A Robust Machine Code Proof Framework for Highly Secure Applications

David Hardin

Advanced Technology Center
Rockwell Collins

Eric Smith
Stanford University

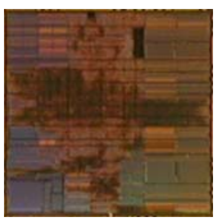
Bill Young
University of Texas at Austin

UNCLASSIFIED

Overview

- Rockwell Collins Introduction
- AAMP7G Microprocessor
  - MILS Certification
- SHADE Program
  - AAMP7G tools
  - Microcryptol Verifying Compiler
  - AAMP7G Instruction Set Formal Model
  - 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

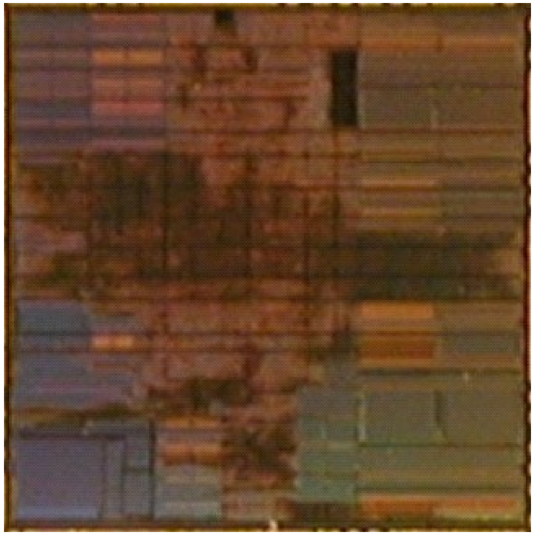
The Problem – High-Assurance for Security Applications

- 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

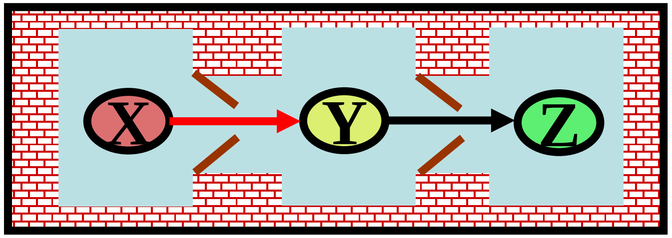
- Developed by RCI Advanced Technology Center
- Used in RCI GPS and Information Assurance products
- High Code Density
- Low Power Consumption (250 mW)
- 100 MHz operation
- Screened for full military temp range
- Implements *intrinsic partitioning*

AAMP7 die



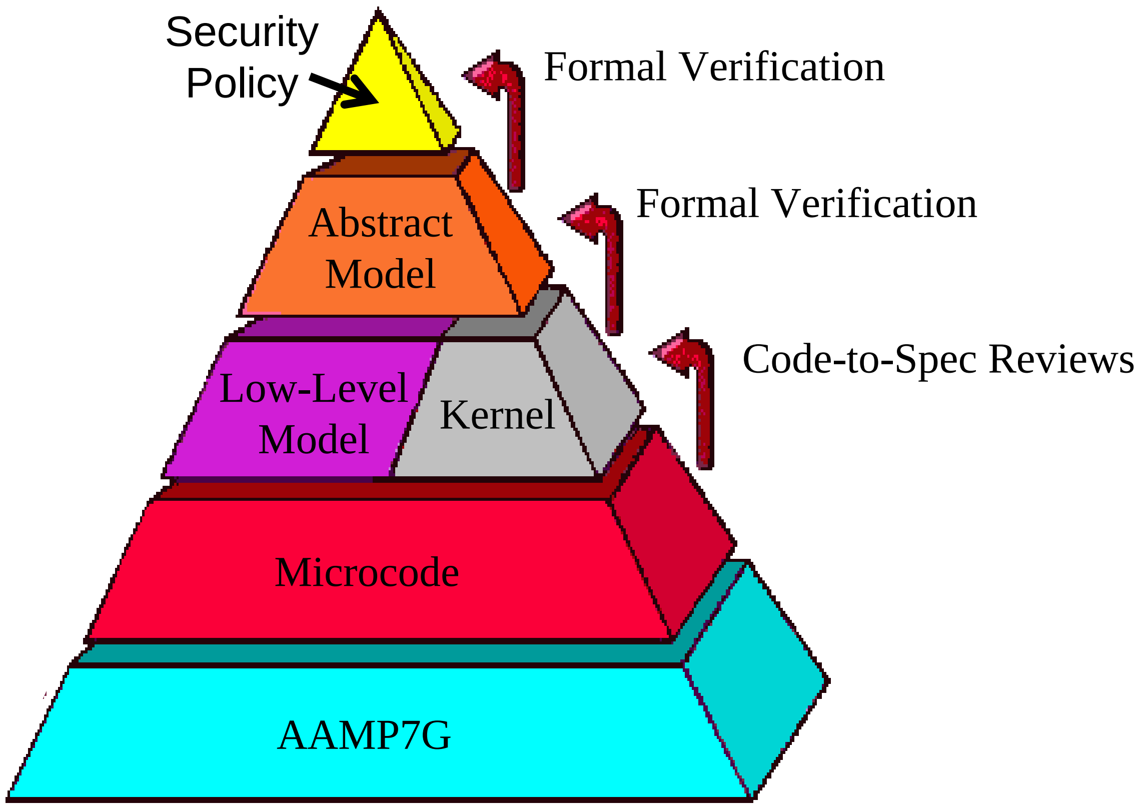
Intrinsic partitioning

- Computing Platform Enforces Data Isolation
- "Separation Kernel in Hardware"



