UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
CURSO DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

Isac de Souza Campos

Explorando Árvores de Decisão Em Um Fluxo de Síntese Para Circuitos Aproximados

Florianópolis
2022

Isac de Souza Campos

Explorando Árvores de Decisão Em Um Fluxo de Síntese Para Circuitos Aproximados

Relatório da disciplina Trabalho de Conclusão de Curso do curso de Ciências da Computação da Universidade Federal de Santa Catarina.
Orientadora: Prof. Cristina Meinhardt, Dra.
Coorientador: Prof. Mateus Grellert, Dr.

Florianópolis
2022

RESUMO

A etapa de síntese lógica no desenvolvimento de circuitos integrados tem se tornado mais desafiadora nos últimos anos devido ao crescimento na complexidade dos circuitos projetados. A escala nanométrica dos atuais transistores permite maior integração em um mesmo chip, implicando em funções mais complexas, com mais entradas e mais termos a serem otimizados. Muitos dos métodos de otimização lógica tradicional atingem seu limite de otimização em poucas dezenas de entradas, ou conseguem minimizar funções complexas ao custo de grandes tempos de execução. Algoritmos de *Aprendizado de Máquina* vêm se tornando mais comuns em diversas áreas da tecnologia, incluindo a área de ferramentas para o projeto de circuitos integrados, conhecida como *Electronic Design Automation*. Explorar essas ferramentas visando otimizar o tempo de execução da síntese lógica e analisar seu comportamento nos resultados de área e potência são os objetivos deste trabalho. Desta forma, este trabalho propõe um fluxo de síntese partindo de uma Tabela Verdade do circuito como entrada e apresentando soluções de síntese voltadas a baixo consumo energético, adotando, em alguns casos, Computação Aproximada. A otimização lógica desenvolvida é baseada em Árvores de Decisão, permitindo que a minimização lógica produza saídas exatas, assim como também possibilitando que alguma incerteza seja inserida no sistema através de restrições na profundidade da Árvore, por exemplo. A exploração de aproximação nas soluções minimizadas pode levar a circuitos com melhor eficiência energética mantendo níveis aceitáveis de precisão para aplicações tolerantes a erro. O fluxo de síntese proposto permite a comparação entre a síntese utilizando ferramentas tradicionais com aquela obtida pelo método de Árvore de Decisão. A saída da minimização lógica é direcionada para o fluxo OpenROAD. O OpenROAD é um projeto de *Electronic Design Automation* que utiliza diversas ferramentas *open-source* integradas e permite uma síntese *standard cell* mapeada para uma tecnologia ASIC. Os resultados observados mostram o quanto abordagens de Computação Aproximada podem ser promissoras, tendo reduzido a média de área, atraso e potência estudados para boa parte dos casos avaliados. Essa média teve um aumento quando se aplica Árvore de Decisão definindo uma acurácia de 100% quando comparado com o fluxo OpenROAD sem Árvore de Decisão, mas reduziu continuamente com a diminuição da precisão do circuito escolhida. Com uma acurácia de 90%, por exemplo, a média de área e potência do conjunto estudado diminuiu 41,49% e 47,19%, respectivamente, em comparação com os resultados obtidos com o OpenROAD.

Palavras-chave: Síntese Lógica. OpenROAD. Aprendizado de Máquina. Árvore de Decisão.

ABSTRACT

The logic synthesis step in the development of integrated circuits has become more challenging in recent years due to the growth in the complexity of designed circuits. The nanometric scale of current transistors allows greater integration on the same chip, implying more complex functions, with more inputs and more terms to be optimized. Many of the traditional logic optimization methods reach their optimization limit in a few dozen inputs, or manage to minimize complex functions at the cost of long execution times. Machine Learning Algorithms are becoming more common in several areas of technology, including the area of tools for the design of integrated circuits, known as Electronic Design Automation. Exploring these tools in order to optimize the execution time of the logic synthesis and analyze their behavior in the area and power results are the objectives of this work. Thus, this work proposes a synthesis flow starting from a circuits' Truth Table as input and presenting synthesis solutions aiming low energy consumption, adopting, in some cases, Approximate Computing. The developed logic optimization is based on Decision Trees, allowing the logical minimization to produce exact outputs, as well as allowing some uncertainty to be inserted in the system through restrictions in the depth of the tree, for example. The approximation in the minimized solutions can lead to circuits with better energy efficiency maintaining acceptable levels of accuracy for error tolerant applications. The proposed synthesis flow allows the comparison between the synthesis using traditional tools and the one obtained by the Decision Tree method. The output of logical minimization is directed to the OpenROAD flow. OpenROAD is an Electronic Design Automation project that uses several integrated open-source tools and allows a standard cell synthesis mapped to an ASIC technology. The observed results show how promising Approximate Computing approaches can be, having reduced the average of the area, delay and power experienced for most of the evaluated cases. This average had an increase when Decision Tree is applied defining an accuracy of 100% when compared to the OpenROAD flow without Decision Tree, but it continuously decreased with the chosen circuits' precision reduction. With an accuracy of 90%, for example, the average area and power of the studied set decreased by 41.49% and 47.19%, respectively, compared to the obtained OpenROAD results.

Keywords: Logic Synthesis. OpenROAD. Machine Learning. Decision tree.

SUMÁRIO

1 INTRODUÇÃO

O mundo contemporâneo tem a tecnologia como parte fundamental do seu funcionamento. Estruturas sociais e econômicas do século 21 se formaram com base em dispositivos eletrônicos e dependem destas mesmas ferramentas para manterem-se ativas. Segundo estudo da Fundação Getúlio Vargas (FGV) (MEIRELLES, 2020) publicado em 2020, o número de dispositivos digitais (computadores, *notebooks, tablets* e *smartphones*) em uso no Brasil atualmente está em torno de 440 milhões de unidades, o que corresponde a aproximadamente dois dispositivos *per capita* no país. Além dos aparelhos pesquisados pela FGV, a tecnologia eletrônica está presente em diversos outros utensílios cotidianos como, televisão, microondas, geladeira, ar-condicionado, etc.

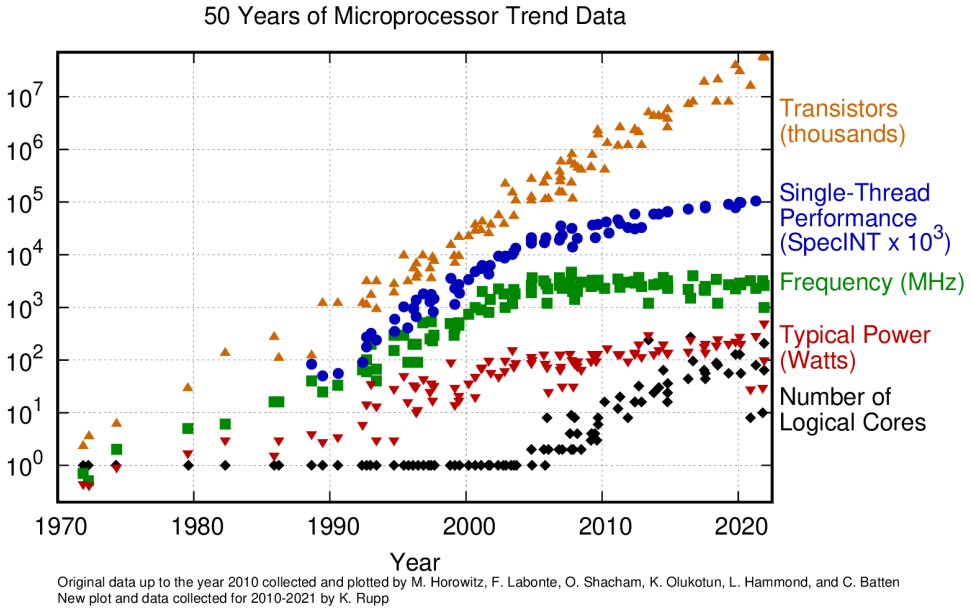
A alta demanda destes produtos gera interesse em seus aprimoramentos por parte do mercado. O número de transistores por componentes cresceu de maneira acelerada com o passar das décadas, possibilitando um aumento de desempenho das tecnologias que os utilizam. Esse comportamento já havia sido previsto em meados da década de 60, pelo cofundador da *Intel Corporation*, Gordon Moore, que estimou um prazo de dois anos para que a quantidade de transistores em um circuito eletrônico dobrasse (MOORE *et al.*, 1965). A Figura 1 apresenta o crescimento exponencial do número de transistores de 1970 a 2020 (RUPP, 2022). O gráfico também detalha a evolução da performance de *single-threads*, estratégia adotada para aumentar o número de instruções realizadas pelo processador, mesmo que a frequência permaneça estabilizada nas últimas décadas. Também é possível observar uma estabilização na potência, e a tendência de crescimento para o número de núcleos lógicos dos circuitos.

Essa ampliação no número de transistores nos circuitos permite o aumento da complexidade das funções realizadas em *hardware*. Por um lado isso é positivo, pois possibilita a construção de sistemas mais avançados e a criação de tecnologias modernas. Por outro lado, torna seu processo de produção mais difícil e demorado, exigindo aprimoramentos na etapa de desenvolvimento que acompanhem a velocidade desses avanços.

O projeto de circuitos usualmente envolve gerar uma descrição fabricável do circuito partindo de uma descrição de alto nível, utilizando linguagens de descrição de hardware. Esse processo é realizado com ferramentas que realizam duas grandes etapas: (1) síntese lógica, que traduzem a descrição de alto nível para uma versão intermediária mais próxima dos dispositivos utilizados na fabricação (e.g., *Standard Cells*), chamada *netlist*; e (2) síntese física, que realiza posicionamento das células, geração de caminhos de roteamento etc.

A otimização lógica, um dos passos da síntese lógica, é uma das etapas com potencial de proporcionar benefícios na produção e qualidade dos circuitos, gerando interesse da indústria pela sua melhoria. Esses procedimentos geram ganhos no desempenho com a redução do número de termos da função e profundidade lógica, impactando área, atraso

Figura 1 – Evolução das características dos circuitos ao longo dos anos



Fonte: (RUPP, 2022).

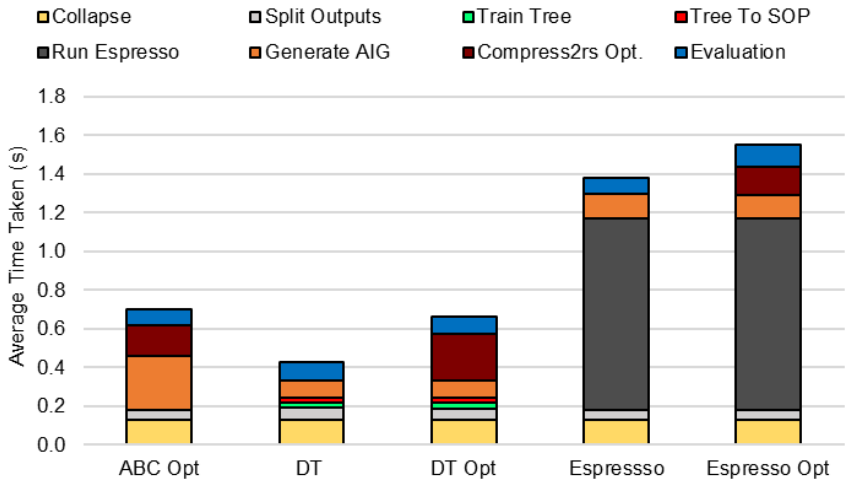
de propagação, e consumo energético. Alguns algoritmos clássicos que desempenham esse papel são o Mapa de Karnaugh (KARNAUGH, 1953), Quine-McCluskey (QUINE, 1955) e Espresso (RUDELL; SANGIOVANNI-VINCENTELLI, 1987; BRAYTON, Robert K. *et al.*, 1984). Porém, esses algoritmos apresentam um elevado custo computacional, especialmente para o processamento de circuitos que excedam 10 *bits* de entrada. O Espresso, por exemplo, pode se tornar inviável nesses casos por conta da alta complexidade do seu algoritmo, exigindo um tempo de execução consideravelmente alto.

Uma alternativa para o aprimoramento da síntese lógica é a aplicação de Aprendizado de Máquina. Recentemente, técnicas de Aprendizado de Máquina passaram a ser utilizadas na área de EDA (*Electronic Design Automation*), possuindo ainda bastante espaço para crescimento (BEEREL; PEDRAM, 2018; PANDEY, 2018). Estas técnicas proporcionam mecanismos poderosos para alcançar objetivos complexos, desde que soluções possam ser obtidas através de dados. Dentre os métodos existentes em Aprendizado de Máquina, um que se destaca para resolução de problemas binários é a Árvore de Decisão. Este procedimento gera como resultado uma estrutura de dados em formato de árvore, que pode facilmente ser convertida para uma função lógica de Soma de Produtos ou Produto de Somas, por exemplo.

Além disso, estudos prévios demonstraram um bom desempenho quando comparado a abordagens clássicas, como o já citado Espresso (ABREU, Brunno A. de *et al.*, 2021). Isso pode ser notado ao analisar-se o gráfico da Figura 2, que mostra uma comparação entre essas técnicas de otimização lógica, incluindo Espresso e uma abordagem de síntese lógica

baseada em Árvore de Decisão explorada em (ABREU, Brunno A. de *et al.*, 2021). As barras do gráfico representam o tempo médio de execução das ferramentas, diferenciadas por cor, envolvidas na síntese lógica para 32 *benchmarks* analisados. Em (ABREU, Brunno A. de *et al.*, 2021), os métodos envolvendo Árvore de Decisão são denominados de *DT* (*Decision Tree*) e *DT Opt* (*Decision Tree and Optimization*), com este segundo sendo composto por otimizações executadas pelo *software* ABC, que também será discutido neste trabalho. Observa-se o elevado tempo de processamento da ferramenta Expresso, praticamente duplicando o tempo médio de execução nestes casos avaliados.

Figura 2 – Comparativo do tempo de execução para diferentes abordagens de síntese lógica



Fonte: (ABREU, Brunno A. de *et al.*, 2021).

A ferramenta ABC (BERKELEY LOGIC SYNTHESIS AND VERIFICATION GROUP, 2019), usada na abordagem *DT Opt* da Figura 2, tem como propósito a conversão de formatos intermediários a fim de tornar possível a integração entre técnicas já conhecidas e o fluxo proposto neste trabalho. Algoritmos de otimização também estão presentes no ABC, possibilitando a redução do circuito final e caminho crítico, além da melhoria com gasto energético proporcionada por esses aprimoramentos. A combinação das ferramentas será explorada com o intuito de potencializar os possíveis ganhos e analisar os diferentes comportamentos e resultados.

