# Verifying Centaur's Floating Point Adder

Sol Swords
sswords@cs.utexas.edu

April 23, 2008

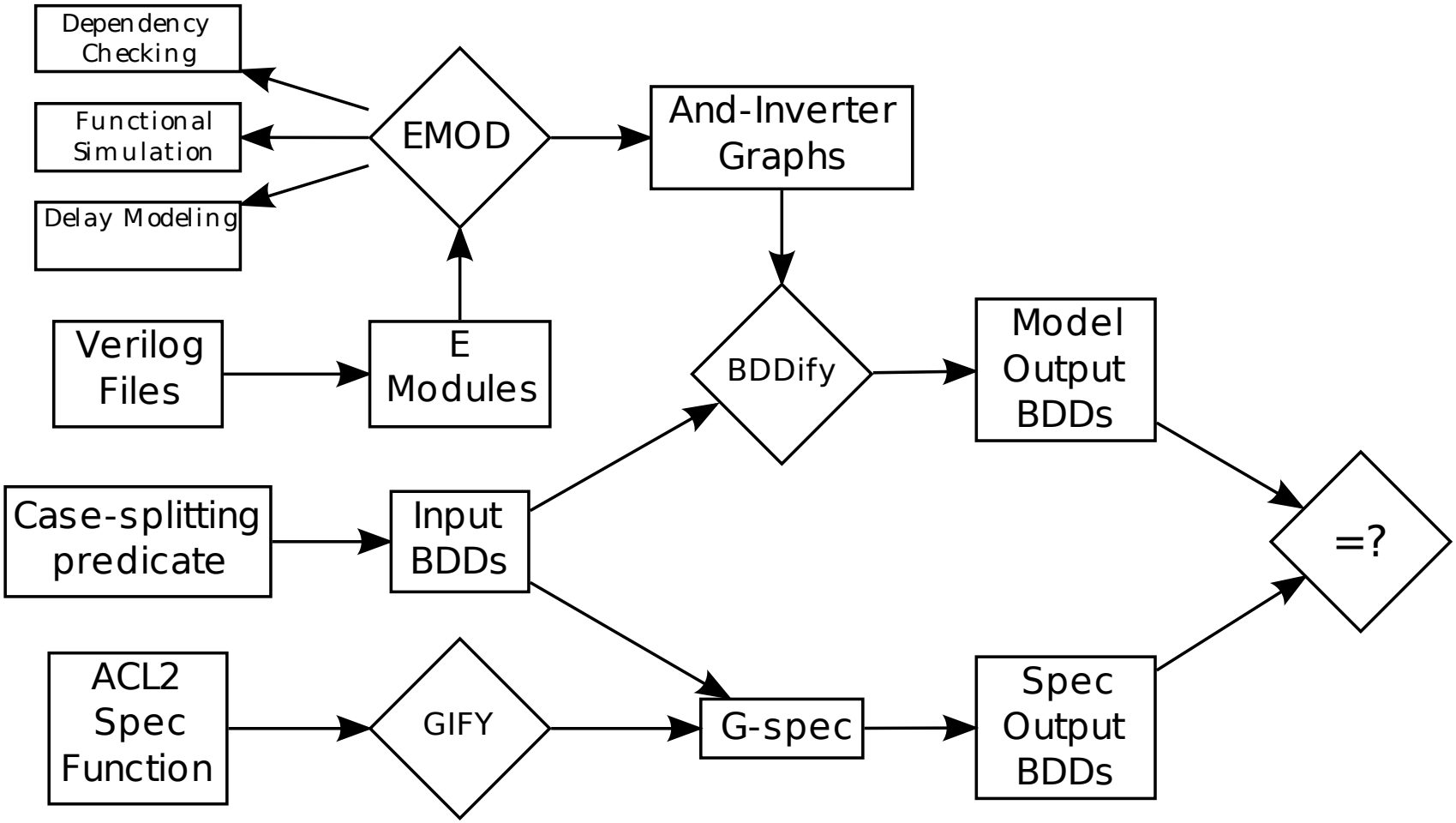
# Problem

Given:

