Formal Verification of LabVIEW Diagrams

Mark Reitblatt

Dept. of Computer Sciences, University of Texas
National Instruments, Inc.

Outline

- LabVIEW Overview
- ACL2 Overview
- Overview of approach
- Walk through example verification
- Conclusion

Project History

- Jeff Kodosky started playing around in 2004 with the idea of verifying a LabVIEW program
- Warren Hunt and J Moore met on occasion with Jeff and Jacob Kornerup over a couple of years, culminating with NI engaging Grant as an intern in 2005
- Summer 2007: Alternate approach developed with Matt Kaufmann models LabVIEW programs, including loop structures, directly as ACL2 functions. At the end of the summer Grant Passmore left for Edinburgh and transferred his work to the author
- Current: Matt continued contracting, approach has been fully automated, expanded and used to verify a dozen examples

Credit

- To reiterate

Credit

- To reiterate
  - Joint work with Matt Kaufmann

Credit

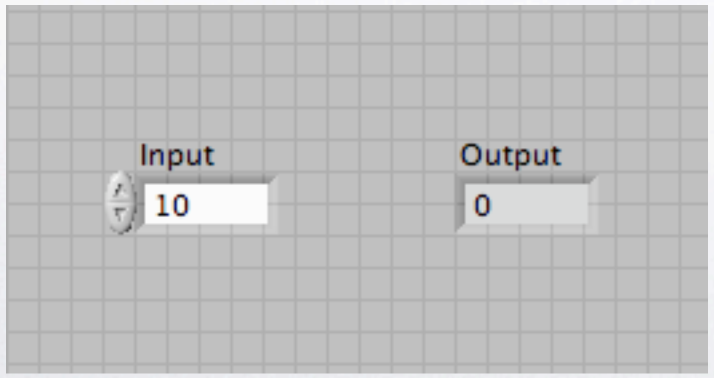
- To reiterate
  - Joint work with Matt Kaufmann
  - Building off work with Matt and Grant

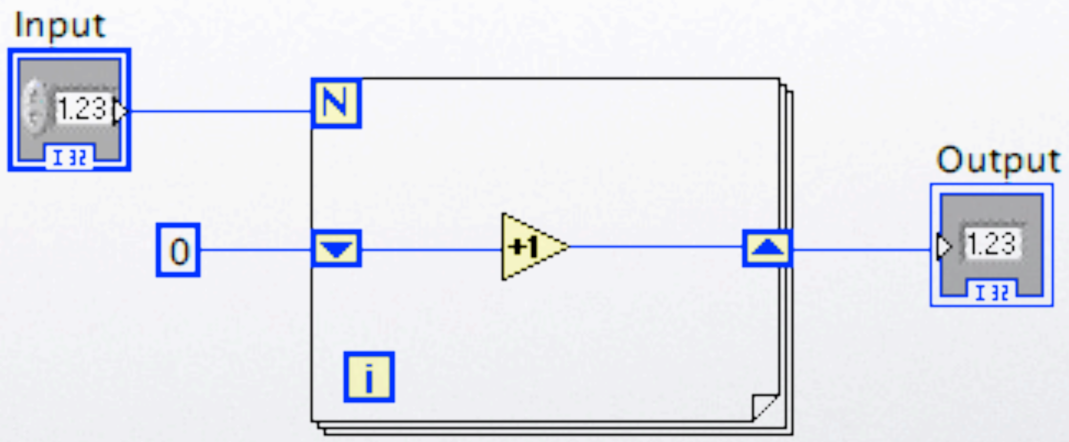
Credit

- To reiterate
  - Joint work with Matt Kaufmann
  - Building off work with Matt and Grant
- Project fully funded by National Instruments, Inc.

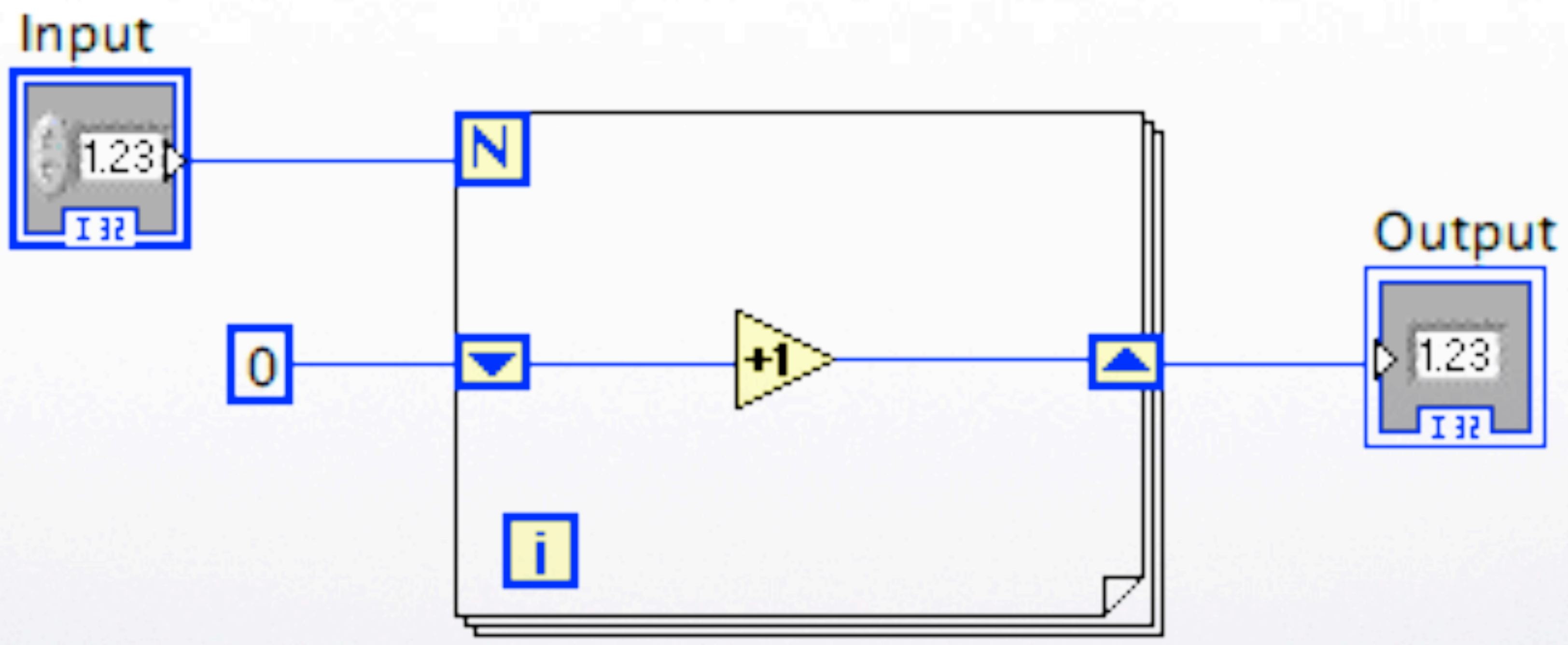
LabVIEW (in brief)

- Graphical dataflow language (G) with control structures
- Shift register memory elements
- Separate Front (user interface) and Back (implementation) panels