Outro aspecto interessante dos métodos de Aprendizado de Máquina é o fato de permitirem treinar modelos simplificados, sem um detalhamento completo, ao custo de uma taxa de acerto (acurácia) reduzida. Isso pode ser explorado para aplicação em outra área também em crescimento e bastante promissora: a computação aproximada. Algumas aplicações possuem certa tolerância a perda de qualidade sem comprometer seu funcionamento. Exemplos bastante conhecidos são as técnicas de compressão de vídeo, que aceita uma pequena taxa de erros na reprodução dos pixeis sem que isso seja perceptível pelo sistema visual humano, viabilizando a diminuição da qualidade em troca de um melhor desempenho. A computação aproximada é um paradigma emergente na área de EDA que tem sido discutido nos últimos anos (SCARABOTTOLO *et al.*, 2020) e pode

ser explorado na etapa de síntese lógica (VENKATARAMANI; KOZHIKKOTTU *et al.*, 2019), principalmente pensando no projeto visando eficiência energética.

Este trabalho tem como objetivo usar o método de Árvore de Decisão para minimização lógica na etapa de Síntese Lógica. O método permite a síntese de funções exatas e a aplicação de Computação Aproximada. A alteração do resultado de acurácia para os circuitos pode ser feita através da manipulação nos parâmetros da Árvore. O fluxo desenvolvido permite que o usuário possa escolher mapear a tecnologia desenvolvida para uma FPGA (*Field-Programmable Gate Array*) ou criar um circuito dedicado pela metodologia *standard cell* de geração de circuitos ASIC (*Application-Specific Integrated Circuit*) de acordo com a necessidade do projeto. Neste trabalho, será priorizada a abordagem de integração para a metodologia *standard cell*, utilizando a ferramenta de código aberto *Open-Road* (ROVINSKI *et al.*, 2020). Os resultados são comparados com abordagens clássicas e ferramentas disponíveis, comerciais e *open-source*, para uma melhor compreensão do impacto desta estratégia no desempenho da síntese.

1.1 OBJETIVOS

Neste sentido, o objetivo geral deste trabalho é desenvolver um fluxo de síntese para funções exatas e aproximadas explorando árvores de decisão na otimização lógica. Além disso, define-se os seguintes pontos como objetivos específicos:

- Propor um método de otimização lógica baseado em árvores de decisão;
- Integrar o método em fluxo de síntese de circuitos integrados ASIC;
- Avaliar o fluxo proposto comparando com ferramentas de otimização lógica;
- Disponibilizar a solução integrada em ambientes *open-source*.

1.2 ORGANIZAÇÃO DO TEXTO

O próximo Capítulo apresenta os principais conceitos para o entendimento do trabalho desenvolvido. O Capítulo 3 analisa trabalhos que tratem de assuntos relacionados ao projeto proposto. Já o Capítulo 4 discute a proposta e o desenvolvimento do fluxo de síntese baseado em Árvores de Decisão, apresentando os resultados para o estudo de caso abordado no Capítulo 5. O Capítulo 6 apresenta o fluxo como se encontra, após aplicação de Computação Aproximada e integração com o OpenROAD, além de avaliar os resultados obtidos. As conclusões gerais sobre o que se observou durante o desenvolvimento e desempenho do fluxo é visto no Capítulo 7.

2 CONCEITOS

Para a elaboração e entendimento da da ferramenta proposta neste trabalho, é importante descrever as estruturas e ferramentas utilizadas a fim de entender-se o motivo para suas escolhas. Estruturas de dados eficientes alinhadas a ferramentas de qualidade podem fornecer um ganho considerável de desempenho. Este Capítulo apresenta os principais conceitos de Síntese Lógica, as relações entre estruturas clássicas de síntese lógica e a estrutura de árvores de decisão, e, finalmente, discute algumas ferramentas consideradas na execução deste trabalho.

2.1 SÍNTESE LÓGICA

O projeto de circuitos integrados (CIs) é um processo amplo que envolve várias etapas de abstração e otimização que vão desde a elaboração do *design* pelo projetista até o circuito final. A fase de Síntese Lógica é de grande importância no fluxo e tem por objetivo converter uma definição de alto nível da tecnologia para uma de mais baixo nível que contenha detalhes necessários para os próximos passos.

A descrição de um *hardware* em alto nível é feita através de uma Linguagem de Descrição de Hardware (*HDL - Hardware Description Language*). Por questão de praticidade, a descrição do circuito é inicialmente feita de maneira comportamental, ou seja, indicando o funcionamento da lógica através de equações Booleanas ou algoritmos em linguagem de alto nível sem necessitar muito detalhamento. Esse formato é mais adequado para o uso humano pois aproxima-se mais da nossa linguagem, mas não é suficiente para que se possa dar continuidade ao fluxo de síntese, pois carece de informações importantes como a lista de conexões (*netlist*) que deverá ser gerada para as etapas de posicionamento e roteamento do circuito, por exemplo.

Assim, é necessário converter a descrição comportamental em estrutural. Primeiro é feita uma Síntese de Alto Nível, que converte a descrição comportamental da tecnologia em uma descrição de *hardware* como, por exemplo, *RTL* (*Register Transfer Level*). A partir desta etapa aplica-se a Síntese Lógica. Essa fase divide-se em três: transformação do RTL em estruturas intermediárias; mapeamento para a tecnologia do circuito final (*FPGA*, *ASIC*); otimizações dependentes da tecnologia mapeada. Além dos aprimoramentos pós mapeamento tecnológico, também aplicam-se técnicas de otimização lógica nas estruturas intermediárias.

A primeira fase aplica transformações que não dependem da tecnologia do circuito final, dependendo apenas do comportamento funcional desejado, e gera uma estrutura intermediária, que pode ser, por exemplo, uma expressão Booleana, uma rede Booleana ou um grafo (representado como um Grafo de *Ands* Inversoras *AIG*, Árvore, ou outra estrutura).

Uma possível forma de realizar a síntese independente de tecnologia é utilizando

métodos de síntese de funções Booleanas. Um método de síntese de funções Booleanas busca representar um dado comportamento lógico, geralmente representado por uma tabela verdade, em uma expressão Booleana contendo literais e operadores Booleanos. Existem diversos métodos de síntese de funções Booleanas e cada método pode apresentar diferentes abordagens. Uma função Booleana pode ser sintetizada em uma estrutura de dois níveis, como somas-de-produtos (SoP) ou produto-de-somas (PoS), ou em uma expressão fatorada de estrutura multi-nível (MICHELI, 1994). Existem abordagens onde a representação de funções pode ser restringido ao uso de operações de AND e OR, enquanto outras abordagens utilizam também a operação de OU-Exclusivo (XOR).

Em seguida, o mapeamento tecnológico mapeia partes da estrutura intermediária em células com informações relativas a tecnologia do circuito final. Por fim, a otimização dependente de tecnologia realiza otimizações nas células mapeadas, como redimensionamento de células e duplicação de lógica.

2.1.1 Síntese Lógica Aproximada

A aplicação da estratégia de Computação Aproximada pode ser feita em diferentes etapas do projeto de circuito, sendo uma delas a de Síntese Lógica. Nesse estágio é mais fácil automatizar os processos, o que permite a aplicação de aproximação sem necessitar de um alto nível de expertise por parte do projetista e possibilita implementar a estratégia em qualquer qualquer circuito, mesmo sem um amplo conhecimento prévio deste (SCARABOTTOLO *et al.*, 2020).

Essas técnicas visam modificar a função que representa o circuito de modo a reduzir seu tamanho final ao passo em que obtém-se um nível aceitável de erro quando comparada à original. Alterar algumas saídas de uma Tabela Verdade propositalmente é uma das formas de aproximar a lógica do circuito. Outra forma é através da manipulação da função *Booleana* que o representa. Em alguns casos, como será abordado neste trabalho, altera-se a representação do circuito num formato de Árvore intermediário à Tabela Verdade e o *Booleano*. Independente do método que se use, o propósito é adquirir uma função menor do que a original sem grandes perdas de precisão na saída.

2.2 ESTRUTURAS DE DADOS

Os problemas de otimização lógica envolvem a busca por similaridades nas equações que permitam a redução dos termos necessários para definir uma mesma função. Encontrar estas vizinhanças em circuitos com várias entradas é um problema computacional grande, onde as combinações entre o número de entradas (variáveis do sistema) crescem rapidamente. Nestes problemas, adotar uma estrutura de dados adequada é de fundamental relevância. Este trabalho detalha duas estruturas de dados: 1) o *And Inverter Graph* (AIG) por ser atualmente a estrutura de dados adotada pelas ferramentas estado-da-arte

na área (BIERE, 2007); e 2) árvores, por serem as estruturas de dados de onde derivam as ideias exploradas em algoritmos de aprendizagem de máquina baseados em árvores de decisão.

2.2.1 AIG

A estrutura de dados utilizada em qualquer algoritmo ou programa computacional é de extrema importância quanto ao desempenho e qualidade do resultado final gerado. Uma conhecida estrutura na área de tecnologia é a de grafos. Um grafo é uma rede de pontos, chamados nodos, conectados por linhas ou, quando o sentido da ação é importante, setas (BERGE, 2001). Eles podem ser utilizados para representar uma grande variedade de situações cotidianas e são aplicados na representação de mapas, estruturas sociais, diagramas, etc.

Em específico, este trabalho tem interesse no grafo denominado *And Interver Graph*. Como o nome sugere (Grafo de *And* Inversoras, em tradução livre), AIG serve para representar portas lógicas *AND* inversoras, também conhecidas como *NAND*. Uma propriedade interessante dessa porta lógica é a capacidade de se representar qualquer circuito utilizando-a, como nota-se analisando a Figura 3. A Figura 3a demonstra a representação de um Inversor, replicando um mesmo valor nas duas entradas da NAND. Para gerar o comportamento de uma AND basta conectar-se duas NANDs em série, anulando a negação, como vê-se na Figura 3b. O Teorema de De Morgan pode ser explorado para gerar o comportamento de uma OR, como visto na Figura 3c.

Uma mesma função pode ser representada por diversas combinações no formato AIG. A Figura 4 exemplifica um caso da função F = (**a** *AND* **b** *AND* **c**), descrita inicialmente por três portas *AND* e convertida para um formato com 2 portas. Casos como este podem aparecer como subgrupos de uma função maior em diversos pontos do grafo, causando grande impacto no seu tamanho. Algoritmos de otimização fazem uma varredura no AIG refatorando cenários como o da função F para que se reduza a área sem impactar o atraso do circuito.

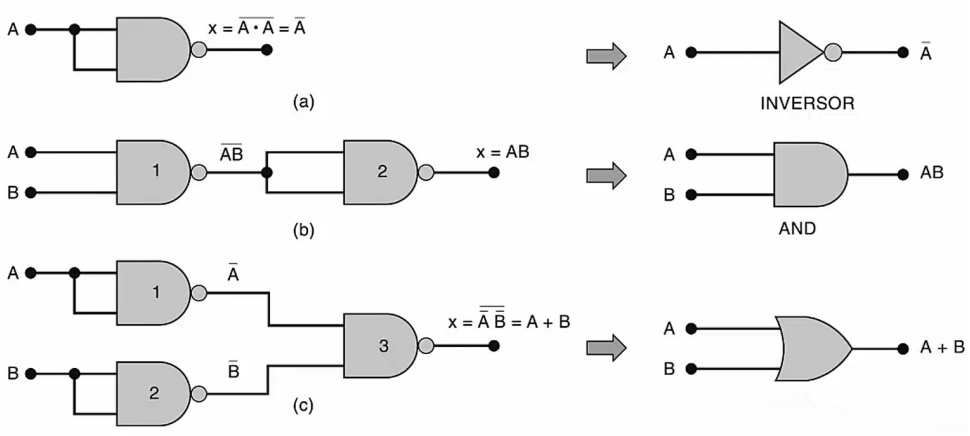
O AIG vem sendo adotado pela maioria das ferramentas de síntese lógica atuais. Sendo assim, a manipulação desta estrutura será adotada no fluxo proposto neste trabalho, sendo inteiramente manipulada através da ferramenta de síntese lógica ABC no decorrer do fluxo proposto.

2.2.2 Árvores de Decisão

No estudo de estruturas de dados, árvores representam estruturas não lineares (em contraponto a listas, por exemplo) que possuem características particulares quanto à sua estrutura e seus atributos. Algumas características dessa estrutura são:

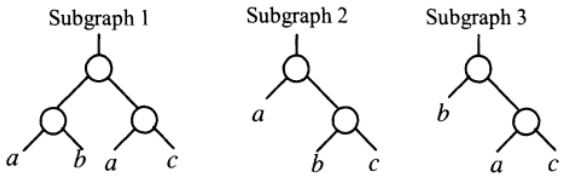
- **Raiz**: nó inicial da Árvore;

Figura 3 – Representação de Funções lógicas com a porta NAND



Fonte: `https://www.youtube.com/watch?v=eASj8pWIpJo`

Figura 4 – Otimização de um subgrupo do AIG



Fonte: (MISHCHENKO; CHATTERJEE; BRAYTON, 2006)

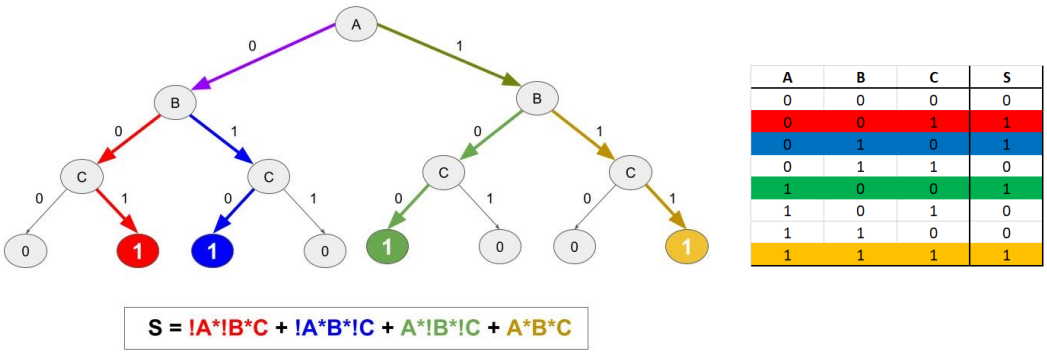
- **Grau**: número de filhos de um nó;
- **Nível (Profundidade)**: distância de um nó até a raiz;
- **Altura**: maior nível da Árvore;
- **Folha**: nó de grau zero.

