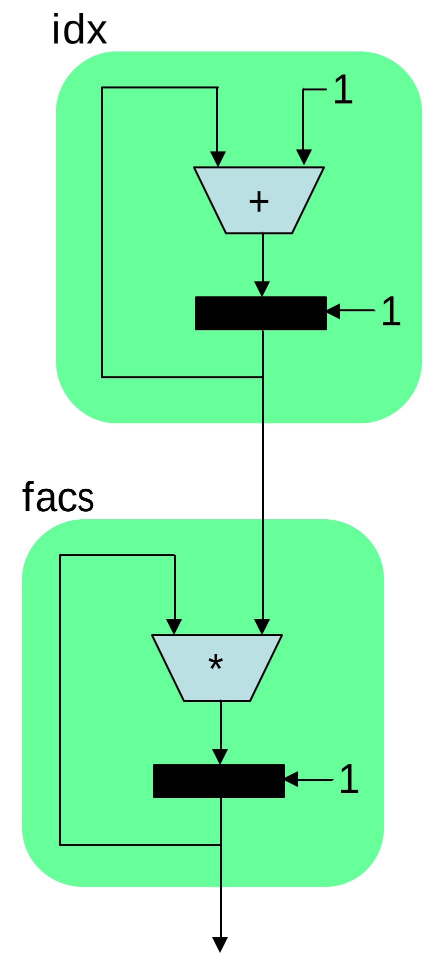
- Cryptographic systems need to be correct
  - NSA is a demanding customer
- Cryptographic systems are difficult, expensive to certify
  - A verifying compiler could markedly reduce code-to-spec review costs and reduce time-to-market for cryptographic devices
- Reference Cryptol specifications for common crypto algorithms are available
- A domain-specific language, such as Cryptol, seems to present lower risk than attempting a verifying compiler for a general-purpose programming language
- Cryptol is a Galois Connections design, so we can state its specification precisely
- The AAMP7G is an “easy” code generation target (think JVM)
- The AAMP7G is a Rockwell Collins design with a precise specification
- Theorem prover technology has matured sufficiently to make this program feasible

Compiler Architecture



Example: factorial (mod 2^8)

```
fac : B^32 -> B^8;
fac i = facs @@ i
  where {
    rec
      idx : B^8^inf;
      idx = [1] ## [x + 1 | x <- idx];
    and
      facs : B^8^inf;
      facs = [1] ## [x * y | x <- facs
                           | y <- idx];
  };
```

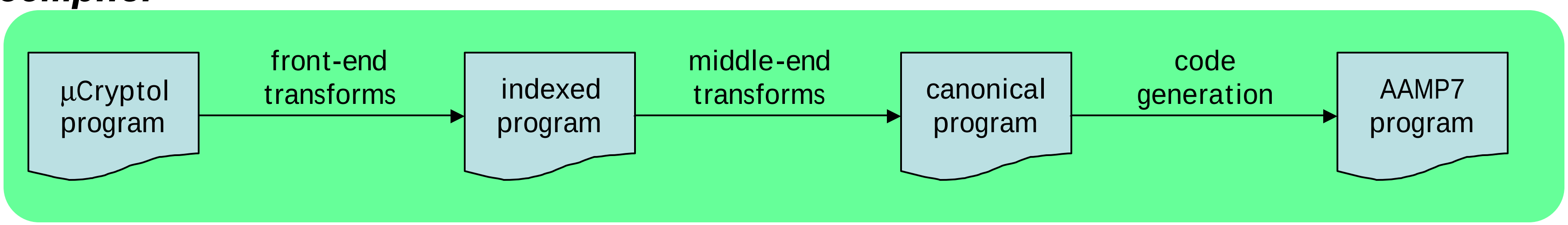


Stream values:

```
idx  = [1, 2, 3, 4,  5,   6,   7,   8, …]
facs = [1, 1, 2, 6, 24, 120, 208, 176, …]
```

