Verification of Prefix Adder Circuits Using HOL

Steve Barrus
CS 6110

4th May 2005

Abstract

Prefix adder circuits are designed for speed at the expense of size and complexity. These circuits can be difficult to verify for correctness because they are so complex. Many large circuit suffer from this same problem. In this project, we explore the use HOL to verify correctness of two prefix adder circuits, a Ladner-Fischer adder and a Brent-Kung adder. For this task, HOL was used to prove netlist equivalence between each of the prefix adders and a simple ripple-carry adder circuit for the 16-bit and 32-bit cases. If the netlists are equivalent to a ripple carry adder, the prefix adder circuits are assumed to be correct. This project focuses on proving logical netlist equivalence. The results show that this approach is feasible even for large 32-bit adder circuits, much more so than exhaustive simulations.

Introduction

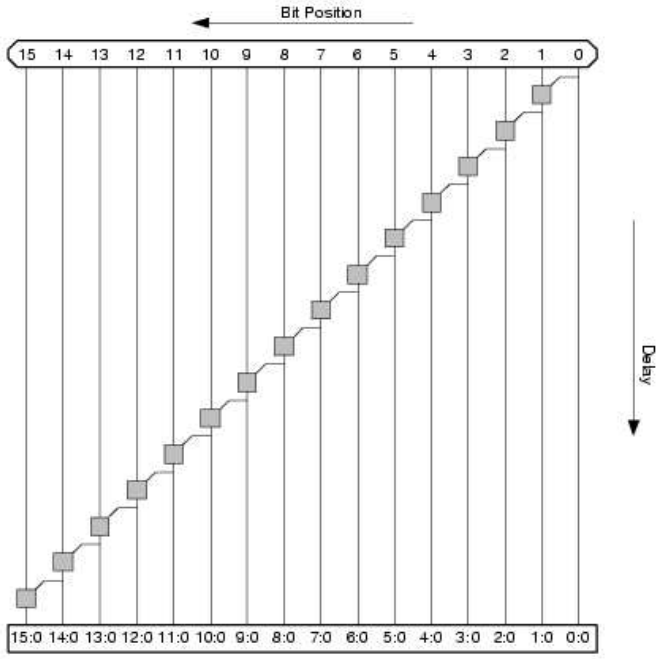
Addition of binary numbers is very common task for computers, calculators and other similar devices. A considerable effort has been made to make addition of twos-complement binary numbers go as fast as possible. An example of this effort can be seen by look at the wide variety of prefix adder circuits that have been developed. These include circuits like the Skylansky, Knowles, Kogge-Stone, Han-Carlson, Brent-Kung and Lander-Fischer adders. The designs of these circuits, like all circuits, have to be have to be checked for correctness before they can be used in a commercial product. Correctness is often checked by running simulations on the design using various circuit design tools. Exhaustive simulations are one way of showing complete design correctness, but exhaustive simulations can not realistically be done on circuits with a large number of inputs. 32-bit adder circuits have 65 inputs (64 plus carry-in) which means that there are 2^{65} possible test cases and not all of those test cases can be checked in a reasonable amount of time. There is also the issue of knowing what the correct results should be. A common way of checking results is to compare the output from the device under test (DUT) with the output of a known good circuit using the same input. For example, if you wanted to check the results of an adder, you would probably compare it to the output of the ADD instruction on the computer running the simulation. If all cases are tested, the circuits are

functionally equivalent. The is the same as showing that the netlist of the DUT logical equivalent to the device producing the results for comparison.

In this project, we will consider the Brent-Kung and Ladner-Fischer adders to be the devices under test and a straight-forward ripple-carry adder circuit to be the devices to be compared against. Cases for both 16-bit and 32-bit inputs were considered and shown to have logically equivalent netlists. While this project only explores using this method only for adder circuits, it could potentially be applied to all types of combination digital circuits. These adders circuits were chosen as interesting test cases for this project because binary addition is such a common operation and an important one to get right.