AAMP7G Formal Verification

Common Criteria
EAL7 Proof Obligations

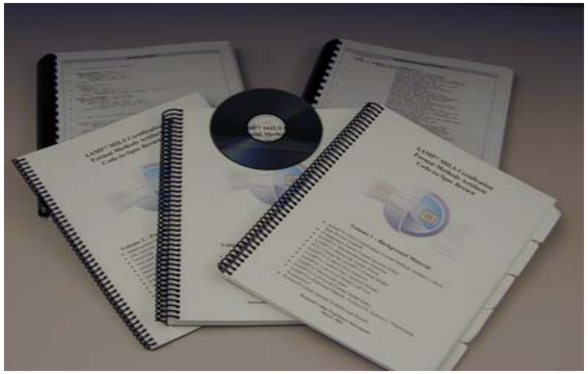


AAMP7G Intrinsic Partitioning Formal Verification

ADVANCED COMPUTING SYSTEMS

<u>Program Accomplishments</u>

- **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



National Security Agency

Information Assurance Directorate

Information Assurance Directorate
Approval Certificate

The NSA certifies the Advanced Architecture MicroProcessor 7 Government Version (AAMP7G) as a Multiple Independent Levels of Security (MILS) device capable of simultaneously processing unclassified through Top Secret Codeword information, as well as performing JOSEKI decrypt and Type 1 signature verification. The MILS capability of the AAMP7G has been verified using Formal Methods techniques as specified by the EAL-7 level of the Common Criteria. This Certification applies to the AAMP7G which consists of the AAMP7 microprocessor ASIC loaded with the verified boot code that performs the cryptographic functions. This ASIC/boot code combination is packaged in three forms, Rockwell Collins' part numbers 984-3199-001, 984-3200-001 and 984-3201-001.

This certificate allows the use of the AAMP7G in cryptographic applications. The use of the AAMP7G in a system must be accomplished in accordance with the Security Features Users Guide for AAMP7G, 0N686613.

NSA does not make, by virtue of this certification, any statement or other representation regarding the use of the security features of the AAMP7G within a system context. The decision to use the AAMP7G must be made in a total system context employing an Information System Security Engineering process.

Daniel G. Wolf
Information Assurance Director

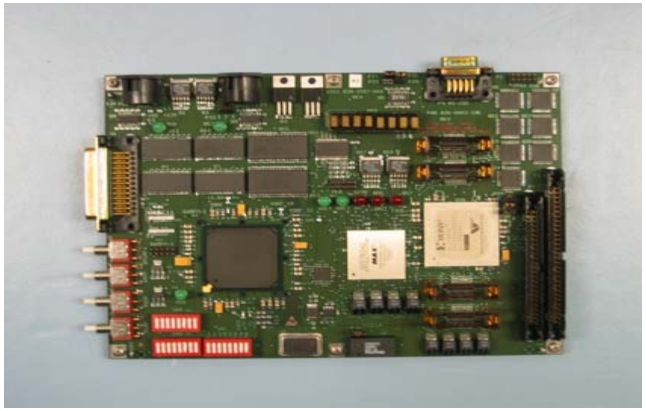
9 May 2005
Date

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

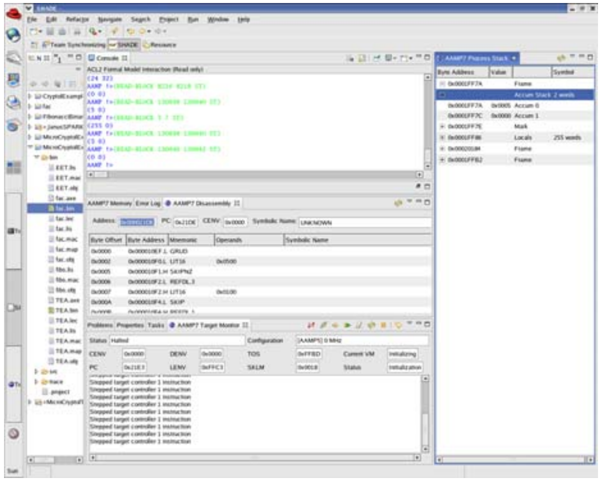
Secure, High Assurance Development Environment (SHADE)

