Project Abstract

Thanawat Techaumnuaiwit

Contents

Background

The problem that we are trying to work on is *digital hardware decompilation*. A digital circuit or a *netlist* is a "graph" of digital logic gates (or components). Some example logic gates are the boolean functions, AND, XOR, OR, NOT, etc, and they work like you would expect - they are "pure" functions that take some inputs and give you an output:

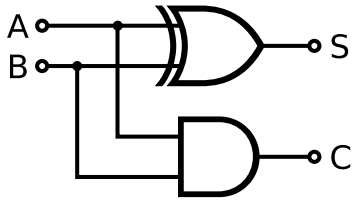

Figure 1: A Half Adder circuit

If an output of a gate is used by another gate, then the design is entirely "stateless" or "combinatorial". However, in digital logic design, an output can be *fedback* as input to itself - which allows these circuits to have memory.[1]

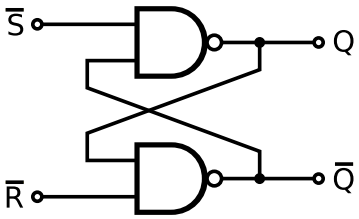

Figure 2: An SR-Flip-Flop

On the hardware, the *only* components that exist are 1-bit logic gates. Everything else needs to compile down to that level. Most programmers these days don't write their programs in assembly, and most digital logic designers don't write their digital designs using just 1-bit logic gates either! They write their digital logic design in higher level HDLs, i.e. SystemVerilog, CLash(Haskell based HDL), etc. These languages have "higher level" language features like multi-bit operators, loops, modules, etc. However, in the *lower netlist* level, these higher level language features **must be compiled away!**.

[1] See: https://en.wikipedia.org/wiki/Flip-flop_(electronics)

To summarize **high level** Verilog/SystemVerilog is a netlist with higher level verilog features(loops, modules, etc). In the **low level** Verilog - these features were compiled away and all you are left with is a **netlist** of only 1-bit logic gates. The problem of hardware decompilation is to take the **netlist** of 1-bit logic gates and turn them back into higher level SystemVerilog.

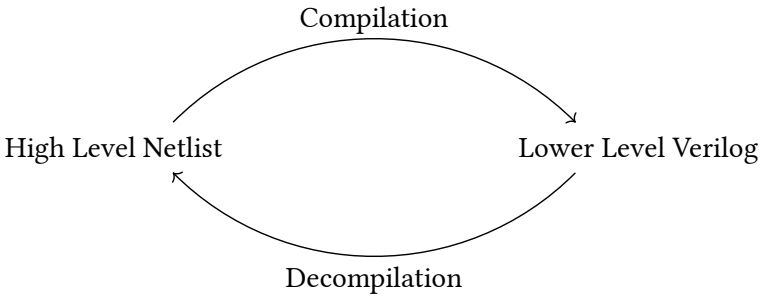

Figure 3: Our problem space

Problem statement

It's very difficult, almost impossible, to achieve perfect decompilation - even in the software space where the problem space is arguably easier. Our goal with the project is to break down the steps into "passes" that *extends* the hardware decompilation problem into mulitiple stages. This approach mirrors compilation - since compilation also happens in multiple stages as well.

Module Finding

One possible stage is finding a known standard libary module inside a bigger netlist. In this stage, we are given a high-level verilog module of the stadard library component. We can take this module and compile it into a netlist. Then, we can try to take this netlist and see of we can find it in a bigger netlist - where the high-level representation is potentially unknown.

The approach we are taking to do this is to use a data structure called an EGraph. A "netlist" have a syntactical representation, as well as a graph representation. A EGraph allows you to efficiently and compactly keep a graph and it's "equivalent" graphs in a datastructure. We can then "traverse"[2] this egraph and "extract out" parts of the netlist that are equivalent to the standard libary module that we want to find.

A simple example: Finding Half Adders in a Full Adder

The simple example is to find a half adder circuit in a full adder circuit. A half-adder circuit adds 2 1-bit binary inputs together. However, adding 2 1-bit numbers will overflow when you try to add `0b1` and `0b1`, so a real adder needs to account for the carry-in and overflow. This adder is a "full adder" and it can be implemented using 2 half adders.