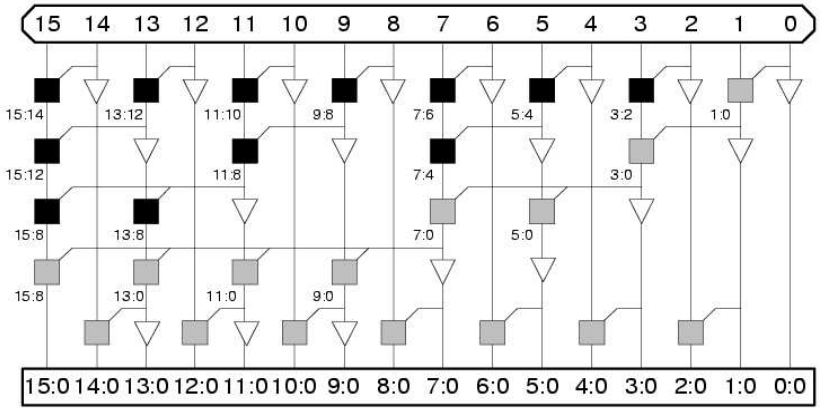
Background

Adder circuits are can be slow because it tasks so much time to generate the final carry signal. The final carry signal depends of the carry from the previous stage and that carry depends on the one before that and so on. Because of this inherent dependency, ripple-carry adders take linear time to produce the final results. Prefix adders, also called tree adders, are designed to take this linear carry network and make it logarithmic. Here is a diagram of a simple carry network in a 16-bit ripple carry adder.



This diagram shows just the carry chain. It doesn't include the propagate/generate signaling at the top and the XOR array at the bottom to produce the final sum. The gray boxes are special gates used specifically for carry generation in tree adders. This particular carry chain has an obvious linear structure.

If you compare it to the diagram of a Ladner-Fischer structure, you can see that a ripple-carry circuit is not the most efficient way to generate the carry signals.



This diagram shows a much more complex, but faster structure for generating all of the carry signals in fewer stages than the ripple carry. The Brent-Kung adder (not shown) has a similar structure to the Lander-Fischer adder. These diagrams can be translated to circuit schematics or Verilog code to produce an adder circuit. Similar diagrams were used to make the circuits used in this project.

Project Details

This section outlines the steps taken in this project to produce the final set of circuit proofs. It explains the how the circuits were modeled, the HOL definitions involved, and the methods used to prove the goals.

Modeling Circuits in HOL

Using HOL, a digital circuit can be modeled by considering power (`Vdd`) to be the boolean value *true* and ground (`gnd`) to be the boolean value *false*. Transistors can be modeled using implication, negation and equality. A PMOS transistor can be logically modeled as `~g ==> (d = s)` and and NMOS transistor is modeled as `g ==> (d = s)`. This translates to the source and drain being connected based on the value at the gate. This captures how a real MOS-FET works and it gives the same logical behavior that is seen in circuit simulators like Verilog-XL. Once you have PMOS transistors, NMOS transistors, power and ground, you then have all of the basic building blocks for any combinational digital circuit.

This project was started out by building a library of logic gates, or cells, out of the previously mentioned models for NMOS and PMOS transistors. For example an inverter was modeled as

```
INV_STR(a,y) = ?n1 n2.  VDD(n1) /\ GND(n2) /\ NMOS(a,n2,y) /\
                        PMOS(a,n1,y)
```

This is considered the structural model constructed from the basic building blocks and closely resembles a netlist for an inverted constructed in a circuit design tool. The terms `a` and `y` represent the input/output pins of the circuit and existentially quantified variables `n1` and `n2` represent the internal wires. For each cell, a behavior model was also constructed and the two were proved to be logically equivalent with HOL. The behavior model for an inverter shown below.

```
INV_BEH(a,y) = (y = ~a)
```

This can be proved to be equivalent by the following HOL statement where `NET_RW_TAC` is a rewrite tactic that automatically includes transistor definitions as well as power and ground.

```
prove (“INV_STR(a,y) = INV_BEH(a,y)“, NET_RW_TAC
  [INV_BEH_DEF,INV_STR_DEF] THEN METIS_TAC []);
```

Having both a behavioral and structural definition of each primitive cell not only shows correctness of the cell, but also provides more to work with when expanding modules that use these cells. It can save effort to uses the simpler behavioral definition at times instead of the larger structural definition (as seen later on). This was done for all of the following cells: INV, NAND, AND, NOR, OR, XOR, GRAY, BLACK, and FULLADDER. Once the cells were modeled and verified, there was a foundation in place to build the adder circuits. A few higher level modules had to be added for propagate/generate signaling (`PG16` and `PG32`) and XOR arrays (`XOR16` and `XOR32`).

