Automating Constraint-Aware Datapath Optimization using E-Graphs

Samuel Coward
Numerical Hardware Group
Intel Corporation
Email: samuel.coward@intel.com

George A. Constantinides
Electrical and Electronic Engineering
Imperial College London
Email: g.constantinides@imperial.ac.uk

Theo Drane
Numerical Hardware Group
Intel Corporation
Email: theo.drane@intel.com

***Abstract*—Numerical hardware design requires aggressive optimization, where designers exploit branch constraints, creating optimization opportunities that are valid only on a sub-domain of input space. We developed an RTL optimization tool that automatically learns the consequences of conditional branches and exploits that knowledge to enable deep optimization. The tool deploys custom built program analysis based on abstract interpretation theory, which when combined with a data-structure known as an e-graph simplifies complex reasoning about program properties. Our tool fully-automatically discovers known floating-point architectures from the computer arithmetic literature and out-performs baseline EDA tools, generating up to 33% faster and 41% smaller circuits.**

I. Introduction

Industrial Register Transfer Level (RTL) design requires engineers to exploit all possible optimization opportunities in increasingly complex designs. In particular, floating-point hardware is the subject of intense scrutiny given its wide ranging applications. These modules are almost always designed by hand [1]–[3]. State-of-the-art electronic design automation (EDA) tools are currently unable to match human designs.

Combining program analysis and e(quivalence) graph rewriting techniques we exceed the optimization capabilities of existing EDA tools for hardware design, with the ability to automatically exploit optimization opportunities generated by conditional branches in designs. We will first discuss key topics and prior work. In Section III we will introduce the theory underpinning sub-domain equivalences and how conditional branches are captured in e-graphs to compute tight approximations to intermediate program values. In Section IV we will discuss how constraint-aware optimization can be implemented in an RTL optimization tool, built on top of the `egg` e-graph library. We will then consider a floating-point subtractor design case study in Section V, with further results in Section VI. The paper contains the following novel contributions:

- expressibility of sub-domain equivalences in an e-graph to enable constraint-aware datapath optimization,
- a method and tool to automate the production of run-time branching / muxing conditions, leading to optimized hardware via constraint-aware optimization,
- a case study demonstrating the automated production of a highly-optimized floating-point subtractor,
- evaluation on benchmarks showing the generality of the method.

979-8-3503-2348-1/23/$31.00 ©2023 IEEE

II. Background

E-graphs are a data structure that represents equivalence classes (e-classes) of expressions compactly [4], [5]. Nodes in the e-graph represent functions or arguments which are grouped into e-classes. Directed edges connect a node to its child e-classes, representing the function's inputs. An e-graph is grown via the application of rewrites (e.g. $\texttt{x+0} \rightarrow \texttt{x}$), which define equivalences over expressions. Constructive rewrite application means that the left-hand side remains in the data structure after application, avoiding the phase ordering problem [6]. The e-graph grows monotonically, representing an increasing design space of equivalent implementations. An example e-graph before and after rewriting is presented in Figure 1, which also shows how interval enclosures can be attached to each e-class from which we can learn useful properties [5], [7]. It is possible to learn such properties via abstract interpretation as we will show in Section III.

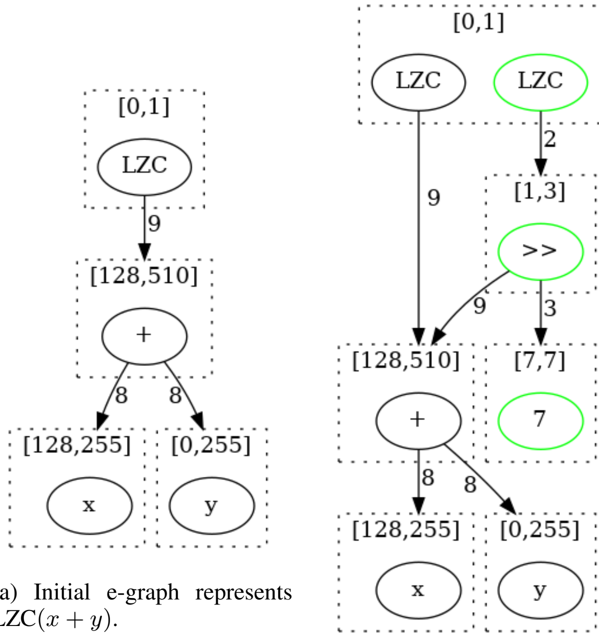
Compilers use program analysis to enable optimizations such as dead code elimination [8], [9]. Abstract interpretation is a theory used to over-approximate program properties [10]. A weak but cheap to compute interpretation is interval arithmetic, which approximates variables by their input ranges and operators by their natural interval extension. Relational domains such as the polyhedral domain [11] capture correlation and are more powerful but are more complex to compute.