The half adder circuit, in verilog looks like:

```
module half_adder
  (
    input i_bit1,
    input i_bit2,
    output o_sum,
    output o_carry
   );
```

[2] catamorphically. I wonder how we could apply recursion schemes to EGraphs. Maybe we can have different types of extraction schmes?

```
  assign o_sum   = i_bit1 ^ i_bit2;  // bitwise xor
  assign o_carry = i_bit1 & i_bit2;  // bitwise and

endmodule // half_adder
```

The full-adder circuit can be implemented as 2 half adders:

```
module full_adder
  (
   input i_bit1,
   input i_bit2,
   input i_cin,
   output o_sum,
   output o_carry
   );

   wire   half_sum1;
   wire   half_carry1;
   wire   half_carry2;

   half_adder half_adder1(
        .i_bit1(i_bit1),
        .i_bit2(i_bit2),
        .o_sum(half_sum1),
        .o_carry(half_carry1)
        );
   half_adder half_adder2(
        .i_bit1(i_cin),
        .i_bit2(half_sum1),
        .o_sum(o_sum),
        .o_carry(half_carry2)
        );
   assign o_carry = half_carry1 | half_carry2;
endmodule // full_adder
```

Modules in Verilog are considered “high level” features. In order to translate this Verilog to a netlist, the netlist compiler(Yosys in our case) needs to “flatten” the module by replacing the module stubs with it’s implementation. The following Yosys outlines the ssteps of “compiling” the high-level full_adder module to a lower level netlist:

```
read_verilog ./half-adder.v
prep -top half_adder;
flatten;
write_lakeroad ./half-adder.egg

read_verilog -sv ./full_adder.v
prep -top full_adder
flatten;
write_verilog ./full_adder_sythv
write_lakeroad ./full_adder.egg
design -reset
```

We can represent the netlist using the an EGraph (the `write_lakeroad` line). For this example, *no optimizations were applied to the netlist.* Given a bigger Egraph contained in `full_adder.egg`, we can

try to find the exact match of the smaller EGraph, `half_adder.egg`. This problem is also known as *subgraph isomorphism*.[3]

[3] https://en.wikipedia.org/wiki/Subgraph_isomorphism_problem

The EGraph of the half adder looks like this:

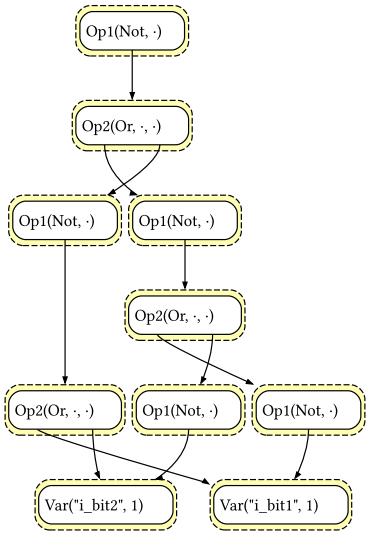

Figure 4: Half Adder EGraph

We can construct a "rewrite rule" in Egglog(a Datalog language for EGraphs) to find the exact match of the Half Adder in the full adder netlist. The rewrite rule looks like this:

```
(rule
  ((= (Op2 (Xor) i_bit1 i_bit2) o_sum)
   (= (Op2 (And) i_bit1 i_bit2) o_carry))
  ((let half-adder (Module "half-adder" (Op2 (Concat) i_bit1 i_bit2)))
    (union o_sum (Op1 (Extract 1 0) half-adder))
    (union o_carry (Op1 (Extract 2 1) half-adder))
) :ruleset rewrites)
```

What this rewrite rule means is that, `i_bit1` and `i_bit2` (inputs to the circuit) are essentially arbitrary circuits *that are connected to an* XOR and an AND gate. If we find this "pattern" in the graph, we can "label" it by adding a new "Module" node to label the half adder.