Árvores são comumente utilizadas para otimizar operações de pesquisa, pois sua estrutura hierárquica permite que essa operação seja realizada em tempo logarítmico (desde que restrições de balanceamento sejam respeitadas).

No contexto de EDA, árvores encontram aplicação no uso de *Binary Decision Trees* (BDT) ou Árvores de Decisão Binárias. Essas árvores são utilizadas para representar funções lógicas, possibilitando que ferramentas de síntese lógica explorem as propriedades destas estruturas na implementação dos seus algoritmos. A Árvore vista na Figura 5 é um exemplo de BDT. Essa Estrutura de Dados é uma representação canônica de uma Tabela Verdade completa, como a que vimos na mesma Figura. As Árvores de Decisão, por outro lado, podem gerar representações mais compactas dependendo da tecnologia em desenvolvimento, além de permitirem reduzir seu tamanho e formato com a manipulação de parâmetros.

A obtenção dessas estruturas se dará pela execução do algoritmo de Árvore de Decisão, que consiste em dividir sequencialmente um grupo de dados de entrada até que se chegue a resultados unitários, contidos nos nodos folha, encontrando um padrão

Figura 5 – Representação do processo de conversão de Árvore para Soma de Produtos



A	B	C	S
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Fonte: O autor.

comportamental na divisão da estrutura para cada circuito analisado. Os dados de entrada tratados no fluxo proposto são os *bits* de entrada do circuito sintetizado e o *bit* de resultado esperado para determinada entrada é encontrado nos nós folhas gerados.

A forma com que essas divisões ocorrem é definida através da análise de cálculos baseados no princípio da entropia. Uma variável *gini* contém o resultado da aleatoriedade de um conjunto pertencente a um nodo. Quanto mais misturado e despadronizado esse conjunto for, maior o valor de *gini*. Desta forma, deseja-se continuar a execução do algoritmo de modo a se atingir um valor zero para cada nodo da Árvore. Quando isso ocorre, tem-se um nó folha. A Árvore está finalizada quando não há mais nodos com *gini* diferente de zero a serem processados.

Este formato será obtido ao final do processo de aprendizagem executado pelo método de Árvore de Decisão usado no fluxo deste trabalho. A partir do arquivo resultante, consegue-se facilmente convertê-lo para outros reconhecidos pelas ferramentas envolvidas no processo. A altura da Árvore será de grande importância para o desempenho da técnica desenvolvida. Esse dado está ligado ao caminho crítico do circuito, tornando importante a sua redução sempre que viável, visando possibilitar o aumento da frequência do *clock* e diminuição do tempo de execução de suas operações.

2.3 FERRAMENTAS

Um *software* bem escrito é um ótimo complemento a um algoritmo eficiente. As ferramentas integradas ao fluxo deste trabalho são tradicionais na área de síntese lógica e possuem diversos estudos publicados com demonstrações de seus desempenhos. Todos possuem código aberto, enquadrando-se nos requisitos exigidos para o que está sendo aqui proposto.

2.3.1 ABC

Desde 1987, a Universidade da Califórnia (UC) desenvolve ferramentas de síntese lógica, tendo criado os programas MIS (BRAYTON, Robert K *et al.*, 1987), SIS (SENTOVICH *et al.*, 1992) e MVSIS (CHAI *et al.*, 2003), até chegar no ABC em 2005.

O ABC é um sistema CAD (*Computer-Aided Design*) que aplica transformações lógicas escaláveis baseadas em AIGs utilizando-se de algoritmos de otimização para este formato, e demonstrou desempenho superior ou igual ao de seus predecessores (BRAYTON; MISHCHENKO, 2010), exigindo menos memória e tempo de execução. Essas vantagens o permitem trabalhar com circuitos maiores do que os anteriores conseguiam.

Outra característica é a possibilidade de conversão entre os formatos comumente envolvidos nessa etapa, como: PLA, EQN, AIG, Verilog, etc. Isso dispensa a necessidade de outras tecnologias de conversão, juntando praticamente todos os recursos necessários nessa fase em um único sistema.

Por fornecer uma grande gama de funcionalidades relacionadas à síntese lógica e ter demonstrado, com o passar do tempo, maior capacidade de processamento para circuitos com entradas maiores do que outras ferramentas conseguem rodar, o ABC tem sido explorado em outras ferramentas de síntese lógica, como o Yosis (WOLF; GLASER; KEPLER, 2013). Por estas características, o ABC é visto como uma ótima ferramenta auxiliar para o nosso fluxo.

Neste trabalho, a ferramenta tem o objetivo de aplicar conversões de formatos entre as diferentes etapas da síntese lógica. Além disso, algoritmos de otimização são executados durante o processo com o intuito de obter-se a menor área e caminho crítico possíveis ao final da síntese. Será usado também um *script* contendo sequências de comandos que visam balancear a lógica do circuito em questão, sempre com interesse em ganhos de desempenho.

O ABC será relembrado posteriormente neste mesmo trabalho durante a apresentação e descrição de outra ferramenta de código aberto crucial para o funcionamento do fluxo planejado: o OpenRoad.

2.3.2 Scikit-Learn

Os últimos anos proporcionaram grande avanço às tecnologias de Inteligência Artificial (IA). Vários serviços e tarefas já contam com a presença desses algoritmos para auxiliar ou executar por completo suas funcionalidades.

O compartilhamento de códigos abertos pode ter sido essencial para o desenvolvimento acelerado desses algoritmos. Eles permitem que cientistas e desenvolvedores trabalhem em conjunto trocando conhecimento e aprimorando estratégias de aprendizagem das IAs.

O Scikit-Learn (SK) é um módulo composto por diversos algoritmos de IA baseados

em métodos de Aprendizagem de Máquina. Ele é escrito em Python e, como boa parte das ferramentas desenvolvidas nessa linguagem, tem o código aberto, permitindo não apenas sua utilização, mas a modificação e especialização do código quando necessário.

Através da configuração de parâmetros dos métodos disponíveis no SK, é possível manipular o caminho a ser seguido no processo de aprendizagem de modo a lapidar o resultado com base no que se deseja. Usando a Árvore de Decisão como exemplo, pois será o método aqui utilizado, pode-se citar as configurações de balanceamento e profundidade. Se a funcionalidade pra qual a ferramenta será usada necessita de uma Árvore com menor profundidade, isso é especificado através desses parâmetros no código.

A facilidade no uso deste módulo e as várias possibilidade de configuração dos algoritmos tornou o SK popular e utilizado em uma variedade de áreas da tecnologia. Pelos mesmos motivos, tornou-se atraente para o que se propõe neste trabalho e, por esta razão, foi escolhido para desempenhar a etapa de Aprendizagem de Máquina.

2.3.3 OpenRoad

O projeto de CIs envolve diversas etapas que necessitam de algoritmos complexos e robustos para boas otimizações do *design* desenvolvido. Tamanha complexidade gera grande valor e interesse em ferramentas que os executem com bons resultados. Essas ferramentas são geralmente privadas, dificultando o desenvolvimento de CIs em algumas áreas de pesquisa sem financiamento adequado. *Softwares* desenvolvidos por companhias privadas permitem apenas o acesso dos funcionários da empresa aos algoritmos em questão, limitando o compartilhamento de conhecimento agregado ao projeto.

Ademais, os altos custos das ferramentas junto com o elevado grau de expertise necessária para utilizá-las gera grandes riscos na tentativa de criar novos *designs* (AJAYI; CHHABRIA *et al.*, 2019). Como uma alternativa a essas barreiras e tendo como lema democratizar o desenvolvimento de *hardwares*, surge o projeto OpenRoad.

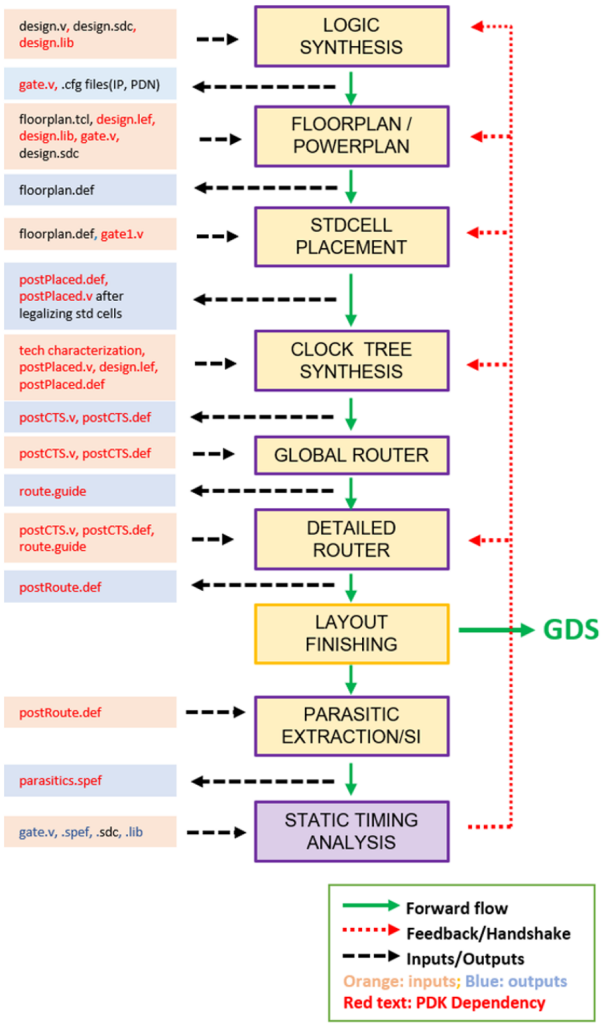
O projeto foi criado pelo programa DARPA IDEA (LIM, s.d.) e apresentado em julho de 2018 no evento *Electronics Resurgence Initiative* (ERI) *Summit* em São Francisco, Califórnia. O OpenRoad foi criado pelo departamento de defesa americano e visa desenvolver um compilador inteiramente automatizado de *hardwares* de propósito geral (OPENROAD, 2022), possibilitando a criação de tecnologias avançadas até mesmo por pesquisadores sem conhecimento muito aprofundado em EDA.

A Figura 6 mostra o fluxo completo do *OpenROAD*. Diferentes ferramentas são usadas para cada uma dessas etapas, e como o objetivo do projeto é ser inteiramente de código aberto, todas as tecnologias intermediárias usadas no processo também o são. Algumas dessas ferramentas e as etapas onde são utilizadas podem ser vistas na Tabela 1.

O fluxo visto nessa Tabela tem como entrada descrições de circuits na linguagem Verilog. Primeiro é feita a Síntese Lógica com o auxílio do Yosys e ABC. O ABC é integrado ao próprio Yosys, permitindo que a versão anterior do projeto deste trabalho

fosse modificada para remover o ABC que vinha sendo usado até então. Finalizada a Síntese Lógica passa-se por etapas como *Floorplan*, Posicionamento de Pinos e Conexões *Chip-Level*. A ferramenta responsável pelo primeiro leva o mesmo nome do processo que executa (*Floorplan*), já as outras são *ioPlacer* e *ICeWall*, respectivamente. Os procedimentos de Posicionamento Macro e Global, muito importantes por impactarem principalmente na área do CI, são executados pelas ferramentas *TritonMacroPlacer* e *RePlAce*. A Análise de *Timing* é feita pelo *OpenSTA*. Da mesma forma que os processos de Posicionamento são importantes pra área, os de Roteamento visam reduzir o tamanho dos fios otimizando os caminhos percorridos por esses. No *OpenROAD*, o Roteamento Global é feito por *FastRoute* e *Antenn Checker*, enquanto o Detalhado é executado pelo *TritonRoute*. Por fim, vemos na Tabela 1 que o Preenchimento de Metal é calculado pela ferramenta *Meta Fill*. Outras ferramentas e processos intermediários integram o fluxo *OpenROAD* completo e podem ser verificados na documentação do projeto.

Figura 6 – Funcionalidades do fluxo OpenRoad



Fonte: (AJAYI; BLAAUW, 2019)

Tabela 1 – Algumas Ferramentas Usadas no OpenROAD

Ferramenta	**Função**
Yosys e ABC	Síntese Lógica
Floorplan	Inicializa o Floorplan
ioPlacer	Posicionamento de Pinos
ICeWall	Conexões Chip-Level
TritonMacroPlacer	Posicionamento Macro
RePlAce	Posicionamento Global
OpenSTA	Análise de *Timing*
FastRoute e Antenna Cheker	Roteamento Global
TritonRoute	Roteamento Detalhado
Metal Fill	Preenchimento de Metal

Fonte: `https://theopenroadproject.org/`

2.4 COMENTÁRIOS FINAIS

A Síntese Lógica abrange várias etapas com diferentes possibilidades de combinações de processos que visam a otimização lógica do CI projetado e impacta consideravelmente o desempenho do mesmo. Há alguns anos esse processo poderia demorar horas, ou até dias para ser executado, dependendo do tamanho e complexidade do circuito em desenvolvimento.

Uma Estrutura de Dados de bom desempenho que passou a ser amplamente utilizada recentemente foi o AIG, que representa o circuito em formato de um grafo de NANDs, permitindo aplicar-se várias técnicas de remoção de redundâncias lógicas e balanceamento do grafo. O *ABC* possui uma variedade de funções para tratamento e otimização de AIGs que serão usadas através do *Yosys* neste trabalho, na fase de Síntese Lógica do *OpenROAD*.

Outra estrutura que chama a atenção é a Árvore de Decisão. Métodos de Aprendizagem de Máquina tem ganhado espaço no meio científico, incluindo EDA. Tendo em vista que Árvores podem ser facilmente convertidas em equações lógicas e o método de Árvore de Decisão possibilita, ainda, aplicação de Computação Aproximada por meio da manipulação de alguns parâmetros, a utilização desse método se torna ainda mais atraente.

Unir essas estratégias com um bom planejamento pode ter desfechos interessantes. Mais adiantes neste trabalho serão detalhados os processos de desenvolvimento do fluxo proposto e avaliados os seus resultados.

3 TRABALHOS RELACIONADOS

Por possibilitar a redução na área e consumo de energia dos circuitos desenvolvidos, a computação aproximada se tornou alvo de diversas pesquisas nos últimos anos, uma vez que o tamanho e complexidade das tecnologias vêm aumentando a preocupação com esses quesitos. Serão apresentados alguns exemplos de estudos conhecidos e recentes nessa seção para que se tenha uma noção de como encontra-se o estado da arte atualmente sobre síntese lógica aproximada.

A computação aproximada pode ser aplicada em diferentes áreas da computação, que vão desde usá-la em algoritmos de *Deep Learning* até o desenvolvimento de CIs, como mencionado em (VENKATARAMANI; CHAKRADHAR *et al.*, 2015). Neste trabalho tem-se interesse no segundo caso.