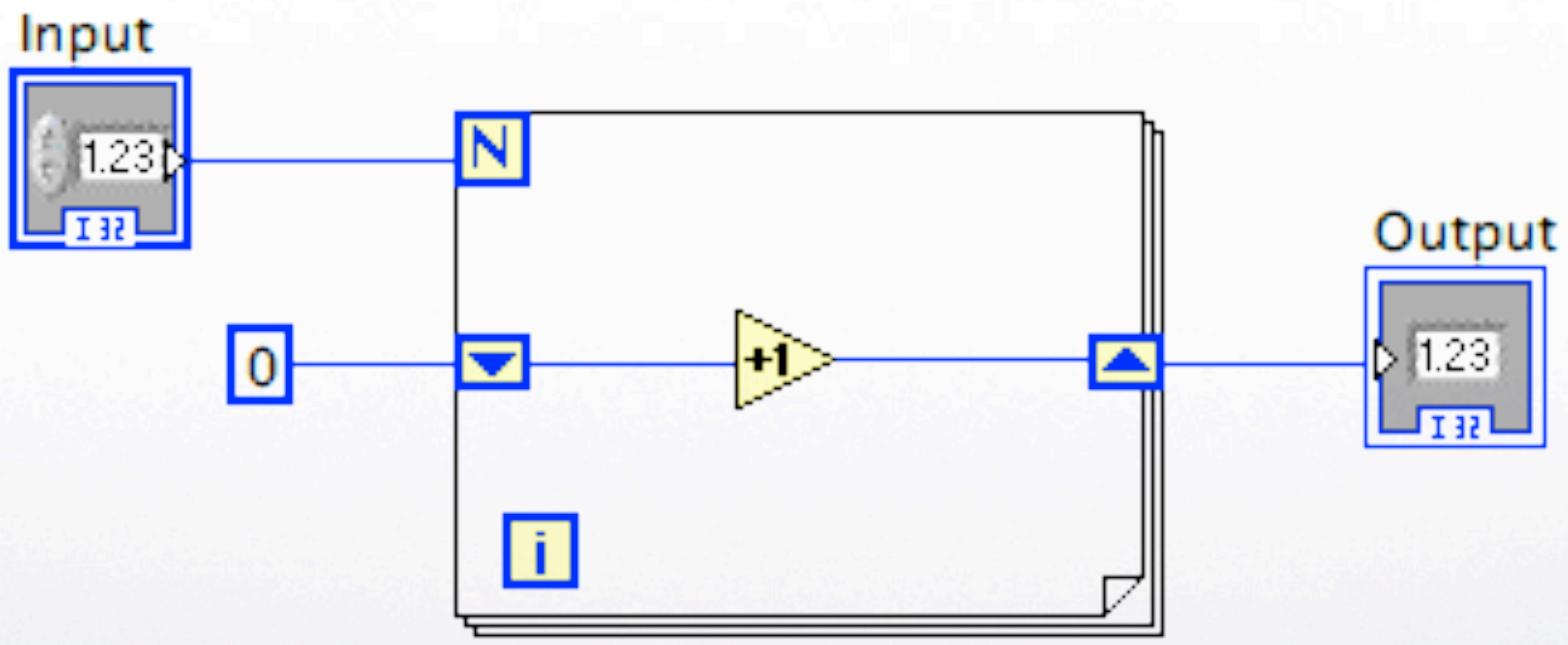


LabVIEW For-Loops

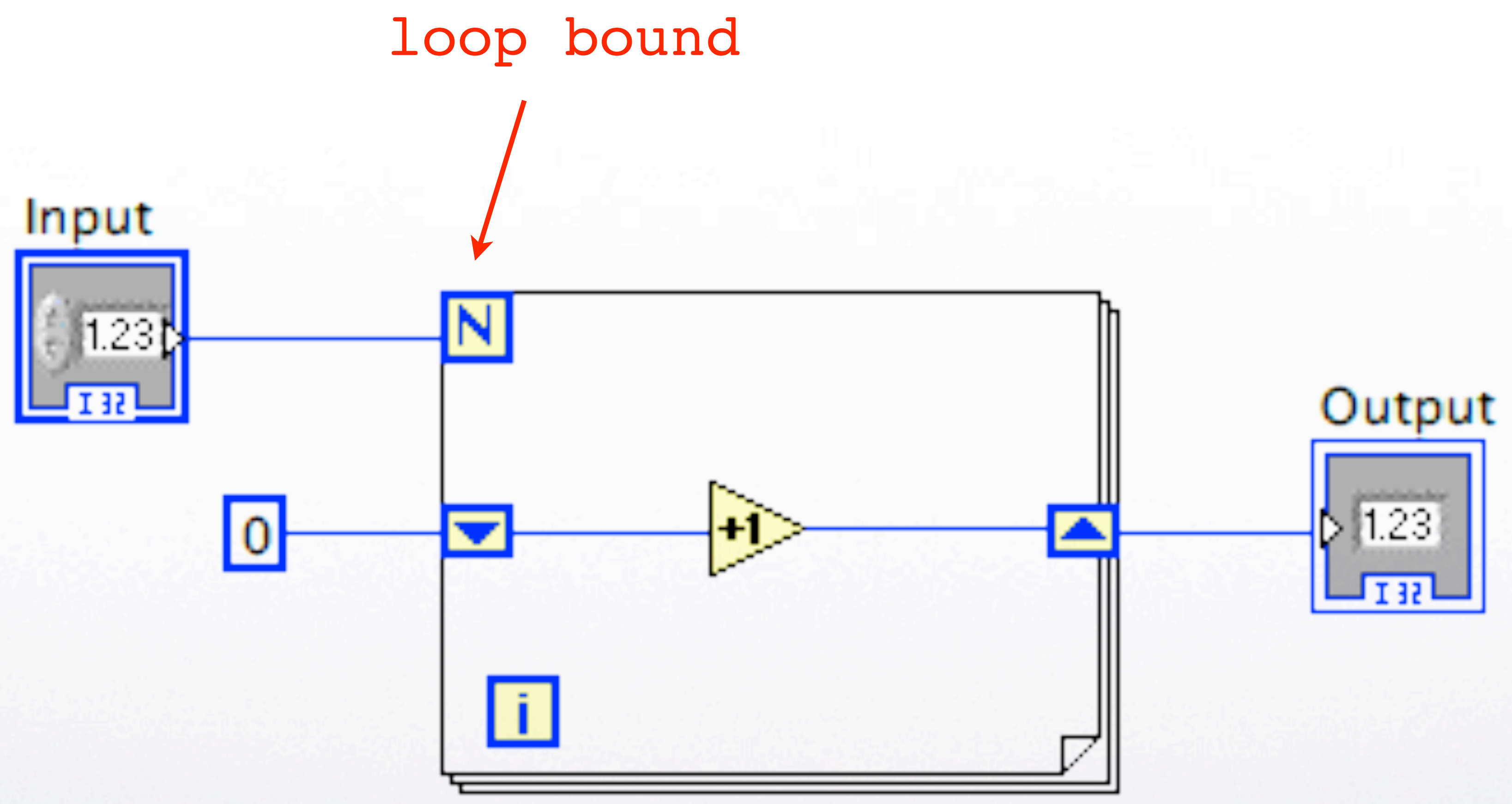


LabVIEW For-Loops

loop bound

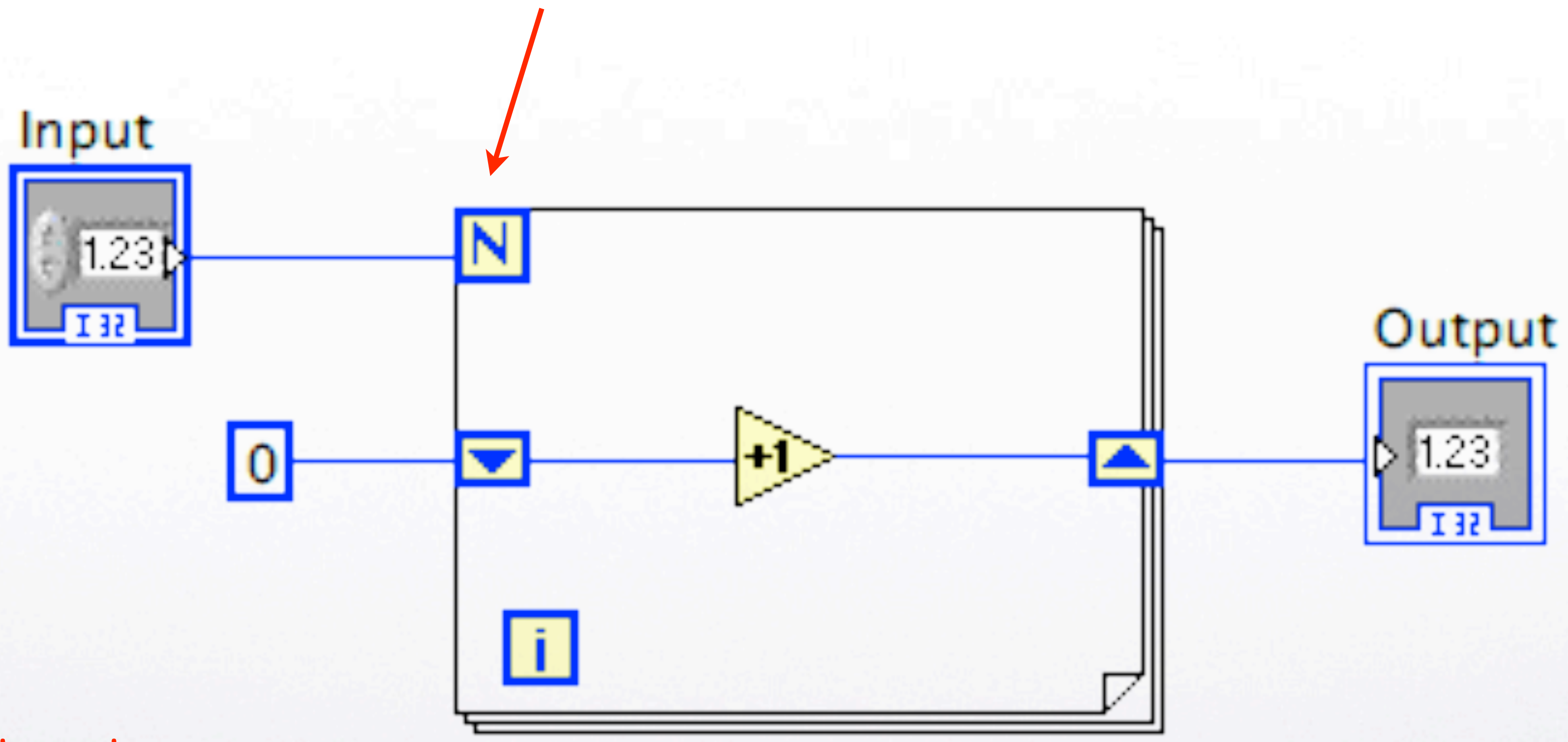


LabVIEW For-Loops



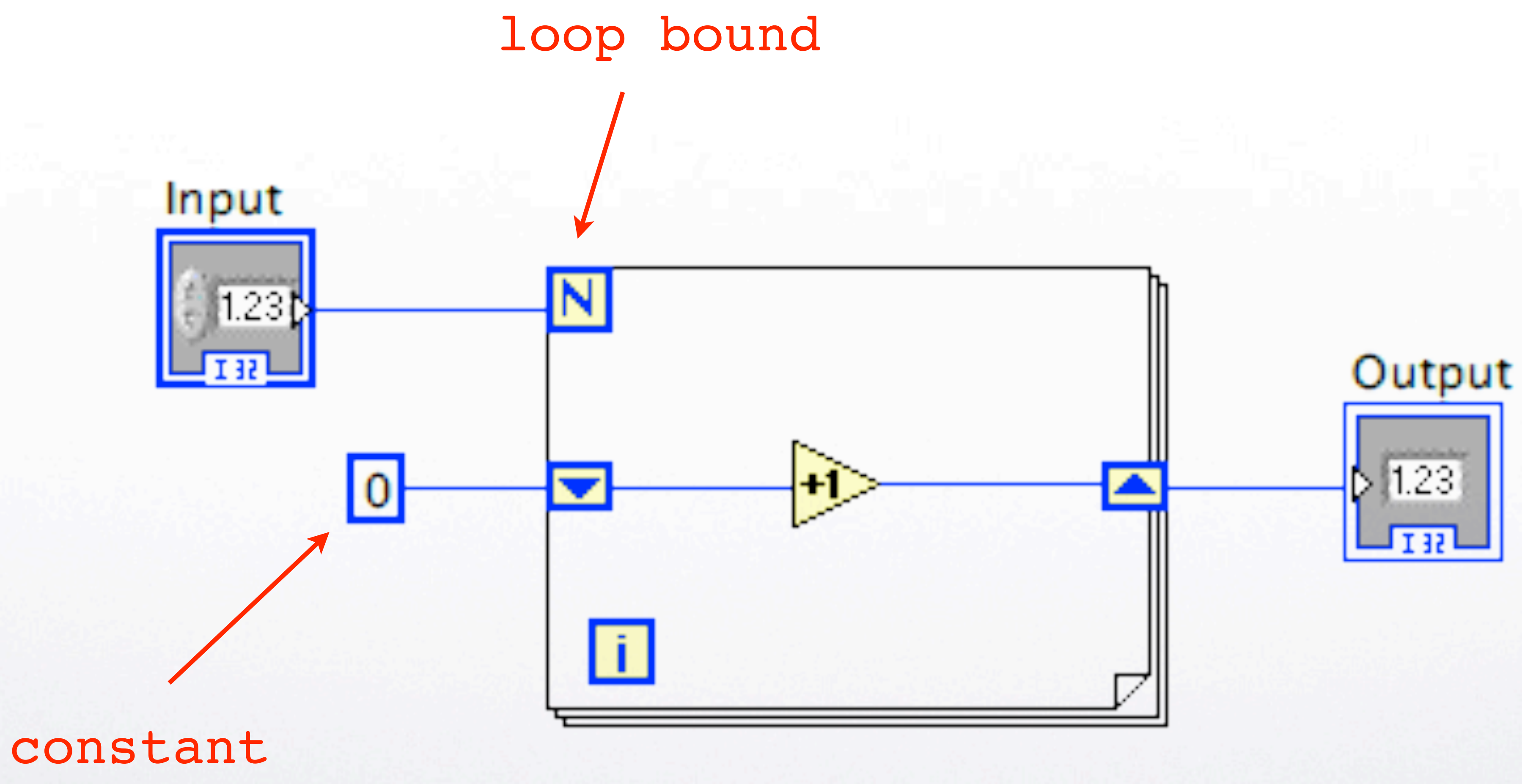
LabVIEW For-Loops

loop bound

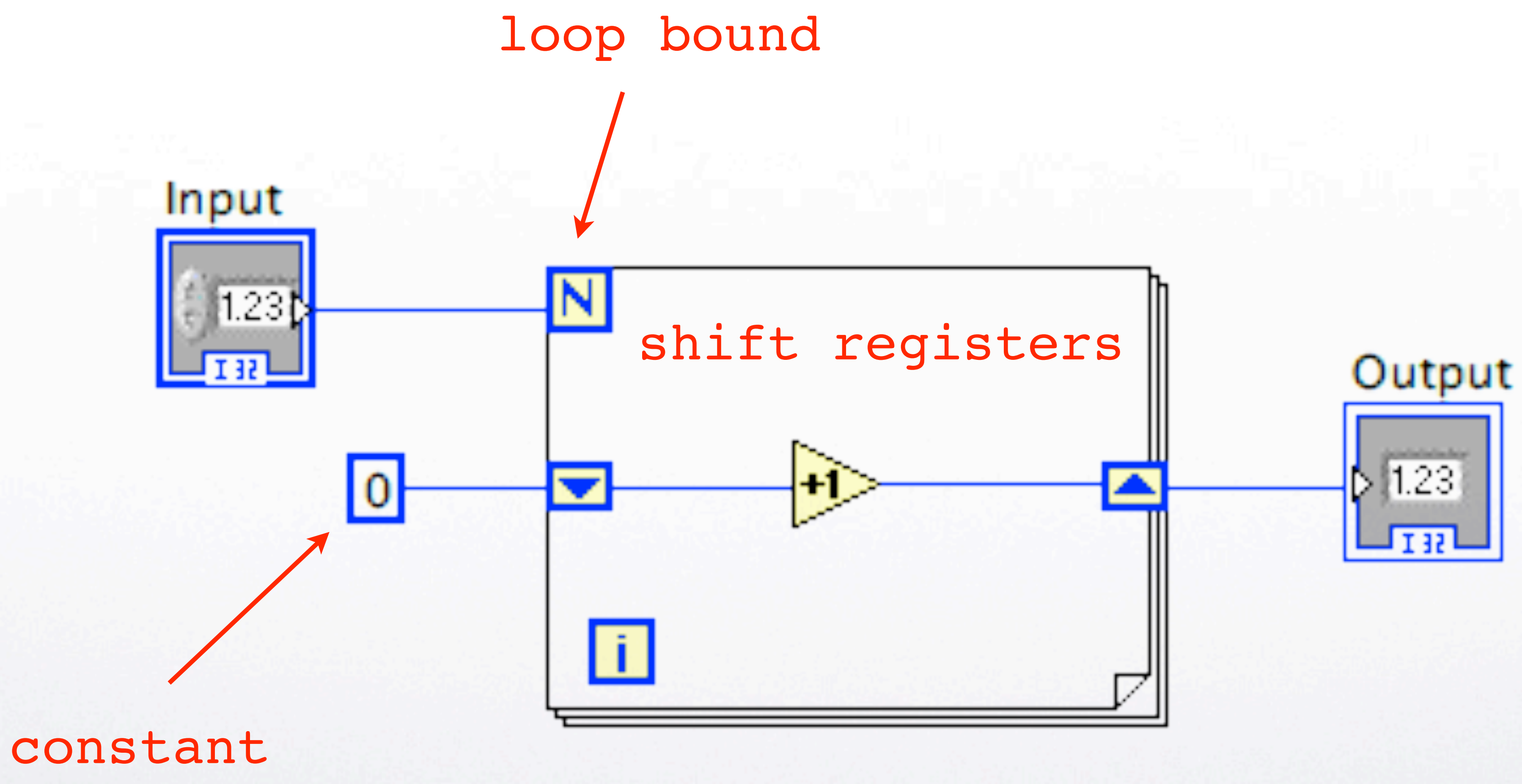