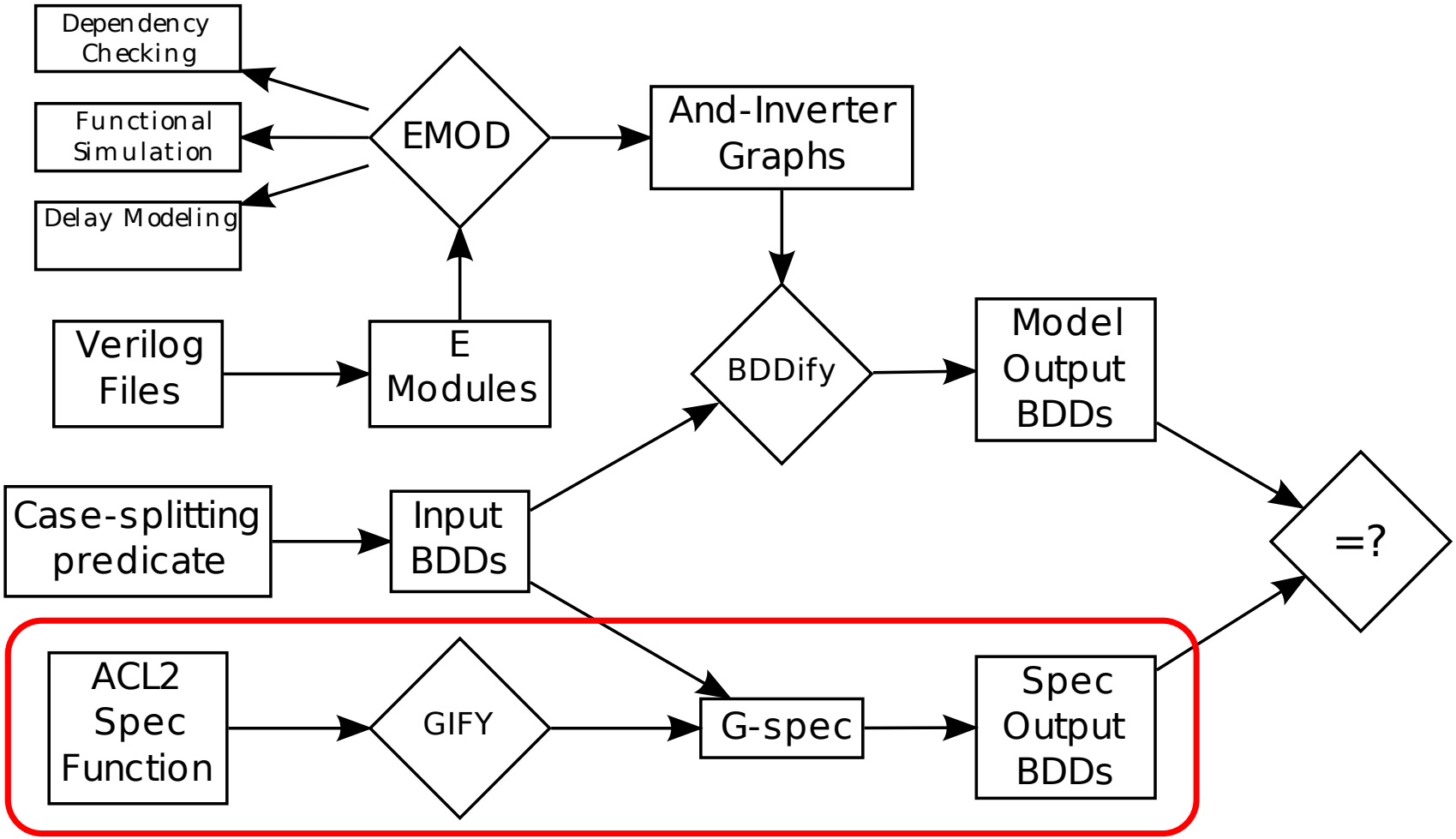
- Verilog RTL for the Centaur CN processor's FADD unit,
- Opcode and instructions for running a floating-point addition,
- An ACL2 specification function for floating point addition,

Prove, to the extent possible, that the design implements the spec.

