A Mechanically Verified AIG-to-BDD Conversion Algorithm

Sol Swords and Warren A. Hunt, Jr.
{sswords,hunt}@cs.utexas.edu

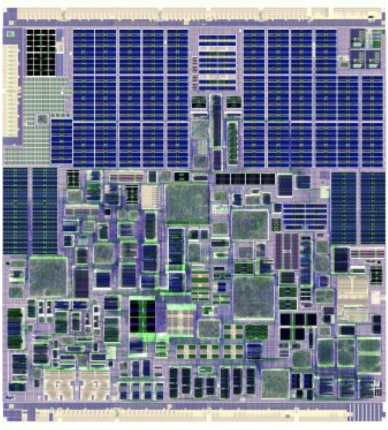
The University of Texas at Austin

Centaur Technology, Inc.

July 12, 2010

Overview

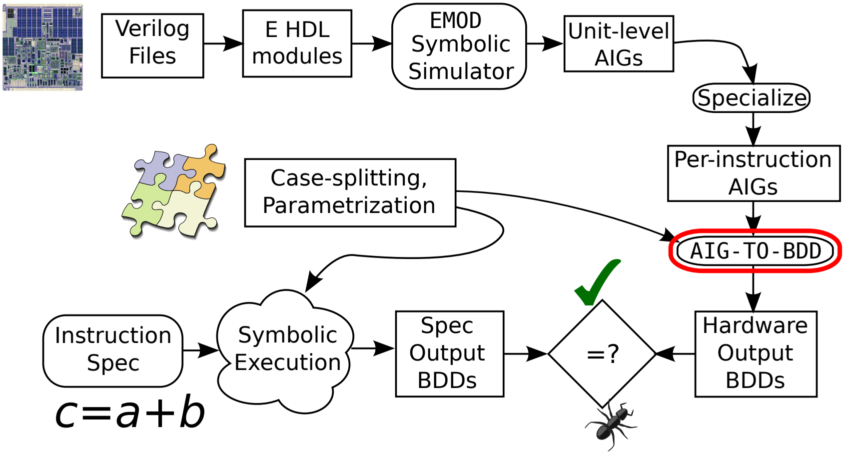
We have implemented and verified in ACL2 an algorithm `AIG-TO-BDD` that computes a BDD representation from an And/Inverter graph (AIG).



Part of a hardware verification flow used at Centaur Technology.

- Uses automated Boolean reasoning to check hardware designs against ACL2 specs.
- Produces ACL2 theorems as the end result.
- Successful application on many operations including floating point addition, multiplication.

Toolflow



Verilog Files
E HDL modules
EMOD Symbolic Simulator
Unit-level AIGs
Specialize
Per-instruction AIGs
Case-splitting, Parametrization
AIG-TO-BDD
Instruction Spec
c=a+b
Symbolic Execution
Spec Output BDDs
=?
Hardware Output BDDs

Sample theorem

```
(implies (and (32-bitsp a)
              (32-bitsp b))
         (equal (fp+-hardware a b)
                (fp+-spec a b)))
```

- This theorem has nothing to do with BDDs or AIGs.
- Proof by reflective procedures — little "conventional theorem proving."
- Conventional theorem proving used to show soundness of these proof procedures.

Sample theorem: Method

```
(implies (and (32-bitsp a)
              (32-bitsp b))
         (equal (fp+-hardware a b)
                (fp+-spec a b)))
```

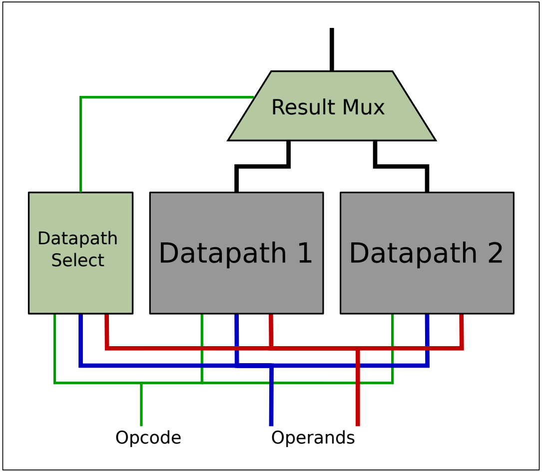
Strategy for proving the theorem:

1. Assign independent BDD variables to each input bit of a, b.
2. *Symbolically execute* `fp+-hardware` and `fp+-spec` on these symbolic inputs, obtaining BDDs representing the bits of the results.
3. Compare results for equality to finish the proof.