Existem diversas estratégias para implementação de Computação Aproximada em CIs. Circuitos somadores, multiplicadores e divisores, por terem funcionalidades bem conhecidas, permitem o desenvolvimento de lógicas de aproximação feitas manualmente, inclusive. Em (MASADEH; HASAN; TAHAR, 2018) faz-se uma comparação da área, potência, atraso e acurácia de diferentes estratégias de aproximação manual em multiplicadores.

Essa abordagem, porém, não é aplicável a quaisquer circuitos, gerando a necessidade de algoritmos e ferramentas especializadas para essa tarefa. Esses casos são analisados em (SCARABOTTOLO *et al.*, 2020), onde utiliza-se diferentes ferramentas de Síntese Lógica Aproximada (SLA) como ABACUS (NEPAL *et al.*, 2014), BLASYS (HASHEMI; TANN; REDA, 2018), entre outros, para detectar a técnica com maior benefício comparando o percentual do Erro Quadrático Médio das aproximações com seus circuitos originais. Além disso, o estudo também faz observações sobre alguns desafios em aberto na área de SLA, como a dificuldade de lidar com escalabilidade, ou a configuração da acurácia levando em conta parâmetros externos, tais quais o percentual de bateria de determinado dispositivo ou uma qualidade mínima exigida pra alguma funcionalidade, por exemplo.

Além de aplicar-se computação aproximada em algoritmos de ML, como mencionado anteriormente, pode-se também usar os próprios algoritmos de ML para gerar aproximação no projeto de CI.

Uma preocupação na aplicação de táticas de Computação Aproximada (AxC) é a qualidade da saída obtida com a aproximação. Algumas aplicações podem ser mais tolerantes do que outras, exigindo uma calibragem na acurácia obtida. Uma forma de contornar esse problema é implementar diversos componentes de aproximação e escolher o mais adequado de acordo com a entrada recebida ou preferência do usuário. Em (MASADEH; AOUN *et al.*, 2020) é explorado o uso de Árvore de Decisão como seletor, garantindo que o método de ML defina o projeto mais adequado, entre 20 disponíveis, levando em conta o valor de entrada.

O método de Árvore de Decisão também é usado em (ABREU, Brunno Alves de *et al.*, 2022), porém na etapa de Síntese Lógica dos projetos desenvolvidos, e não para seleção da tecnologia a ser executada, como em (MASADEH; AOUN *et al.*, 2020). Neste trabalho (ABREU, Brunno Alves de *et al.*, 2022) é feita a descrição e análise da ferramenta *C2PAx* (*Complexity-Aware Constant Parameter Approximation*) desenvolvida pelos autores do artigo. Usando Árvore de Decisão, o *C2PAx* visa definir parâmetros constantes em comparadores e limita a profundidade máxima das árvores para forçar uma aproximação, reduzindo, assim, recursos do circuito como: área, potência, atraso e consumo de energia. Os resultados se mostraram promissores ao final, entregando o que se esperava quanto à redução das grandezas observadas e melhorando o desempenho das tecnologias, uma vez que o caminho crítico também foi diminuído.

A Tabela 2 mostra uma análise comparativa dos assuntos abordados em cada um dos trabalhos relacionados com o que foi desenvolvido no projeto aqui apresentado. Os trabalhos foram classificados de acordo com a abordagem de Computação Aproximada (AxC), sendo automatizada, aplicada na Síntese Lógica (LS), adotando Árvores de Decisão para a síntese aproximada, e se projetam a integração com algum fluxo de síntese. O levantamento permite visualizar que praticamente todas as propostas exploram AxC automatizada, e que dentre as abordagens para síntese lógica, apenas (ABREU; GRELLERT; BAMPI, 2020) e este trabalho exploram Árvore de Decisão para minimizar as funções lógicas. Além disso, dentro do nosso conhecimento, até o momento este trabalho é pioneiro em integrar a síntese lógica baseada em Árvores de Decisão com um fluxo de síntese *standard cell*, permitindo a avaliação completa do impacto da adoção de Árvores de Decisão na etapa de otimização lógica.

Tabela 2 – Visão Geral dos Trabalhos Relacionados

Referência	**AxC em CIs**	**AxC Automatizada**	**AxC LS**	**Árvore de Decisão para AxC**	**Integração com Fluxo de Síntese**
(VENKATARAMANI; CHAKRADHAR *et al.*, 2015)	✓	✓	x	x	x
(SCARABOTTOLO *et al.*, 2020)	x	✓	✓	x	x
(MASADEH; HASAN; TAHAR, 2018)	✓	x	x	x	x
(MASADEH; AOUN *et al.*, 2020)	✓	✓	✓	x	x
(ABREU; GRELLERT; BAMPI, 2020)	✓	✓	✓	✓	x
Este Trabalho	✓	✓	✓	✓	✓

Fonte: O Autor.

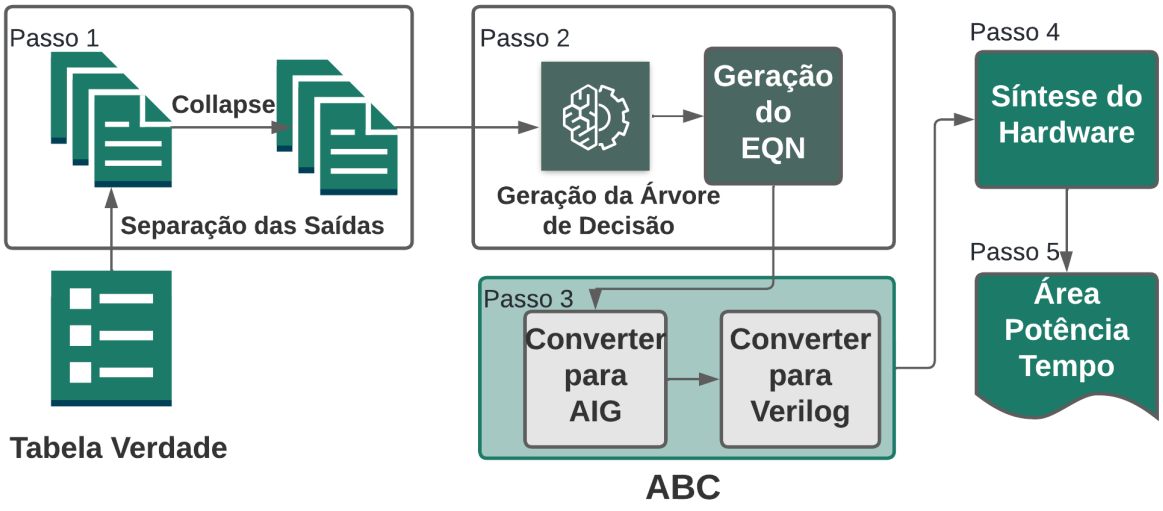
4 PROPOSTA E DESENVOLVIMENTO

A síntese de circuitos utilizando computação aproximada se distancia da tradicional síntese de circuitos integrados, onde a exigência por precisão é uma forte afirmativa. Entretanto, cada dia mais o aprendizado de lógica ou a síntese parcial de funções se torna mais relevante em dois principais contextos: 1) o aumento da síntese de funções altamente complexas com elevado número de entradas, e 2) a síntese voltada para eficiência energética.

Este projeto propõe a utilização de Árvore de Decisão para o aprendizado e minimização lógica destas funções Booleanas, fornecendo um fluxo de síntese adaptado tanto para a síntese de funções completas como de funções incompletas.

A Figura 7 apresenta a visão geral do fluxo proposto. O fluxo é dividido em cinco passos: Inicialização, Geração da Árvore de Decisão, Conversão de formatos, Síntese do *Hardware* e Análise. A seguir, serão detalhados cada um destes passos.

Figura 7 – Fluxo de execução



Fonte: O autor.

4.1 PASSO 1: INICIALIZAÇÃO

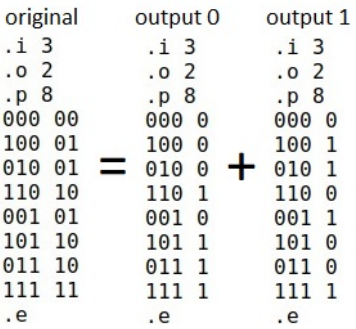
O fluxo de síntese desenvolvido inicia com a leitura de um arquivo de descrição da função, detalhando uma tabela verdade completa ou incompleta, no formato PLA (*Programmable Logic Array*). Esta Tabela pode conter uma representação compacta, com *bits* de entrada do tipo *Don't Care* (DC), ou com valores fixos. Para possibilitar a leitura do arquivo e correta execução da aprendizagem por parte das ferramentas de geração de árvores de decisão, os casos em que a Tabela Verdade possuir DCs deverão passar por uma etapa responsável por estendê-la para o formato com *bits* fixos antes de seguir com o processo. A ferramenta ABC possui a funcionalidade *collapse*, que executa a conversão

para o formato estendido e gera como saída outro arquivo que representa um circuito equivalente ao inicial. A ferramenta ABC é de código livre, podendo ser integrada nesta proposta de fluxo de síntese.

Com a Tabela Verdade já na versão estendida, uma separação das saídas é feita tornando o circuito com n saídas em n circuitos com uma saída. Essa abordagem foi adotada pois o *IWLS 2020* tratava apenas de circuitos com uma saída, mas para trabalhos futuros pretende-se alterar esse comportamento para que a separação das saídas não seja necessária.

Este projeto considera circuitos realistas que podem conter múltiplos *bits* de saída. Adotou-se uma solução clássica de síntese lógica chamada *two-level minimization* onde a otimização é inicialmente realizada em cada função lógica individual, ou seja, cada bit da saída é tratado como um problema de otimização individual, e um próximo passo identifica semelhanças entre as funções individuais otimizadas, permitindo otimização conjunta das funções de saída. Desta forma, este primeiro passo divide esses circuitos com n saídas em n descrições de funções, cada uma de um *bit*. As Tabelas resultantes dessa operação podem ser vistas no exemplo da Figura 8. Na figura, os dados de cabeçalho que definem o tamanho da entrada ($.i$) e saída ($.o$) não são atualizados com a separação do arquivo. Porém, por não serem importantes para o prosseguimento do processo, isso não aferará o resultado final da síntese. Feita esta divisão, cada Tabela passa pelos procedimentos de síntese lógica de maneira independente e, após a execução do fluxo, as descrições de cada função são reunidas.

Figura 8 – Tabela Verdade com duas saídas dividida em duas Tabelas com uma saída



Fonte: O autor.

4.2 PASSO 2: GERAÇÃO DA ÁRVORE DE DECISÃO

Inicialmente investigou-se a adoção da ferramenta de geração de árvores de decisão C5.0 (PANDYA; PANDYA, 2015). O C5.0 teve um bom desempenho de aprendizagem quando adotado no aprendizado de funções incompletas, propostas no *contest IWLS 2020* (RAI *et al.*, 2021). Porém, em alguns casos, a ferramenta não chegava a acurácias satisfatórias, principalmente quando a entrada da função estava completamente definida.

Optou-se, então, pelo SK (PEDREGOSA *et al.*, 2011), uma abordagem mais recente que vem crescendo nos últimos anos. Este é um módulo em Python que possui diversos métodos de Aprendizado de Máquina em código aberto, o que permite alterações no algoritmo para aplicações mais específicas. Outro fator de motivação para o uso do SK é a compatibilidade com o código usado no restante do fluxo desenvolvido, que também é programado em Python. Isso permite que leituras e escritas de arquivos com diferentes formatos sejam evitadas, pois os dados transmitidos pelo fluxo podem ser passados como parâmetros de funções ao invés de arquivos temporários.

O fluxo inicia com a geração de uma Árvore de Decisão correspondente a cada uma das funções utilizando o SK. Os parâmetros definidos para a Árvore possibilitam limitar sua profundidade lógica, permitindo uma pré-configuração que garanta a redução do atraso da tecnologia, uma vez que a profundidade representa o caminho crítico do circuito em produção. Essa limitação é definida pelo parâmetro *max_depth* da Árvore. Outro parâmetro que pode ser usado para reduzir a Árvore é o *cpp_alpha*, utilizado para podar a Árvore gerada. Essa técnica pode ser ainda mais promissora que o uso de *max_depth*, pois pode gerar reduções mais suaves do que a limitação da profundidade. Essa estratégia, porém, será implementada e analisada em trabalhos futuros.

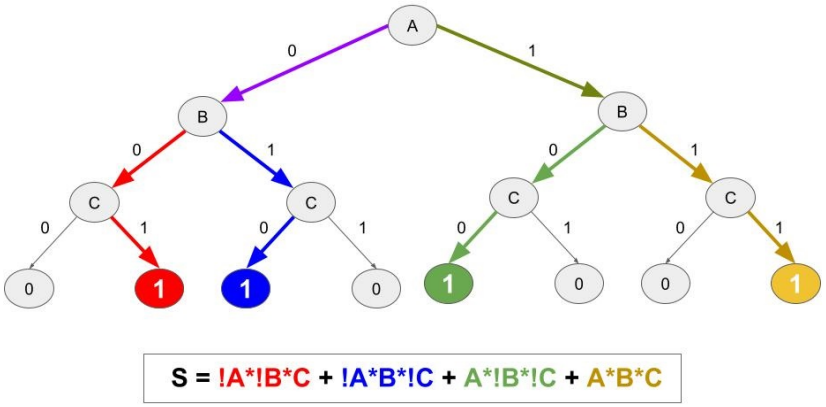
O formato resultante do aprendizado, ou seja, a descrição da Árvore de Decisão gerada, não é compreendido pelo ABC, tornando necessário o desenvolvimento de um algoritmo de conversão para um formato a ser lido pela próxima ferramenta no fluxo de otimização.

Árvores podem ser facilmente convertidas em equações lógicas do tipo Soma de Produtos (*Sum Of Products/SOP*) ou Produto de Somas (*Producto Of Sums/POS*) que são representados por arquivos no formato *EQN*. A Figura 9 ilustra como o algoritmo de conversão funciona partindo de uma árvore e definindo a equação correspondente no formato de soma de produtos. Cada nível da Árvore representa um valor de entrada da Tabela Verdade. Como se trata de uma Soma de Produtos, devem ser considerados os nó folhas com valor 1 e escrito na equação o caminho feito para se chegar a este nó, negando as entradas em que se sai através de uma seta que represente o valor 0. Por exemplo, para se chegar ao nó folha de valor 1 em vermelho, parte-se de A para B através da seta 0 (!A), depois parte-se de B também por 0 chegando em C (!A * !B) e por último chega-se ao resultado final 1 partindo de C pela seta de mesmo valor (!A * !B * C). Esse processo se repete até que todos os nó folhas de valor 1 tenham sido transcritos em forma de equação *booleana*.

