# Modeling Algorithms in SystemC and ACL2

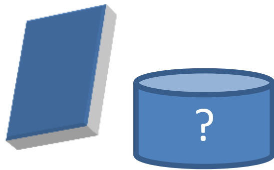
John O'Leary, David Russinoff

Intel Corporation

# Algorithm Design

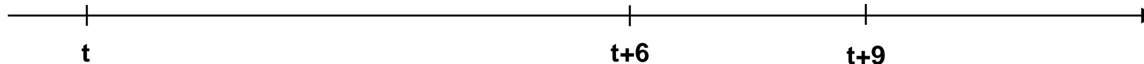
Architects

Designers



DC
PrimeTime
...

Forte
Jasper Gold
...



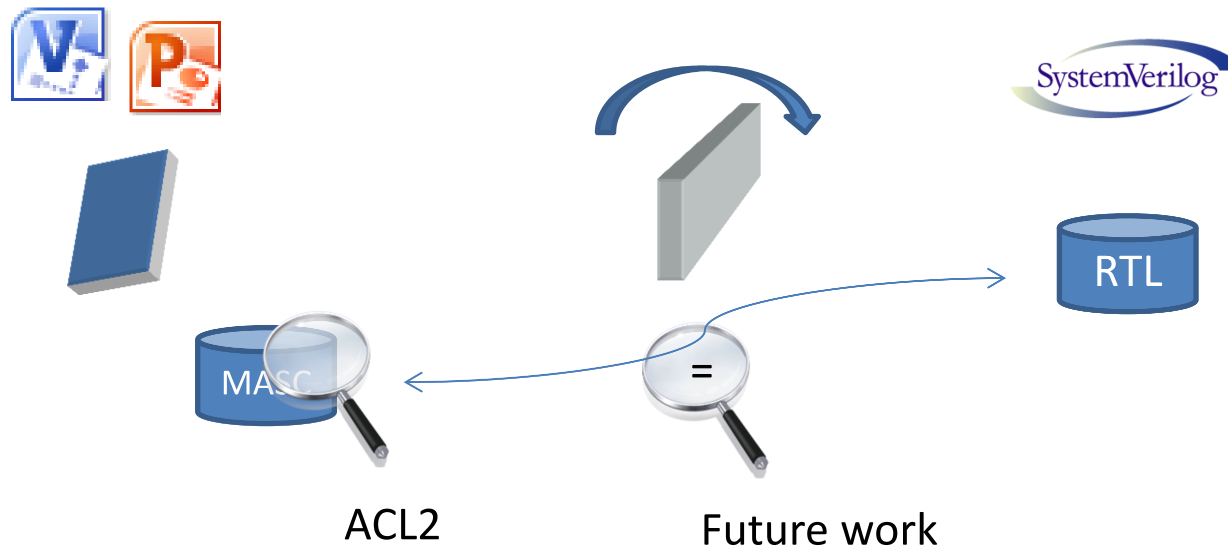
# A recent experience

- A design implements a new square root algorithm
  - High performance
  - High complexity
  - High risk
- Architect provided a pen-and-paper argument for correctness (and an executable model in C++!)
- Development of RTL + validation environment required a substantial investment of design & validation resources
- The lack of a convincing proof of correctness led to late RTL change and validation reset

# Algorithm Modeling & Validation

Architect-ville

Designer-town



# MASC

A *lingua franca* for specifying arithmetic algorithms and communicating between architects, designers, and validators

- A semantically simple and easily understood subset of C, suitable for executable modeling of arithmetic algorithms
- Extended with the SystemC register class library, supporting bit-level modeling of integer and fixed-point registers of arbitrary width and precision
- Amenable to mathematical reasoning and source-level formal analysis

# Example

```
// Second phase of encoder (compute BE[7:0]):

array<8, int> Encode(ui16 S, ui16 C) {
  array<8, int> result;
  uint lo = 14;
  for (uint i=0; i<8; i++) {
    int d = S.range(lo+1, lo).to_uint() + C[lo] - 2;
    result.elt[i] = -d;
    lo -= 2;
  }
  if (result.elt[7] % 2) {
    result.elt[7] = 0; // result.elt[7]--;
  }
  return result;
}
```