One other useful definition was one that described how to translate a bit-vector into a number. For a single bit, *true* translates to 1 and *false* translates to 0.

```
val BVS_DEF = Define ‘(BVS T = 1) /\ (BVS F = 0)‘;
```

This can be extended to work for an arbitrary number of bits. The `BVL` definition describes how that is done.

```
val BVL_DEF = Define ‘(BVL [] = 0) /\ (BVL (h::t) = (if h = T
            then 2 * BVL t + 1 else 2 * BVL t))‘;
```

These definitions were used for verifying the correctness of the of the full-adder cell and could also be used for checking the correctness of larger adders, especially if they are defined recursively.

Creating HOL-Style Netlists

Creating HOL-style netlists is an easy enough task. Netlists in other forms (like Verilog) can be mapped almost directly to HOL. However, for large circuits, doing it by hand it can be tedious and increases the risk of inadvertently introducing errors during translation. A computer program is well-suited for the job of taking a Verilog netlist and converting it to a HOL-friendly netlist. As part of this project, a Perl program was constructed to read in arbitrary Verilog structural (modules and wires only) code and produce a HOL definition of a netlist described in the Verilog file. The Verilog::Netlist Perl module was used to parse the input and determine which signals and modules the netlist contains.

The tool that was dveloped is called `holnetlist` and is fairly simple to use. It takes in a list of Verilog files on the command line and then sends the HOL netlist to standard output. The output can then be copy directly into HOL (provided there were no errors). The output has the a form similar to the following example.

```
Module1(in1,in2,out1) = ?wire1 wire2.  Cell1(in1, wire1) /\
        Cell1(in2,wire2) /\ Cell2(wire1, wire2,out1)
```

The nets in this example are `in1`, `in2`, `out1`, `wire1`, and `wire2`. The nets `in1`, `in2`, and `out1` are the inputs and the outputs (module ports) and are free variables. The other nets `wire1`, and `wire2` are wires internal to the module and are bound under existential quantification. The wires are therefore bound to follow the changes out the inputs just like an actual circuit. The Verilog code that would generate the sample output would look something like the following.

```
module Module1(in1,in2,out);
  input in1, in2;
  output out;
  wire wire1, wire2;
  Cell1(.in(in1), .out(wire1));
  Cell1(.in(in2), .out(wire2));
  Cell2(.in1(wire1), .in2(wire2), .out(out));
endmodule
```

The Verilog code closely resembles the equivalent HOL definition. This shows that the tool does not have much translation to do. The difficult task of parsing the input is done by the Verilog::Netlist Perl module. This leaves the `holnetlist` tool the translation task which was done in under 80 lines of Perl code.

Building a Sat-Based Tactic

Sat-based method can be used for proving that two large boolean equation are equal. HOL provides an interface to external satisfiability tools like grasp, zchaff and sato called HolSatLib. The HolSatLib library provides a method

called satOracle that can be used for tautology checks. Given a CNF boolean expression, satOracle will return a theorem that contains an assignment that satisfies the expression. If the formula is not satisfiable, satOracle will produce a formula of the form `|- ~formula` . If one wishes to show that a formula is satisfiable for all assignments, then you can give satOracle the negated formula and if it produces a theorem with the negated input, then you can prove that the the original formal holds in all cases.

Building a sat-based tactic, `MY_SAT_TAC`, was one of the final and most challenging parts of the project. Building a tactic in HOL can be difficult if you have never done it before. Fortunately, HOL provides several examples to work from. `MY_SAT_TAC` uses several conversions and rules to produce a theorem that is passed to `ACCEPT_TAC` to prove the goal on the top of the goal stack. The tactic will only succeed if all of the conversions work and satOracle is able to prove that `|- ~cnf` given the input of `cnf`. The first thing that the tactic has to do is negate the goal and convert it to CNF form. Then that can be passed to satOracle to generate a theorem. That theorem is combined with the original CNF conversion theorem to show `|- ~t = F`. This then converted to a theorem that can be used to prove the goal with `ACCEPT_TAC`.

Solving the Final Goals

Proving two large higher-order logic formulas are equal was somewhat difficult in HOL. Many of the general-purpose tactics and rewriter struggle with the 32-bit and even 16-bit adder netlists. The goals had to be broken up into smaller subgoals. The original goals were all of the form `Netlist1 = Netlist2`. These goals were first broken up into to subgoals with `EQ_TAC`. After that, one goal from each of the pairs of subgoals could be solved with the sat-based tactic after some rewriting, simplification and a conversation to get rid of unnecessary existential variables. This half of the goal was actually the fastest to prove after the sat-based tactic was working.