constant

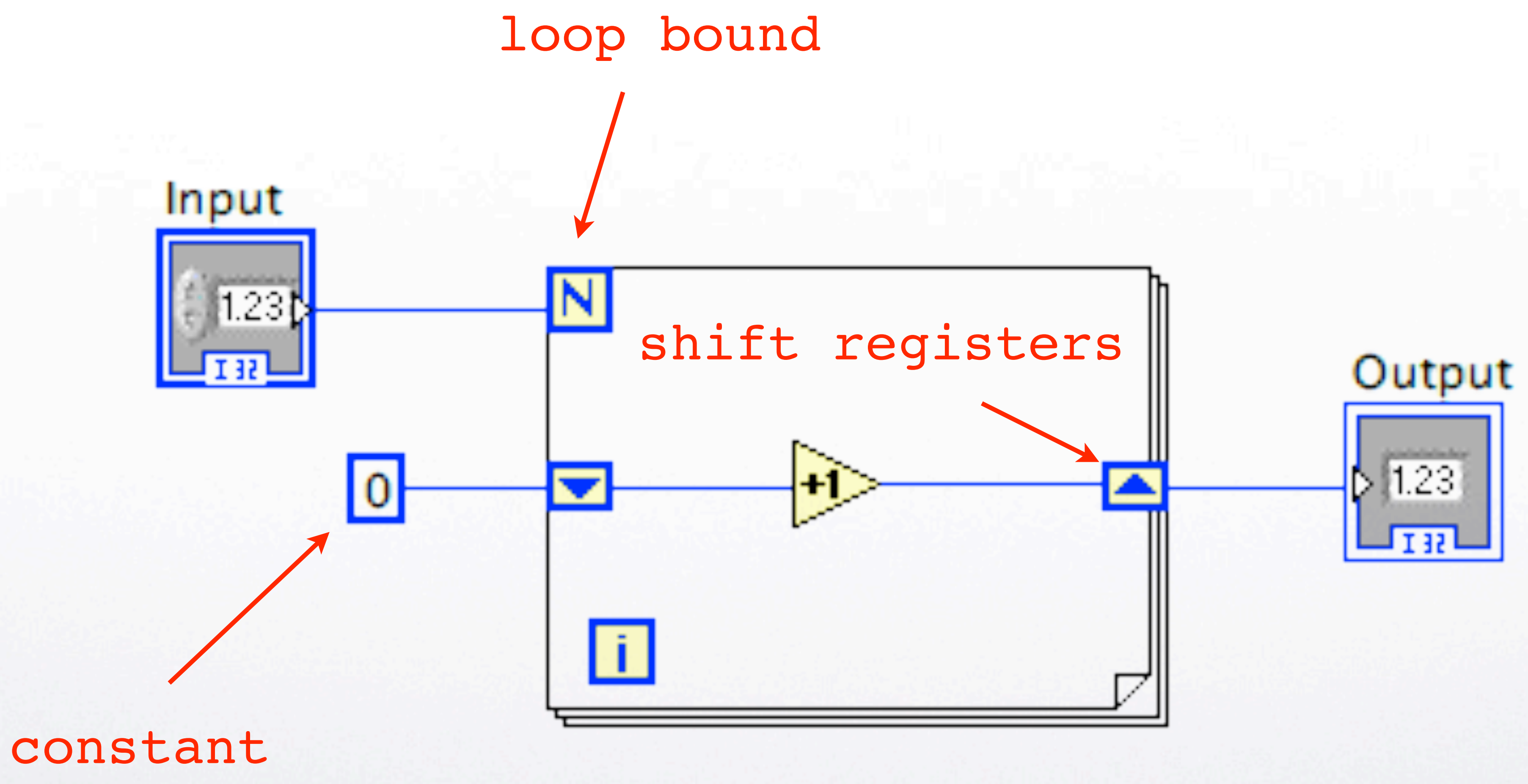
LabVIEW For-Loops



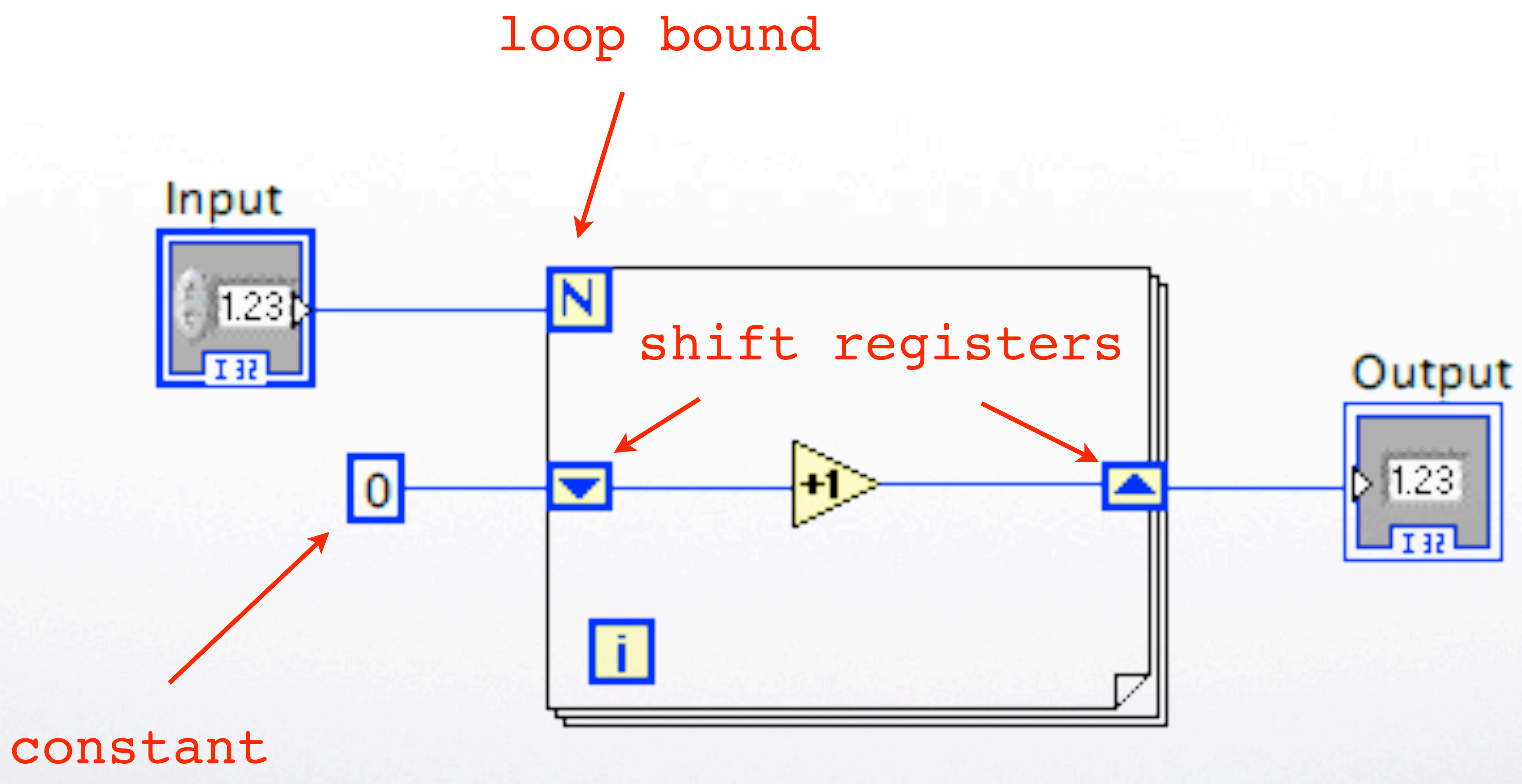
LabVIEW For-Loops



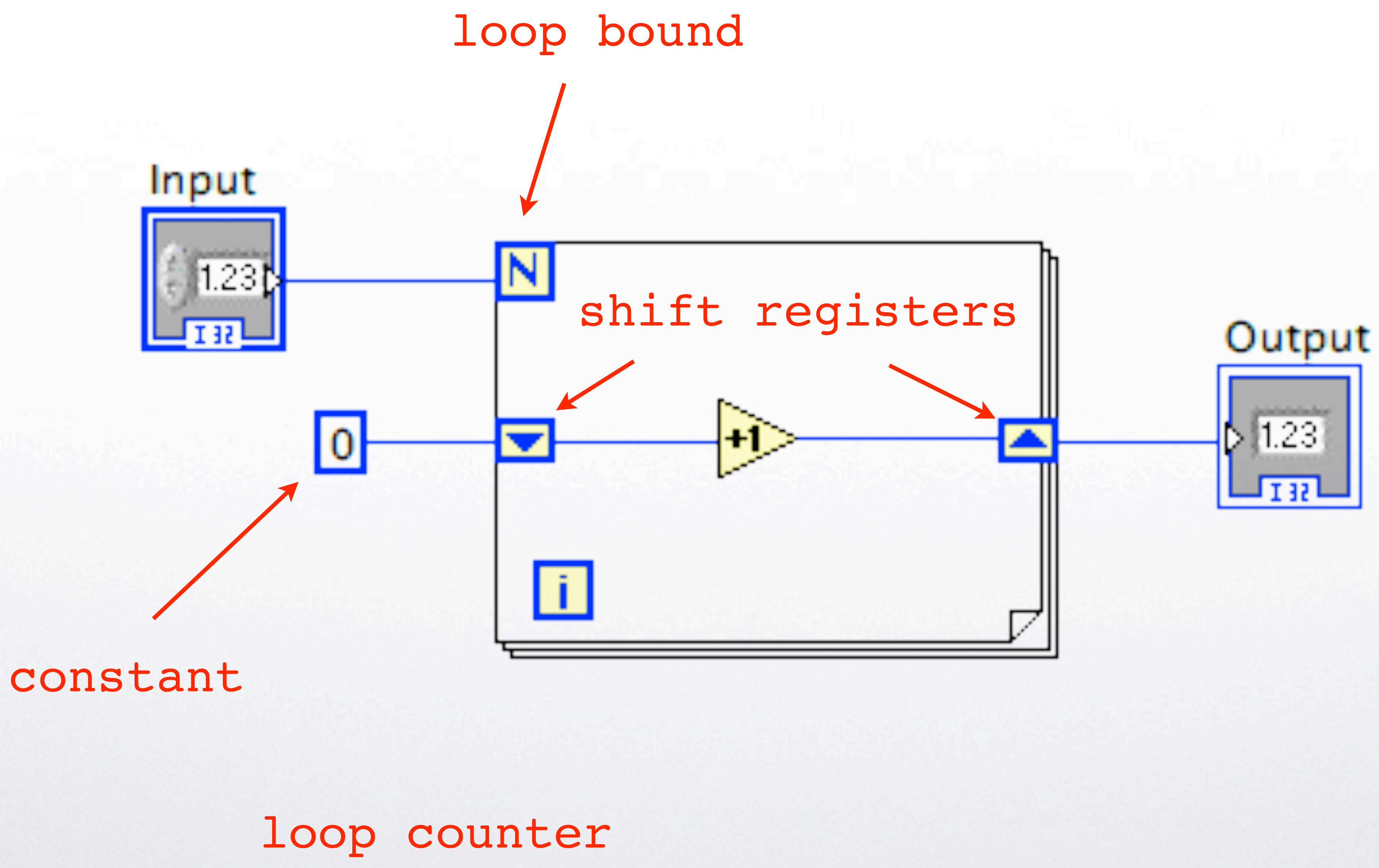
LabVIEW For-Loops



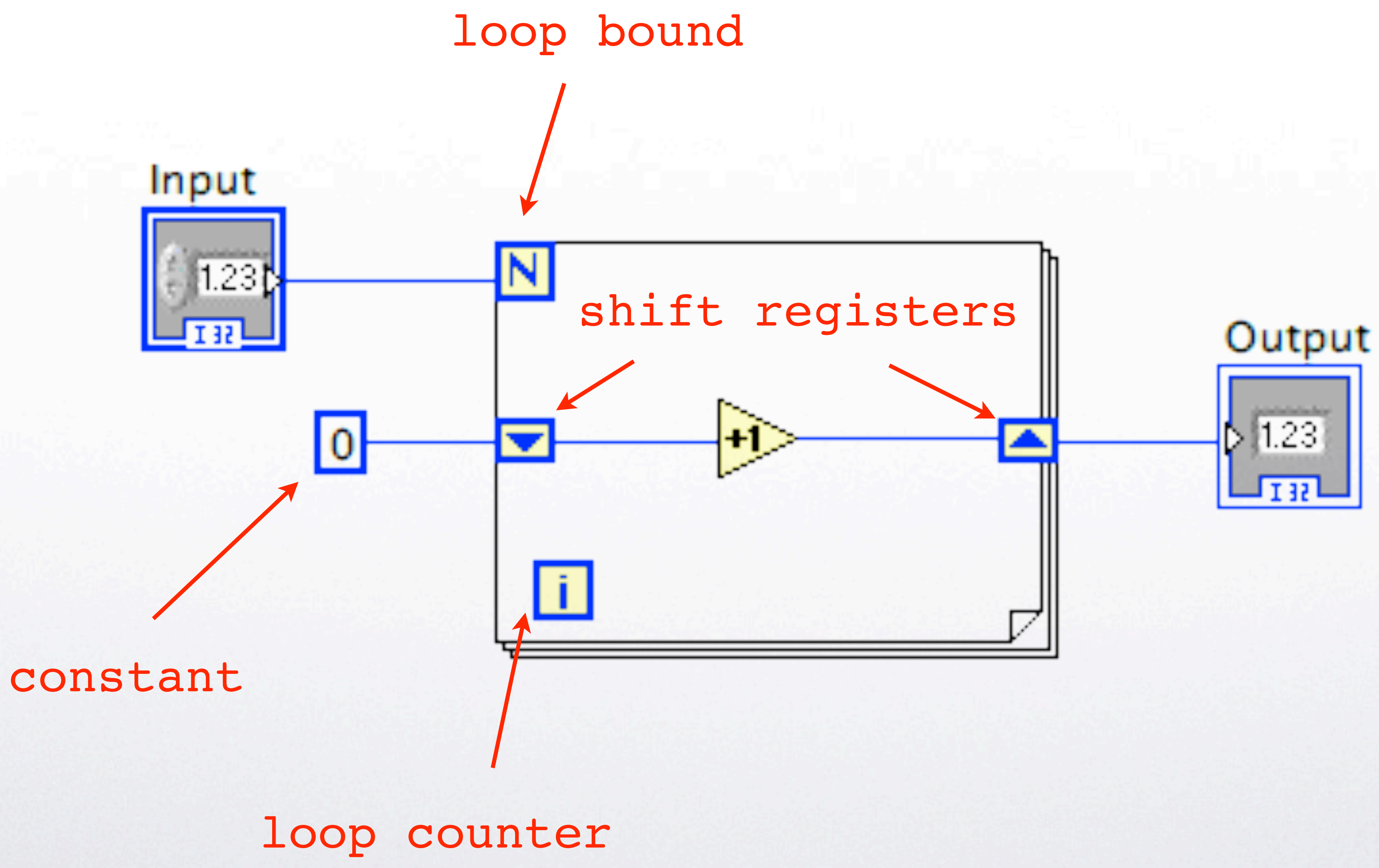
LabVIEW For-Loops



LabVIEW For-Loops



LabVIEW For-Loops



ACL2

- Programming Language
- Formal Logic
- Automated Theorem Prover

ACL2 The Language

- Subset of Lisp
  - S-Expressions
  - Untyped
- First Order
- Applicative
  - Purely functional
- Total
  - All functions defined on all inputs
  - I.E. everything terminates