# Masc subset of SystemC

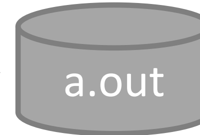
- Data types:
  - bool, uint, int, enumerated types
  - sc_uint, sc_int, sc_biguint, sc_bigint, sc_ufixed, sc_fixed
  - Arrays and structs of the above
- Modeling constructs:
  - Assignment to function-local variables only
  - The usual control constructs: if/else, for, while, do-while
  - Parameter passing by value only (special constructs for passing and returning arrays and tuples)
  - Only pure functions are allowed
  - assert(), with its usual meaning
- Not supported:
  - Everything else: pointers, I/O, C++ OO features, SystemC events, …
- For the full story see the paper

# Masc tool flow

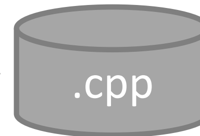
Masc is compilable SystemC

*<Arbitrary C++ code>*
```
// Masc begin
```
*<Masc code>*
```
// Masc end
```
*<Arbitrary C++ code>*

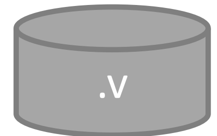
g++ -lsystemc

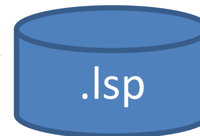



Pre-process

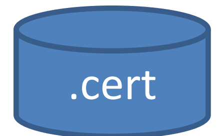



HLS




ACL2




Annotations in the source delineate code in the Masc subset and guide the preprocessor

# Support for functional programming

```
array<8, uint> sum (array<8,uint> a,
                    array<8,uint> b) {
    …
}
```

---

```
mv<uint,uint> bar (uint x, uint y) {
    return mv<uint, uint>(y,x);
}

// (u,v) = bar (x,y)
bar(x,y).assign(u,v)
```

# MASC -> ACL2 Translation

- MASC primitive types map to signed integers
- MASC arrays, structs map to ACL2 alists
- Expressions over these translate in the obvious way
  - Some shifts auto-inserted to handle sc_fixed, sc_ufixed
- Each MASC function translates to an ACL2 defun…

# Translating Statements

- Inside a function, statements are translated bottom up into a nest of lets

```
// (u,v) = bar (x,y)
bar(x,y).assign(u,v)        < (X Y), (U V), (BAR X Y) >
                               (U V Z),
;                              (MV-LET (U V)
                                       (BAR X Y)
                                       (+ U V)) >
z = u + v                   < (U V), (Z), (+ U V) >
```

- Some optimizations for readability

# Loops

- A separate recursive function is generated for each loop

```
for (int j=5; j >= -3; j--) {
    u = x + 3*u;
}
```

```
; Invoked as (LOOP-0 (5 X U))
(DEFUN LOOP-0 (J X U)
    (DECLARE (XARGS :MEASURE (NFIX (- J (1- -3)))))
    (IF (AND (INTEGERP J)
             (>= J -3))
        (LET ((U (+ X (* 3 U))))
             (LOOP-0 (- J 1) X U))
       U))
```

# Loop translation

```
array<17, ui3> Booth (ui32 x)
{
   array<17, ui3> a;

  // Pad with 2 leading zeroes
  // and 1 trailing zero:
  ui35 x35 = x << 1;

  // Compute booth encodings:
  for (int k=0; k<17; k++) {
    a.elt[k] =
      Encode(x35.range(2*k+2, 2*k));
  }
  return a;
}
```

```
(DEFUN BOOTH-LOOP-0 (K X35 A)
       (DECLARE (XARGS :MEASURE (NFIX (- 17 K))))
       (IF (AND (INTEGERP K) (< K 17))
           (LET ((A (AS K
                        (ENCODE (BITS X35 (+ (* 2 K) 2) (* 2 K)))
                        A)))
                (BOOTH-LOOP-0 (+ K 1) X35 A))
           A))
(DEFUN BOOTH (X)
       (LET ((X35 (BITS (ASH X 1) 34 0)) (A NIL))
            (BOOTH-LOOP-0 0 X35 A)))
```