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.**



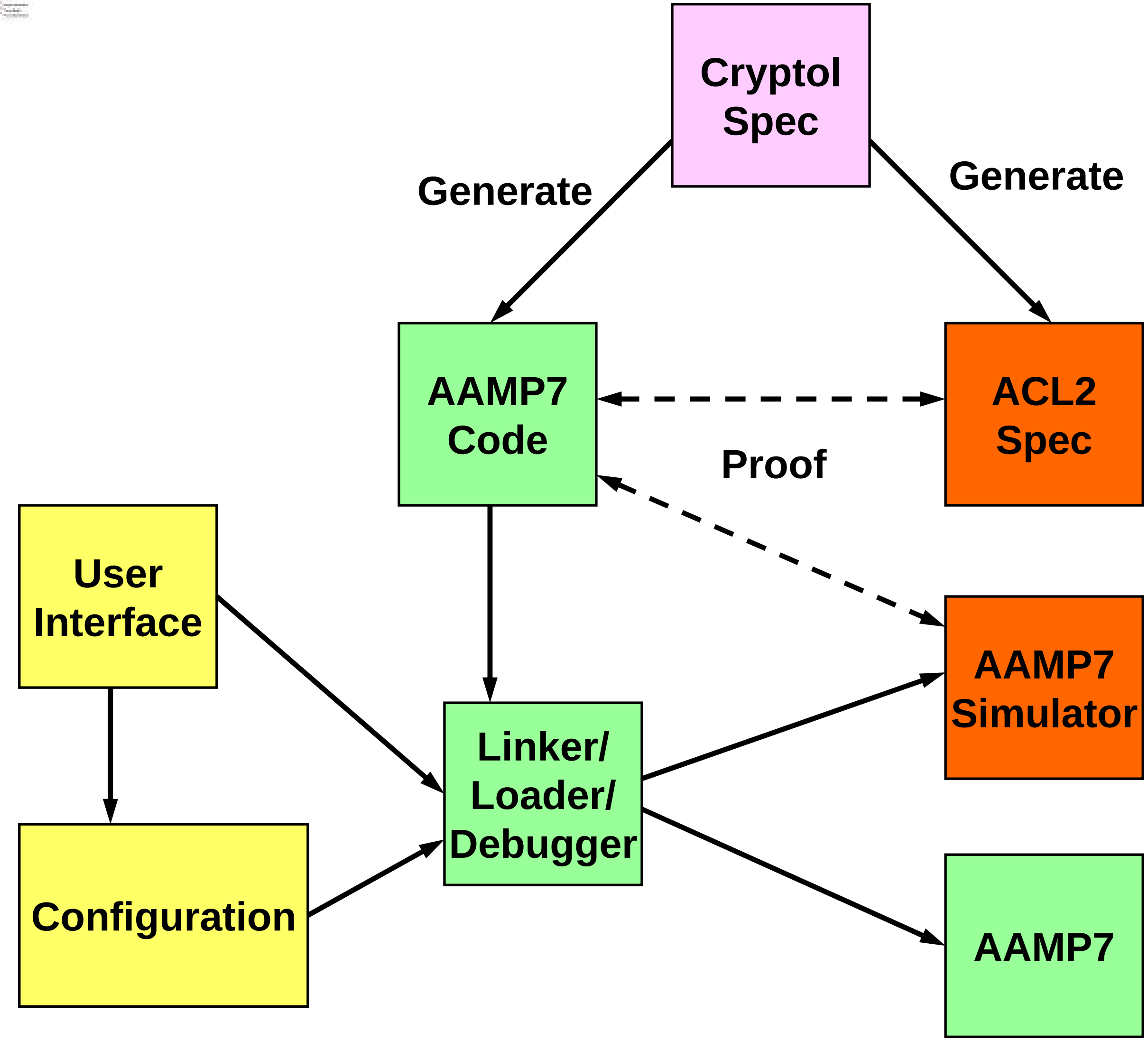
AAMP7G development board



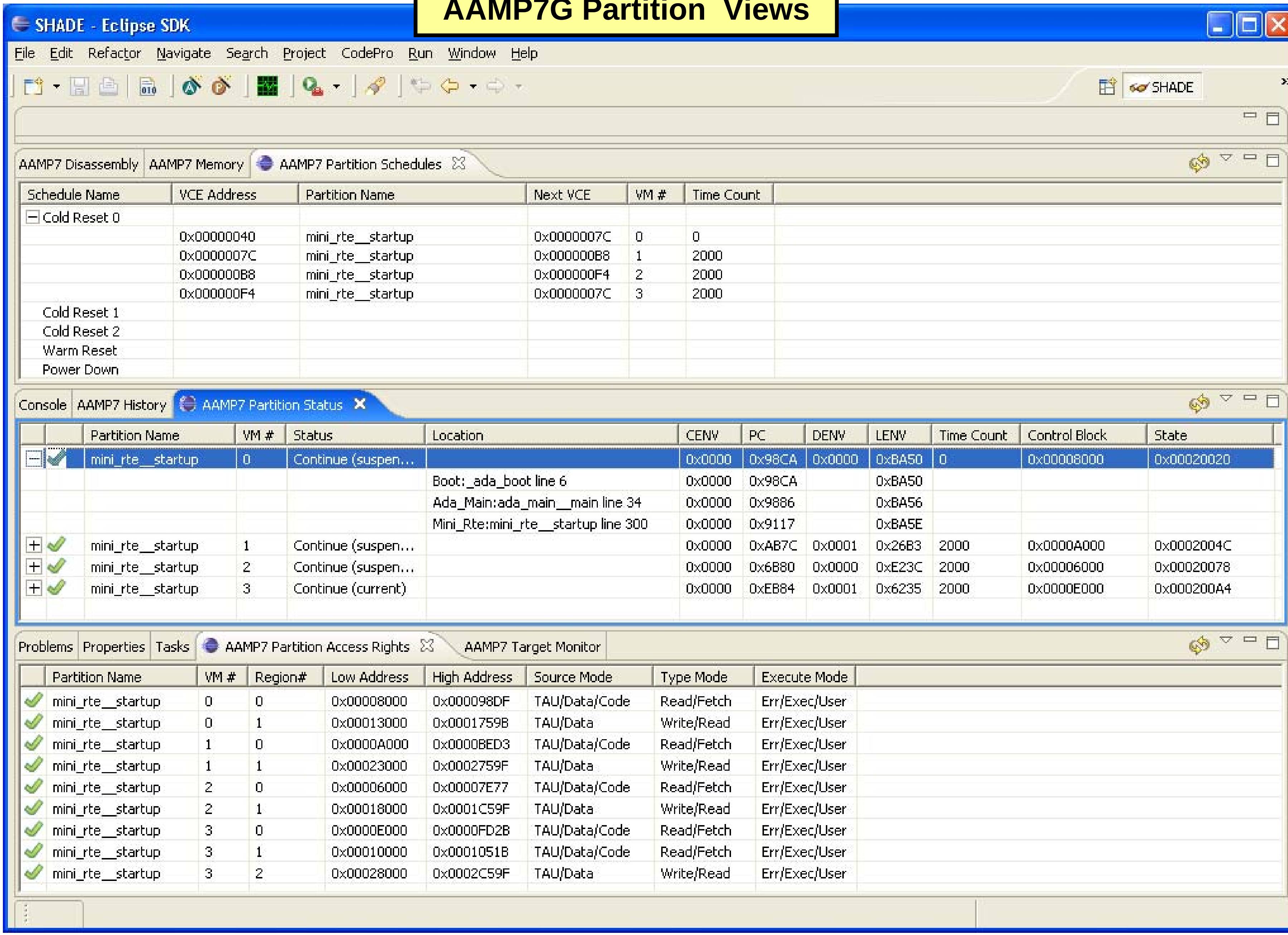
Eclipse-based AAMP7G development environment