# Overview


Dependency Checking
Functional Simulation
Delay Modeling
EMOD
And-Inverter Graphs
Verilog Files
E Modules
BDDify
Model Output BDDs
Case-splitting predicate
Input BDDs
=?
ACL2 Spec Function
GIFY
G-spec
Spec Output BDDs

# Spec to BDDs


Dependency Checking
Functional Simulation
Delay Modeling
EMOD
And-Inverter Graphs
Verilog Files
E Modules
BDDify
Model Output BDDs
Case-splitting predicate
Input BDDs
=?
ACL2 Spec Function
GIFY
G-spec
Spec Output BDDs

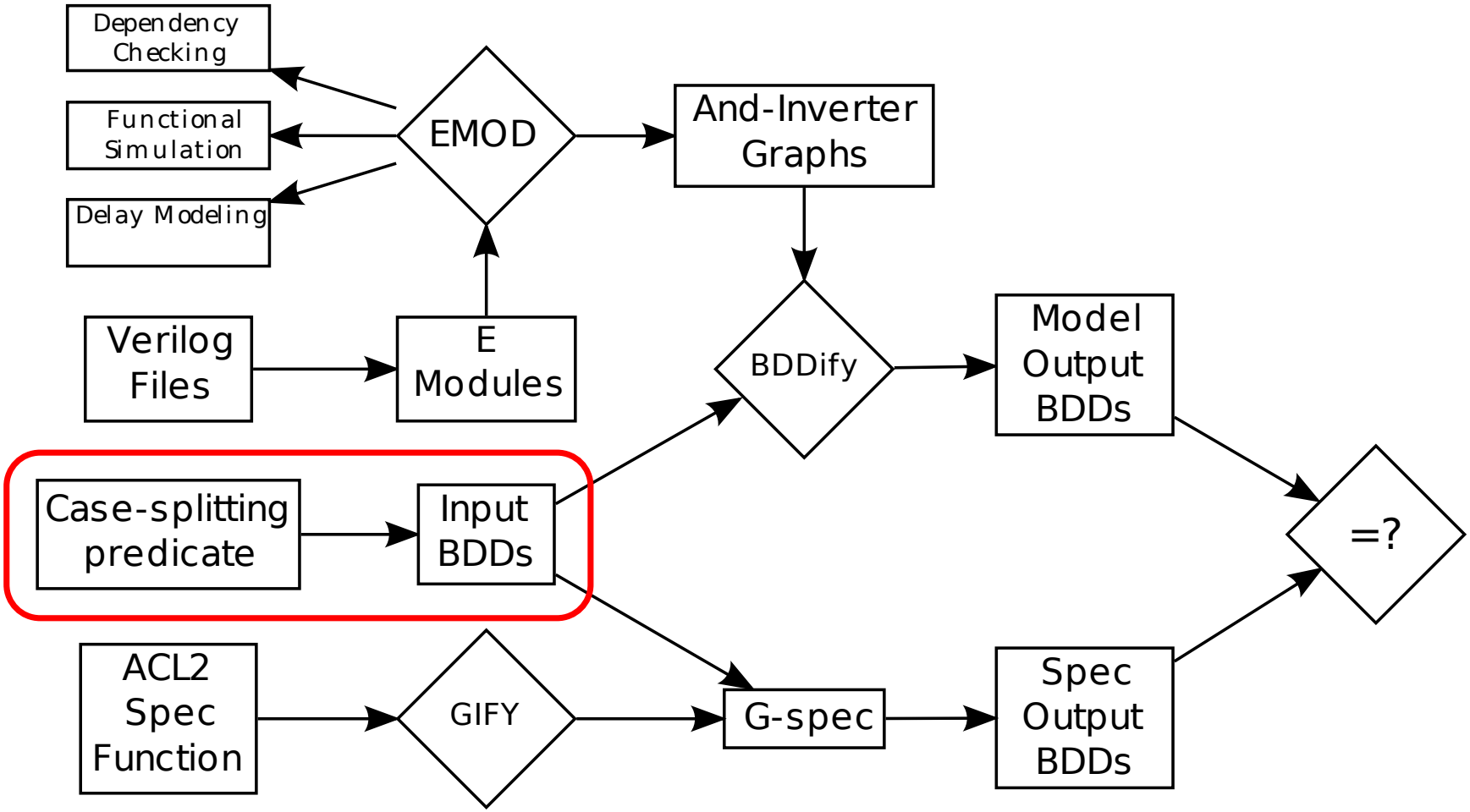
# Gify quick summary

- Want the spec represented as BDDs - Boolean functions, one for each output bit, over the bits of the input
- `(GIFY 'SPEC)` defines the (Common Lisp) function `G-SPEC` which now operates on symbolic objects.