On the datapath optimization problem, syntactic rewriting techniques have been explored [12], with one contribution using e-graphs [13]. Unfortunately, syntactic rewrites alone cannot express the deep transformations required for floating-point hardware design. We will show that such transformations depend upon knowledge of intermediate values and domain restrictions under which the branches are executed.

III. Theory

A. Sub-Domain Equivalences

E-graphs represent expressions, drawn from a set *Expr* with variables evaluated over a domain $\mathcal{D}$. We consider a concrete semantics of expressions $[\![\cdot]\!]. \in \mathit{Expr} \rightarrow \mathcal{D}^n \rightarrow \mathcal{D}$, evaluating an expression as a function, e.g. $[\![\mathtt{x}+\mathtt{1}]\!]$ is a function mapping values of the variable $\mathtt{x}$ to values of the expression $\mathtt{x}+\mathtt{1}$. This



(a) Initial e-graph represents LZC$(x + y)$.

(b) LZC(a) → LZC$(a \gg 7)$. Green nodes are newly added.

Fig. 1: An e-graph before and after rewriting. The LZC node denotes a leading-zero counter. Edges are labelled with bitwidths. The rewrite is valid due to the input constraint, $x \geq 128$, which implies that LZC$(x + y) \leq 1$, namely $x + y$ has at most one leading zero.

allows us to say that two expressions, e_a and e_b, are congruent, $e_a \cong e_b$, iff $[\![e_a]\!] = [\![e_b]\!]$.

This notion of congruence is strict and enforces equivalence across the entire domain $\mathcal{D}^n$. Under a weaker congruence relation, e.g. equivalence on a sub-domain, many additional congruences may hold. Expressions, e_a and e_b, can be said to be *congruent under* $c \in$ *Expr* iff

$$\forall v \in \mathcal{D}^n,\ [\![c]\!](v) \Rightarrow [\![e_a]\!](v) = [\![e_b]\!](v)$$

i.e. in every model where $[\![c]\!]$ holds, $[\![e_a]\!] = [\![e_b]\!]$. We denote this $e_a \cong_c e_b$.

Input constraints or conditional branches can constrain the domain of operands for a given operator, potentially exposing additional optimization opportunities. The leading zero counter (LZC) circuit described in Figure 1, demonstrates how an input constraint can be exploited to build an equivalent but smaller and faster circuit.

Conditional branches initiated via an if statement in software or a mux in hardware will be of particular interest in this work. As an example, the following C expressions are equivalent, even though fabs$(x) \neq x$ in general.

`(x > 0 ? fabs(x) : 0)` $\cong$ `(x > 0 ? x : 0)`

Capturing these possibilities within datapath design optimization is essential. Sub-domain equivalence relations allow developers to optimize each branch of their code under different conditions, often resulting in better performance. This is exactly the motivation behind introducing case splits into any design. Examples can be seen in Sections V and VI.

B. Sub-Domains in E-Graphs

The preceding examples show the necessity to capture the set of possible variable values under which one cares about the correct evaluation of an expression, and to do so requires automated reasoning about the set of values an expression can take. Abstract interpretation is a well-known approach to this problem in the field of program analysis [14]. In our work, we abstract the set of 'care' values of an expression as a finite union of integer intervals, as this is sufficient to be able to reason about the industrial datapath designs we encounter. To be more precise, with each expression we associate an element of the set $\mathcal{A}$:

$$\mathcal{A} = \left\{ \bigcup_{i=1}^{n} [a_i, b_i] \mid a_i \leq b_i,\ a_i, b_i \in \mathbb{Z},\ n \in \mathbb{N} \right\}.$$

For a given $e \in$ *Expr*, we compute $\mathcal{A}[\![e]\!] \in \mathcal{A}$, using interval arithmetic extended to unions of intervals incurring additional computational complexity. Following the technique in [7], each e-class of expressions is associated with an element of $\mathcal{A}$, which represents a conservative approximation of all evaluations of that e-class, as shown in Figure 1. Such an approach is shown to produce a more accurate program analysis [7].