4.3 PASSO 3: CONVERSÃO DE FORMATOS

Feita a descrição da árvore como uma equação, pode-se carregar a descrição da função gerada pela árvore de decisão na ferramenta ABC para uma segunda etapa de otimização lógica ou para aproveitar os recursos de conversão de formatos de arquivos

Figura 9 – Representação do processo de conversão de Árvore para Soma de Produtos



Fonte: O autor.

disponíveis no *software*. Dessa forma, é possível converter o arquivo para o formato AIG.

Além disso, alguns algoritmos de otimização disponíveis no ABC visam rebalancear o grafo e reduzir o número de nodos e profundidade, que se correlacionam com as portas lógicas e caminho crítico, respectivamente. A integração com a ferramenta ABC permite que o projetista escolha entre adotar mais passos de otimização lógica nesta etapa ou seguir com a otimização obtida na geração da árvore de decisão.

Nesta etapa também pode-se extrair alguns dados referentes ao estado atual do circuito para possíveis análises e comparações, como o número de nodos e a profundidade lógica, caso seja necessário.

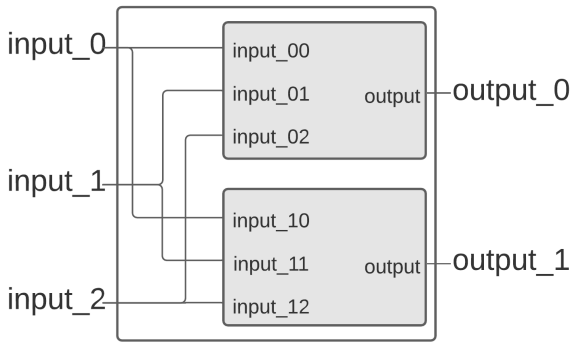
Com as otimizações executadas, aplica-se uma última conversão à estrutura de dados resultante para o formato Verilog, que é salvo em arquivo. Ao final, com todos os circuitos parciais prontos, contendo a otimização lógica para cada função, ou seja, para cada um dos *bits* de saída, deve-se uni-los novamente em uma *Top Level Entity* a fim de obter-se o circuito completo de volta, como pode ser visto na Figura 10. Agora, é possível mapear este resultado para a plataforma de síntese desejada, seja uma FPGA ou fazer a síntese de um *hardware* dedicado (ASIC), dependendo da finalidade do projeto.

4.4 SÍNTESE DO HARDWARE

O fluxo proposto permite a síntese de circuitos completos ou aproximados. A aproximação pode ser resultado de funções incompletas, onde as informações fornecidas sobre as funções não correspondem a totalidade da tabela verdade, ou pode ser resultado de condições na geração das Árvores de Decisão, adotadas para reduzir o tamanho das árvores, e, consequentemente, reduzir o número de portas lógicas e potência necessárias.

Com o objetivo de avaliar a independência da estratégia de otimização lógica proposta quanto ao fluxo de síntese de hardware adotado, este trabalho apresenta um

Figura 10 – Circuito final após reunificação dos circuitos parciais



Fonte: O autor.

primeiro estudo de caso integrando a solução de minimização lógica com uma ferramenta comercial de síntese para FPGAs, e uma ferramenta comercial de síntese ASIC baseada em biblioteca de células. Após, este trabalho centralizou os esforços em integrar com a ferramenta OpenRoad, que permite a síntese ASIC baseada em biblioteca de células.

Na Seção 5 serão apresentados os resultados da execução do fluxo na análise de alguns *benchmarks* do estudo de caso. Os valores obtidos pelas ferramentas tradicionais são comparados com os valores gerados pela Árvore de Decisão, tanto no mapeamento para uma FPGA quanto na geração de um circuito ASIC adotando um fluxo comercial de síntese *standard cell*. Pode-se adiantar, porém, que os números comprovam a possibilidade de utilização dessas técnicas no desenvolvimento dos circuitos, sendo, inclusive, mais eficientes do que os métodos clássicos em alguns pontos.

Com a versão mais atual integrada ao *OpenRoad* poucos passos mudaram. Boa parte do processo se comporta de maneira equivalente ao reportado no experimento inicial, mas algumas etapas foram adaptadas de modo a aproveitar-se o melhor da ferramenta. As alterações implementadas ao fluxo para esta integração e para a aplicação de Computação Aproximada serão abordadas no Capítulo 6.

4.5 ANÁLISE

Para a etapa de testes, primeiro adotou-se um conjunto de *benchmarks* clássicos, similar aos vistos em (ABREU, Brunno A. de *et al.*, 2021). Estes *benchmarks* possuem funções lógicas de até 16 entradas. Atualmente, a ferramenta ABC limita a operação de expansão de Don't Cares nas Tabelas Verdades para funções até 16 *bits*. Entretanto, o fluxo pode funcionar para funções incompletas com valores maiores, onde somente parte das combinações de entradas serão especificadas sem Don't Cares.

Os resultados serão comparados quanto ao tamanho do circuito gerado, a potência dinâmica e estática, ao atraso crítico do circuito (*delay*) e ao tempo de execução. Após, serão também considerados os resultados para sínteses visando circuitos aproximados.

4.6 DESENVOLVIMENTO

Apesar do envolvimento de diversas tecnologias no fluxo proposto, a automatização do processo e etapas ligadas à Árvore de Decisão e Computação Aproximada foram desenvolvidas inteiramente em linguagem de programação *Python*. Isso facilita a integração com o SK, pois este é também um módulo Python, e reduz o número de arquivos temporários a serem escritos e lidos, melhorando o desempenho do fluxo quanto ao tempo de execução. O código em questão é encontrado no site de versionamento *GitLab*.

Várias decisões foram tomadas durante o desenvolvimento deste fluxo, adaptando tempo de projeto com os objetivos iniciais. Algumas limitações foram observadas referentes as ferramentas auxiliares adotadas e refletem em considerações sobre a execução do fluxo proposto. A principal consideração necessária foi quanto à limitação da função *collapse* da ferramenta ABC (e Yosys). Este passo é necessário para transformar a descrição compacta ou parcial da tabela verdade em uma descrição completa, sem DCs. As ferramentas possuem uma limitação quanto ao número de entradas para as funções a serem expandidas, limitando a 16 entradas. No momento, com as ferramentas de geração de árvores estudadas, a função completa é necessária para que a geração das árvores de decisão seja precisa. Visando fornecer o fluxo de síntese para funções exatas e aproximadas, este trabalho optou por definir esta restrição de 16 entradas para as funções a serem otimizadas. Algumas tentativas foram realizadas para realizar a extensão da tabela verdade sem a ferramenta ABC, entretanto, observou-se que o desafio computacional para esta solução demandaria esforços de tempo correspondentes a um novo projeto de pesquisa. Além disso, a descrição parcial de funções com mais de 16 entradas pode ser adaptada no fluxo proposto, gerando desta forma soluções sempre aproximadas.

Fora este contratempo, o modo de desenvolvimento do fluxo permitiu a fácil integração da solução de otimização lógica com as demais etapas da síntese de circuitos integrados, permitindo a disponibilidade de um fluxo final completo, da descrição dos circuitos como tabela-verdades até a síntese de *hardware*. Não foram realizadas otimizações nas etapas de síntese física específicas para cada *benchmark* avaliado, para permitir uma avaliação inicial mais direta dos efeitos da síntese com árvores de decisão proposta. Entretanto, observa-se grande espaço de exploração de projeto com o ajuste e configuração de opções relacionadas as etapas de síntese física nas ferramentas comerciais ou do OpenRoad a serem utilizadas. Tal exploração está além dos objetivos deste trabalho, também podendo ser objetivo de novas pesquisas.

5 ESTUDO DE CASO

Nesta seção será apresentado um comparativo entre os resultados gerados pela Árvore de Decisão, explorando o fluxo da Figura 7, e outras abordagens populares na síntese lógica (CAMPOS *et al.*, 2022). As estratégias comparadas são as seguintes:

1. otimização com Árvore de Decisão (AD),
2. ferramenta tradicional ABC (ABC),
3. síntese direta com o *software Quartus* ou com fluxo de síntese da Cadence (SD).

O item 1 já foi devidamente explicado nas seções anteriores desta monografia. O item 2 pula a etapa de Aprendizagem de Máquina, fazendo-se também desnecessária a conversão do formato EQN para AIG ou Verilog. Neste caso, apenas lê-se o arquivo PLA utilizando o ABC e converte-se ao formato AIG. É executado, então, o *script* de otimização *compress2rs* (RIENER *et al.*, 2019), que consiste em aplicar uma sequência de balanceamentos, refatorações e reduções na estrutura do AIG e gera-se o Verilog equivalente.

O item 3 passa por processo semelhante ao anterior, apenas pulando a etapa de otimizações via *compress2rs*.

Utiliza-se o *Quartus* para obter os resultados no mapeamento para FPGA, e o fluxo de síntese da *Cadence* no desenvolvimento *ASIC*. No mapeamento para FPGA foi utilizado um dispositivo da família *Ciclone V* de modelo *5CGXFC7C7F23C8*, que contém 15 mil elementos lógicos, 56.480 módulos de lógica adaptativa, representados pela sigla ALM *Adaptive Logic Modules*, memória embarcada de 6,70 Mbit, 240 entradas e saídas e tensão de 1,1 V. Já a síntese *ASIC* foi executada com a ferramenta da *Cadence*, *Genus*, com a síntese para a biblioteca padrão de células *STMicroelectronics* 65 nm com uma frequência de 500 MHz.

Foram utilizados na comparação dois *benchmarks* disponibilizados pelo *International Workshop on Logic Synthesis' 93*. Um dos *benchmarks* é o *ALU4*, uma Unidade Lógico-Aritmética de 4 *bits* (*4-bit Arithmetic-Logic Unit*), e o segundo é o *PDC*. A Tabela 3 mostra as características de entrada e saída em cada um dos casos estudados.

Tabela 3 – Características dos Benchmarks

Benchmark	**Entradas**	**Saídas**
ALU4	14	8
PDC	16	40

Fonte: O autor.

A Tabela 4 apresenta os resultados obtidos para as três análises mapeadas para a FPGA, apresentando os resultados de atraso e utilização de recursos. Não foram observados

os valores de potência por não terem sido definidos os estímulos de entrada para estes circuitos.

A Tabela 5 apresenta os resultados de potência estática, dinâmica e total, número de células, área e o equivalente em portas NAND2 para a síntese das três análises utilizando as ferramentas comerciais para síntese *standard cell*. A síntese ASIC atingiu resultados próximos de área e potência no fluxo da AD e SD para o ALU4. Já com o PDC o resultado ficou consideravelmente distante, com uma redução de 61% no número de células usando a AD quando comparado com a SD. Observando a área total vemos que isso representa 58% de redução na área, além de uma melhoria de 38% nos dados de potência. No caso da FPGA, obteve-se uma melhoria em quase todas as métricas analisadas em ambos os circuitos ao se usar a AD, gerando apenas um leve decremento de 3,2% no atraso do ALU4, que foi compensado com uma boa redução nos recursos utilizados de 53,2%. Tivemos também uma redução bastante interessante no uso de recursos do PDC com o fluxo de AD (20% em relação ao pior caso), que demonstra a grande capacidade da proposta.

É interessante notar que os valores das sínteses FPGA e ASIC demonstraram que o *script* do ABC pode restringir o aprimoramento executado por outras ferramentas, aumentando o número de células e/ou recursos em ambos os *benchmarks* avaliados em comparação às demais atuações.

Tabela 4 – Resultados de Performance para os Diferentes Fluxos do Mapeamento para FPGA.

Benchmark		**Síntese FPGA**	
	Método	**Atraso (ns)**	**Recursos (#ALMs)**
Alu4	AD	23.7	275
Inputs: 14	ABC	25.2	624
Outputs: 8	SD	23	588
PDC	AD	15.2	26
Inputs: 16	ABC	19	108
Outputs: 40	SD	18.2	111

Fonte: O autor.

Tabela 5 – Resultados de Performance para os Diferentes Fluxos da Síntese ASIC.

Benchmark	**Síntese ASIC 65 nm @500MHz**						
	Método	**Potência Estática (nW)**	**Potência Dinâmica (µW)**	**Potência Total (µW)**	**# Células**	**Área (µm2)**	**NAND2 Eq.**
Alu4	AD	1.25	1517.4	1518.6	675	3665	1762
Entradas: 14	ABC	1.31	1529.8	1531.1	710	3810	1832
Saídas: 8	SD	1.21	1485.4	1486.6	657	3529	1697
PDC	AD	0.28	325.8	326.1	88	533	256
Entradas: 16	ABC	0.56	565.3	565.9	262	1412	679
Saídas: 40	SD	0.57	521.6	522.1	223	1277	614

Fonte: O autor.

6 INTEGRAÇÃO COM OPENROAD E COMPUTAÇÃO APROXIMADA

Nos últimos anos o compartilhamento de conhecimento entre desenvolvedores cresceu significativamente com as ferramentas de código aberto. Uma das comunidades que faz isso de maneira ampla atualmente é a de desenvolvedores Python, ajudando na disseminação e criação de novas tecnologias. O próprio SKLearn, utilizado neste projeto e bastante popular, é um dos vários exemplos de resultado positivo obtido com esse modelo de cooperação. A área científica pode tirar proveito dessas ferramentas e ajudar na criação de novas, ampliando o horizonte da tecnologia. Com esse objetivo, decidiu-se pela utilização apenas de *softwares* de código aberto no desenvolvimento deste projeto. Sendo assim, o *OpenRoad* foi escolhido para a integração com o fluxo de síntese.

Esta decisão, porém, não foi tomada de imediato. No primeiro momento do projeto, o mapeamento foi planejado para ser feito apenas para *FPGA* com o auxílio da ferramenta *Quartus*, como já descrito neste trabalho. Portanto, algumas adaptações precisaram ser implementadas para possibilitar a integração com o o novo fluxo.