- Approximate, hypothetical contract of a G-function:

```
(equal (eval-g (g-foo a b c) vals)
       (foo (eval-g a vals)
            (eval-g b vals)
            (eval-g c vals)))
```

where `EVAL-G` maps a symbolic object to a concrete object.

# Input BDD Generation

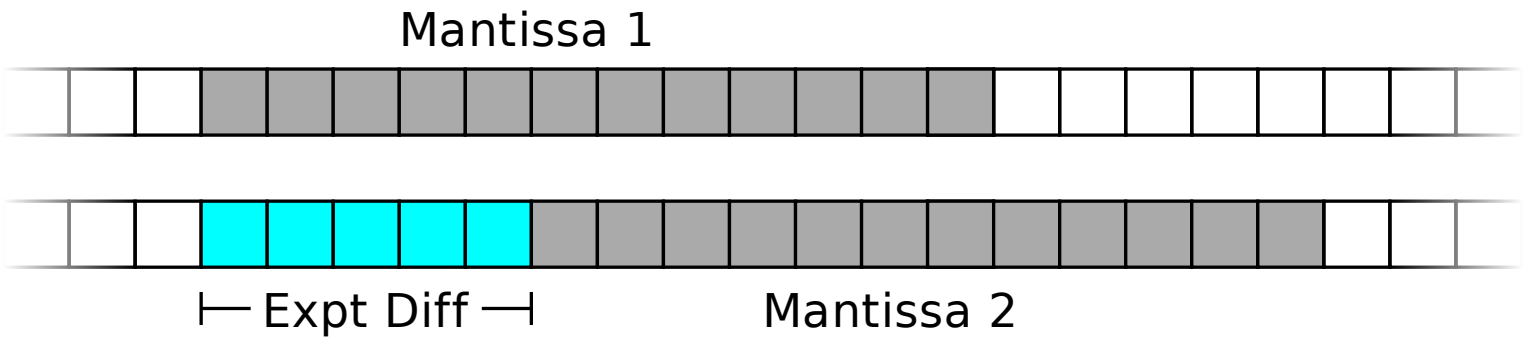


# Case Splitting

- BDDs for fully general FP addition are too big.
  - We have built them for the single-precision case: 2-4 hours computation, 20 million hash-conses. Not happening for double-precision.
- Case-splitting lets the BDD order be chosen for each case
  - Also makes it easier to eliminate irrelevant intermediate computations (more later.)