We introduce an additional operator, `ASSUME`, that will be used to encode sub-domain equivalences. `ASSUME` takes two operands, an e-class containing equivalent *Expr* to evaluate and a set of e-classes containing *Expr*, encoding conditions that may be assumed to be true when evaluating the first argument. We achieve this effect by appending an additional *special* element to $\mathbb{Z}$, forming a new domain $\mathbb{Z}' = \mathbb{Z} \cup \{*\}$. Naturally extending $[\![\cdot]\!]$ notation to e-classes, the semantics under a single constraint are:

$$[\![\texttt{ASSUME}(x, c)]\!] = \begin{cases} [\![x]\!] & \text{if } [\![c]\!] \\ * & \text{else.} \end{cases} \quad (1)$$

Under multiple constraints, a * is returned if any of the conditions do not hold. The semantics of all other functions except the ternary operator $\cdot ? \cdot : \cdot$ are extended to this new domain by also returning * iff at least one of their operands is *. The ternary operator requires special treatment as it returns a * only if the branching condition itself is a * or one of the *reachable* branches returns a *. In this way, we can think of * as precisely capturing the code 'failing an assertion'. The key benefit of this construction is that we may now define $\cong_c$ in terms of congruence over the whole domain, that is

$$x \cong_c y \Leftrightarrow \texttt{ASSUME}(x, c) \cong \texttt{ASSUME}(y, c). \quad (2)$$

Thus we may reason automatically about sub-domain congruences using the e-graph machinery which applies for whole-domain congruences.

The consequences of the constraints are realized in the abstraction of `ASSUME`, if the assumed conditions constrain

the expression under evaluation. The abstraction of ASSUME with a single constraint is:

$$\mathcal{A}[\![\texttt{ASSUME}(x,c)]\!] = \mathcal{A}[\![x]\!] \cap I, \text{ where} \tag{3}$$

$$I = \begin{cases} (-\infty, c') & \text{if } [x < c'] \quad \in c \\ (c', \infty) & \text{if } [x > c'] \quad \in c \\ [c', c'] & \text{if } [x == c'] \in c \\ (-\infty, c') \cup (c', \infty) & \text{if } [x \neq c'] \quad \in c \\ (-\infty, \infty) & \text{else.} \end{cases} \tag{4}$$

Since c is an e-class, it may include multiple equivalent *Expr*, therefore we test whether any of the interpretable constraints are members of c. We define *Constr* $\subseteq$ *Expr*, denoting the generalized set of constraints appearing on the right of the if statements in (4). An ASSUME with a set of constraints represents a further restriction of the domain via additional intersections. To demonstrate, suppose $\mathcal{A}[\![\text{x}]\!] = [-3, 3]$ then,

$$\mathcal{A}[\![\texttt{ASSUME(x, x>0)}]\!] = [-3, 3] \cap (0, \infty) = [1, 3].$$

Examples in Section IV-B demonstrate how this theory enables additional optimizations. Our experience shows that, when combined with the rewriting described in Section IV, it is only necessary to reason about the limited and computationally-efficient set of constraints described in (4).

IV. Implementation

We applied the theory described above to an RTL optimization tool that is built on the egg e-graph library [5]. We optimize hardware designs at the RTL level of abstraction, operating on combinational logic on unsigned bitvectors. The tool parses input (System) Verilog using Yosys and sv2v [15], converting it into an e-graph with bitwidth annotations, following the approach in [13]. A set of parameterized and generalized constraint-aware rewrites at the word level is developed for this work. For concision of notation we exclude bitwidth annotations when describing rewrites in this paper. An online repository[1] summarizes rewrites described in this paper. Rewrites are automatically applied to the e-graph for a number of iterations, then a delay optimized expression is extracted from which a System Verilog implementation is generated.

A. Bitwidth Reduction

In RTL, expressions are evaluated over unsigned bit-vectors, therefore arithmetic is computed with respect to some modulo. When propagating finite unions of integer intervals, we use a conservative approximation to modular intervals.

$$[l, u] \mod p = \begin{cases} [l \mod p, u \mod p] & \text{if } \left\lfloor \frac{l}{p} \right\rfloor == \left\lfloor \frac{u}{p} \right\rfloor \\ [0, p-1] & \text{else.} \end{cases} \tag{5}$$

By propagating finite unions of integer intervals throughout the e-graph, corresponding to each classes' possible outputs, we enable bitwidth reduction. We maintain a bitwidth for each operand in the internal representation and are able to shrink this if we discover that the values which that operand can take would be representable in a smaller bitwidth. When combined with the ASSUME node abstraction described above and the rewrites described in Section IV-B below, we generate tighter approximations throughout, meaning that bitwidths can be squeezed to their minimum required precision.

