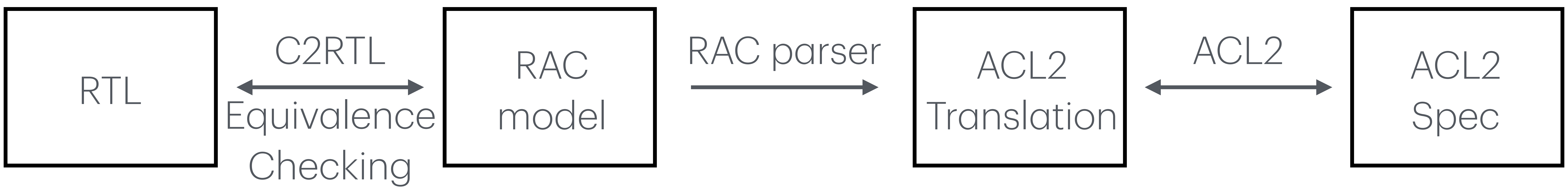
On Automating Proofs of Multiplier Adder Trees using the RTL books

Mayank Manjrekar, Arm Inc.

May 12, 2025

Verification Methodology




```
module adder16 (
    input  wire [15:0] a,
    input  wire [15:0] b,
    output wire [15:0] sum,
);

    wire [16:1] carry;  // internal carry signals

    assign sum[0]   = a[0] ^ b[0];
    assign carry[1] = (a[0] & b[0])
                    | (a[0] & carry[0])
                    | (b[0] & carry[0]);

    genvar i;
    generate
        for (i = 1; i < 16; i = i + 1) begin : ripple
            assign sum[i]    = a[i] ^ b[i] ^ carry[i];
            assign carry[i+1] = (a[i] & b[i])
                              | (a[i] & carry[i])
                              | (b[i] & carry[i]);
        end
    endgenerate

endmodule
```

```
// RAC begin

typedef ac_int<16, false> ui16;
typedef ac_int<17, false> ui17;

ui16 add16(ui16 a, ui16 b) {
  ui16 sum = 0;
  ui17 carry = 0;

  for (uint i = 1; i < 16; i++) {
    sum[i] = a[i] ^ b[i] ^ carry[i];
    carry[i+1] = (a[i] & b[i])
               | (b[i] & carry[i])
               | (carry[i] & a[i]);
  }

  return sum;
}

// RAC end
```

```
(IN-PACKAGE "RTL")

(INCLUDE-BOOK "rtl/rel11/lib/rac" :DIR :SYS

(SET-IGNORE-OK T)

(SET-IRRELEVANT-FORMALS-OK T)

(DEFUND ADD16-LOOP-0 (I B A SUM CARRY)
  (DECLARE (XARGS :MEASURE (NFIX (- 16 I)))
  (IF (AND (INTEGERP I) (< I 16))
      (LET ((SUM (SETBITN SUM 16 I
                          (LOGXOR (LOGXOR (
                                  (BITN CAR
            (CARRY (SETBITN CARRY 17 (+ I 1
                            (LOGIOR (LOGIOR

                                    (LOGAND
        (ADD16-LOOP-0 (+ I 1) B A SUM CARRY
    (MV SUM CARRY)))

(DEFUND ADD16 (A B)
  (LET ((SUM 0) (CARRY 0))
    (MV-LET (SUM CARRY)
            (ADD16-LOOP-0 1 B A SUM CARRY)
      SUM)))
```

```
(defun add16-spec (a b)
  (bits (+ a b) 15 0))
```

Integer Multiplication

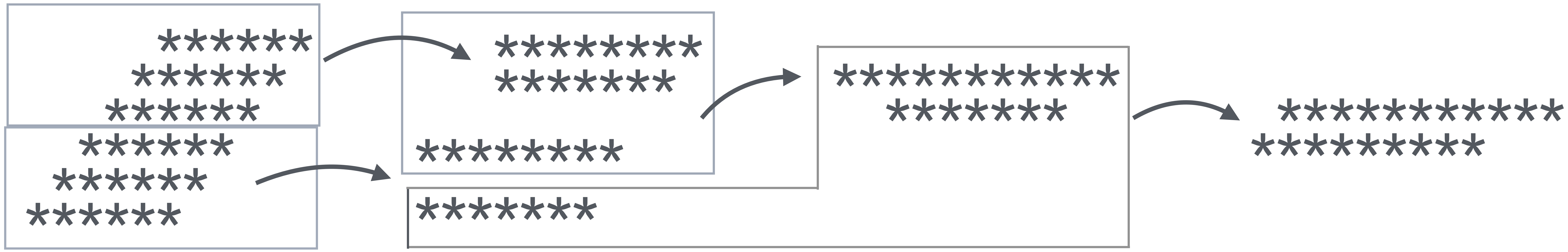
- Building block for many other arithmetic operations, e.g., Floating-point multiplication, Division.
- Fully automatic methods do not scale well.
- A typical design may be divided into 2 steps:
  - Generation of partial products
  - Summation of partial products
- Verification is also a two step process
  - Prove that the sum of the partial products is equal to the product of the inputs
  - **Verify the summation procedure.**

```
      ******
    x ******
------------
      ******
     ******
    ******
   ******
  ******
 ******
------------
************
```