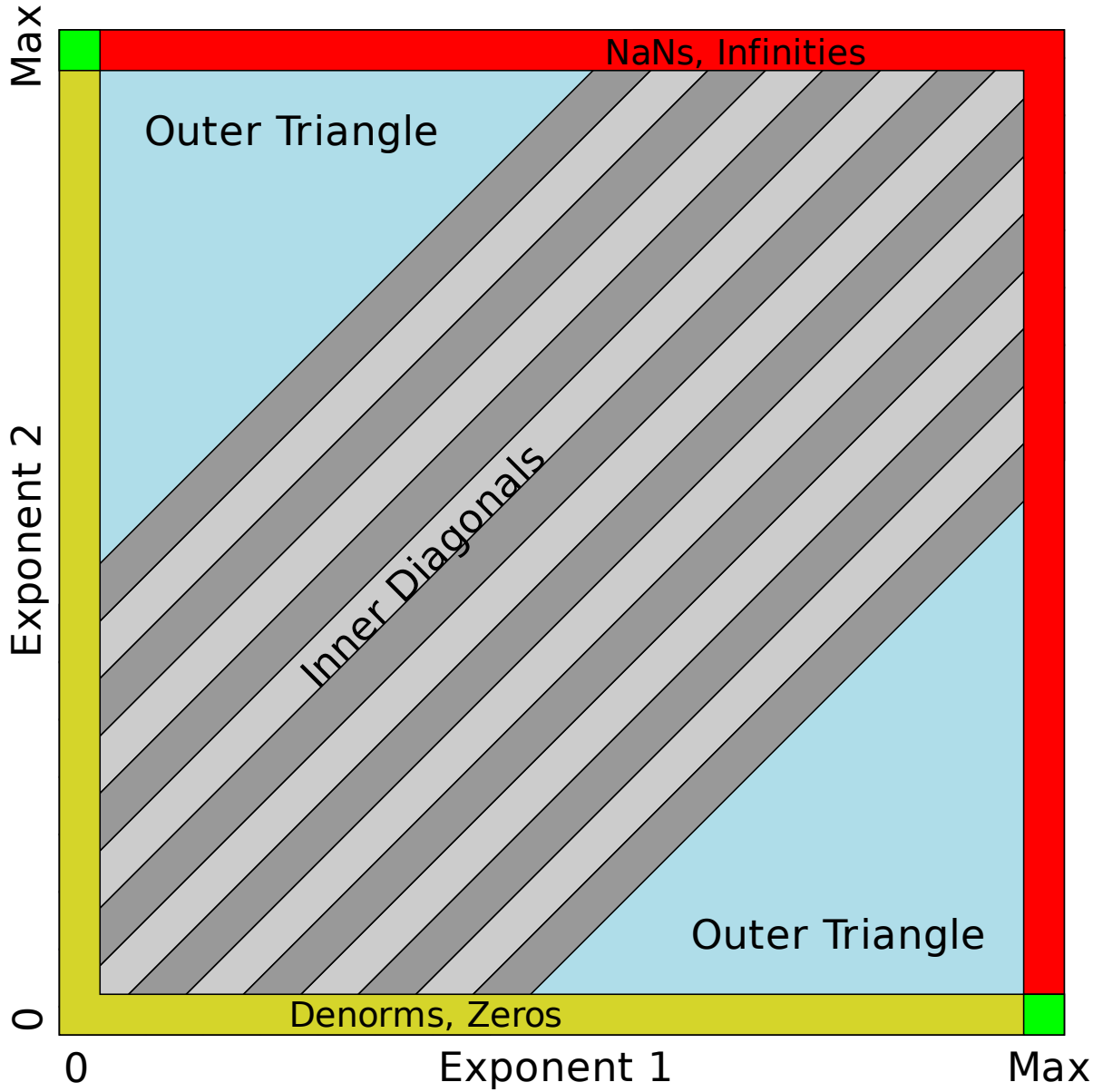
Where do we split?