RCI subcontractors: Galois Connections, University of Texas at Austin

SHADE Summary



AAMP7G Partition Views



AAMP7 Partition Schedules

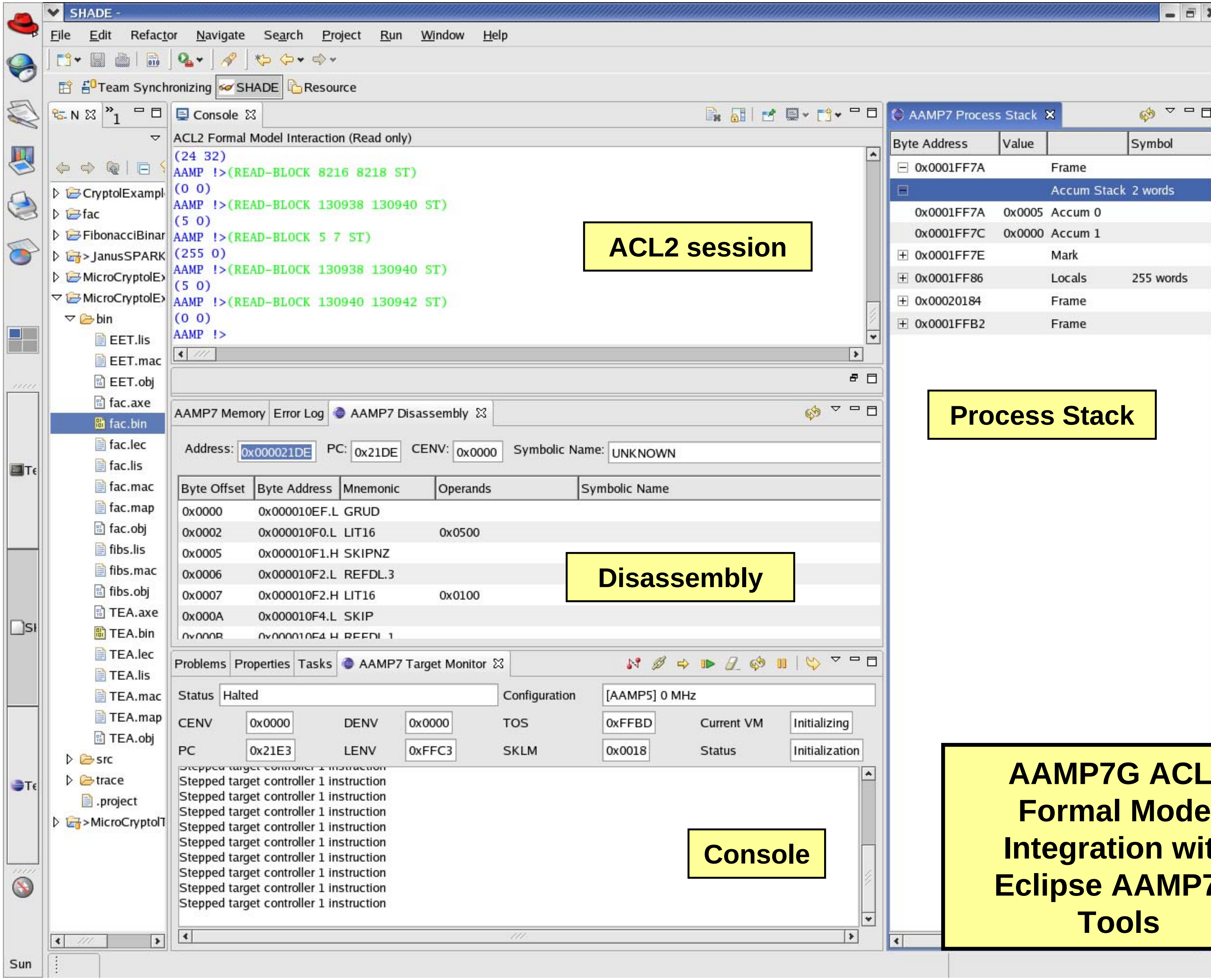
Schedule Name	VCE Address	Partition Name	Next VCE	VM #	Time Count
⊟ Cold Reset 0					
	0x00000040	mini_rte__startup	0x0000007C	0	0
	0x0000007C	mini_rte__startup	0x000000B8	1	2000
	0x000000B8	mini_rte__startup	0x000000F4	2	2000
	0x000000F4	mini_rte__startup	0x0000007C	3	2000
Cold Reset 1					
Cold Reset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

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

AAMP7 Partition Access Rights

	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



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**

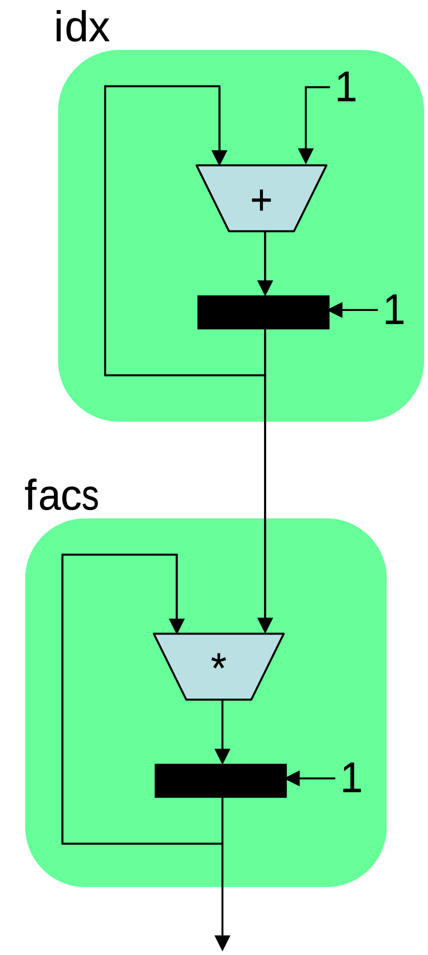
- **µCryptol**
  - **Cryptol subset, tailored for systems with constrained memory**
  - **Formal semantics**
  - **Designed for verification**
  - **Creating a verifying compiler targeting the AAMP7G**
  - **See paper in HCSS06 Proceedings**