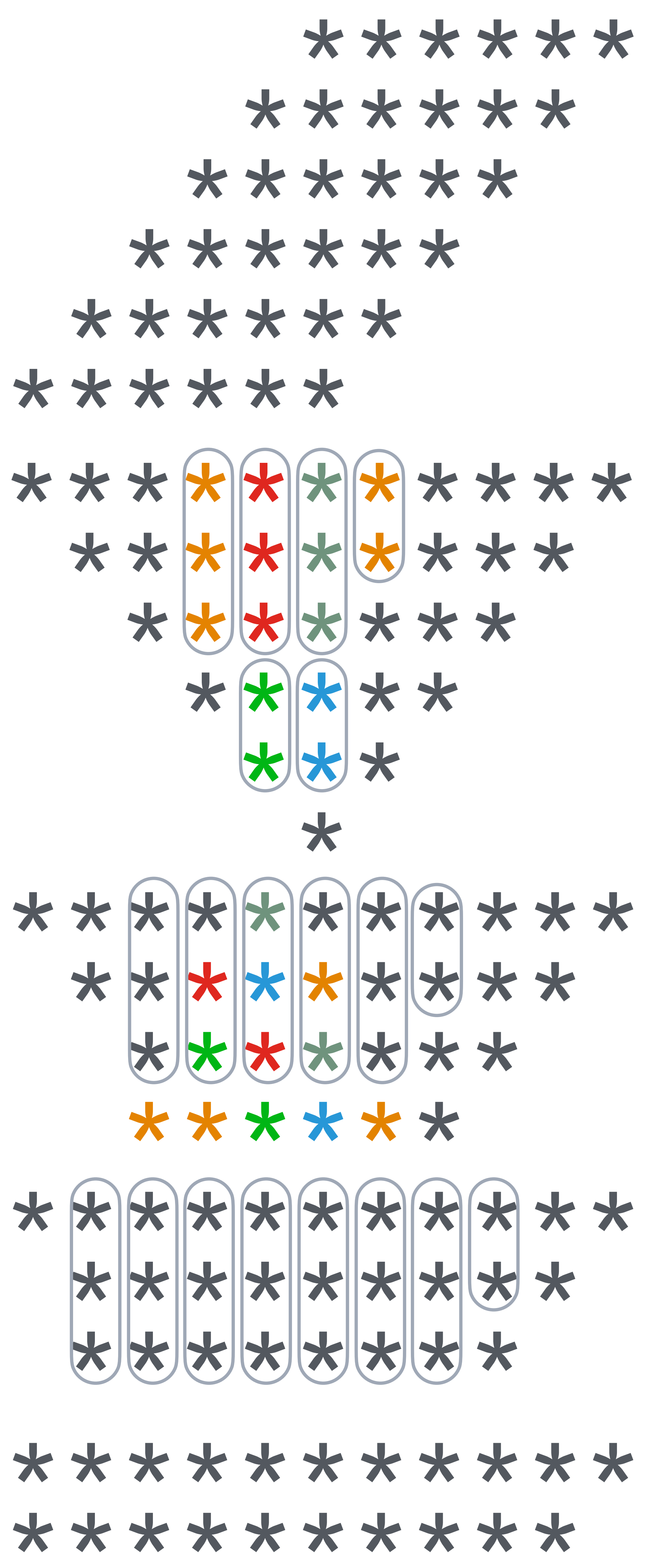
3:2 compressor trees

- The following lemma is used to reduce the number of partial products to two
- Full-adder theorem:

```
(defthm add-3
  (implies (and (integerp x) (integerp y) (integerp z))
           (= (+ x y z)
              (+ (logxor x y z)
                 (* 2 (logior (logand x y) (logand x z) (logand y z)))))))
```



Dadda Compression Tree



- Compression tree RTL is generated by an automated script.
- 3:2 compressors applied bitwise along each column.

Compress Function

RAC model

```
ui12 compress(ui12 pp0, ui12 pp1, ui12 pp2, ui12
pp3, ui12 pp4, ui12 pp5) {

    ui12 l0_pp0 = 0;
    l0_pp0.set_slc(2, pp0.slc<4>(2));
    l0_pp0[6] = pp1[6];
    l0_pp0[7] = pp2[7];
    l0_pp0[8] = pp3[8];
    l0_pp0[9] = pp4[9];
    l0_pp0[10] = pp5[10];

    ...

    ui12 fpp1 = 0;
    fpp1[1] = pp1[1];
    fpp1.set_slc(2, l2_pp0to2_s.slc<8>(2));
    fpp1[10] = l2_pp0to2_c[10];

    return fpp0 + fpp1;
}

ui12 mul6(ui6 a, ui6 b) {
    ui12 pp[6] = gen_pp_array(a, b);
    return compress(pp[0], pp[1], pp[2], pp[3],
pp[4], pp[5]);
}
```