Stage 1: Compile to indexed form

SHADE Compiler



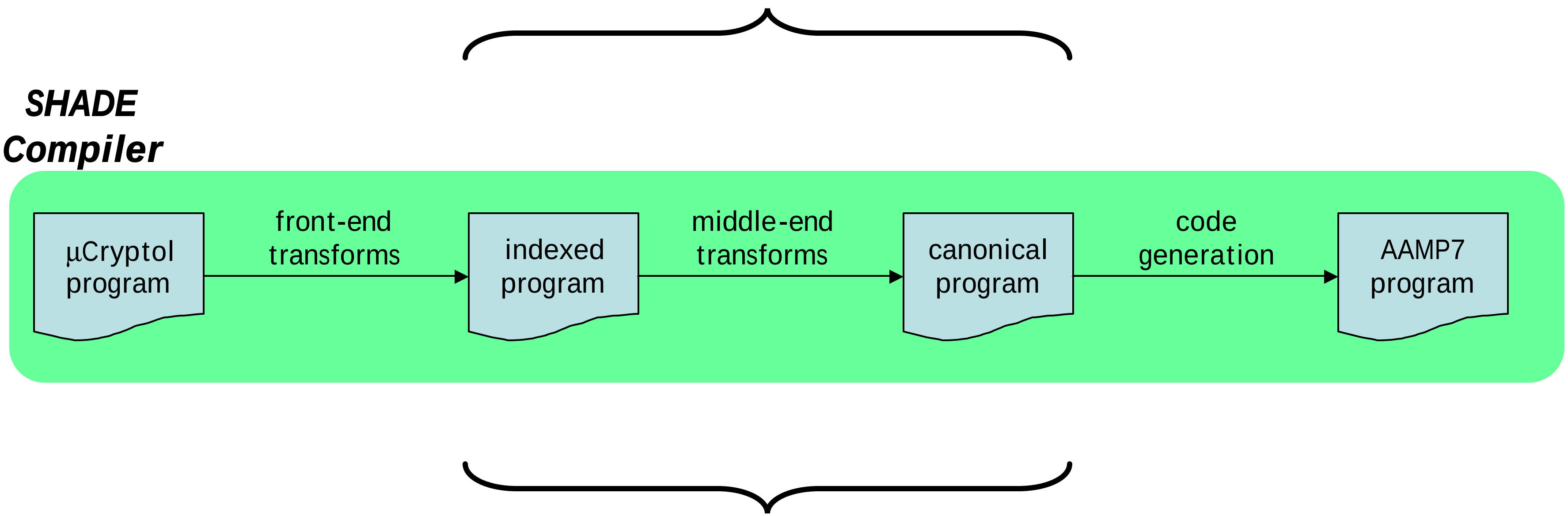
Stage 1: Compile to indexed form

- **Each stream represented as first-order function taking index to stream element**
- **Nested definitions lambda-lifted to top-level**
- **Pattern-matching and stream/vector comprehensions compiled away**
- **Program can now be shallowly embedded into ACL2**

```
idx : nat -> B^8;
idx n =  if n = 0 then 1
         else idx (n-1) + 1;

facs : nat -> B^8;
facs n =  if n = 0 then 1
          else facs (n-1) * idx (n-1);

fac : B^32 -> B^8;
fac i = facs (toNat i);
```