ACL2 Syntax

- S-Expressions
  - Primary syntax is ()
- Prefix notation
  - `(f x)` instead of `f(x)`
- Predicates end in a `-p` by convention
- Use `defun` to define a function

ACL2 (cont.)

```
(defun fib (i)
  (if (or (zp i) (= i 1))
      1
      (+ (fib (- i 1))
         (fib (- i 2)))))
```

- (zp x) returns false if x is a natural number and x > 0
- Note that zp recognizes all non-integers

ACL2 The Logic

- Definitional Principle

ACL2 The Logic

- Definitional Principle
- Use `defthm` to name, define and prove a new theorem

ACL2 The Logic

- Definitional Principle
- Use `defthm` to name, define and prove a new theorem
    - Theorems are stored as rules (usually rewrite)

ACL2 The Logic (cont.)

```
(defthm fib-is-bigger-than-n
  (implies (integerp n)
           (>= (fib n) n)))
```

- integerp recognizes integers
- fib is defined on every ACL2 object
  - But fib(n) >= n is not true for all objs.

ACL2 The Theorem Prover

- ACL2 proves theorems with existing theorems and function definitions
- User guides the process with hints and theory control
  - A theory is a list of enabled rules and definitions

ACL2 The Thm. Prover

```
ACL2 !>(defthm fib-is-bigger-than-n
  (implies (integerp n)
  (>= (fib n) n)))

([ A key checkpoint:

Goal'
(IMPLIES (INTEGERP N) (<= N (FIB N)))

*1 (Goal') is pushed for proof by induction.

])

Perhaps we can prove *1 by induction.  One induction
scheme is suggested
by this conjecture.

We will induct according to a scheme suggested by (FIB
N).  This suggestion
was produced using the :induction rule FIB.  If we let
(:P N) denote
*1 above then the induction scheme we'll use is
(AND (IMPLIES (AND (NOT (OR (ZP N) (= N 1)))
                   (:P (+ -1 N))
                   (:P (+ -2 N)))
              (:P N))
     (IMPLIES (OR (ZP N) (= N 1)) (:P N))).
This induction is justified by the same argument used
to admit FIB.

When applied to the goal at hand the above induction
scheme produces
five nontautological subgoals.

*1 is COMPLETED!
Thus key checkpoint Goal' is COMPLETED!

Q.E.D.

Summary
Form:  ( DEFTHM FIB-IS-BIGGER-THAN-N ...)
Rules: ((:COMPOUND-RECOGNIZER ZP-COMPOUND-RECOGNIZER)
        (:DEFINITION =)
        (:DEFINITION FIB)
        (:DEFINITION NOT)
        (:EXECUTABLE-COUNTERPART <)
        (:EXECUTABLE-COUNTERPART FIB)
        (:EXECUTABLE-COUNTERPART INTEGERP)
        (:EXECUTABLE-COUNTERPART NOT)
        (:FAKE-RUNE-FOR-LINEAR NIL)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:INDUCTION FIB)
        (:TYPE-PRESCRIPTION FIB))
Warnings:  None
Time:  0.01 seconds (prove: 0.00, print: 0.00, other:
0.00)
 FIB-IS-BIGGER-THAN-N
```

Formal Verification

- Application of formal methods for correctness proofs of SW/HW
- Uses formal specifications of behavior
- Utilizes theorem provers and decision procedures to complete proofs

Problem at Hand

- We desire to verify LabVIEW/G diagrams
- LabVIEW/G lacks an assertion primitive
- LabVIEW/G lacks a formal semantics

Solution

- Add assertion block to LabVIEW/G

Solution

- Add assertion block to LabVIEW/G
- Construct semantics for LabVIEW/G in ACL2

Solution

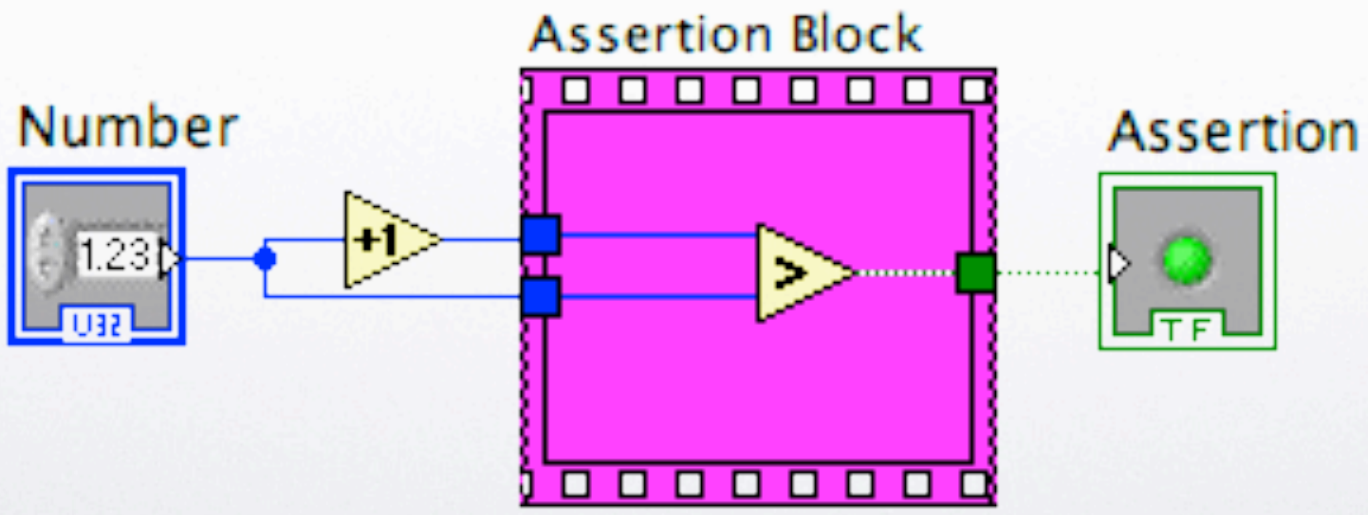
- Add assertion block to LabVIEW/G
- Construct semantics for LabVIEW/G in ACL2
- Convert assertions into ACL2 proof obligations

Solution

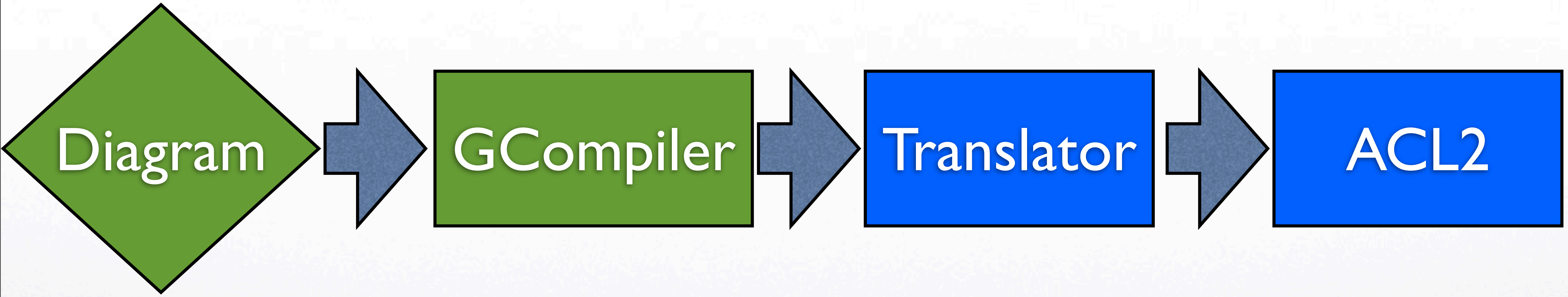
- Add assertion block to LabVIEW/G
- Construct semantics for LabVIEW/G in ACL2
- Convert assertions into ACL2 proof obligations
  - Use LabVIEW semantics for semantics of assertions

Our Approach

- "assertion" blocks are written in LabVIEW/G
  - This allows simulation, validation



Translation



Written in LabVIEW/G

Written in ACL2

Our Approach (cont.)

Our Approach (cont.)

- Translate LabVIEW/G diagrams into ACL2 functions (shallow embedding)

Our Approach (cont.)

- Translate LabVIEW/G diagrams into ACL2 functions (shallow embedding)
- 1-1 correspondence between function nodes,wires and ACL2 functions

Naming

Naming

- LabVIEW/G doesn’t allow naming of (most) nodes

Naming

- LabVIEW/G doesn’t allow naming of (most) nodes
- Human readability is essential to understanding proofs

Naming

- LabVIEW/G doesn't allow naming of (most) nodes
- Human readability is essential to understanding proofs
- Auto-naming of nodes based on type

Naming (cont.)

- Fn nodes are named as fntype-number

Naming (cont.)

- Fn nodes are named as fntype-number
  - ADD-1

Naming (cont.)

- Fn nodes are named as fntype-number
  - ADD-1
- Constant nodes are named by value

Naming (cont.)

- Fn nodes are named as fntype-number
    - ADD-1
- Constant nodes are named by value
    - CONSTANT[0]-2

Naming (cont.)