ACL2 translation

```
(DEFUND COMPRESS (PP0 PP1 PP2 PP3 PP4 PP5)
  (LET* ((L0_PP0 0)
         (L0_PP0 (SETBITS L0_PP0 12 5 2 (BITS PP0 5 2)))
         (L0_PP0 (SETBITN L0_PP0 12 6 (BITN PP1 6)))
         (L0_PP0 (SETBITN L0_PP0 12 7 (BITN PP2 7)))
         (L0_PP0 (SETBITN L0_PP0 12 8 (BITN PP3 8)))
         (L0_PP0 (SETBITN L0_PP0 12 9 (BITN PP4 9)))
         (L0_PP0 (SETBITN L0_PP0 12 10 (BITN PP5 10)))

         ...

         (FPP1 0)
         (FPP1 (SETBITN FPP1 12 1 (BITN PP1 1)))
         (FPP1 (SETBITS FPP1 12 9 2 (BITS L2_PP0TO2_S 9 2)))
         (FPP1 (SETBITN FPP1 12 10 (BITN L2_PP0TO2_C 10))))
    (BITS (+ FPP0 FPP1) 11 0)))
```

CTV-CP Algorithm

- Process terms of the form

```
(implies *hyps*
   (equal (compress pp0 pp1 pp2 pp3 pp4 pp5)
          (bits (+ pp0 pp1 pp2 pp3 pp4 pp5) 15 0)))
```

- Consider the LHS of conclusion. Expand `compress` function call.

- Dive into the sequence of bindings, building a substitution context till the inner-most term is reached.

- Build an *internal representation* of the inner-most term to represent its bitwise expansion

- Loop: Match all sum and carry terms and rewrite $\rightleftarrows$ Make the inner-most substitution

- If the final representations match, we are done!

CTV-CP

BVFSL data-structure

- Expressions in the *compress* function are parsed into a format that is equivalent to their bitwise expansion

- ```
  bvfsl := (cons bvfs bvfsl)
         | nil

  bvfs  := (cons bvf num)


  bvf   := bv
         | ‘(:fas bvf bvf bvf)
         | ‘(:fac bvf bvf bvf)


  bv    := ‘(:bit symbol num)
         | ‘(:v 0)
         | ‘(:v 1)
  ```

```
i-bvfsl:
(cons a b) |-> (i-bvfs a) + (i-bvfsl b)
nil        |-> 0

i-bvfs:
(cons a n) |-> (i-bvf a) << n

i-bvf:
‘(:fas a b c) |-> (i-bvf a) ^ (i-bvf b) ^ (i-bvf c)
‘(:fac a b c) |->  (i-bvf a) & (i-bvf b)
                 | (i-bvf b) & (i-bvf c)
                 | (i-bvf c) & (i-bvf a)
i-bv:
'(:bit s n) |-> (bitn s n)
'(:v 0)     |-> 0
'(:v 1)     |-> 1
```
```

CTV-CP

BVFSL data-structure

Expressions	BVFSL form	Interpretation (bitwise expansion)
Final Sum: (bits (+ fpp0 fpp1) 11 0))	'(((:bit fpp0 0) 0) … ((:bit fpp0 11) 11)) ((:bit fpp1 0) 0) … ((:bit fpp1 11) 11)))	(+ (ash (bitn fpp0 0) 0) … (ash (bitn fpp0 11) 11) (ash (bitn fpp1 0) 0) … (ash (bitn fpp1 11) 11))
(setbitn l0pp0s 12 4 (logxor (bitn l0pp0 4) (bitn l0pp1 4)))	(((:bit l0pp0s 11) 11) … ((:bit l0pp0s 5) 5) ((:fas (:bit l0pp0 4) (:bit l0pp1 4) (:v 0)) 4) ((:bit l0pp0s 3) 3) … ((:bit l0pp0s 0) 0)	(+ … (ash (logxor (bitn l0pp0 4) (bitn l0pp1 4) 0) 4) …)

Properties of CTV-CP

- The inner-loop avoids blow-up of terms — the number of BVFS terms are never larger than *(# initial partial products) x multiplication width*
- Equivalent to outside-in rewriting, but avoids hash-consing overhead
- Efficient, verified clause processor
  - Runtime for 53x53 multiplier adder tree <1s

Comparison with other approaches

- VeSCMul:
  - Custom rewriter + custom set of rewrite rules on S-C terms
  - Normalize LHS and RHS, and compare
  - Incompatible with RTL library

- GL:
  - Stalls with medium size designs when given the main conjecture.
  - However, works when applied bitwise, i.e., on conjectures of the form:

```
(implies *hyps*
   (equal (bitn (compress pp0 pp1 pp2 pp3 pp4 pp5) *concrete-n*)
          (bitn (+ pp0 pp1 pp2 pp3 pp4 pp5) *concrete-n*)))
```

Future work

- Extend CTV-CP to handle vector multipliers.
- Continue investigating techniques to increase proof automation using the RTL books.

Thank You!