# Split by Exponent Difference

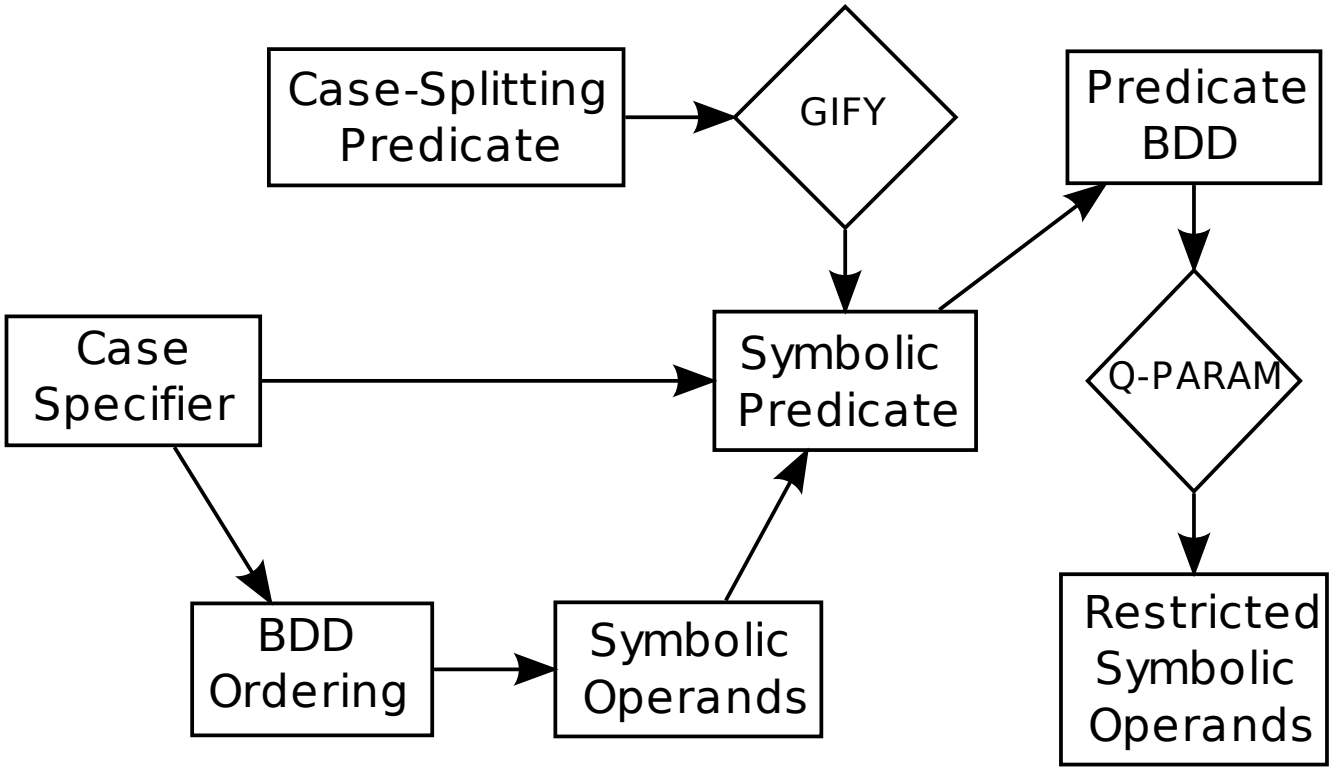



Why?

- Adding the same mantissas at the same exponent difference is the same addition operation
- Best BDD order for addition has bits in order of significance
- Separates Near Path from Far Path cases
- Can consolidate cases where mantissas don't overlap


Max
NaNs, Infinities
Outer Triangle
Exponent 2
Inner Diagonals
Outer Triangle
0
Denorms, Zeros
0
Exponent 1
Max

# Input generation detail



# Case-splitting Predicate

Define as an ACL2 function: `(ops-ok op1 op2 case)`

- `case` specifies which of the cases to accept
- Equals `t` if the operands fit that case, `nil` otherwise.
- Gify this function to get `g-ops-ok`
- Use the Gified function to get a BDD that shows when the symbolic operands satisfy the predicate.

# Parameterized Inputs

For the inputs to symbolic simulations we want symbolic values that

- Always satisfy the predicate
- Cover all possible inputs that satisfy the predicate.

Implemented by function `(Q-PARAM P N)`

- `P` - predicate BDD
- `N` - Number of variables to create parameterized values for
- `(Q-PARAM P N)` makes a list of `N` BDDs which
  - evaluate to values satisfying `P` for all variable settings
  - are general enough so that every set of values satisfying `P` can be generated.