The other half did not work with the sat-based approach. However, the general purpose tactics like `METIS_TAC` were able to handle this side after identifying points (nets) in the netlists that were that same between the two circuits. Since both circuits have nets for the final carry out to each bit, `Q.EXISTS_TAC` could be used to tell HOL that those points were equal. Then calling `REPEAT STRIP_TAC` broke the goal down enough that `METIS_TAC` could finish off the proofs. The final proofs took hours for HOL to solve with most of the time being taken by `METIS_TAC`. Even though it took so long, this is an acceptable penalty when avoiding exhaustive simulations. There are probably more efficient ways to solve this final step in the proof, but this method does not require interaction and therefore it could potentially be automated.

Discussion

This project met all of the goals of in the original proposal. The proposal was to verify a 32-bit Ladner-Fischer adder and a 32-bit Brent-Kung adder and that is what this project did. It also verified the 16-bit versions along the way as planned for in the proposal. The actual work progress didn't match up with the work schedule, but in the end the original goals were achieved. There was also more that came from this project than was originally planed for. The HOL netlist generator and the sat-based tactic were tools that had to be created to get the work done, but they were not planned for in the original proposal. In general, it is pleasing to see a proposal match a finished product so closely.

The progress of this project was slowed by obstacles at times, but also accelerated by the power of the theorem prover. The definitions that were required for this project were mostly straight-forward and so they were not too difficult to get into HOL. One of the major problems was the size of some of the definitions (like the ones for the adders). The `holnetlist` tools was developed to overcome this issue and it worked out fairly well. The fact that it is so easy to describe circuit netlists in HOL is a real testimony to its expressive power.

Many of the general-purpose tactics do not really scale to formulas the size of the netlists in this project. Before this tactic could be used the netlists had to be broken down into more manageable sized pieces. There were also cases where there was not an easy way to breakdown the goal. In this case, the sat-based approach was used to solve the goal. Overall, HOL was well-suited for this project.

Conclusion

The results of this project show that a netlist equivalence method aided by HOL can be used show correctness of fairly large circuits. The approached used in this project successfully proved that a Ladner-Fischer adder and a Brent-Kung adder are logically equivalent to a simple ripple-carry adder of the same width. This was shown for 16-bit and 32-bit versions of each of the circuits. These circuits were large enough to demonstrate how this approach should scale to other types of circuits.

Future Work

This project opens up a wide range possibilities for future work. Other types of circuits could make use of this framework for correctness verification. The HOL netlist program could be improved. It could be made to work with other types of netlists (other than Verilog). It could even be made to take netlist definitions from HOL and generate Verilog code.

The sat-based tactic could also be improved to work with a wider range of goals. It could be made more intelligent so that it could deal with existential and universally quantified variables and not require only free variables. The

goal of improving this tactic could be having it prove the top-level goal of the 32-bit adder proofs (after definition rewrites).

If the sat-based tactic were improved enough, netlist equivalence proofs could potentially be automated. A script could be written to take two Verilog netlists, convert them to HOL definitions and prove equivalence with some type of intelligent tactics. Once this process is automated, it could then be optimized for speed. The proofs in this project take hours and it would be fantastic if it could be done in minutes instead.

Thoughts About HOL

HOL is a powerful tool. It has an incredible array of theorems, tactics, converters and other tools that make it flexible and robust. However, there is a steep learning curve involved with using HOL. It seems like every time you use HOL, you learn something new about it. It is even harder to learn if you don't know ML, but once you know some ML basics, HOL is much easier to use. It can be fun to explore the inter-workings of HOL and at time frustrating because you can not get it to do what you want it to do.

Aside from the steep learning curve, one of drawback of HOL is the interface. An interactive command line seems like a perfectly logical choice for a tool like this, but it's has some shortcomings. It's not very forgiving if you make a mistake because you can't use the arrow keys to go back and make changes. It is also missing some features. It would be nice if it supported command line history at least. It would be even better if it had tab-completion for theorems and tactics. That would be very useful, but I think it is really a problem with Moscow ML and not really HOL. Most of the interface problems can be overcome by using emacs or a readline wrapper program (like rlwrap), so it's not a huge problem.

Overall, HOL is useful and it does what it is designed to do, theorem proving. It worked well for this project and it should be useful for other similar projects. HOL is a tool that is worth learning how to use.