[1] https://figshare.com/s/e3ab2850662d24991cbc

TABLE I: ASSUME node rewrites, describing creation, propagation and simplifications. The op operator can represent any operator defined in the intermediate language.

Left-Hand Side	Right-Hand Side
$a\,?\,b\,:\,c$	$a\,?\,\texttt{ASSUME}(b,\ a)\,:\,\texttt{ASSUME}(c,\ \sim a)$
$\texttt{ASSUME}((a \texttt{ op } b),\ c)$	$\texttt{ASSUME}(a,\ c) \texttt{ op } \texttt{ASSUME}(b,\ c)$
$\texttt{ASSUME}(\texttt{ASSUME}(a, b),\ c)$	$\texttt{ASSUME}(a,\ b \cup c)$
$\texttt{ASSUME}((a\,?\,b\,:\,c),\ a)$	$\texttt{ASSUME}(b,\ a)$
$\texttt{ASSUME}((a\,?\,b\,:\,c),\ \sim a)$	$\texttt{ASSUME}(c,\ \sim a)$

B. Enabling Sub-Domain Equivalences in E-Graphs

We introduced the ASSUME operator above and its abstraction. Focusing on RTL optimization, Table I describes rewrites to create, propagate and exploit ASSUMEs. The ASSUMEs will be induced by mux statements via the first rewrite in Table I. Recall that ASSUMEs allow us to express multiple equivalence relations in the e-graph. We use rewrites of the form $\texttt{ASSUME}(x, c) \rightarrow \texttt{ASSUME}(y, c)$ to encode $x \cong_c y$.

The need for ASSUME nodes is illustrated via an example, `a==0 ? a : -a`, equivalent to, `a==0 ? 0 : -a`. Naively applying `a`$\rightarrow$`0`, to the e-graph merges the *non-equivalent* expression, `a==0 ? 0 : -0`, into the e-graph. Using Table I and the rewrite `ASSUME(a, a==0)` $\rightarrow$ `ASSUME(0, a==0)`, we produce the optimal expression.

$$\texttt{(a==0)?a : -a} \rightarrow$$
$$\texttt{(a==0)?ASSUME(a,a==0) : ASSUME(-a,} \sim \texttt{(a==0))} \rightarrow$$
$$\texttt{(a==0)?ASSUME(0,a==0) : ASSUME(-a,} \sim \texttt{(a==0))}$$

By construction we can treat ASSUMEs as assignment statements in the implementation phase, meaning that they can be ignored in the optimized expression generated after rewriting.

The remainder of Table I is valuable in more complex scenarios. The second rewrite propagates ASSUME nodes down through the e-graph towards the inputs, essential because the constrained sub-expression may occur at any point in the tree (or not at all). The third combines nested ASSUME nodes by combining the two sets of conditions, which could be implemented as an AND of all the conditions. The final two rewrites prune unreachable branches. We discover the validity of additional sub-domain rewrites by evaluating the abstractions of the ASSUME nodes and propagating this knowledge upwards through the e-graph.

In Section III-A we described a C expression. We now have a sequence of rewrites to prove the desired equivalence.

$$\texttt{(x>0)?fabs(x)} \qquad \texttt{: 0} \rightarrow$$
$$\texttt{(x>0)?ASSUME(fabs(x),x>0) : ASSUME(0,} \sim \texttt{(x>0))} \rightarrow$$
$$\texttt{(x>0)?fabs(ASSUME(x,x>0)) : ASSUME(0,} \sim \texttt{(x>0))} \rightarrow$$
$$\texttt{(x>0)?ASSUME(x,x>0)} \qquad \texttt{: ASSUME(0,} \sim \texttt{(x>0))}$$

TABLE II: Condition rewrites, used to transform members of *Expr* into members of *Constr*.