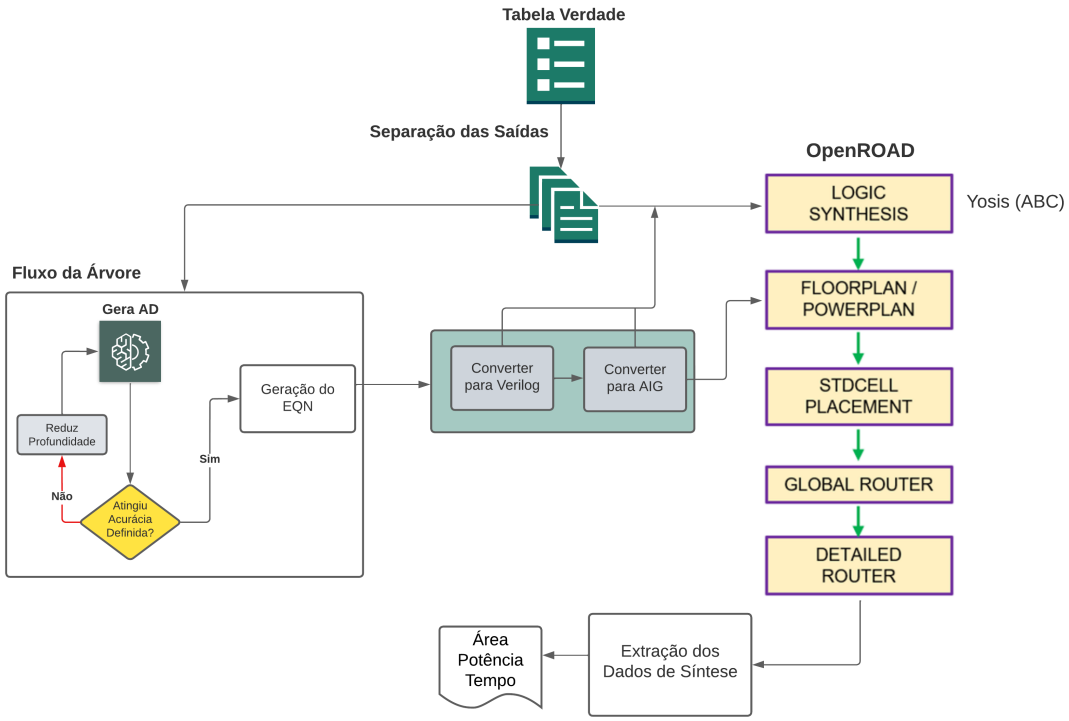
Primeiro definiu-se o *OpenRoad* como dependência do projeto no repositório *git* do mesmo. Dessa forma é possível atualizar o *OpenRoad* através do repositório do projeto aqui proposto, caso seja necessário obter a funcionalidade de alguma nova atualização da ferramenta. Tais atualizações podem impactar no restante do fluxo e devem ser executadas com bastante cautela. O passo seguinte foi corrigir a extração dos dados de síntese para o formato de relatórios do OpenRoad. Por último, substituiu-se o *ABC* usado no fluxo antigo para o ABC que já está integrado ao *Yosys* dentro do fluxo *OpenRoad*, reduzindo o número de *softwares* envolvidos e evitando a redundância destes.

Logo após a implementação da nova abordagem, partiu-se para a aplicação da lógica de Computação Aproximada. Assim como observado em (ABREU, Brunno Alves de *et al.*, 2022), a aproximação foi forçada através da limitação do parâmetro que define a profundidade máxima da Árvore de Decisão. Um valor de acurácia (percentual) deve ser definido pelo usuário como entrada na execução do *software* caso deseje usar a funcionalidade. A etapa de aprendizagem é executada normalmente e, ao final, o valor da acurácia obtida é comparada com o que se deseja pelo usuário. Caso esteja acima do que foi definido, a execução é feita novamente com uma profundidade inferior à obtida na primeira etapa e a Árvore atual é mantida em memória. Esse comportamento se mantém em *loop* até que a acurácia resultante seja menor do que a definida e, então, a Árvore anterior mantida em memória é retornada para dar continuidade à síntese.

A Figura 11 mostra o diagrama do fluxo final já com a lógica de Computação Aproximada e integração com o projeto OpenROAD que gerou os resultados avaliados. As etapas de Separação de saídas e conversão para Verilog e AIG são realizadas com a ferramenta Yosis, invocando o ABC nativo internamente. Após a geração unificada da descrição Verilog otimizada para as n-saídas da função, segue-se o fluxo de síntese da

ferramenta OpenRoad até a conclusão do roteamento sem violações de regras de projeto. A síntese neste experimento foi configurada sem restrição de *timing*, para permitir a comparação do atraso.

Figura 11 – Fluxo completo após implementação de Computação Aproximada e integração com o OpenROAD



Fonte: O autor.

6.1 ANÁLISE DOS RESULTADOS

A fim de analisar os impactos gerados por essas alterações, foram executadas seis combinações diferentes de fluxo. Uma com o fluxo exclusivo do *OpenRoad*, outra com a utilização de AD com 100% de acurácia para analisar-se o impacto da AD sem aproximação e outras quatro com aproximação, com as acurácias definidas em: 95%, 90%, 80% e 75%. Primeiro serão comparados os dados do fluxo *OpenRoad* sem AD e o fluxo com integração com a AD em 100% de acurácia. O número de entradas e saídas de cada *benchmark* é mostrado na Tabela 6.

Os gráficos das Figuras 12, 13, 14 e 15 demonstram o resultado de cada um dos *benchmarks* avaliados o fluxo OpenROAD sem AD, pra AD com 100% de acurácia e outras duas com uso da técnica de aproximação: AD 90% e AD 75%. Os resultados para as outras versões aproximadas foram omitidos dos gráficos para facilitar a visualização.

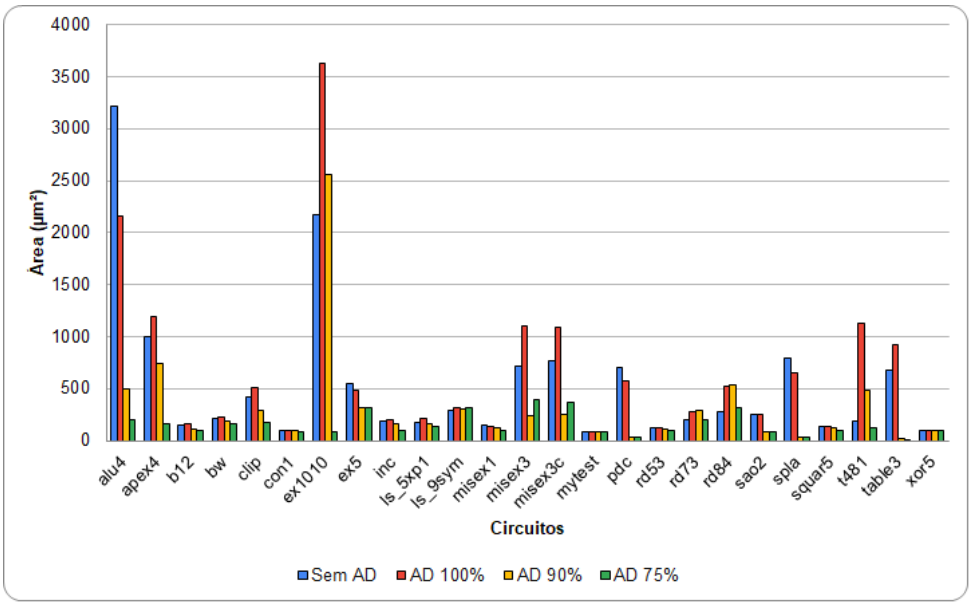
As médias de área ficaram em 544,20 µm² para o *OpenRoad*, 650,40 µm² o *AD 100*, 318,40 µm² o *AD 90* e 154,76 µm² o *AD 75*. Como visto antes, a área do *AD 100*

Tabela 6 – Número de Entradas e Saídas dos *Benchmarks*

DESING	INPUTS	OUTPUTS	DESING	INPUTS	OUTPUTS
alu4	14	8	misex3c	14	14
apex4	9	19	mytest	2	1
b12	15	9	pdc	16	40
bw	5	28	rd53	5	3
clip	9	5	rd73	7	3
con1	7	2	rd84	8	4
ex1010	10	10	sao2	10	4
ex5	8	63	spla	16	46
inc	7	9	squar5	5	8
ls_5xp1	7	10	t481	16	1
ls_9sym	9	1	table3	14	14
misex1	8	7	xor5	5	1
misex3	14	14			

Fonte: O autor.

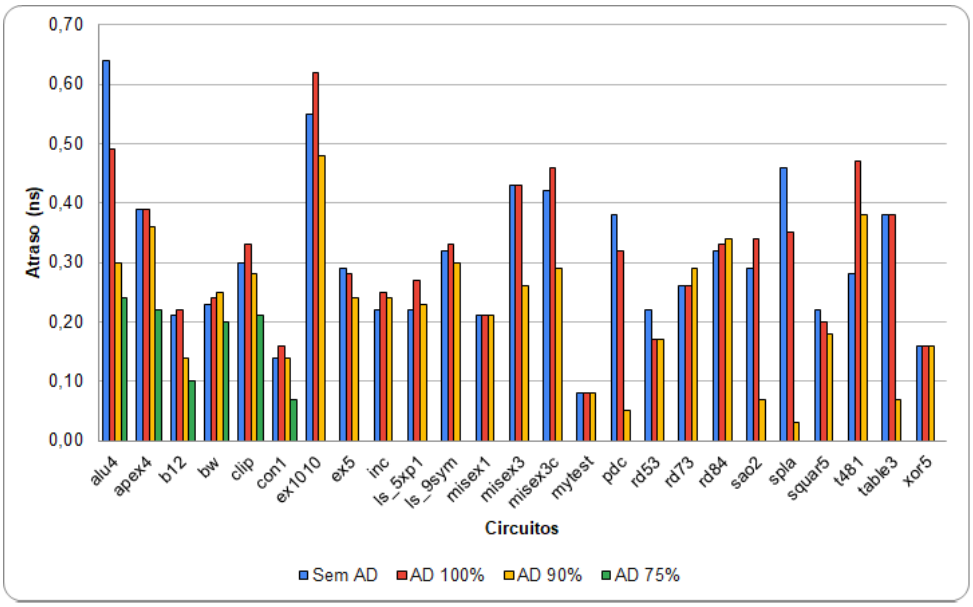
Figura 12 – Comparação de área para os *benchmarks* avaliados e as técnicas exatas e aproximadas



Fonte: O Autor.

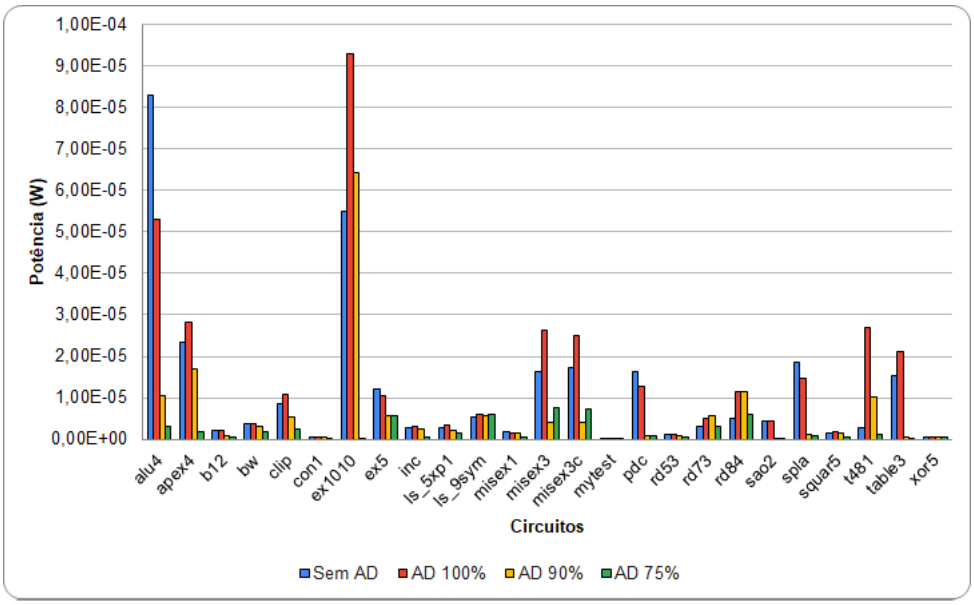
aumentou em relação ao *OpenRoad*, mas adotando aproximação, tanto a versão *AD 90* como a versão*AD 75* tiveram uma redução em relação ao *OpenROAD* e a síntese AD 100%. O método *AD 90* diminuiu 41,49% em comparação com o *OpenROAD*, enquanto o *AD 75* reduziu a área em até 71,56%. No número de células, as médias foram 453,20, 590,64, 260,88 e 46,48. Como era de se esperar, as diferenças ficaram semelhantes às da área, uma vez que o número de células está diretamente ligado à área do circuito. Para essa grandeza a redução da abordagem *AD 90* e *AD 75* em relação ao *OpenROAD* foi de 42,43% e 89,74%, respectivamente. Por último tem-se a potência, que aumentou 21,48% em *AD*

Figura 13 – Comparação de atraso para os *benchmarks* avaliados e as técnicas exatas e aproximadas



Fonte: O Autor.

Figura 14 – Comparação de potência para os *benchmarks* avaliados e as técnicas exatas e aproximadas

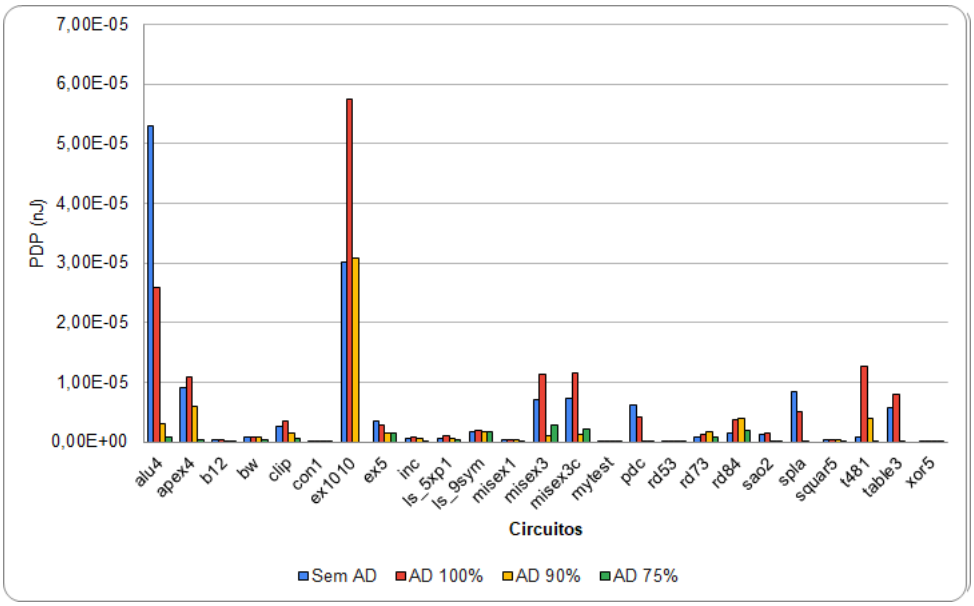


Fonte: O Autor.