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

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];
  };
```



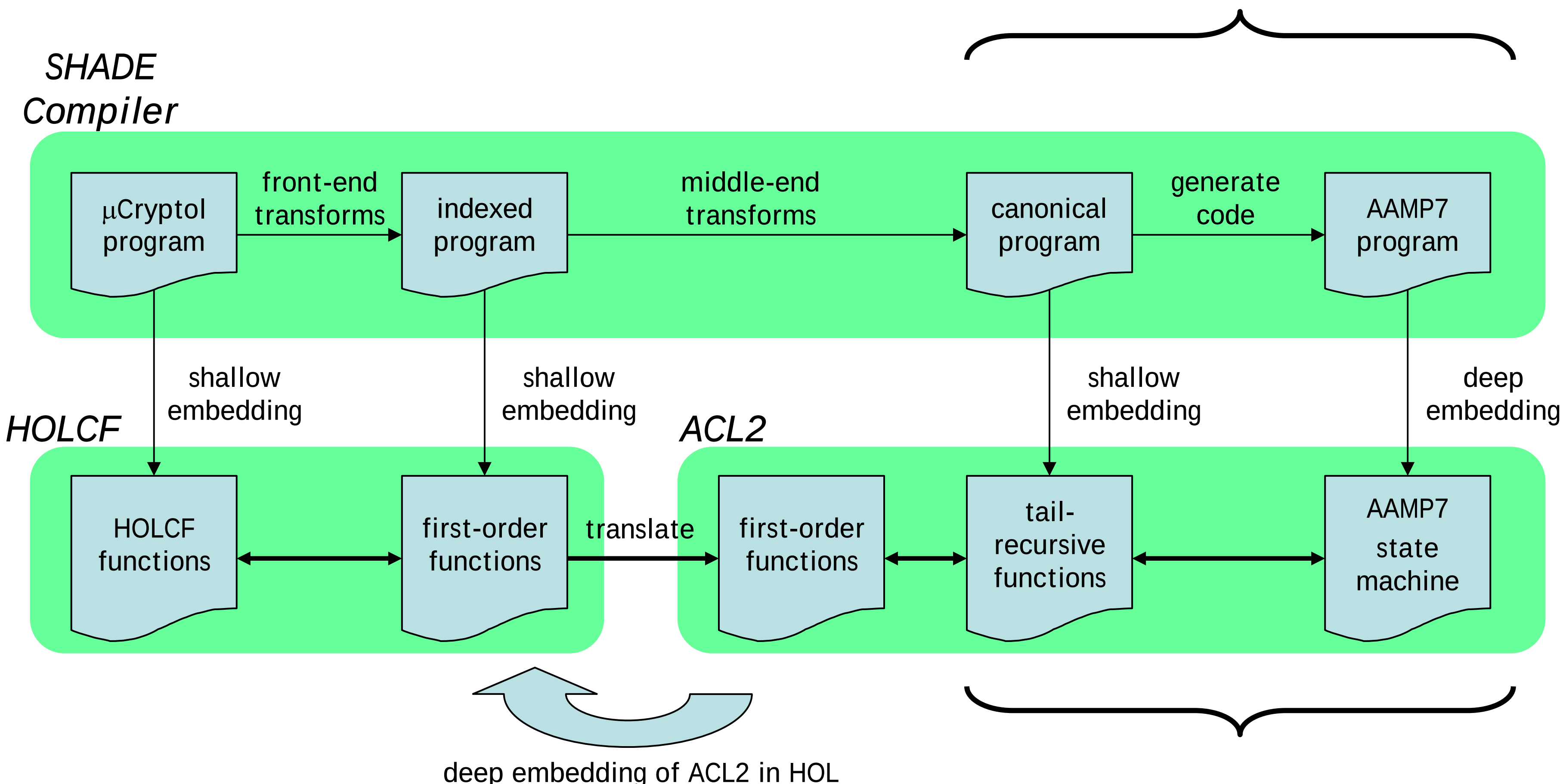
Stream values:

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

Extended Verification Architecture

ADVANCED COMPUTING SYSTEMS

Focus of this talk



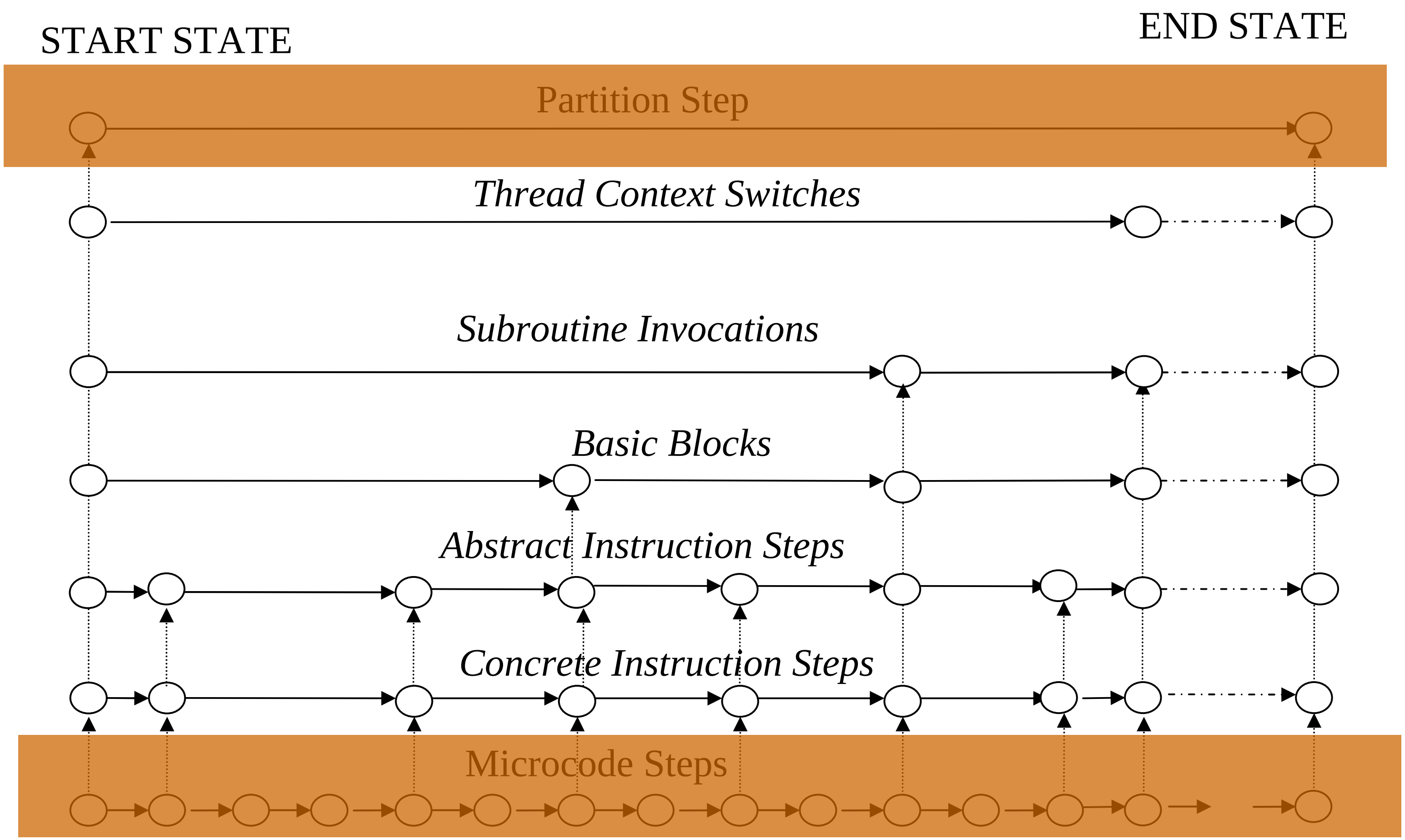
Machine code proofs

- **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**

AAMP7G Instruction-Set Formal Model

- **Provides instruction-level simulator for the AAMP7**
- **Written in ACL2**
  - ***~100 KSLOC with all RCI support books***
  - ***~500 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**
- **Models complex instruction set, including exception handling, trap handling, thread context switching, floating point, etc.**

Layers in the AAMP7G instruction-level model



Instruction Abstraction

- **Concrete instruction set level similar to microcode implementation**
- **Abstract level models the overall effect of executing the instruction without necessarily modeling every microstep, e.g.:**