Transformation Rules	Inversion Rules
$a < b \rightarrow a - b < 0$	$\sim (a = b) \rightarrow a \neq b$
$a \leq b \rightarrow a < b + 1$	$\sim (a > b) \rightarrow a \leq b$
$a > b \rightarrow a - b > 0$	$\sim (a \geq b) \rightarrow a < b$
$a \geq b \rightarrow a > b - 1$	$\sim (a < b) \rightarrow a \geq b$
$a = b \rightarrow a - b = 0$	$\sim (a \leq b) \rightarrow a > b$
$a = b \rightarrow 0 = b - a$	

`fabs(ASSUME(x,x>0))` $\rightarrow$ `ASSUME(x,x>0)` is proven valid via (4), as $\mathcal{A}[\![\texttt{ASSUME(x,x>0)}]\!] = \mathcal{A}[\![\texttt{x}]\!] \cap (0, \infty)$.

C. Condition Rewriting

In Section III-B we described condition rewriting as a technique to mitigate the restrictions imposed by (4). Using the rewrites described in Table II, the tool attempts to transform $c \rightarrow c'$, where $c \in$ *Expr* and $c' \in$ *Constr*.

Consider an e-graph containing `ASSUME(a-b, a>b)`. `a>b` $\notin$ *Constr* $\Rightarrow$ $\mathcal{A}[\![\texttt{ASSUME(a-b, a>b)}]\!] = \mathcal{A}[\![\texttt{a-b}]\!]$. By rewriting `a>b` $\rightarrow$ `a-b>0`, we merge `a-b>0` $\in$ *Constr* into the constraint e-class, triggering a refinement via (4), $\mathcal{A}[\![\texttt{ASSUME(a-b, a-b>0)}]\!] = \mathcal{A}[\![\texttt{a-b}]\!] \cap (0, \infty)$.

An e-class can contain many equivalent representations of a constraint so there is no need to find the single ideal representation. At the same time, the tool is also rewriting the expression under evaluation for optimization purposes. But a side benefit is that it may also discover how the particular imposed constraint impacts the expression.

In RTL design we often encounter conjunctions or disjunctions of conditions. We handle logical and/or via mux rewrites.

$$(a \wedge b) \,?\, c : d \rightarrow a \,?\, (b \,?\, c : d) : d \tag{6}$$

$$(a \vee b) \,?\, c : d \rightarrow a \,?\, c : (b \,?\, c : d) \tag{7}$$

These rewrites break conjunctions and disjunctions into simpler *Expr* that the tool can reason about. Rewriting mux operations further mitigates the restrictions imposed by (4).

D. Delay Modeling

The final e-graph contains many functionally equivalent implementations of the input RTL. In this work, we target maximal performance and extract the design with the shortest critical path delay. If multiple designs achieve identical delay, we extract the smallest area circuit amongst them.

We take a similar approach to previous work on multiplier design for FPGAs using e-graphs and construct a theoretical model of delay [16]. For each operator we compute an estimate based on a fixed component architecture for the total number of two-input gates on the operator's critical path as a function of operator precision. At each operator the total delay to the output is the maximum delay across all its children plus its own delay. Using a theoretical model enables efficient design space exploration and avoids long logic synthesis runtimes.

We extract the best design from the e-graph using `egg`'s standard extraction algorithm [5] combined with a delay/area weighted sum objective function. For a performance prioritized optimization, common sub-expressions are not a significant factor, therefore we did not take an integer linear programming approach to extraction as elsewhere [13].

V. Case Study: Floating-Point Subtract

To demonstrate the capabilities of such an approach we used the tool to automatically optimize a hardware implementation of a floating-point subtractor. These subtractors are amongst the most well studied hardware components and are the target of deep optimization efforts. Specifically we will demonstrate how the tool is able to optimize a half-precision floating-point subtractor, that computes $2^{ea} \times 1.ma - 2^{eb} \times 1.mb$, producing a half-precision floating-point output. We focus on the subtraction case because it is well-known to be harder [1] due to the potential for cancellation. For simplicity, we consider the case where the output is rounded towards zero. We ignore exception handling and denormal, `NaN` and infinite inputs.

The input design in Figure 2a, is a naive, easy to write, implementation of the mantissa calculation for floating-point subtraction. In the naive design, the mantissas are concatenated with preceding zeros to ensure that after the alignment, all bits are retained. This results in a 42 bit subtractor, the result of which is normalized, using an LZC and a right shift. Discarding the leading one from this result, the output mantissa is then the 10 most significant remaining bits.