- Fn nodes are named as fntype-number
  - ADD-1
- Constant nodes are named by value
  - CONSTANT[0]-2
  - Third instance of the constant ‘0’

Naming (cont.)

- Wires are named a little differently
- Because it’s dataflow, each wire retrieves one terminal from one node
- Wire named after its source

CONSTANT[0]-2<_T_0>

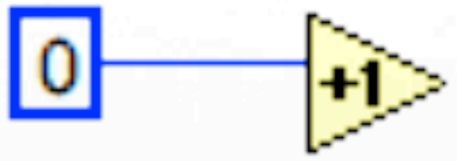
Naming (cont.)

- Diagram inputs are named by label
- Diagram structures are also named by label
- Function terminals are named by LabVIEW term-name field
- Output terminal of assertion diagrams is named `:ASN`

ACL2 Model

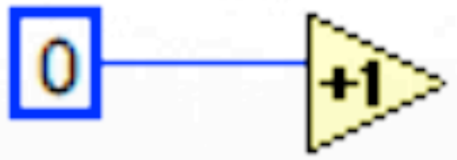
- Nodes have input and output terminals (wire ports)
- Each node takes a record (IN) as input
  - Returns output record
- Wires extract values from records

Translation



```
(DEFUN-N CONSTANT[0]-0 (IN)
           (S* :|_T_0| 0))

(DEFUN-W CONSTANT[0]-0<_T_0> (IN)
           (G :|_T_0| (CONSTANT[0]-0 IN)))

(DEFUN-N INCREMENT-0 (IN)
           (S* :X+1 (1+ (CONSTANT[0]-0<_T_0>
                                    IN))))
```

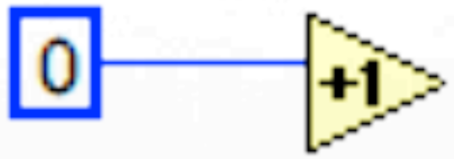
Translation



```
(DEFUN-N CONSTANT[0]-0 (IN)
           (S* :|_T_0| 0))

(DEFUN-W CONSTANT[0]-0<_T_0> (IN)
           (G :|_T_0| (CONSTANT[0]-0 IN)))

(DEFUN-N INCREMENT-0 (IN)
           (S* :X+1 (1+ (CONSTANT[0]-0<_T_0>
                          IN))))
```

- (S* :key1 val1 :key2 val2 ...) creates new record binding :keyi to vali (“set”)

Translation

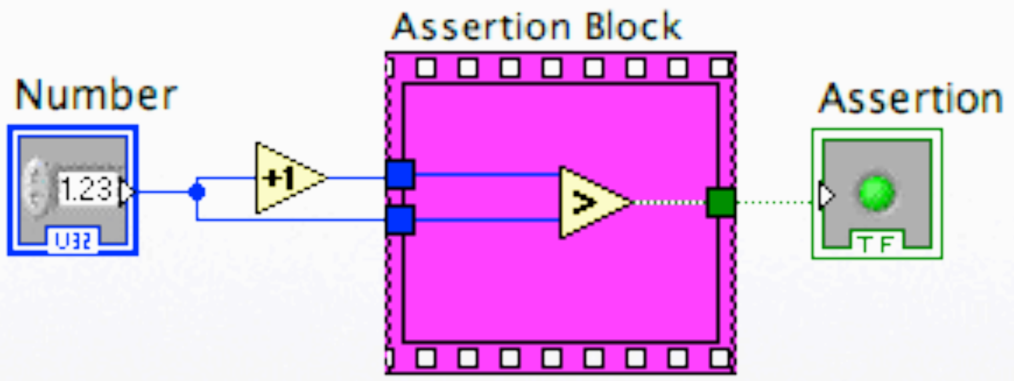


```
(DEFUN-N CONSTANT[0]-0 (IN)
         (S* :|_T_0| 0))

(DEFUN-W CONSTANT[0]-0<_T_0> (IN)
         (G :|_T_0| (CONSTANT[0]-0 IN)))

(DEFUN-N INCREMENT-0 (IN)
         (S* :X+1 (1+ (CONSTANT[0]-0<_T_0>
                        IN))))
```

- (S* :key1 val1 :key2 val2 ...) creates new record binding :keyi to vali ("set")
- (G :key rec) returns the value associated with :key in rec ("get")

Our Approach (cont.)

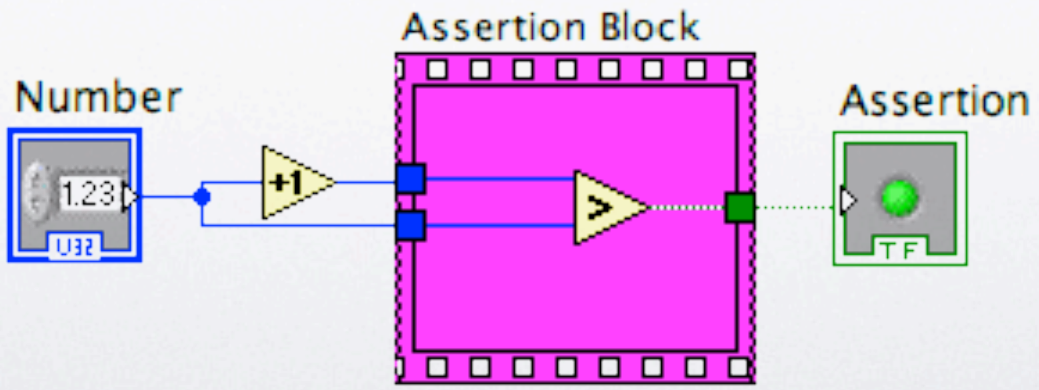
- Translate assertions into proof obligations



```
(DEFTHM ASSERTION-BLOCK-HOLDS
        (IMPLIES (AND (NATP (G :NUMBER IN)))
                 (G :ASN (ASSERTION-BLOCK IN))))
```

Caveats

- We use unbounded arithmetic, so this is a theorem for us, but not for LabVIEW/G
- We view this as verifying a slightly "idealized" form of LabVIEW/G



LabVIEW Loops

- We separate for-loop structures into 4 ACL2 functions
- $step function
  - Executes loop body and binds outputs to next iteration inputs

```
(DEFUN FOR-LOOP$STEP (IN)
  (S :|_T_4| (G :|_T_1| (|_N_5| IN)) IN))
```

LabVIEW Loops (cont.)

- $loop function
  - Compares loop counter to loop bound
  - Updates loop counter and calls $step fn

```
(DEFUN FOR-LOOP$LOOP (N IN)
(DECLARE (XARGS :MEASURE (NFIX (- N (G :LC IN)))))
(COND ((OR (>= (G :LC IN) N)
           (NOT (NATP N))
           (NOT (NATP (G :LC IN))))
         IN)
      (T (FOR-LOOP$LOOP N (S :LC (1+ (G :LC IN))
           (FOR-LOOP$STEP IN))))))
```

LabVIEW Loops (cont.)

- $init function
  - Binds loop variables to initial values

```
(DEFUN FOR-LOOP$LOOP$INIT (IN)
       (S* :LC 0
           :|_T_2| (CONSTANT[10]-1<_T_0> IN)
           :|_T_4| (CONSTANT[0]-0<_T_0> IN)))
```

LabVIEW Loops (cont.)

- Top function
  - Binds loop bound and calls $loop fn with results of $init fn

```
(DEFUN-N FOR-LOOP (IN)
  (FOR-LOOP-SRN$LOOP (CONSTANT[10]-1<_T_0> IN)
      (FOR-LOOP-SRN$LOOP$INIT IN)))
```

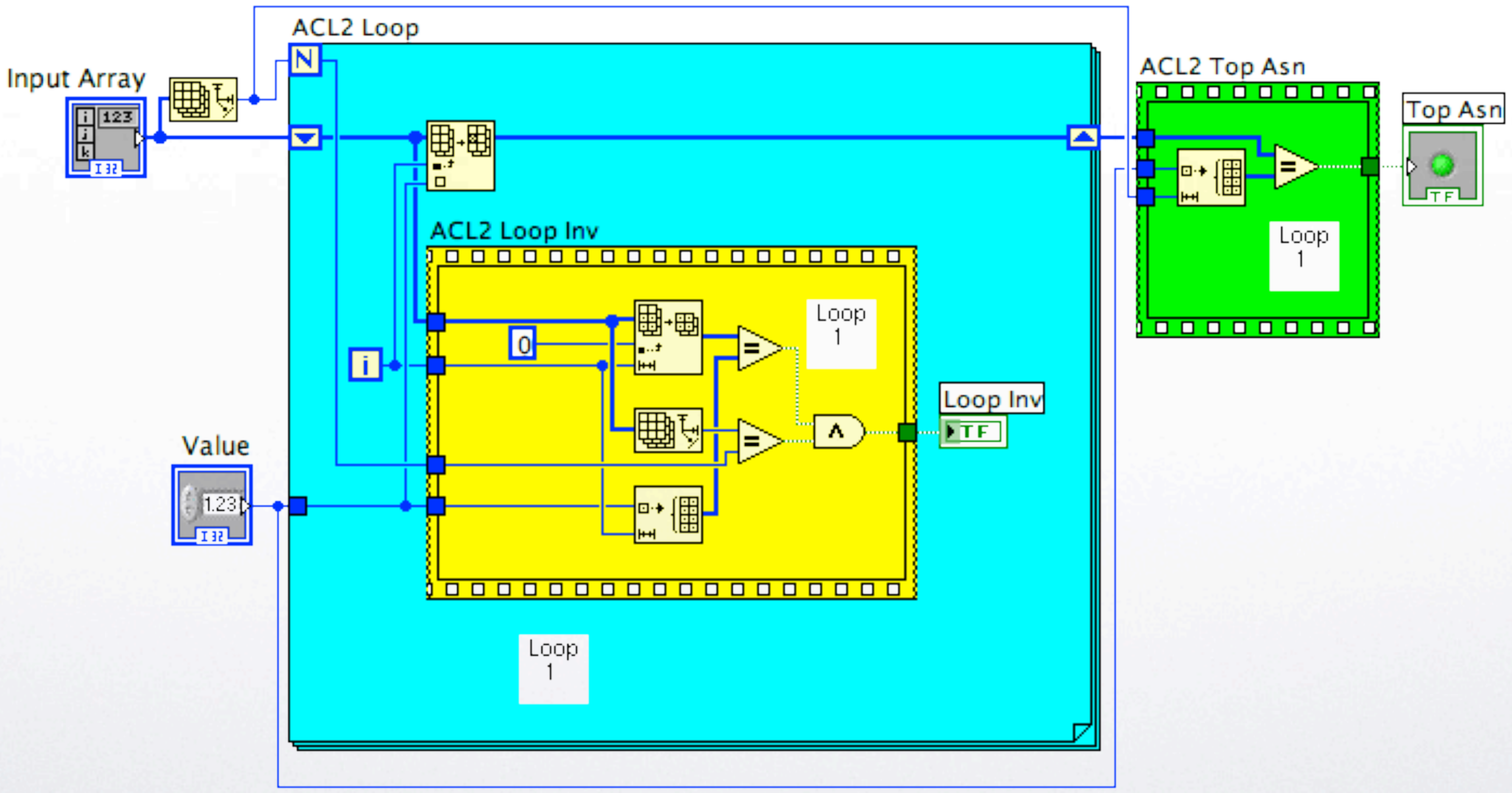
LabVIEW Structures

- LabVIEW loops are split into inner and outer structures
  - Inner structures are called "Self-reference Nodes" (SRN)
  - SRN nodes contain the body of the loop
  - Outer nodes map external values to internal names