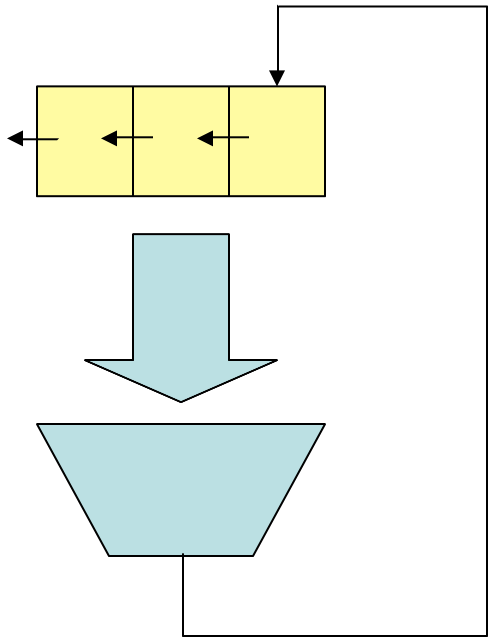
Stage 2: Compile to canonical form

SHADE Compiler



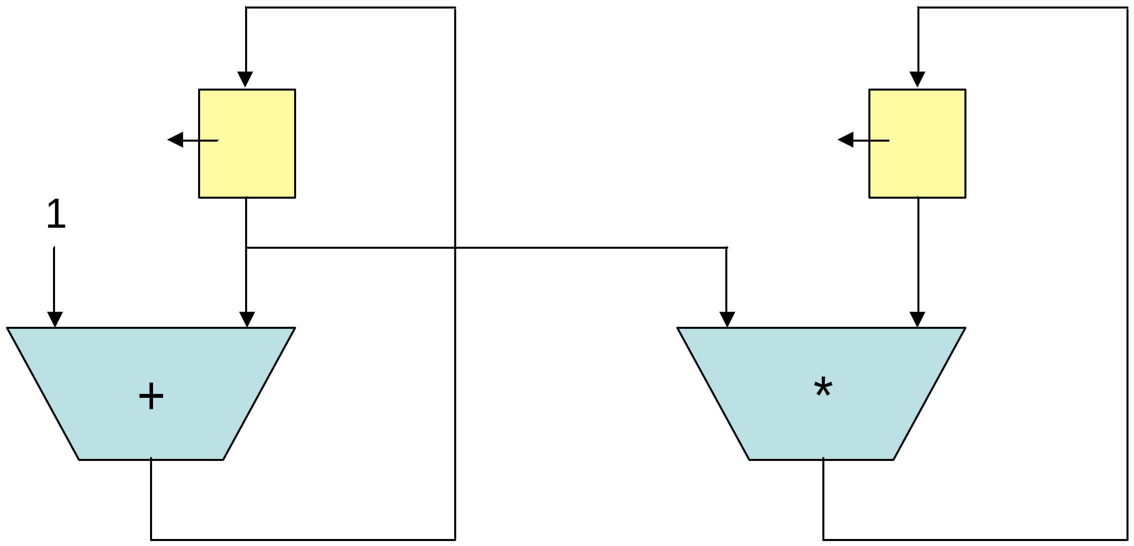
Stage 2: Compile to canonical form

- **Each clique of mutually recursive stream functions represented by single tail-recursive function**
- **Each tail-recursive function takes an extra tuple of *history buffers***
- **Stream dependency analysis calculates minimum length of each history buffer**
- **Complex Cryptol primitives left unchanged, some simple ones are inlined**



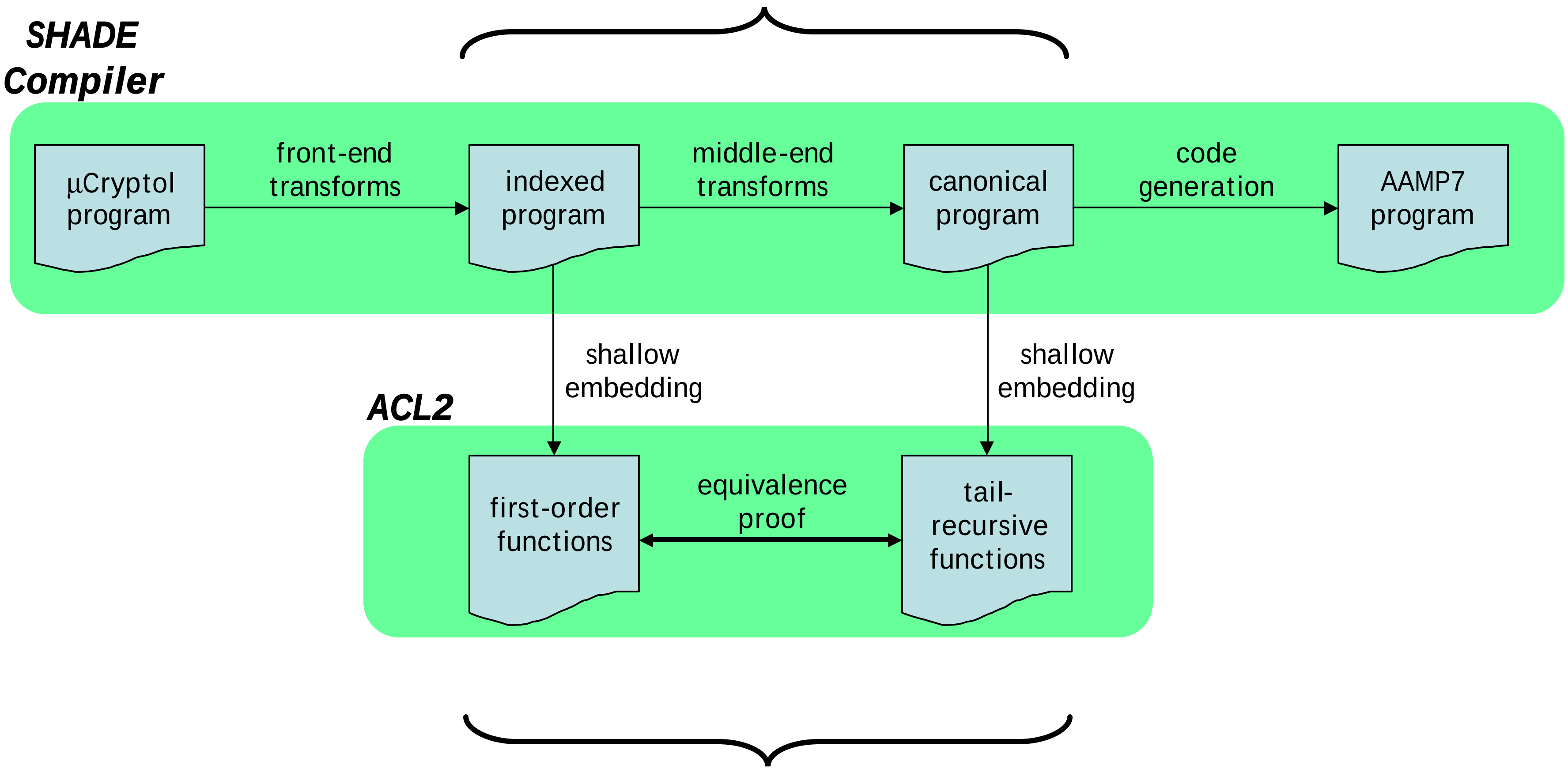
Stage 2: Compile to canonical form

- Factorial program contains two single-element history buffers
- Running time
  - Factorial in indexed form: quadratic
  - Factorial in canonical form: linear