```
(defun vm-addu-expected-result (st)
  (modify st
    :pc (inc-pc 1 st)
    :tos (inc-tos 1 st)
    :memtmp8 *addu-opcode*
    :memtmp (get-stack-word 1 st)
    :ram (modify-ram st :stack-word 1 (+ (get-stack-word 0 st)
                                         (get-stack-word 1 st))
    )))
```

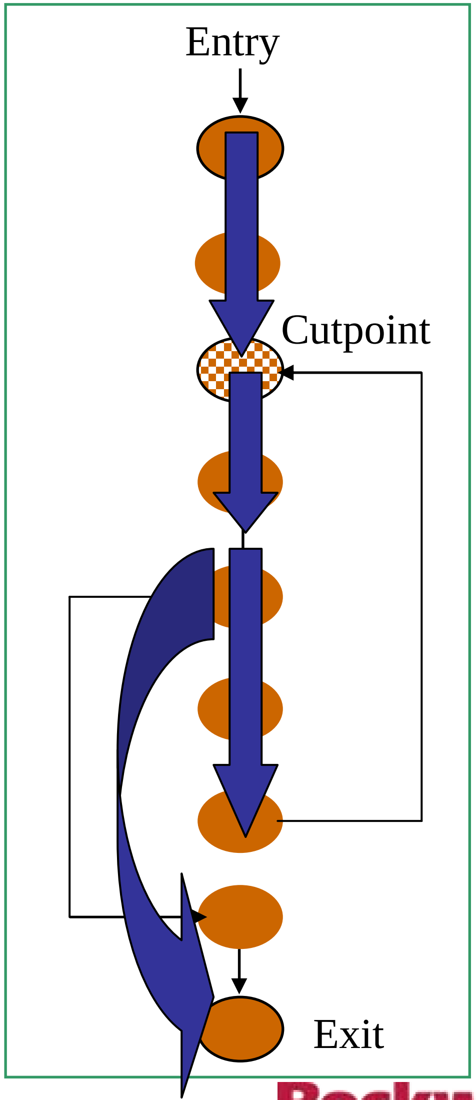
We couldn't have done this 10 years ago…

ADVANCED COMPUTING SYSTEMS

- **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**
- **Underlying memory implementation now uses Jared Davis' fast memories, described at this workshop**
  - **Results in 20x speedup on short simulation runs; higher on longer runs**
  - **4000 instructions/sec simulating complex instruction set with simulated memory management unit**
- **New bitvector library, "super-ihs", extends ACL2 Integer Hardware Specification (IHS) library**
- **We make extensive use of David Greve's Parameterized Congruences ("nary"), also described at this workshop**
- **Partial correctness technique depends on defpun, first discussed by Manolios and Moore in 2000**

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 (LPAR'06, to appear)
- 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**
- **See paper in Proceedings for more details**

Example Program – Iterative Factorial