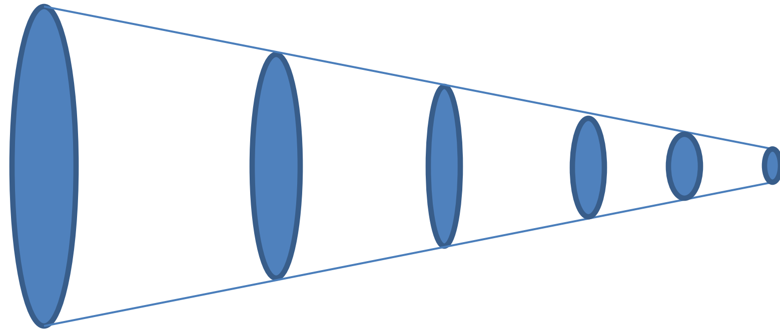
# Proofs about loops

```
(defthm booth-recursion-1
  (implies (and (natp k)
                (natp j)
                (< j k))
```

```
(defthm booth-lemma
  (implies (and (bvecp y 32)
                (natp k)
                (< k 17))
           (equal (ag k (booth y))
                  (cat (neg (theta k y)) 1 (abs (theta k y)) 2))))
```

```
                (<= k j)
                (< j 17))
           (equal (ag j (booth-loop-0 k y35 a))
                  (encode (bits y35 (+ (* 2 j) 2) (* 2 j))))))
```

# Application: Division and Sqrt



- In a nutshell:
  - Form initial estimates of $Q_0, R_0$
  - Compute $Q_i, R_i$ for $i > 0$ by recurrence
  - Stop when $Q_i$ is sufficiently accurate to enable IEEE rounding
- Correctness depends on
  - fidelity of the initial estimates,
  - accuracy of various approximations used in the recurrence,
  - validity of several tricks and optimizations (e.g. biased Booth encoding)
- Proofs are nontrivial

# Outline: Division algorithm

```
mv<uf64i1, int> Execute(bool div, Prec prec, RMode rmode,
                        uf64i1 X, uf64i1 Y)
{

  ufxi2 R0;      // partial remainder
  ufxi2 QP;      // partial quotient

  <… Initialize …>

  for (uint j=1; j<=NL; j++) {

    <… Iteration …>

  }

  <… Cleanup …>

  // return rounded result and exponent correction
  return Rounder(rmode, prec, QP);

}
```

# Correctness of division

```
(encapsulate (((div) => *) ((prec) => *) ((rmode) => *)
              ((x) => *) ((y) => *) ((e0) => *)) … )

(defun significand ()
  (mv-nth 0 (mv-list 2 (EXECUTE (div) (prec) (rmode) (x) (y)))))

(defun expcorr ()
  (mv-nth 1 (mv-list 2 (EXECUTE (div) (prec) (rmode) (x) (y)))))

(defun ufval (x f) (/ x (expt 2 f)))
(defun x$ () (ufval (x) 63))
(defun y$ () (ufval (y) 63))
(defun significand$ () (ufval (significand) 63))

(defthm correctness-of-div
  (implies (divp)
           (and (member (expcorr)  '(-1 0 1))
                (< (significand$) 2)
                (>= (significand$) 1)
                (= (rnd (/ (x$) (y$)) (decode-rmode (rmode)) (prec))
                   (* (expt 2 (expcorr)) (significand$))))   ))
```

# Division and Sqrt: Wins

- An accessible and well-documented reference implementation of divide and sqrt (~600 loc)
- Pen and paper analysis of the algorithms
- Mechanized proofs of correctness
- Our model is the main resource used by multiple projects to understand the algorithms and implementation tricks
- Our proof framework was used to verify several proposed optimizations to the algorithm

# Summary

- We have defined a SystemC subset that’s sufficient for modeling arithmetic algorithms and amenable to formal reasoning
- We have implemented a translation path that checks adherence to the subset and generates
  - A formal model in the ACL2 theorem prover
  - Synthesizable SystemC for input to Cadence CtoS
- Architects have developed several significant models in the language:
  - Reciprocal approximation
  - Division/square root
  - Byte compression
- We’ve mechanized proofs of correctness for each of these

# Future work

- Extensions to the MASC subset
  - Full support for sc_* register classes
  - Templates – extensively used by some customers
- Streamline reasoning about extracted code
  - The ACL2 definitions we extract tend to lack modularity
- Investigate links to VP & RTL
  - HLS
  - C vs RTL equivalence checking
    - Calypto SLEC, Synopsys Hector, homegrown

# Thank you