Stage 2: Verification Architecture

- **Use ACL2 to verify compiler middle-end transformations**

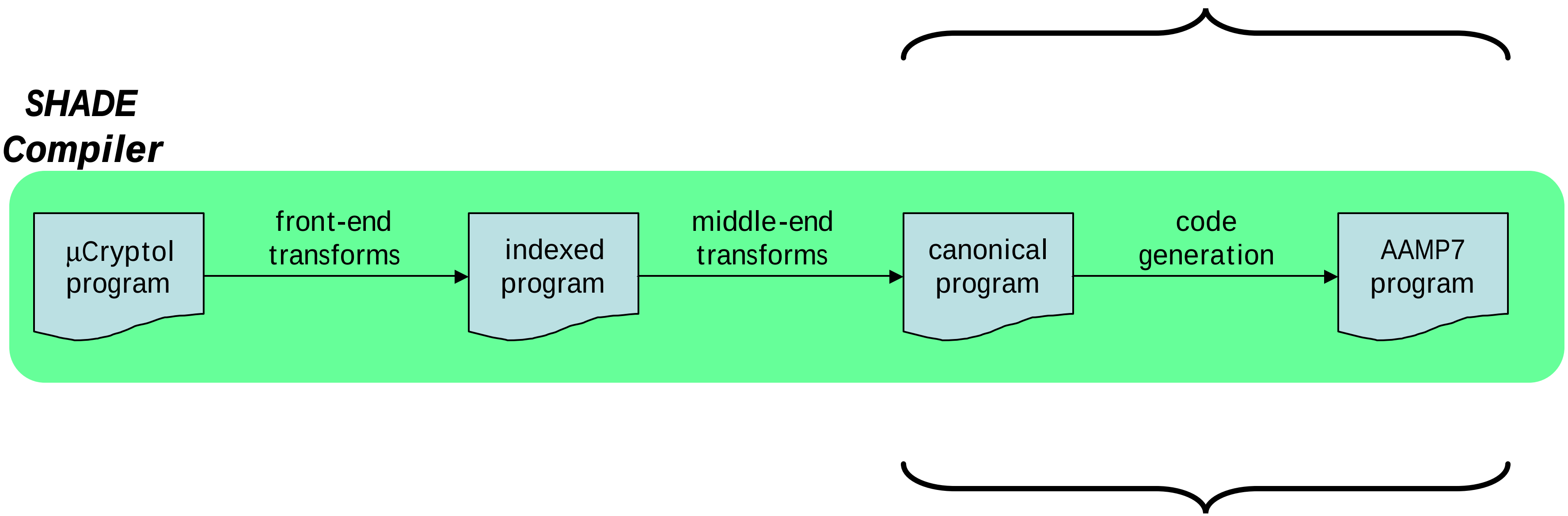


Results

- ACL2 macro that can automatically prove equivalence of indexed to canonical forms, for all examples
  - factorial, alt-factorial
  - Fibonacci, 3-Fibonacci, 5-Fibonacci
  - TEA, AES, RC6
- AES proof takes about 20 minutes on a 1.5 GHz G4 Powerbook
- See paper in HCSS06 Proceedings for more details