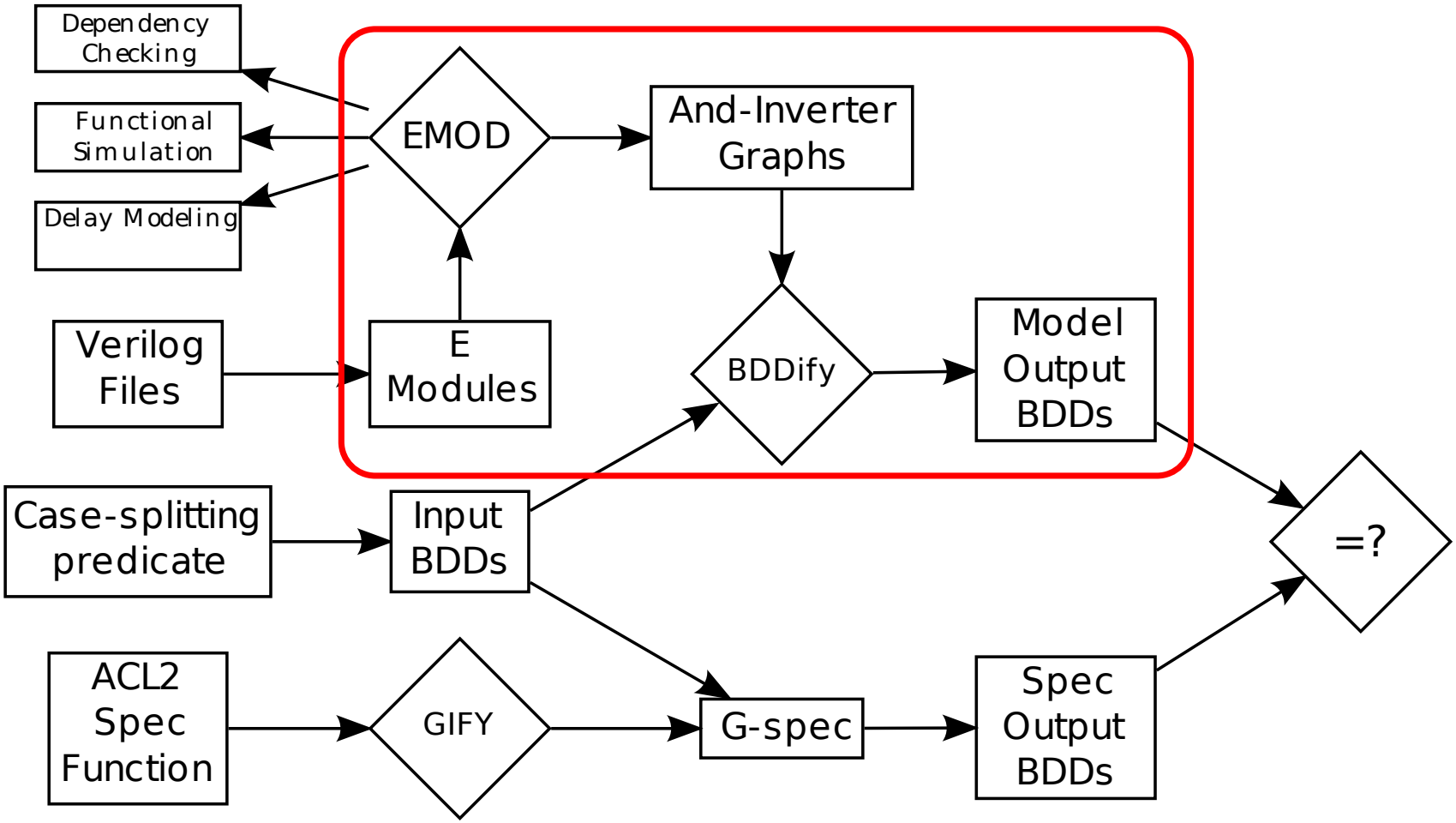
# Q-PARAM theorems: 1

```
(defthm forall-y-p-of-param-of-y-is-true
  (implies
   (and (normp p)                    ;; P is a BDD
        p                            ;; P is satisfiable
        ;; N is an integer
        ;; and is >= the number of variables used in P
        (integerp n)
        (<= (max-depth p) n))
   ;; Every case covered by (Q-PARAM P N) satisfies P.
   (equal (eval-bdd p
                    (eval-bdd-list (q-param p n)
                                   y))
          t)))
```