100 e reduziu 47,19% e 82,14% em *AD 90* e *AD 75* comparados com o OpenROAD.

Entretanto, este comportamento médio depende do conjunto de *benchmarks* considerado. Olhando casos individuais que ficaram dentro do padrão observado pro conjunto completo, isto é, desconsiderando os *outliers*, pode-se destacar o circuito *alu4* como um caso positivo, por proporcionar um bom desempenho para todas as abordagens analisadas em todas as grandezas. Ele teve uma melhora no desempenho já com a aplicação da *AD*

Figura 15 – Comparação de PDP para os *benchmarks* avaliados e as técnicas exatas e aproximadas



Fonte: O Autor.

100 em comparação com o OpenROAD, reduzindo em 32,71% a área, 25,88% o número de células, 23,44% o atraso e 36,14% a potência. Na *AD 90*, essas reduções em relação ao mesmo OpenROAD sem AD foram de 84,61% na área, 84,63% no número de células, 53,12% no atraso e 87,35% na potência. Do lado negativo tem-se o *rd84*, que manteve uma piora no desempenho mesmo com a aplicação da aproximação nas técnicas de AD. Com a *AD 75*, por exemplo, obteve-se para a área um aumento em relação ao OpenROAD de 17,94%, de 32,69% para o número de células, 6,25% para o atraso e 22,08% para a potência.

É importante notar que em *AD 75* três projetos não geraram os resultados de atraso. São eles *ex1010*, *spla* e *table3*. Esse comportamento ocorre pois essas tecnologias tem saídas esparsas e, quando esses valores são fixados em zero, a acurácia obtida já é superior ao limite definido de 75%. Outro ponto que deve ser mencionado é o salto grande na diferença das acurácias visto em alguns casos, o motivo para tal pode ser o uso de uma profundidade máxima no controle da acurácia. Uma forma de contornar a situação pode ser através da manipulação do parâmetro *ccp_alpha* (*Cost Complexity Pruning Alpha*), que aplica uma *poda* à Árvore em questão, reduzindo seu tamanho de maneira mais sutil e sem alterar a sua profundidade necessariamente. Essa abordagem poderia permitir a redução no número de células, área e potência do circuito e em alguns casos até mesmo o caminho crítico. O estudo desta alteração ficará, porém, para trabalhos futuros.

Outro ponto interessante observado, é que ao fixar uma acurácia mínima para o projeto, todas as entradas estão sujeitas a esta restrição. Assim, para circuitos com várias saídas, algumas podem ter a acurácia bem acima do valor limite estabelecido. Desta

forma, a acurácia definida pelo projetista é atendida, embora em geral, a acurácia total considerando todas as saídas seja superior ao limite, possibilitando exploração futura para maiores otimizações. Como um exemplo, a Tabela 7 mostra as acurácias obtidas para cada uma das saídas do circuito ALU4 quando solicitada uma precisão de 90% pelo usuário. O valor que mais se aproximou do requisitado foi da saída 7, com 90,05% de acurácia. O mais afastado é visto na saída 1, com 93,75%. O algoritmo de aproximação descrito anteriormente garantiu que em nenhum dos casos o percentual de acerto ficasse abaixo do solicitado no momento da execução do fluxo. Esse exemplo reflete o comportamento de todos os *benchmarks* processados no desenvolvimento do projeto.

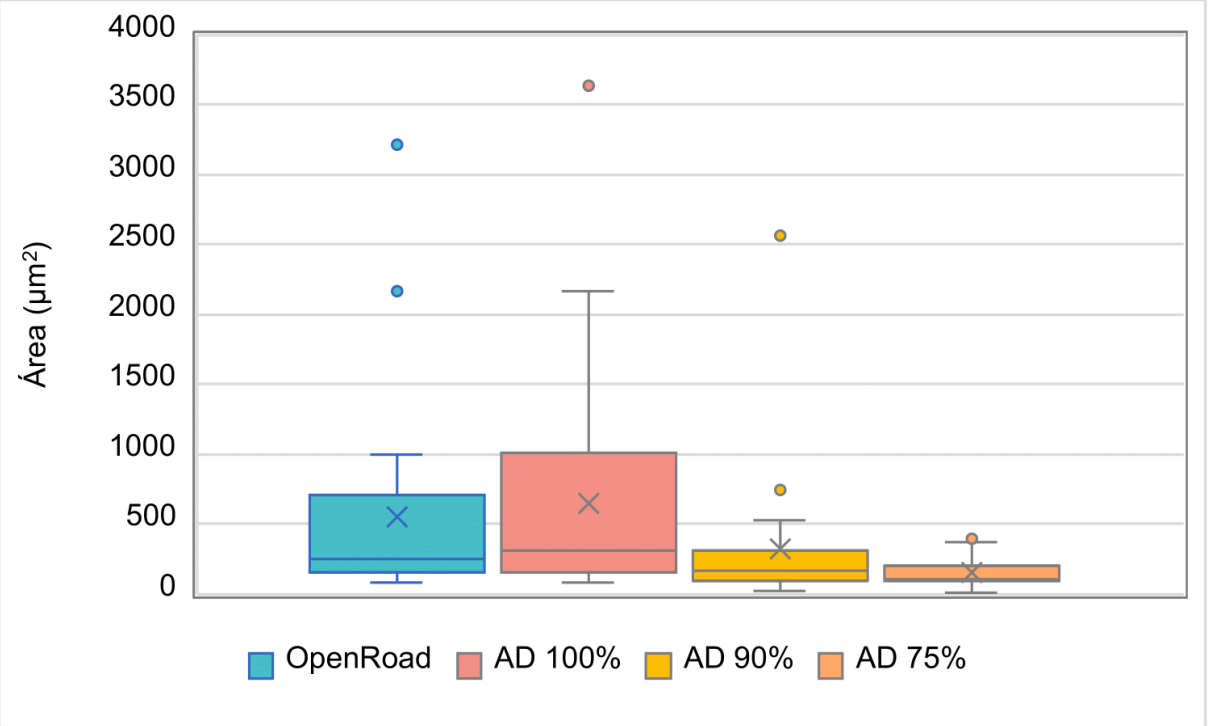
Tabela 7 – Acurácias Das Saídas de ALU4

Acurácias de ALU4 90%	
Saídas	Acurácias
0	92,38
1	93,75
2	90,14
3	90,14
4	93,36
5	93,51
6	91,41
7	90,05

Fonte: O Autor.

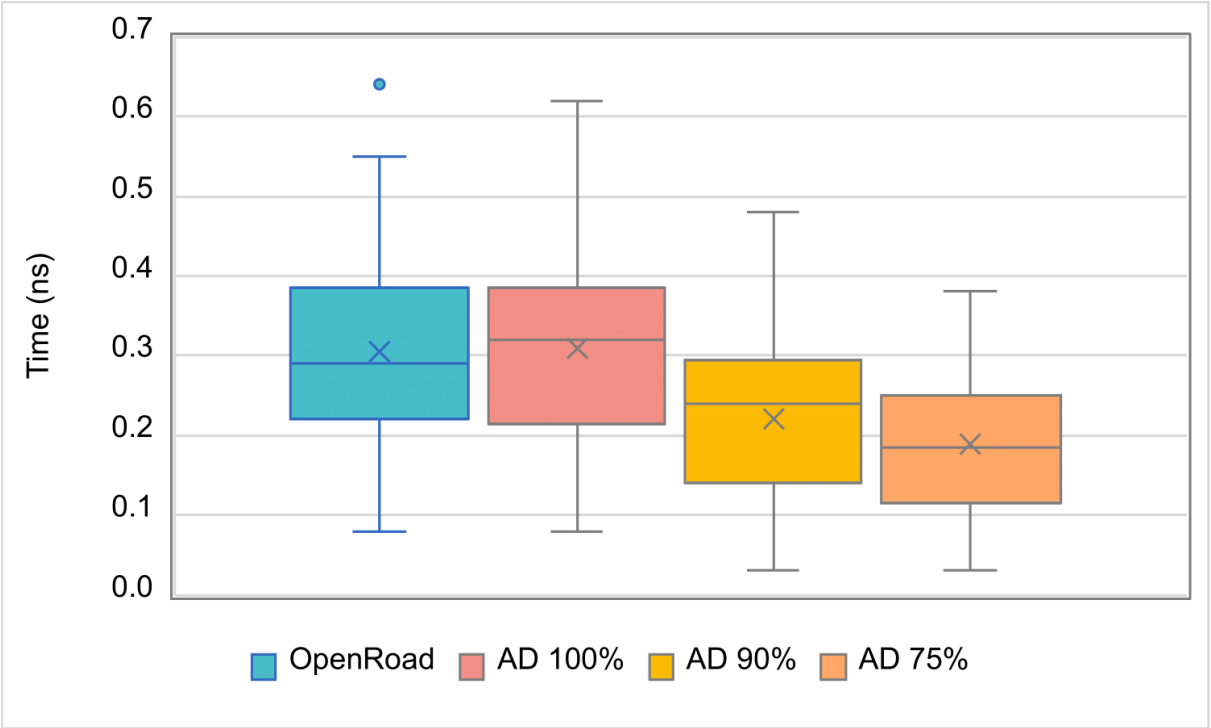
Finalizando a análise, apresenta-se os *boxplots* das Figuras 16, 17, 18 e 19, que-comprovam a tendência de queda nas grandezas avaliadas quando adotada a síntese com aproximação. Esta análise permite que se observe os desvios padrão entre as sínteses dos circuitos considerados e os valores *outliers* de cada abordagem avaliada. Os resultados médios de área, potência e PDP demonstram que *AD 100* teve um aumento em relação ao OpenROAD, com comportamento bem semelhante quanto ao atraso. Entretanto, existem reduções significativas em atraso, potência e área utilizando aproximação, mesmo com valores de acurácia altos como os definidos para a síntese *AD 90*. Esse comportamento também é verificado no produto entre *delay* e potência (PDP) da Figura 19.

Figura 16 – *Boxplot* da área



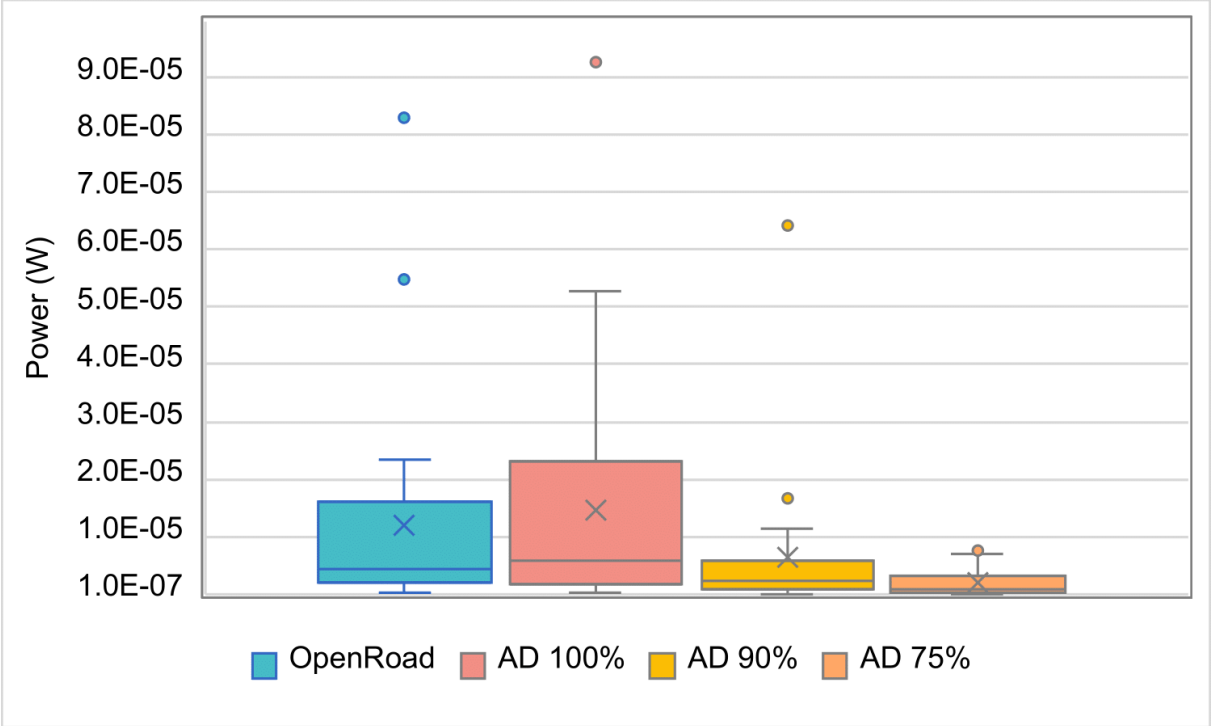
Fonte: O Autor.

Figura 17 – *Boxplot* do atraso



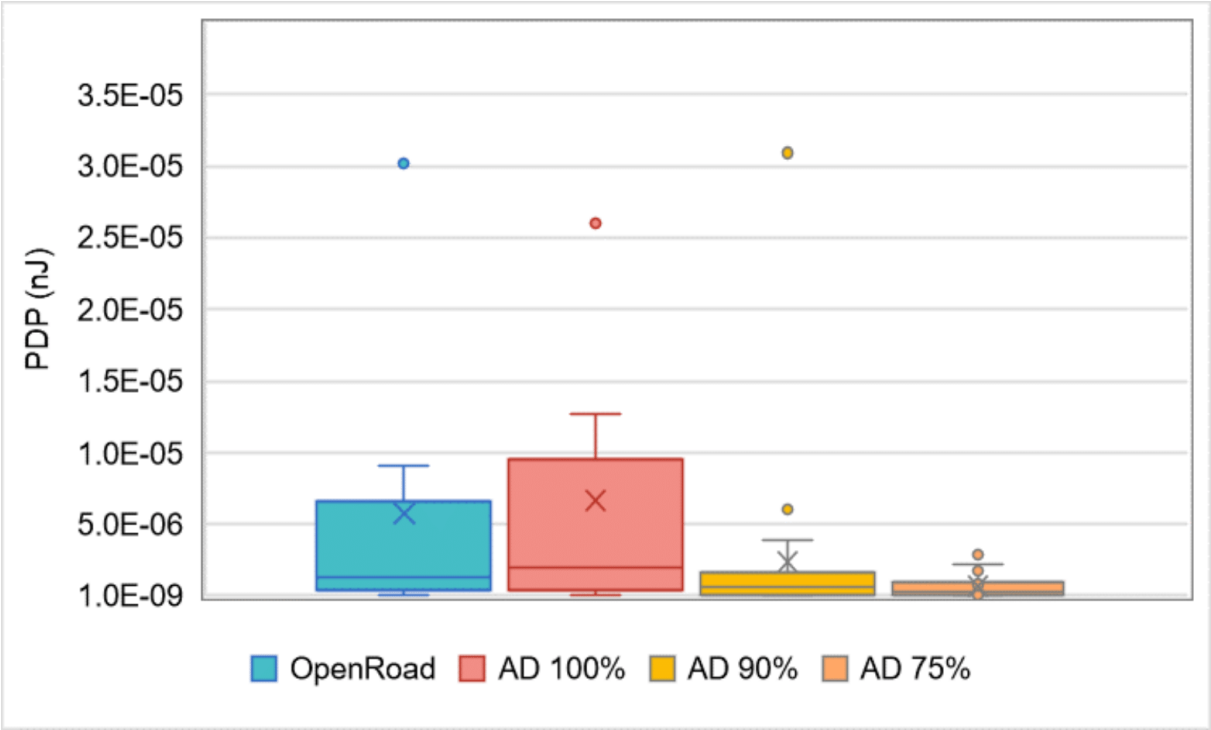
Fonte: O Autor.