Running this egglog rule on the full adder looks like:

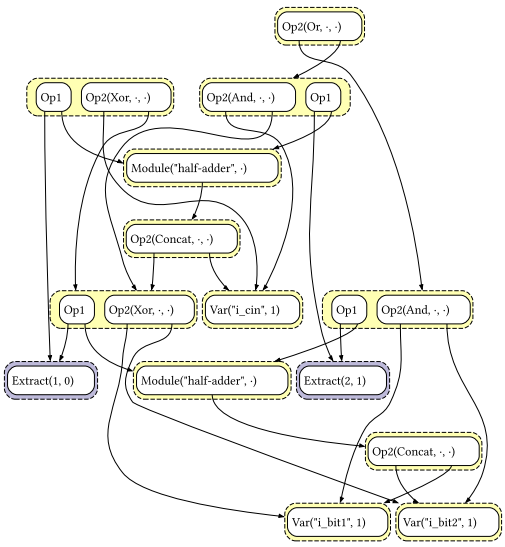

Figure 5: Half Adder EGraph

As you can see from the picture, we found the 2 half adders we were looking for! Unfortunately, this only works because the full adder graph was never optimized or rewritten in such a way that doesn't preserve the "shape" of the half adders we were looking for. For example, the full adder netlist can just have NAND gates[4], and we would never find our half adder netlist since it was compiled to XOR and OR. The issue is that circuits have different representations and optimizations can "merge" gates together, as we will see in the next section.

A harder example: Converting to a different set of base gates

For those not familiar with hardware, the set of "base gates" is the set of logical gates that can be used to implement all other logical gates. What if we converted a simple circuit to a smaller level of base gates? For example:

```
module test(
        input a,
        input b,
        output d,
        output c
);
        assign d = a & b;
        assign c = a ^ b;
endmodule
```

If we compile away the XOR into other gates, can we still find the XOR gate in this bigger circuit? Here's the XOR implementation using a set of base gates that doesn't include XOR:

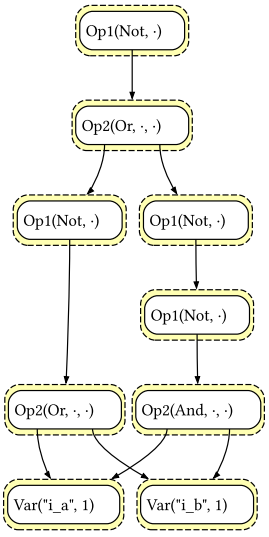

Figure 6: XOR Egraph

Naively finding the exact match on the XOR EGraph on the test1 EGraph doesn't work! It seems like the XOR net is not contained in the test1 net at all!

[4]The NAND gate is a "functionally complete" gate and can be used to represent the other boolean logic gates. https://en.wikipedia.org/wiki/Functional_completeness

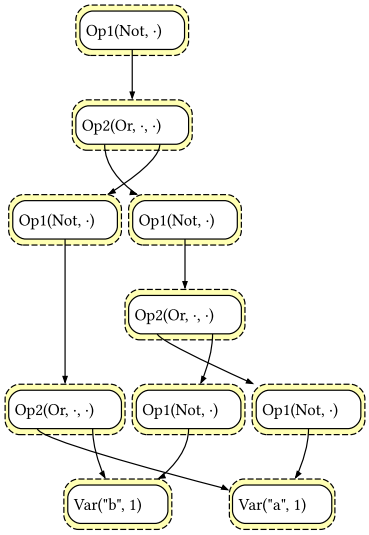

Figure 7: Test1 Egraph before rewrites

However, in order to actually find the XOR gate, we need to "continuously deform"[5] the XOR netlist into a form that can be found in the test1 netlist. We can do this by adding more rewrite rules (i.e. the logical equivalences: https://en.wikipedia.org/wiki/Logical_equivalence), see Listing 1.