Stage 3: Generate machine code

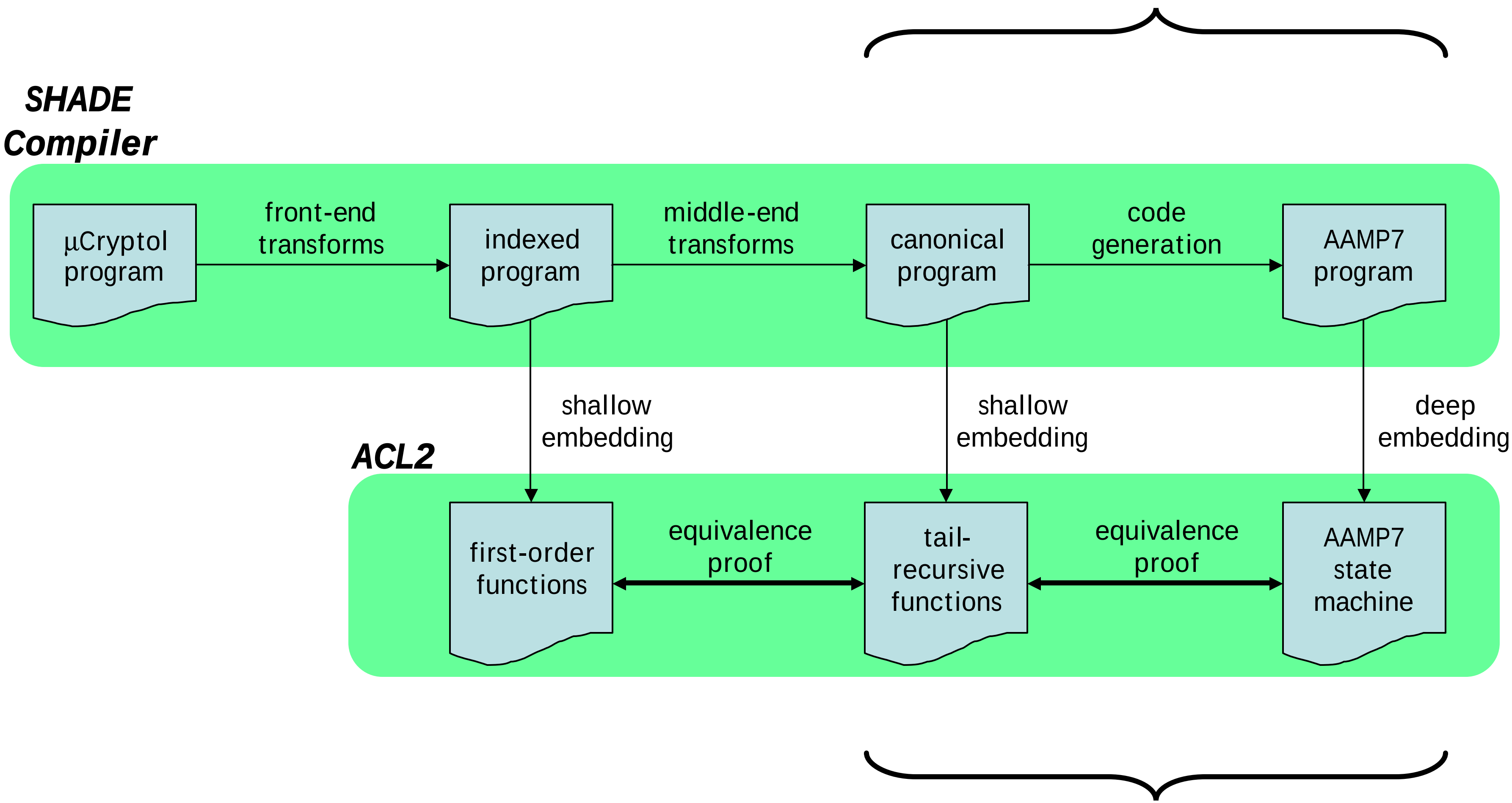
SHADE Compiler



Stage 3: Generate machine code

- **History buffers represented as circular imperative arrays**
  - **Optimized away if history length is small**
- **Compiler statically allocates history buffers**
- **Calls library routines for multiple-word Cryptol primitives such as arithmetic, shift, rotate, etc.**

Stage 3: Verification Architecture

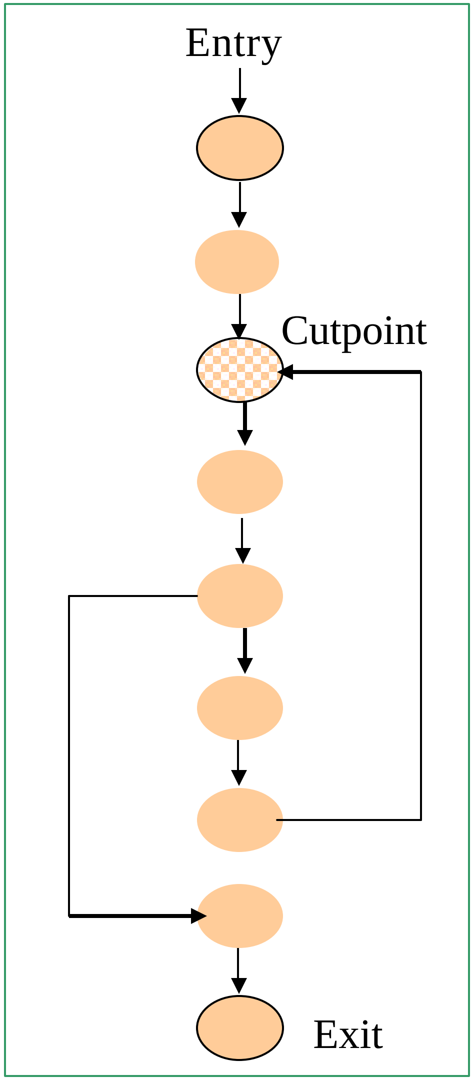


Desired Theorems (in general)

- If machine starts at a state satisfying program's precondition (entrypoint assertion), then
  - *Partial correctness*: if the machine ever reaches an exitpoint state, then the first exitpoint reached satisfies the program's postcondition (exitpoint assertion).
  - *Termination*: the machine will eventually reach an exitpoint
- However, we don't want to
  - write and verify a VCG
  - manually define a *clock function*
    - computes for each program state exactly how many steps are needed to reach the next exitpoint

Underlying Verification Method – Compositional Cutpoint Technique

- **Sound and automatic theorem proving technique for generating verification conditions from a small-step operational semantics**
- **Inspired by J Moore presentation at HCSS 2004**
- **Cutpoints and their state assertions for a given subroutine must be specified**
- **Symbolic simulation of processor model takes us from cutpoint to cutpoint, until we reach subroutine exit**
- **Compositionality: Once cutpoint proof is done for a given subroutine, we don't have to reason about it again if it's called by another subroutine**
- **No Verification Condition Generator required**
- **See *Verification Condition Generation via Theorem Proving* John Matthews, J Moore, Sandip Ray, Daron Vroon, 2006 (submitted for publication)**
- **Has been used it to verify a 600-line JVM program implementing a generic CBC-mode encryption**