The most well-known floating-point subtract optimization known as the near-path/far-path optimization, stems from the observation that the critical path is never fully exercised [1], [3]. It splits the design into two paths. The near path is taken when $|ea - eb| < 2$, requiring only a small alignment shift. The far path is taken when $|ea - eb| > 1$. On this path catastrophic cancellation cannot occur, simplifying the renormalization logic. Figure 1 shows a related optimization.

The tool parses the RTL corresponding to Figure 2a and generates an initial e-graph. Typically LZCs are expressed in Verilog as case statements. To enable operator specific rewrites, we include an LZC operator in the intermediate language, with accompanying rewrites to map appropriate case statements to this operator.

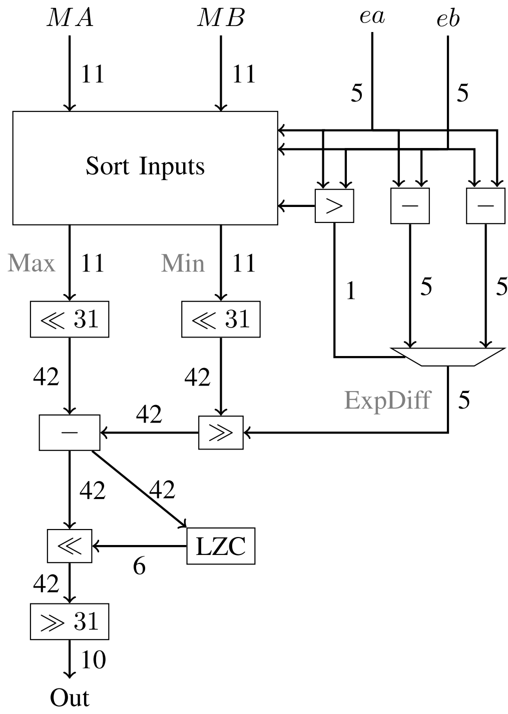
Once the tool has mapped the input design to its intermediate language optimization begins. Firstly, the tool introduces the case split into the e-graph via a rewrite, which is intended to possibly isolate catastrophic cancellation to a single branch.

$$a - (b >> c) \rightarrow (c > 1) \,?\, a - (b >> c) : a - (b >> c)$$

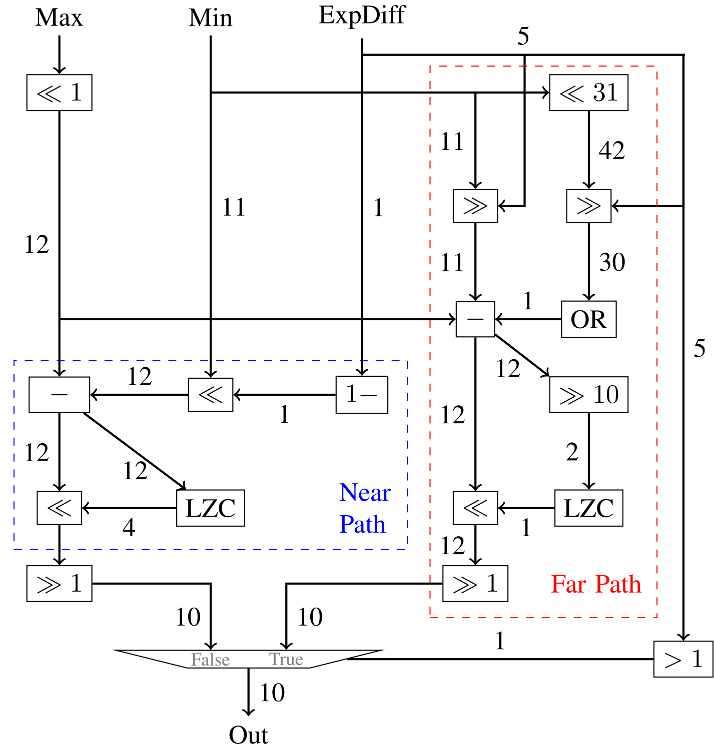
Note that this inserts a mux directly after the subtraction, which may be beneficial in some instances, but in this case using mux propagation rewrites, the tool pushes the mux towards the output.

$$a \;\texttt{op}\; (b \,?\, c : d) \rightarrow b \,?\, a \;\texttt{op}\; c : a \;\texttt{op}\; d$$

This duplicates operations, leaving a mux at the output between two identical branches. Using the rewrites from Table I,



(a) Behavioural architecture, using a 42 bit subtraction.