(Symbolic execution framework described elsewhere.)

Problematic Situation

Suppose datapaths 1 and 2 require different BDD variable orderings.



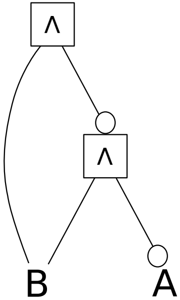
- BDDs blow up if we build both using a single variable ordering.
- Case-splitting strategy: restrict inputs so that select signal is constant.
- But naive symbolic simulation still constructs BDDs for both datapaths.
- AIG to BDD conversion prunes away irrelevant pieces of the hardware.

AIGs as intermediate representation

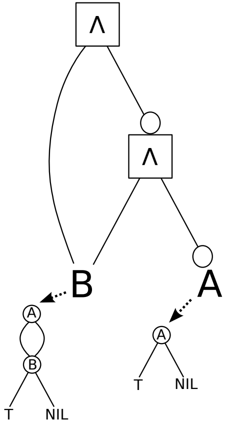
Could compute BDDs directly from E HDL representation, but using AIGs as an intermediate representation has several advantages:

- Easy to build from HDL
- Compact (linear in circuit size)
- Relatively simple data structure – constant, variable, negation, or conjunction
- No names for internal nodes
- **Much simpler to manipulate algorithmically than E!**

Example AIG to BDD Conversion

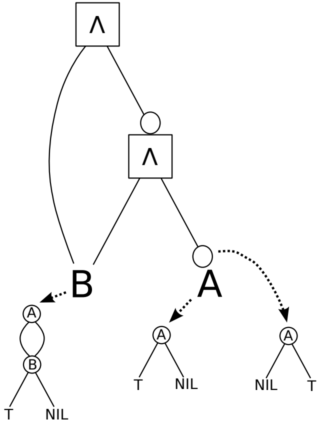


Example AIG to BDD Conversion



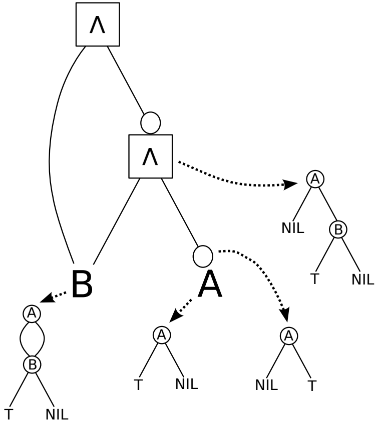
- Assign BDDs to variables

Example AIG to BDD Conversion



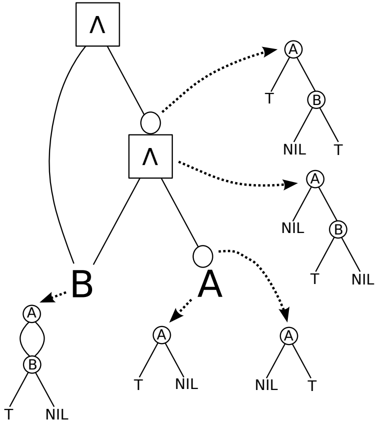
- Assign BDDs to variables
- Negate on INV nodes

Example AIG to BDD Conversion



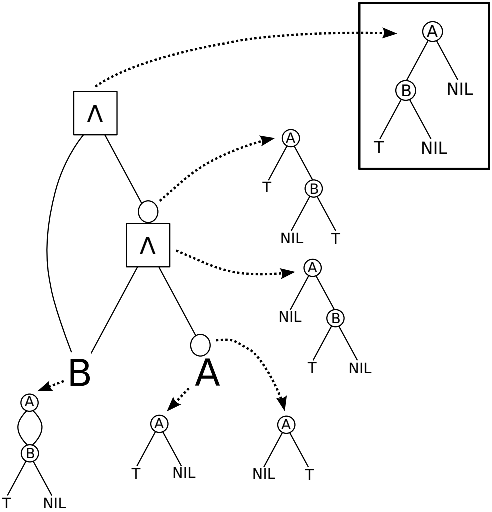
- Assign BDDs to variables
- Negate on INV nodes
- AND on AND nodes

Example AIG to BDD Conversion



- Assign BDDs to variables
- Negate on INV nodes
- AND on AND nodes
- Etc.

Example AIG to BDD Conversion



- Assign BDDs to variables
- Negate on INV nodes
- AND on AND nodes
- Etc.