AAMP7G Machine Code Proofs using Compositional Cutpoint Method

- Preconditions, e.g.
  - Code to be proved is loaded into memory
  - Input parameter is within range for a given algorithm
- Postconditions
  - e.g., fact(x) on top of stack after running AAMP7G machine code for factorial
- Frame Conditions
  - e.g., Only local variables and operand stack memory needed to implement factorial are modified by executing AAMP machine code for factorial
- Compositional Cutpoint Proof Technique
  - No Verification Condition Generator required
- Generation of the above information can be done mostly automatically

Example Program – Iterative Factorial


```
    #x02          ;; Proc Header --  2 words of locals
    #x00
;
    #x10           ;; LIT4 0
    #x11           ;; LIT4 1
; local0 is a counter counting from 1 up to local2
    #xc0           ;; ASNDL 0
;    accumulator lives on stack; initialize it to 1
    #x10           ;; LIT4 0
    #x11           ;; LIT4 1
;    L2 loop top ------------------------------------- CUTPOINT
    #x30           ;; REFDL 0
    #x32           ;; REFDL 2
;    if local0>local2, goto L
    #xa5
    #x0e           ;; GRUD
    #x5b           ;; SKIPNZI
    #x0c           ;;  L (+12)
    #x30           ;; REFDL 0
;   multiply local0 into the accumulator on the stack
    #xa5
    #x2a           ;; MPYUD
;  increment local0
    #x30           ;; REFDL 0
    #x10           ;; LIT4 0
    #x11           ;; LIT4 1
    #xa5
    #x28           ;; ADDUD
    #xc0           ;; ASNDL 0
    #x19           ;; LIT8N
    #x11           ;;  L2 (-18)
    #x59           ;; SKIP
    #x14           ;; LIT4 4
    #x5f           ;; RETURN
```

Machine Code Proofs – Preconditions Example


```
(defund f-precondition (s)
  (declare (xargs :non-executable t))
  (and
      (equal (starting-program-counter) 131394 ;(+ 2 (iter-fact-address)))
      (not (equal 4294967295 (aamp::read-two-local-words 2 s)))
      ;argument isn't the largest int...

      (< (aamp::aamp.lenv s)
         (gacc::read-data-word (aamp::aamp.denvr s) (+ -1 (aamp::aamp.lenv s)) (aamp::aamp.ram s)))

      ;;the operand stack should be empty just after fact is called
      ;;(the argument is passed in as a "local")
      (equal 0 (aamp::stack-height s))

      (aamp::st-p s)
      (aamp::aamp-normal-statep s)
      (aamp::no-code-data-clash (aamp::aamp.cenvr s) (aamp::aamp.denvr s))

      ;;factorial code is loaded where we expect it to be.  Since the program
      ;;begins with two bytes of header, the code actually starts 2 bytes
      ;;before the first instruction.
      (iter-factorial-program-loaded (+ -2 (starting-program-counter)) ;(nth *aamp.pc* st)
                                     (aamp::aamp.ram s))

      (aamp::code-fetches-allowed 100 #x0002 #x0140 (nth 17 s))
      (AAMP::DATA-WRITES-ALLOWED 65536 (NTH 2 S) 0 (NTH 17 S))
      (AAMP::DATA-READS-ALLOWED 65536 (NTH 2 S) 0 (NTH 17 S))))
```