```
     #x04             ;; Proc Header --
     #x00             ;; 4 words of locals
;
     #x10              ;; LIT4 0
     #x11              ;; LIT4 1
     #xc0              ;; ASNDL 0 --- local0 is a counter from 1 up to N
     #x10              ;; LIT4 0  --- local2 is initialized to 1
     #x11              ;; LIT4 1
     #xc2              ;; ASNDL 2
; L2: loop top -------------------- CUTPOINT
     #x30              ;; REFDL 0
     #x34              ;; REFDL 4
; if local0 > N, goto L
     #xa5
     #x0e              ;; GRUD
     #x5b              ;; SKIPNZI
     #x0e              ;;  L (+14)
     #x30              ;; REFDL 0
     #x32              ;; REFDL 2
     #xa5
     #x2a              ;; MPYUD
     #xc2              ;; ASNDL 2 --- local2 = local2 * local0
     #x30              ;; REFDL 0
     #x10              ;; LIT4 0
     #x11              ;; LIT4 1
     #xa5
     #x28              ;; ADDUD
     #xc0              ;; ASNDL 0 --- increment local0
; go to L2
     #x19              ;; LIT8N
     #x13              ;;  L2 (-20)
     #x59              ;; SKIP
; L: return local2
     #x32              ;; REFDL 2
     #x16              ;; LIT4 6
     #x5f              ;; RETURN
```

Machine Code Proofs – Preconditions Example


```
(defun fact-iter-max-words-of-operand-stack () (declare (xargs :guard t)) 4)
;from analysis of the code

(defund fact-iter-precondition (s)
   (declare (xargs :non-executable t))
   (and (standard-precondition (fact-iter-address)
                               (fact-iter-code)
                               (fact-iter-max-words-of-operand-stack)
                               s)
       ;; The routine doesn't work if the argument is the maximum 32-bit
       ;; unsigned value, since in that case the loop never terminates:
       (not (equal 4294967295 (aamp::read-two-local-words 4 s)))))
```

Machine Code Proofs – Postconditions Example


```
;; Factorial, defined in the traditional recursive style
(defun fact (n)
  (if (zp n) 1
    (* n (fact (1- n)))))


(defun fact-iter-words-of-locals-and-args () (declare (xargs :guard t)) 6)
;from dealloc count pushed just before return

(defun fact-iter-words-of-return-values () (declare (xargs :guard t)) 2)
;from height of operand stack just before return


(defun fact-iter-poststate (s0 s)
    (declare (xargs :non-executable t))
    (standard-poststate ((0 ;; top return value
                          2 ;; takes up 2 words
                          ;;the mathematical factorial of the argument:
                          (fact (gacc::read-data-words 2 (aamp::aamp.denvr s0)
                                                         (+ 4 (aamp::aamp.lenv s0))
                                                         (aamp::aamp.ram s0)))
                          ))
                        (fact-iter-max-words-of-operand-stack)
                        (fact-iter-words-of-locals-and-args)
                        (fact-iter-words-of-return-values)
                        s0
                        s))
```

Machine Code Proofs – Assertions at Cutpoint


```
(prove-it  ;; Proof driver macro
  fact-iter ;the name of the routine
  :wormhole t
  :subroutine-calls nil ;makes for faster proofs
  :user-cutpoints
  ;; List of (PC byte offset . assertion) pairs
  ((6 . (and
          ;; First comes an equality claim about the current state, s,
          ;; in terms of the initial state, s0.
          (equal s
              (standard-cutpoint-state
                  :pc 6
                  :locals (
                      (4 2 (aamp::read-two-local-words 4 s0))
                      (2 2 (fact (+ -1 (gacc::read-data-words 2
                                         (aamp::aamp.denvr s0)
                                         (aamp::aamp.lenv s0)
                                         (aamp::aamp.ram s))))))))

         ;;  Precondition still holds (e.g., code has not been modified)
         (fact-iter-precondition s0)

         ;; Asserts that the loop counter at local slot 0 is at most one more
         ;; than the input argument, N (accessed on the AAMP stack at local slot 4)
         (<= (aamp::read-two-local-words 0 S)
             (+ 1 (aamp::read-two-local-words 4 S)))

         ;; Asserts that the loop counter is positive (it starts at 1 and goes upward).
         (< 0 (aamp::read-two-local-words 0 S))))) <hints elided>)
```

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 EAL7**
- **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**
  - **Currently completing first end-to-end equivalence proofs for a simple μCryptol program**