Naive Algorithm

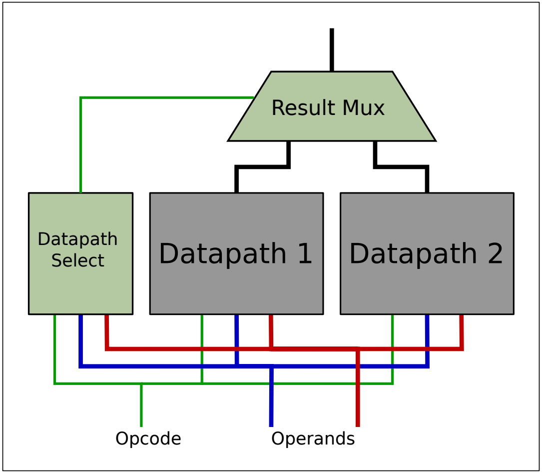
Simple algorithm that satisfies our specification is (`A2B` *x avt*), defined as:

- If *x* is a constant, return *x*
- If *x* is a variable, return (`CDR` (`ASSOC` *x avt*))
- If *x* is an AND node with children *a*, *b*, return (`BDD-AND` (`A2B` *a avt*) (`A2B` *b avt*))
- If *x* is an INV node with child *y*, return (`BDD-NOT` (`A2B` *y avt*)).

Easy to verify. Inefficient in same cases as before. Blindly builds a fully accurate BDD for every node in *x*.

Improved Strategy

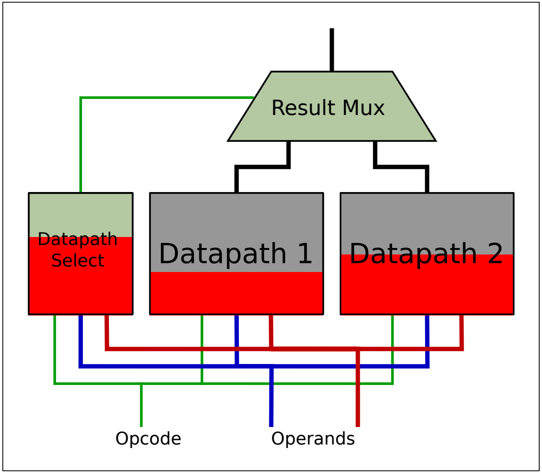
Suppose we select datapath 2, choose appropriate BDD ordering.



- Incrementally produce BDDs, starting with small *size limit*

Improved Strategy

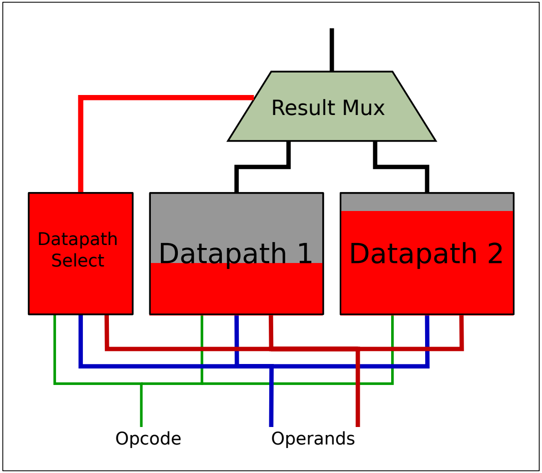
Suppose we select datapath 2, choose appropriate BDD ordering.



- Incrementally produce BDDs, starting with small *size limit*
- Increase size limit on each iteration

Improved Strategy

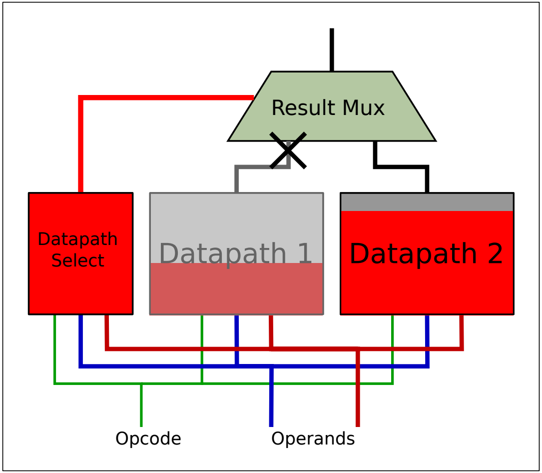
Suppose we select datapath 2, choose appropriate BDD ordering.