Machine Code Proofs – Postconditions Example

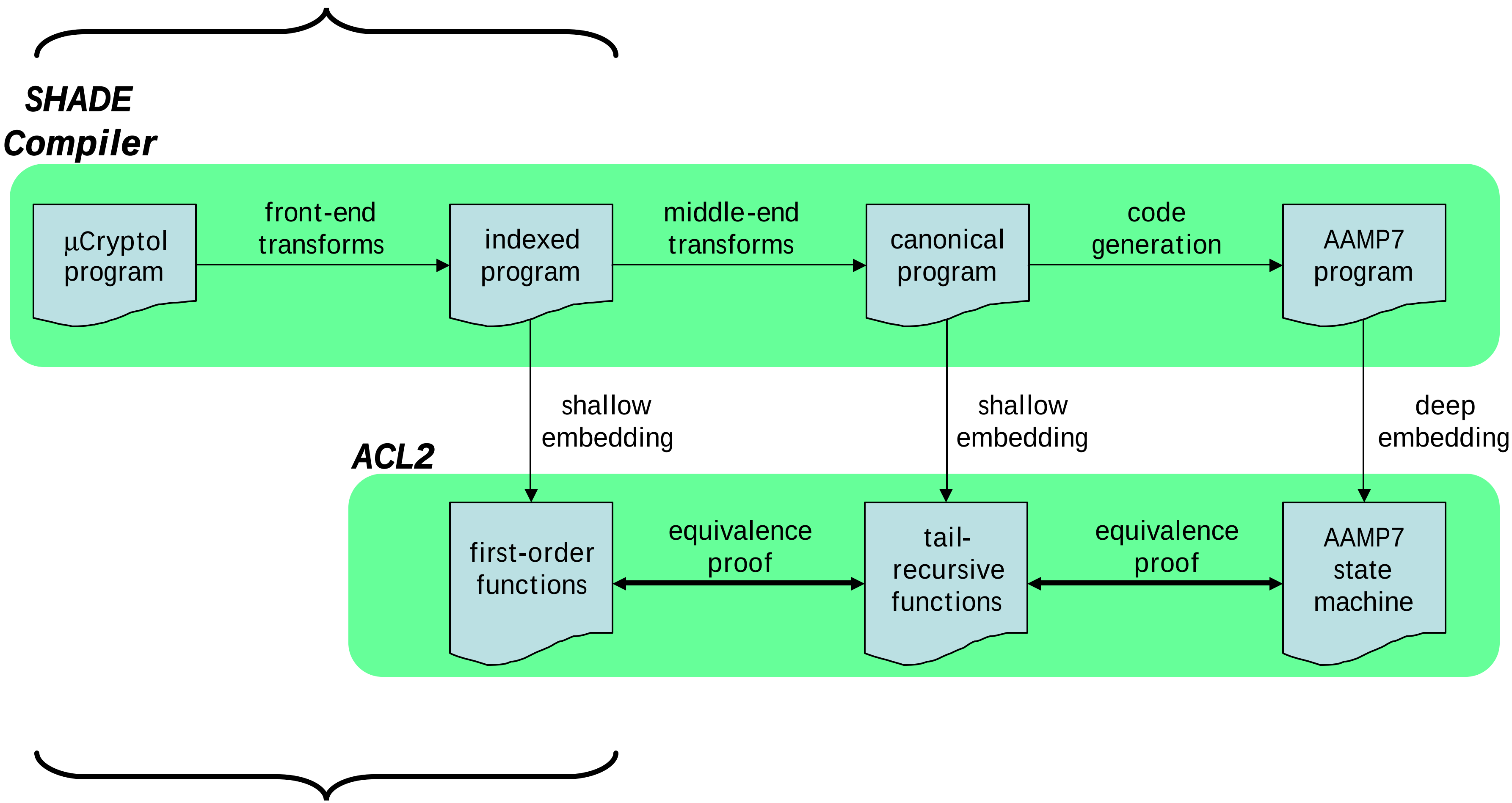

```
(defund f-poststate (s0 s)
  (declare (xargs :non-executable t))
  (aamp::modify s0

              ;;the 4-word stack mark gets popped off along with 4 words of
              ;;args/locals, then two words of RV get pushed on, so the
              ;;stack shrinks by 6.
              :tos (aamp::inc-tos 6 s0)

              :pc (get-saved-callers-pc s0)
              :lenv (get-saved-callers-lenv s0)
              :cenvr (get-saved-callers-cenv s0)

              ;; Memory access temporaries – artifact of the applicative model
              :memtmp (loghead 16 (aamp::aamp.memtmp s))
              :memtmp8 (loghead 8 (aamp::aamp.memtmp8 s))
              :memtmp32 (logext 32 (aamp::aamp.memtmp32 s))
```

<<continued on next slide>>

Machine Code Proofs – Postconditions Example (cont'd.)