# Q-PARAM theorems: 2

```
(defthm exists-y-such-that-x-is-param-of-y
  (implies
   (and ;; X is a list of Booleans that satisfies P
        (boolean-listp x)
        (equal (eval-bdd p x) t)
        ;; X is long enough to cover all variables of P
        (<= (max-depth p) (len x)))
   ;; There exists Y for which (Q-PARAM P (LEN X))
   ;; evaluates to X.
   (let ((y (eval-bdd-list (q-param-inv p (len x)) x)))
     (equal (eval-bdd-list (q-param p (len x))
                           y)
            x))))
```

# Model to BDDs



# Model-side summary

- We read the model from Centaur's Verilog RTL (about 20,000 LOC.)
- Synthesize the Verilog to gates and translate the gates to E
- Results in an ACL2 `defconst` called `|*fadd*|`
- Run several cycles of
  `(emod 'faig |*fadd*|` $<$ *inputs* $>$ $<$ *state* $>$ `)`
  to get a pair of And-Inverter Graphs (AIGs) for each output bit.
- Using the BDD inputs generated by case-splitting, build the BDD for each AIG using an iterative process.

# AIG introduction

An AIG is a recursive data structure:

- Booleans: `T` and `NIL`
- Variables: non-Boolean atoms
- Negation of an AIG $x$: `(CONS X NIL)`
- Conjunction of AIGs $x$ and $y$: `(CONS X Y)`

# AIG to BDD, simple algorithm

Given an assignment of BDDs to the variables present in an AIG, make an equivalent BDD:

```
(defn aig-to-bdd (x al)
  (cond ((booleanp x)        ;; Boolean
         x)
        ((atom x)            ;; Variable
         (cdr (hons-get x al)))
        ((eq (cdr x) nil)    ;; Negation
         (q-not (aig-to-bdd (car x) al)))
        (t                   ;; Conjunction
         (q-and (aig-to-bdd (car x) al)
                (aig-to-bdd (cdr x) al)))))
```

- Too inefficient, can't use.

# AIG to BDD, practical approach

At a conjunction node $A \wedge B$, suppose $A$ can be cheaply translated into a BDD but $B$ cannot.
Observation: We may not need to BDDify $B$ in order to BDDify $A \wedge B$.

- If BDDify$(A) = \texttt{NIL}$, then BDDify$(A \wedge B) = \texttt{NIL}$.
- More generally, if $A$ is never true when $B$ is false, then BDDify$(A \wedge B) =$ BDDify$(A)$.
- Strategy: Set an upper bound on the size of BDDs to work on. If we can detect the above situation before fully BDDifying $B$, we win. Otherwise, may need to increase the upper bound and try again.

# AIG to BDD, practical approach

- Use one of two strategies for dealing with too-big BDDs:
  - More conservative, faster: Use pairs of BDDs to represent upper and lower bounds of the true BDD values. Set the upper bound to `T` or the lower bound to `NIL` when too big.
  - Less conservative, slower: Associate each too-large BDD created with a fresh BDD variable.
- In either case, we may sometimes prune the AIG even when we have no exact BDD results.
- Have ACL2 proofs that both approaches are sound.
- Our approach: Alternate between the two strategies while iteratively increasing the size limit until an exact result is reached.

# Results

- Works pretty well:
  - Single-precision: 8 minutes to verify
  - Double/extended precision: 1 hour each
- Future directions:
  - Fight our way toward an ACL2 theorem.
    - Prove that we really have a proof.
    - Need a logical story for Gification.
  - Adapt the approach to other kinds of hardware.
    - Need to decompose the problem in other ways than by case-splitting on the inputs.