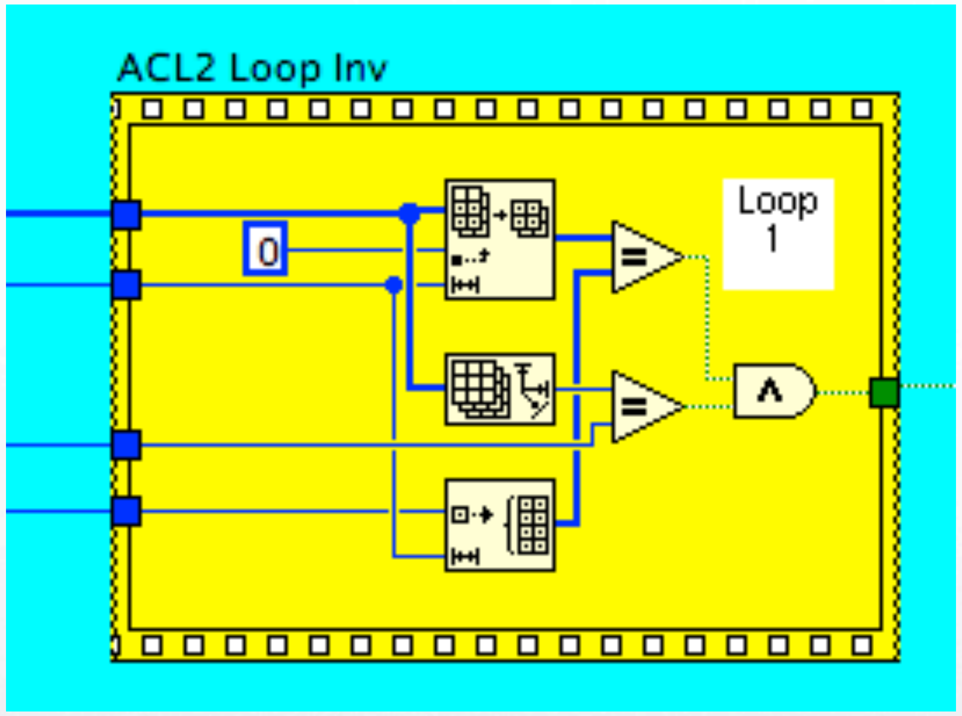
Top Loop Assertions

- Assertions about loops (in general) require inductive proofs
- We split loop assertions into “top” assertions and loop invariants

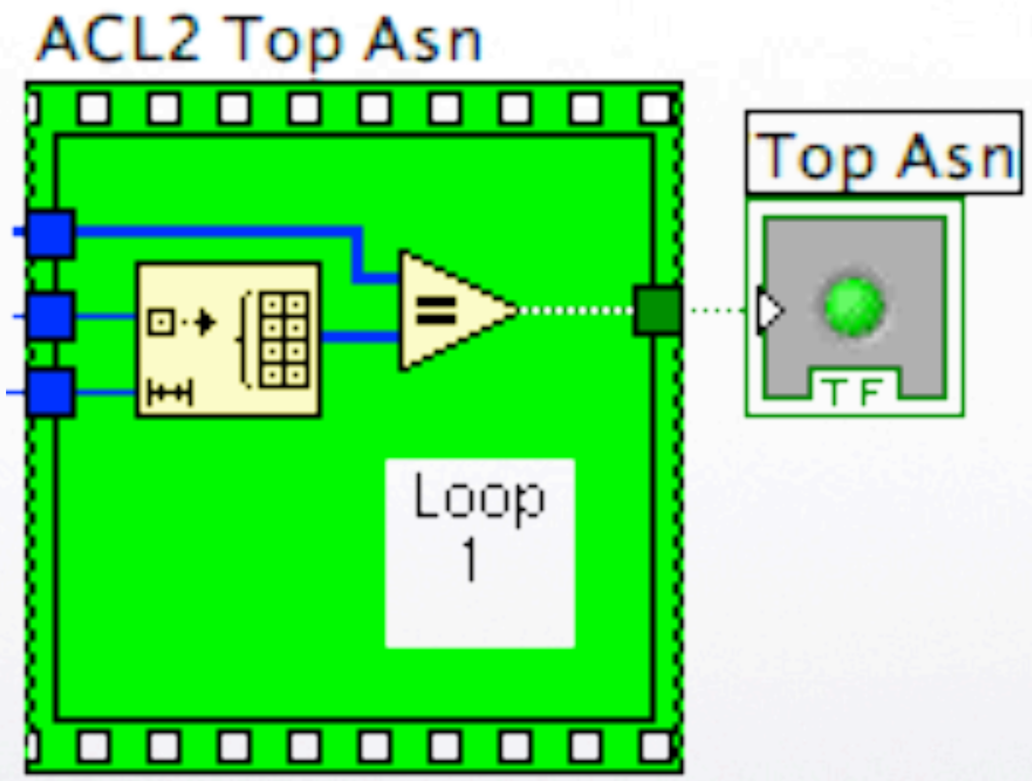
Loop Assertions (cont.)



Loop Invariant



Top Loop Assertion



Proving Loop Assertions

- Hold the user's hand to prove invariants
- Autogenerate highly structured proof scaffolding
- Strictly guide proof process by way of theory control

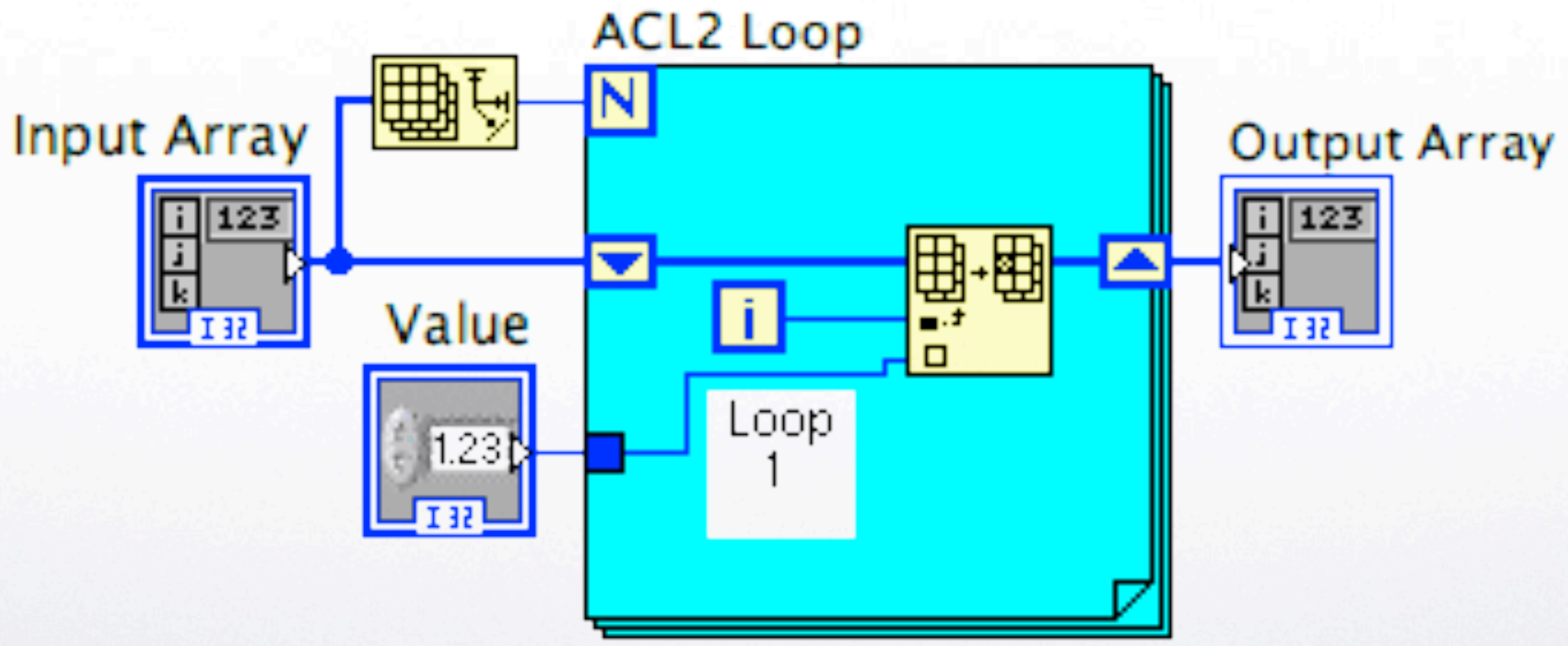
Proof Scaffolding

- Generate 13 lemmas, 6 predicates
- 4 lemmas potentially require user assistance
  - All other lemmas (should) be automatic
- Generated file has ~50 lines of comments
  - User assisted lemmas are marked