(b) Dual path optimized architecture, using two 12 bit subtractors. Near path uses a one bit alignment shift, followed by a larger renormalization stage. Far path uses a 12 bit alignment shift and a single bit renormalization.

Fig. 2: Half-precision floating-point subtractor architectures. Input mantissas have the implicit one appended. Edge labels represent bitwidths. The inputs are sorted according to $ea > eb \vee (ea == eb \wedge MA > MB)$. In the optimized architecture diagram, we omitted input sorting and exponent difference calculation blocks, as they were unchanged by the optimization.

the tool creates and propagates `ASSUME`s down each branch and after several rewriting iterations, the following are created.

$$\texttt{ASSUME}(\text{ExpDiff}, \quad \text{ExpDiff} > 1) \tag{8}$$

$$\texttt{ASSUME}(\text{ExpDiff}, \sim (\text{ExpDiff} > 1)) \tag{9}$$

Computing the abstraction of (8), (4) takes effect immediately. For (9), two sequential condition rewrites transform it into an equivalent *Constr*, ExpDiff < 2. These constrained value ranges are propagated along each branch triggering a chain of branch specific rewrites and bitwidth reductions.

After 11 iterations of rewriting an e-graph of approximately 40,000 nodes and 14,000 classes is grown. The optimized design shown in Figure 2b is extracted from this e-graph, within a total runtime of 22 minutes, the majority of which is spent growing the e-graph. Note that the two branches are reduced to distinct implementations as a result of the constraint-aware rewrites. Further optimizations apply to even this design, but these are often gate level optimizations, a level of optimization we have left to the logic synthesis tools.

The input and optimized RTLs are proven equivalent using the Synopsys Datapath Validation (DPV) tool, a formal equivalence checking tool that runs in minutes on this problem. We synthesized both designs at a range of delay targets using Synopsys Fusion Compiler for a TSMC 7nm cell library. The results are shown in Figure 3. The optimized design can achieve a 33% lower delay in a circuit area 41% smaller than the behavioural architecture.

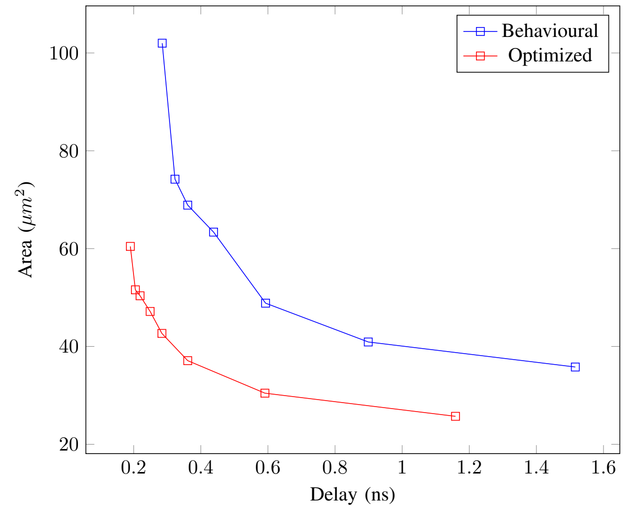


Fig. 3: Area-delay plot of competing floating-point subtractors.

TABLE III: Logic synthesis results using Fusion compiler for circuit delay (ns) and circuit area (μm^2). Each implementation is synthesized at the minimum delay target it can achieve.

Test Case	Behavioural		Optimized	
	ns	μm^2	ns	μm^2
FP Sub	0.285	102.0	0.190 (**-33%**)	60.4 (**-41%**)
float to unorm	0.055	17.6	0.056 (**+2%**)	13.6 (**-23%**)
interpolation	0.245	433.0	0.254 (**+3%**)	353.0 (**-18%**)
unorm to float	0.039	13.4	0.039 (**+0%**)	7.0 (**-48%**)

VI. Further Results

To demonstrate that the results of Section V are not hand-tuned to the case study, this section uses the tool to optimize several different design. We ran the tool for six iterations on smaller test cases generating e-graphs of less than 150 nodes, running in under 0.25 seconds. We demonstrate how the tool can automatically do dead code elimination and generalize optimizations learnt from the floating-point case study. The behavioural and optimized RTLs are proven equivalent using the DPV tool. The competing RTLs are synthesized using Fusion Compiler at the minimum delay target that each implementation can meet. Table III summarizes the results.