Once we add these rewrite rules, we can find the XOR gate in our EGraph:

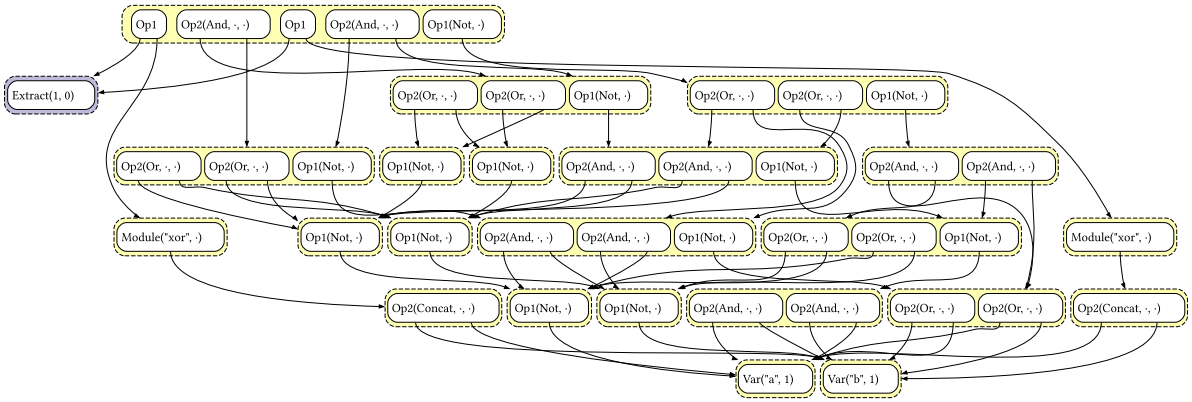

Figure 8: Found XOR Gates

[5] I want to call this subgraph homotopy but I'm not well versed enough in math to see if this fits.. https://en.wikipedia.org/wiki/Homotopy

```
(rewrite
 (Op2 (Or) x (Op2 (And) x y))
 x
 :ruleset deoptimize)

(rewrite
 (Op2 (And) x (Op2 (Or) x y))
 x
 :ruleset deoptimize)
;; commutativity
(birewrite
 (Op2 (And) x y)
 (Op2 (And) y x)
 :ruleset deoptimize)
(birewrite
 (Op2 (Or) x y)
 (Op2 (Or) y x)
 :ruleset deoptimize)

;; idempotence
(rewrite
        (Op2 (And) x x)
        x
        :ruleset deoptimize
  )
(rewrite
        (Op2 (Or) x x)
        x
        :ruleset deoptimize
  )
;; De Morgans

(birewrite
        (Op1 (Not) (Op2 (And) x y))
        (Op2 (Or) (Op1 (Not) x) (Op1 (Not) y))
        :ruleset deoptimize
  )

(birewrite
        (Op1 (Not) (Op2 (Or) x y))
        (Op2 (And) (Op1 (Not) x) (Op1 (Not) y))
        :ruleset deoptimize
  )
```

Listing 1: Added rewrite rules

Ideas

Graph Language

One thing we can take inspiration from is looking at the vast literature in Symmetric Monoidal Categories. I wouldn't say that the work we are doing is that related to that, but I just wanted to mention it because I like SMCs.. The "main idea"(from what my small brain can understand) in SMCs is that equations have a natural graphical representation and the graphical representation can potentially be easier to work with due to their unambiguity - since two equations can be used to represent the same graph.

Unrelated idea: Interpretation of circuits for efficient

This is a stretch, and almost unrelated, to this work but I just wanted to share this work (again) with the EGraph folks, since I think the two fields seem so similar.

One related work(ish) that I want to highlight is the work by Dan Ghica and colleagues on Diagrammatic Semantics for Digital Circuits. Here, Ghica formalizes the semantics of digital circuits by giving operational sematnics on them using graph rewriting semantics. Graph rewriting as operational semantics is well studied in the field of Quantum Computing - i.e. the https://zxcalculus.com/ . Instead of using the Lakeroad language for *optimization* or *decompilation*, what if we could also *interpret* circuits using rewrite rules? This has potential to allow more efficient *simulation* of circuits because the evaluation of SMCs are automatically paralellizable.