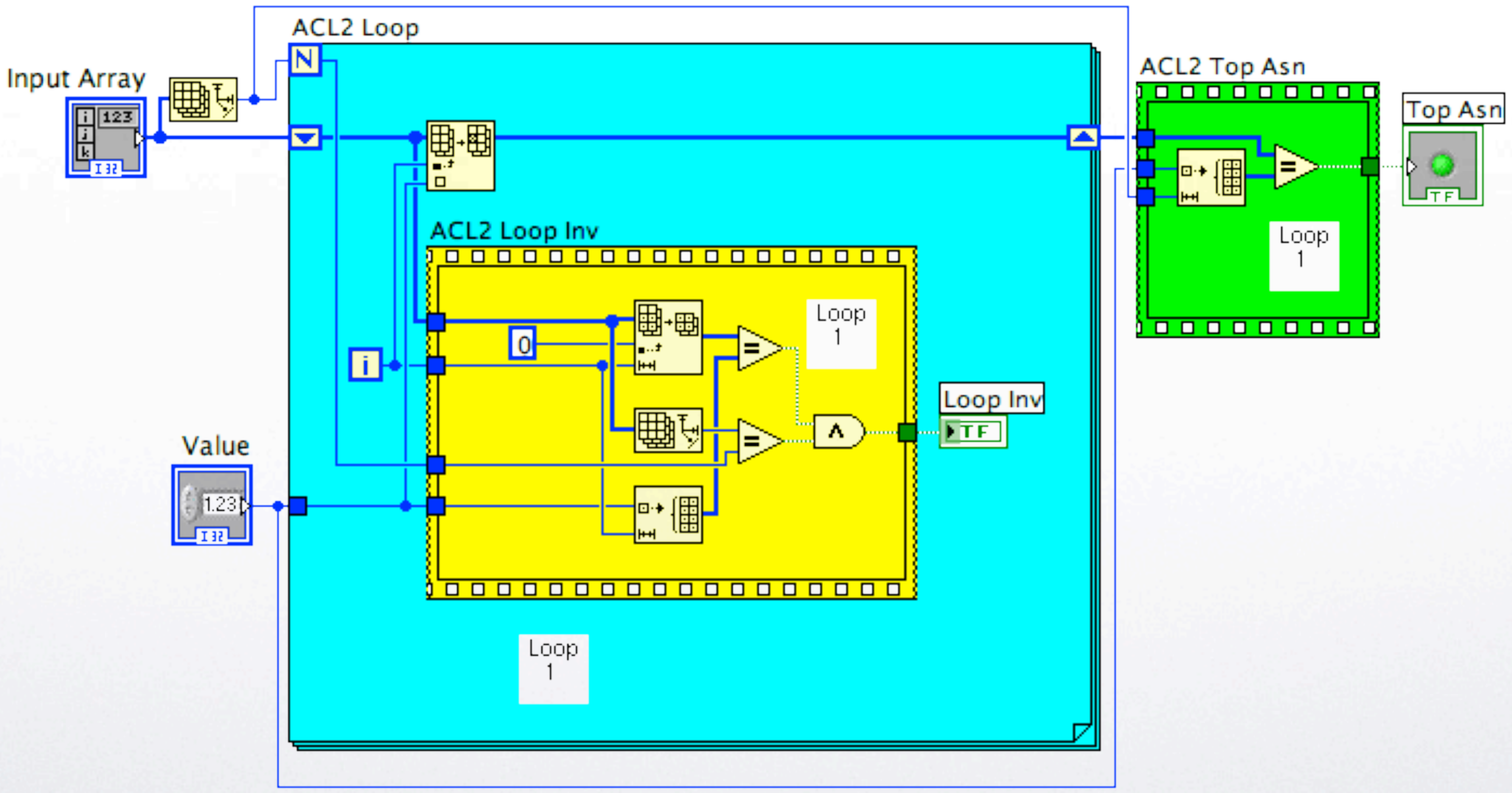
Generic Theory

- We use a generic theory to avoid induction in the invariant proof
  - Use encapsulate to define a generic $step, $loop
  - Prove that if $prop holds on entry to $loop and is preserved by $step then it holds when $loop is run

Example Diagram



Example Diagram



Our Goal

```
(DEFTHM ACL2-TOP-ASN$INV
  (IMPLIES (GAUSS$INPUT-HYPS IN)
           (G :ASN (ACL2-TOP-ASN IN))))
```

Our Goal

```
(DEFTHM ACL2-TOP-ASN$INV
  (IMPLIES (GAUSS$INPUT-HYPS IN)
           (G :ASN (ACL2-TOP-ASN IN))))
```

- But we can’t prove this immediately

Extend Loop Invariant

```
(DEFUN LOOP-INV-SRN$PROP (N IN)
  (DECLARE (IGNORABLE N))
  (AND (LOOP-INV-SRN$HYPS IN)
       (EQUAL N (G :|_T_2| IN))
       (G :ASN (ACL2-LOOP-INV IN))))
```

- LOOP-INV-SRN$HYPS is a type predicate that recognizes the types on the inputs to LOOP-INV-SRN
- ACL2-LOOP-INV is the name of the loop invariant

Loop Inv. is Preserved

```
(DEFTHMDL LOOP-INV-SRN$PROP{FOR-LOOP-SRN$STEP}
  (IMPLIES (AND (NATP (G :LC IN))
                (< (G :LC IN) N)
                (LOOP-INV-SRN$PROP N IN))
           (LOOP-INV-SRN$PROP N
            (S :LC (1+ (G :LC IN))
               (FOR-LOOP-SRN$STEP IN)))))
```

- `Defthmdl` is a macro for `(local (defthmd foo ...))`

Use Generic Theory

```
(DEFTHML LOOP-INV-SRN$PROP{FOR-LOOP-SRN}
  (IMPLIES (AND (NATP N)
                (NATP (G :LC IN))
                (LOOP-INV-SRN$PROP N IN))
           (LOOP-INV-SRN$PROP N (FOR-LOOP-SRN$LOOP N IN)))
  :HINTS
  (("Goal" :BY (:FUNCTIONAL-INSTANCE
                LOOP-GENERIC-THM
                (STEP-GENERIC FOR-LOOP-SRN$STEP)
                (PROP-GENERIC LOOP-INV-SRN$PROP)
                (LOOP-GENERIC FOR-LOOP-SRN$LOOP))
           :IN-THEORY
           (UNION-THEORIES '(LOOP-INV-SRN$PROP{FOR-LOOP-SRN$STEP})
                           (THEORY 'MINIMAL-THEORY))
           :EXPAND ((|FOR-LOOP-SRN$LOOP| N IN))))
  :RULE-CLASSES NIL)
```

Inv Holds on Input, with type hyps

```
(DEFTHML ACL2-LOOP-INV$INV{INIT}
  (IMPLIES (ACL2-LOOP-INV$INV{PRE} IN)
           (LOOP-INV-SRN$PROP (ARRAY-SIZE-0<SIZE(S)> IN)
            (LOOP-INV-SRN$PROP$INIT IN)))
  :RULE-CLASSES NIL)
```

Loop Inv. Holds w/o type hyps

```
(DEFTHML ACL2-LOOP-INV$INV
  (IMPLIES (ZERO-ARRAY$INPUT-HYPS IN)
           (ACL2-LOOP-INV$INV+ IN))
  :HINTS
  (("Goal"
    :IN-THEORY
    (UNION-THEORIES '(ACL2-LOOP-INV$INV{PRE})
                    (THEORY 'MINIMAL-THEORY))
    :USE (ACL2-LOOP-INV$INV$CONDITIONAL
          ACL2-LOOP-INV$INV{PRE}{HOLDS})))
  :RULE-CLASSES NIL)
```

Loop counter = Loop bound

```
(DEFTHML LC$FOR-LOOP-SRN
  (IMPLIES (AND (NATP N)
                (NATP (G :LC IN))
                (<= (G :LC IN) N))
           (EQUAL (G :LC (FOR-LOOP-SRN$LOOP N IN)) N))
  :HINTS (("Goal" :BY (:FUNCTIONAL-INSTANCE
                       LOOP-GENERIC-LC
                       (STEP-GENERIC FOR-LOOP-SRN$STEP)
                       (PROP-GENERIC LOOP-INV-SRN$PROP)
                       (LOOP-GENERIC FOR-LOOP-SRN$LOOP))
           :IN-THEORY (THEORY 'MINIMAL-THEORY)
           :EXPAND ((FOR-LOOP-SRN$LOOP N IN)))))
```

Top Inv. Holds

```
(DEFTHM ACL2-TOP-ASN$INV
  (IMPLIES (GAUSS$INPUT-HYPS IN)
           (G :ASN (ACL2-TOP-ASN IN)))
  :HINTS (("Goal" :IN-THEORY (DISABLE |FOR-LOOP-SRN$LOOP|)
           :USE (ACL2-LOOP-INV$INV
                 LEMMA-2-ACL2-LOOP))))
```

- Uses several small lemmas not shown here

Lemma Library

- Lemmas about LabVIEW primitives essential to automatic proofs
- Primitive definitions are disabled by default to (weakly) remove dependence upon defintions
- Currently ~80 theorems

Not Shown Here

- LabVIEW/G diagrams can use ACL2 functions for specifications
- Existential ghost variables for specifications
  - Pick-a-point strategies
- By-hand approach to compositionality

Future Work

- Compositional Verification
  - Initial Approach done by hand
  - Use encapsulate to export diagram properties
- Use bounded arithmetic
- Use encapsulate for primitive definitions
- Diagrams containing state

Conclusion

- Prototype system for verifying LabVIEW diagrams
- About a dozen (fully automatic) examples completed
- Feasibility of approach has been proven (for state-free diagrams)

Thanks

- J, for advising me these past 2 years
- Matt, for teaching me all about ACL2
- Jacob, JeffK and NI, for making this project possible
- My committee, the ACL2 seminar and Jessica, for helpful comments on the presentation and thesis
- My roommates, Yonatan and David Reaves, for all the support over the past couple of years