The float to unorm design converts a half-precision float (less than or equal to 1 in magnitude) to a unorm11, rounding down, as described in the DirectX specifications [17]. The tool reuses the round-off based optimizations from Section V. The interpolation example is a kernel from an Intel media module, computing an interpolation between four pixels and clamping the output. For certain clamping thresholds, the tool automatically detects that the threshold can never be met and optimizes the clamping away.

$$c \; ? \; a \; : \; b \rightarrow b \;\; \text{if } \mathcal{A}[\![c]\!] == [0,0]$$

This test case relies on rewriting to obtain tight approximations to the range of outputs and only with this rewriting can the tool prove that the clamping is unnecessary. Namely, naive interval arithmetic would not suffice.

The unorm to float design special cases zero inputs, such that they are handled on a separate path. The tool automatically propagates the domain restriction and applies the constraint-aware optimizations generating a smaller circuit that matches the behavioural's performance. In this example the interplay of rewriting and program analysis is invaluable.

VII. Conclusion

This paper combines e-graphs with constraint-aware program analysis to automate RTL optimization exceeding the capabilities of existing EDA tools. By representing multiple equivalence relations in an e-graph we exploit branch specific optimizations and compute tight approximations to intermediate signals using an extension of interval arithmetic. These techniques enable bitwidth reduction, dead code elimination and automated case splitting. We automated the optimization of a floating-point subtraction unit recreating efficient human implementations and saving 41% of area and 33% of delay. In other test cases the tool reduced circuit area by up to 48% with minimal delay penalty demonstrating its generalizability.

Future work will look to refine the delay model by considering a bit-level delay model, and explore multi-objective optimization to generate Pareto curves of designs. Having matched human designers on floating-point unit design, we hope to use the tool to discover novel architectures in the near future. An unexplored interactive tool usage, would be for designers to propose case-splits based on their intuition and have the tool automatically optimize the proposed design.

Acknowledgment

The authors acknowledge the discussions with the creators of the `egg` library, in particular Pavel Panchekha and Max Willsey for the initial idea to use a variation on our `ASSUME` nodes.

References

[1] A. Beaumont-Smith, N. Burgess, S. Lefrere, and C. C. Lim, "Reduced latency IEEE floating-point standard adder architectures," *Proceedings - Symposium on Computer Arithmetic*, 1999.

[2] J. Sohn and E. E. Swartzlander, "Improved architectures for a fused floating-point add-subtract unit," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 59, no. 10, 2012.

[3] P. M. Farmwald, *On the design of high performance digital arithmetic units*. Stanford University, 1981.

[4] C. G. Nelson, "Techniques for program verification," Ph.D. dissertation, Stanford University, 1980.

[5] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "Egg: Fast and extensible equality saturation," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, 2021.

[6] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: A new approach to optimization," in *ACM SIGPLAN Notices*, vol. 44, no. 1, 2009.

[7] S. Coward, G. A. Constantinides, and T. Drane, "Combining E-Graphs with Abstract Interpretation," 5 2022. [Online]. Available: https://arxiv.org/abs/2205.14989

[8] K. D. Cooper and L. Torczon, *Engineering a compiler: Second edition*. Morgan Kaufmann, 2011, vol. 9780120884780.

[9] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, CGO*, 2004.

[10] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, vol. Part F130756, 1977.

[11] L. Chen, A. Miné, and P. Cousot, "A sound floating-point polyhedra abstract domain," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5356 LNCS, 2008.

[12] A. K. Verma, P. Brisk, and P. Ienne, "Data-flow transformations to maximize the use of carry-save representation in arithmetic circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1761–1774, 2008.

[13] S. Coward, G. A. Constantinides, and T. Drane, "Automatic Datapath Optimization using E-Graphs," 4 2022. [Online]. Available: https://arxiv.org/abs/2204.11478

[14] P. Cousot, *Principles of Abstract Interpretation*, 1st ed. MIT Press, 9 2021.

[15] C. Wolf and J. Glaser, "Yosys-A Free Verilog Synthesis Suite," in *Proceedings of Austrochip*, 2013.

[16] E. Ustun, I. San, J. Yin, C. Yu, and Z. Zhang, "IMpress: Large Integer Multiplication Expression Rewriting for FPGA HLS," in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2022, pp. 1–10.

[17] Microsoft, "DirectX-Specs," Tech. Rep., 2022. [Online]. Available: https://microsoft.github.io/DirectX-Specs/