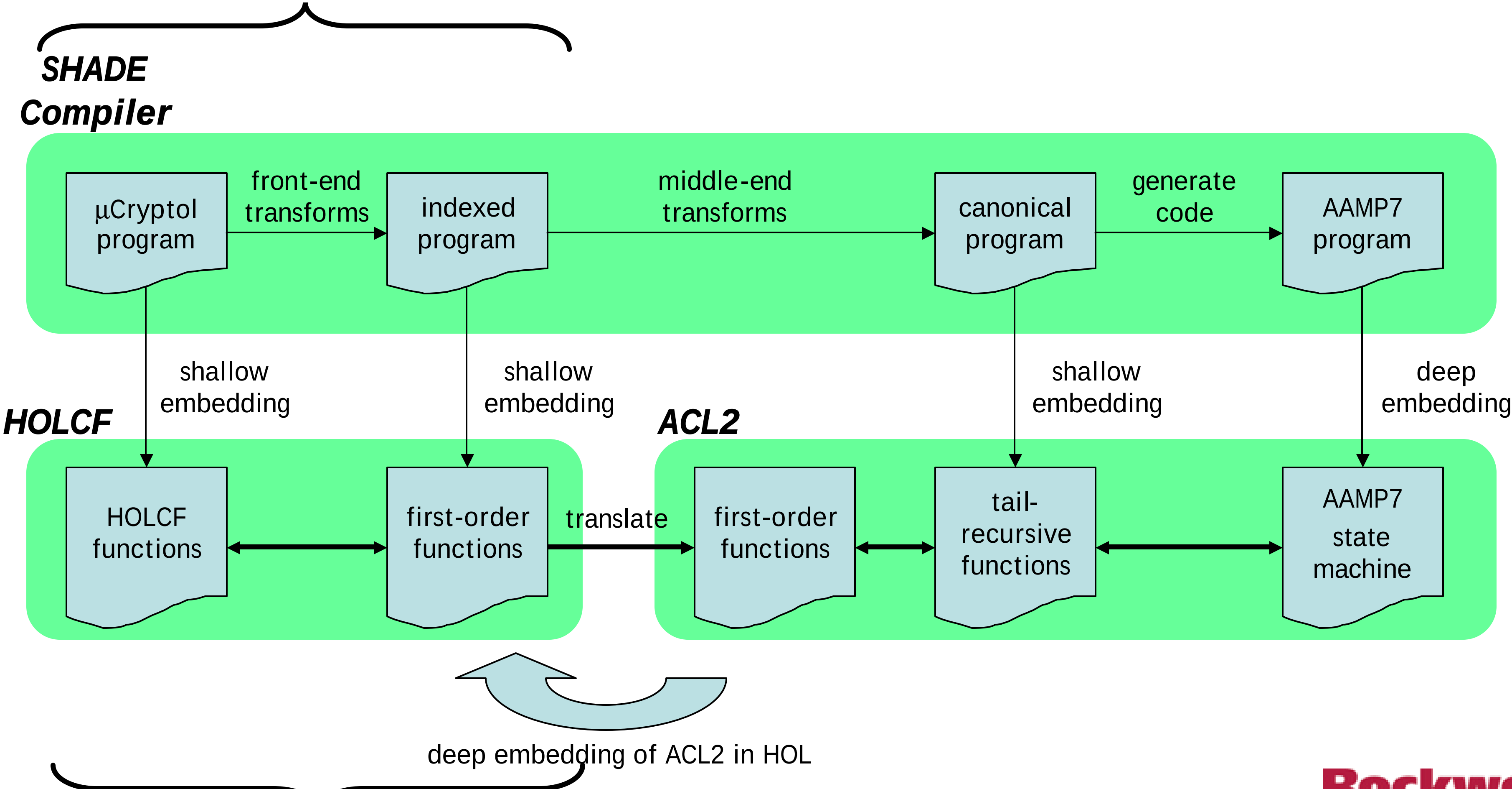

```
:ram (gacc::write-data-words
                2
                (aamp::aamp.denvr s0)
                (+ 6 (aamp::aamp.tos s0))

                ;;the mathematical factorial of the argument:
                (fact
                 (gacc::read-data-words 2 (aamp::aamp.denvr s0) (+ 2 (aamp::aamp.lenv s0))
                                        (aamp::aamp.ram s0)))

                ;;This says we are allowed to make a mess of the entire
                ;;stack.  Restrict this to just the amount used
                ;;(determined from the argument to factorial).

                (copy-over-n-words 12
                         (aamp::aamp.denvr s0)
                         (+ -10 (aamp::aamp.lenv s0))
                         (aamp::aamp.ram s)

                         (aamp::aamp.denvr s0)
                         (+ -10 (aamp::aamp.lenv s0))  ;write starting at address 0
                         (aamp::aamp.ram s0)))))
```

Issue: Verifying compiler front-end



Extended Verification Architecture

- **Use Isabelle/HOLCF to verify front-end compiler transformations**



Translating ACL2 to Isabelle

- **Mike Gordon, Matt Kaufmann, Warren Hunt and James Reynolds are building an ACL2 external oracle for HOL4**
  - Defined ACL2 universe as sexp datatype in HOL
  - Used ACL2 axioms to define ACL2 primitives in HOL
  - Expressions and formulas defined over sexp invoke ACL2
  - Result of ACL2 is trusted
- **John Matthews has developed a prototype sexp datatype for Isabelle, and proved equivalence between a shallow embedding of the μCryptol factorial example into Isabelle/HOLCF and a translated version of indexed form using the sexp datatype**

Bonus Proof!

- **As an extra credit assignment, John Matthews recently proved that the shallow embedding of the µCryptol factorial example into Isabelle/HOLCF provides mathematical factorial, mod 2^8**

```
lemma fac_math_fac_ind:
  “fac$(Def n) = Def (fac_math n mod 2^8)
   /\ ind_idx$(Def n) = Def ((n + 1) mod 2^8)”
```

Summary

Rockwell Collins and partners have developed robust techniques and tools to improve high-assurance system evaluations by:

- Making use of automated theorem provers to provide formal proofs as required by EAL 7
- Producing executable formal models of computing platforms that can also be validated by execution of production tests
- Pioneering techniques for automating hardware, microcode, and software verification
- Designing and implementing a verifying compiler for a subset of the Cryptol language