- Incrementally produce BDDs, starting with small *size limit*
- Increase size limit on each iteration
- Prune AIG using intermediate results

Improved Strategy

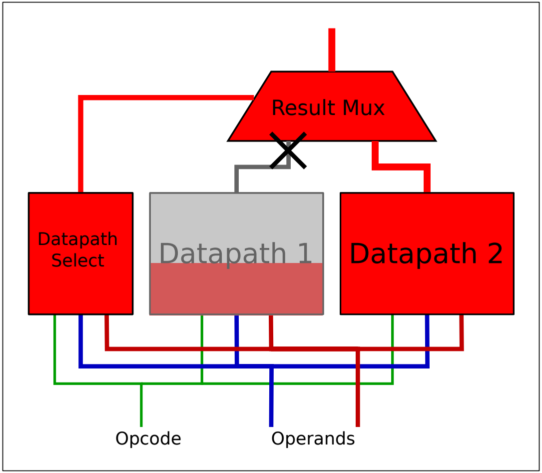
Suppose we select datapath 2, choose appropriate BDD ordering.



- Incrementally produce BDDs, starting with small *size limit*
- Increase size limit on each iteration
- Prune AIG using intermediate results

Improved Strategy

Suppose we select datapath 2, choose appropriate BDD ordering.



- Incrementally produce BDDs, starting with small *size limit*
- Increase size limit on each iteration
- Prune AIG using intermediate results
- Iterate until exact answer is produced.

When Size Limit Is Reached

Don't have to give up completely when BDD size limit is reached.

Bounding method. Track upper/lower bound BDDs, and replace oversized bound with TRUE if upper/FALSE if lower.

- Cheaper, loses a lot of information
- Example: $a \vee (b \wedge a)$ reduces to a even if b is expensive to compute

Variable substitution method. Replace oversized BDDs with fresh variables.

- More expensive, loses less information.
- Example: $b \wedge \ldots \wedge \neg b$ reduces to FALSE even if b is expensive to compute.

Each iteration involves a choice of BDD size limit and one of these two methods.

Top-level Algorithm

(`AIG-TO-BDD` *x avt steps*) $\longrightarrow$ (*success bdd aig*)

x: AIG to be converted

avt: table mapping AIG variables to BDDs

steps: list of pairs (*method*, *limit*) giving the sequence of iterations

success: true if the sequence of iterations yielded an exact result

bdd: the BDD result, equal to (`A2B` *x avt*) if successful

aig: simplified AIG equivalent to *x* under composition with *avt*, even if not successful.

Loop over *steps* building BDDs with the given *method*, *limit*. Update *x* as it gets pruned. Stop when an exact BDD result is obtained or *steps* runs out.

Memoization & Bookkeeping

- Memoize between and within iterations. Three memo tables:
  - *bmemo*: inexact results for bounding method, discarded after each iteration
  - *smemo*: inexact results for substitution method, discarded after each iteration
  - *fmemo*: exact results for both methods, preserved between iterations.
- Additional bookkeeping:
  - *bvt*: mapping from oversize BDDs to new variables for substitution method, discarded after each iteration.

Invariants

Memoization tables must contain accurate entries:

- *fmemo* maps AIGs x to exact BDDs (`A2B` x *avt*)
- *bmemo* maps AIGs x to upper/lower bound BDDs
- *smemo* maps AIGs x to BDDs that are equivalent under the substitutions in *bvt* to the exact BDD (`A2B` x *avt*)

Must be proven within one induction:

- *fmemo*, *bmemo* invariants and correctness of bounding method
- *fmemo*, *smemo* invariants, well-formedness of *bvt*, and correctness of substitution method

Verification Result

Final correctness theorem: If

$$(\mathit{success}\ \mathit{bdd}\ \mathit{aig}) = (\texttt{AIG-TO-BDD}\ x\ \mathit{avt}\ \mathit{steps}),$$

then

- If *success*, then $\mathit{bdd} = (\texttt{A2B}\ x\ \mathit{avt})$,
- $(\texttt{A2B}\ \mathit{aig}\ \mathit{avt}) = (\texttt{A2B}\ x\ \mathit{avt})$ regardless of *success*.

Conclusions

- AIG-TO-BDD statistics:
  - Implementation: 20 definitions, 450 lines.
  - Verification: 24 additional definitions, 160 lemmas, 2350 lines.
- Part of effective verification strategy. Example: extended-precision FP addition verified in ~1 CPU hour
- Verified BDD and AIG operations, AIG-TO-BDD algorithm, symbolic execution engine, ...
- **Flow results in full-fledged ACL2 theorems** ensuring that we really prove what was intended.

Questions?