Figura 18 – *Boxplot* da potência



Fonte: O Autor.

Figura 19 – *Boxplot* do PDP



Fonte: O Autor.

7 CONCLUSÕES

O avanço e modernidade tecnológica correm de maneira acelerada nos últimos anos e indicam que as próximas décadas serão ainda mais intensas. Isso só foi possível devido ao aumento da eficiência nas técnicas de desenvolvimento dos circuitos usados para tais atividades. A Síntese Lógica é um dos processos envolvidos nesse desenvolvimento e merece atenção especial devido ao impacto relevante que causa no tempo para se criar um circuito e eficiência no desempenho do mesmo.

Não só os circuitos têm ficado mais complexos, como algoritmos de IA também. A área de computação vem dando grande foco a este segmento e obtendo resultados bastante interessantes com métodos de Aprendizagem de Máquina. Isso nos permite usar um em benefício de outro, alinhando e equilibrando os avanços nessas tecnologias.

Com o objetivo de usar Aprendizagem de Máquina na execução da Síntese Lógica, desenvolveu-se o fluxo proposto e analisado neste trabalho.

Durante o desenvolvimento do projeto, adoutou-se o *software* de código aberto e síntese ASIC *OpenRoad*. O *OpenRoad* já possui algumas ferramentas integradas ao seu fluxo e é de fácil integração à proposta deste trabalho, tendo como maior diferencial a substituição do ABC pelo *software* Yosys.

Os resultados obtidos com a implementação da lógica de Computação Aproximada comprovaram o ganho de desempenho dos circuitos gerados em boa parte das acurácias avaliadas. O comportamento padrão foi de queda na performance para acurácia de 100% em comparação com o fluxo OpenROAD. Entretanto, observou-se um aumento contínuo desse desempenho acompanhando o aumento da taxa de acurácia pré-definida pelo usuário, como era de se esperar.

Os resultados demonstraram também saltos relativamente elevados nos valores da acurácia quando a profundidade das Árvores é alterada em alguns casos. Isso pode ser uma barreira quando se deseja atingir valores mais precisos na taxa de erro, além de não ser uma boa opção de uso em circuitos pequenos que já possuam uma baixa profundidade na sua representação em formato de Árvore. Uma alternativa para contornar esse comportamento é o uso do parâmetro de poda *cpp_alpha* no lugar de *max_depth*, que poderia suavizar a queda da precisão do circuito. Estudos com essa nova abordagem são planejados para trabalhos futuros.

O projeto apresentado ainda tem bastante espaço para ser explorado e aprimorado. Além da aplicação da técnica de aproximação usando o *cpp_alpha*, pretende-se implementar algoritmos que permitam executar o fluxo sem a necessidade de separação das saídas como é feito atualmente, reduzindo ainda mais o tempo de execução das etapas iniciais e simplificando o fluxo. Da mesma forma que as áreas que o compõem ainda têm muito a oferecer, o projeto aqui apresentado também o tem.

7.1 PUBLICAÇÕES

Durante o desenvolvimento deste trabalho, foram aplicadas diferentes estratégias com o intuito de atingir-se os melhores resultados. Durante o processo, algumas publicações e apresentações foram feitas em seminários e *workshops*.

Após a divulgação de resultados pela equipe organizadora do *contest* do IWLS foi proposta a publicação de um artigo conjunto com as equipes participantes descrevendo suas estratégias. Neste artigo, (RAI *et al.*, 2021), é possível comparar e avaliar as propostas no *contest* e compreender a motivação para cada uma das abordagens. As equipes de Árvore de Decisão e Computação Genética, apesar de terem propostas diferentes na competição, em alguns momentos trabalharam em conjunto, tendo sempre como objetivo o aprimoramento de seus respectivos fluxos. Por esse motivo, algumas publicações em co-autoria foram feitas, como no *Journal of Integrated Circuits and Systems*, (BERNDT, A. A. S. *et al.*, 2022), e no *Symposium on Integrated Circuits and Systems Design* (SBCCI), (BERNDT, A. *et al.*, 2021). Uma avaliação da solução proposta para o *contest* voltada para síntese de circuitos com lógica completamente definida foi publicada no*International Symposium on Circuits and Systems* (ISCAS), (ABREU, Brunno A. de *et al.*, 2021). Além disso, também foi produzida uma de publicação como primeiro autor descrevendo resultados da síntese baseada em árvores explorando o gerador de árvore C5.0 no 36º Simpósio Sul de Microeletrônica (SIM 2021) (CAMPOS *et al.*, 2021a).

Os resultados do estudo de caso apresentados neste trabalho já utilizando o SK para a geração das árvores de decisão foi publicado na conferência IEEE Dallas Circuits and Systems Conference (DCAS) em 2022 (CAMPOS *et al.*, 2022). Os resultados finais do projeto integrado com a ferramenta OpenRoad serão publicados futuramente.

Abaixo segue lista das publicações até o momento:

1. A Decision Tree Synthesis Flow for Precise and Approximate Circuits, 15TH IEEE Dallas Circuits and Systems Conference (DCAS), 2022. (CAMPOS *et al.*, 2022),
2. Exploring a Decision Tree Synthesis Flow for Approximate Circuits, 6TH Workshop on Approximate Computing, 2021 (CAMPOS *et al.*, 2021b),
3. An Automatic Flow for Precise and Approximated Synthesis, SIM - Simpósio Sul de Microeletrônica, 2021 (CAMPOS *et al.*, 2021a).

Além disso, a participação no grupo de pesquisa permitiu a colaboração como co-autor nas seguintes publicações:

1. CGP-based Logic Flow: Optimizing Accuracy and Size of Approximate Circuits, Journal of Integrated Circuits and Systems, 2022 (BERNDT, A. A. S. *et al.*, 2022),
2. Accuracy and Size Trade-off of a Cartesian Genetic Programming Flow for Logic Optimization, 34th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI), 2021 (BERNDT, A. *et al.*, 2021),

3. Fast logic optimization using decision trees, IEEE International Symposium on Circuits & Systems (ISCAS), 2021(DE ABREU *et al.*, 2021),
4. Bootstrapping Logic Optimization Techniques With Cartesian Genetic Programming, SIM - Simpósio Sul de Microeletrônica, 2021 (BERNDT *et al.*, 2021).

REFERÊNCIAS

ABREU, Brunno A. de *et al.* Fast Logic Optimization Using Decision Trees. *In:* 2021 IEEE International Symposium on Circuits & Systems (ISCAS). [*S.l.*: *s.n.*], 2021. P. 1–4.

ABREU, Brunno A.; GRELLERT, Mateus; BAMPI, Sergio. VLSI Design of Tree-Based Inference for Low-Power Learning Applications. *In:* 2020 IEEE International Symposium on Circuits and Systems (ISCAS). [*S.l.*: *s.n.*], 2020. P. 1–5. DOI: 10.1109/ISCAS45731.2020.9180704.

ABREU, Brunno Alves de *et al.* C2PAx: Complexity-Aware Constant Parameter Approximation for Energy-Efficient Tree-Based Machine Learning Accelerators. **IEEE Transactions on Circuits and Systems I: Regular Papers**, v. 69, n. 7, p. 2683–2693, 2022. DOI: 10.1109/TCSI.2022.3169028.

AJAYI, Tutu; BLAAUW, David. Openroad: Toward a self-driving, open-source digital layout implementation tool chain. *In:* PROCEEDINGS of Government Microcircuit Applications and Critical Technology Conference. [*S.l.*: *s.n.*], 2019.

AJAYI, Tutu; CHHABRIA, Vidya A. *et al.* Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project. *In:* PROCEEDINGS of the 56th Annual Design Automation Conference 2019. Las Vegas, NV, USA: Association for Computing Machinery, 2019. (DAC '19). DOI: 10.1145/3316781.3326334. Disponível em: https://doi.org/10.1145/3316781.3326334.

BEEREL, P. A.; PEDRAM, M. Opportunities for Machine Learning in Electronic Design Automation. *In:* 2018 IEEE International Symposium on Circuits and Systems (ISCAS). [*S.l.*: *s.n.*], 2018. P. 1–5.

BERGE, Claude. **The theory of graphs**. [*S.l.*]: Courier Corporation, 2001.

BERKELEY LOGIC SYNTHESIS AND VERIFICATION GROUP. **ABC: A System for Sequential Synthesis and Verification**. [*S.l.*: *s.n.*], 2019. http://www.eecs.berkeley.edu/ alanmi/abc/.html. Acesso em: 10 jan. 2020.

BERNDT, A. A. S. *et al.* Bootstrapping Logic Optimization Techniques With Cartesian Genetic Programming. *In:* SIM - Simpósio Sul de Microeletrônica. [*S.l.*: *s.n.*], 2021. P. 1–4.

BERNDT, Augusto *et al.* Accuracy and Size Trade-off of a Cartesian Genetic Programming Flow for Logic Optimization. *In:* 2021 34th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI). [*S.l.*: *s.n.*], 2021. P. 1–6. DOI: 10.1109/SBCCI53441.2021.9529968.

BERNDT, Augusto André Souza *et al.* A CGP-based Logic Flow: Optimizing Accuracy and Size of Approximate Circuits. **Journal of Integrated Circuits and Systems**, v. 17, n. 1, p. 1–12, 2022.

BIERE, Armin. The AIGER And-Inverter Graph (AIG) format version 20071012. **FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr**, v. 69, p. 4040, 2007.

BRAYTON, Robert; MISHCHENKO, Alan. ABC: An academic industrial-strength verification tool. *In:* SPRINGER. INTERNATIONAL Conference on Computer Aided Verification. [*S.l.*: *s.n.*], 2010. P. 24–40.

BRAYTON, Robert K *et al.* MIS: A multiple-level logic optimization system. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 6, n. 6, p. 1062–1081, 1987.

BRAYTON, Robert K. *et al.* Logic Minimization Algorithms for VLSI Synthesis . **The Kluwer International Series in Engineering and Computer Science**, v. 2, p. 1–194, 1984. Disponível em: `https://doi.org/10.1007/978-1-4613-2821-6`.

CAMPOS, Isac S. *et al.* A Decision Tree Synthesis Flow for Precise and Approximate Circuits. *In:* 15TH IEEE Dallas Circuits and Systems Conference. [*S.l.*: *s.n.*], 2022. P. 1–4.

_____. An Automatic Flow for Precise and Approximated Synthesis. *In:* SIM - Simpósio Sul de Microeletrônica. [*S.l.*: *s.n.*], 2021. P. 1–4.

_____. Exploring a Decision Tree Synthesis Flow for Approximate Circuits. *In:* 6TH Workshop on Approximate Computing. [*S.l.*: *s.n.*], 2021. P. 1–4.

CHAI, Donald *et al.* MVSIS 2.0 user's manual. **Department of Electrical Engineering and Computer Sciences**, 2003.

DE ABREU, Brunno A *et al.* Fast logic optimization using decision trees. *In:* IEEE. 2021 IEEE International Symposium on Circuits and Systems (ISCAS). [*S.l.*: *s.n.*], 2021. P. 1–5.

HASHEMI, Soheil; TANN, Hokchhay; REDA, Sherief. BLASYS: Approximate Logic Synthesis Using Boolean Matrix Factorization. *In:* 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC). [*S.l.*: *s.n.*], 2018. P. 1–6. DOI: `10.1109/DAC.2018.8465702`.

KARNAUGH, M. The map method for synthesis of combinational logic circuits. **Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics**, v. 72, n. 5, p. 593–599, 1953. DOI:

10.1109/TCE.1953.6371932. Disponível em:
https://doi.org/10.1109/TCE.1953.6371932.

LIM, Sung Kyu. **Intelligent Design of Electronic Assets (IDEA)**. [*S.l.*]: DARPA RSS. Disponível em:
https://www.darpa.mil/program/intelligent-design-of-electronic-assets.

MASADEH, Mahmoud; AOUN, Alain *et al.* Decision Tree-based Adaptive Approximate Accelerators for Enhanced Quality. *In:* 2020 IEEE International Systems Conference (SysCon). [*S.l.*: *s.n.*], 2020. P. 1–5. DOI: 10.1109/SysCon47679.2020.9275856.

MASADEH, Mahmoud; HASAN, Osman; TAHAR, Sofiene. Comparative Study of Approximate Multipliers. *In:* PROCEEDINGS of the 2018 on Great Lakes Symposium on VLSI. Chicago, IL, USA: Association for Computing Machinery, 2018. (GLSVLSI '18), p. 415–418. DOI: 10.1145/3194554.3194626. Disponível em:
https://doi.org/10.1145/3194554.3194626.

MEIRELLES, FERNANDO S. Pesquisa Anual do FGVcia. **Uso da TI–Tecnologia da Informação nas Empresas. Fundação Getúlio Vargas**, 2020.

MISHCHENKO, Alan; CHATTERJEE, Satrajit; BRAYTON, Robert. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. *In:* IEEE. 2006 43rd ACM/IEEE Design Automation Conference. [*S.l.*: *s.n.*], 2006. P. 532–535.

MOORE, Gordon E *et al.* **Cramming more components onto integrated circuits**. [*S.l.*]: McGraw-Hill New York, 1965.

NEPAL, Kumud *et al.* ABACUS: A technique for automated behavioral synthesis of approximate computing circuits. *In:* 2014 Design, Automation Test in Europe Conference Exhibition (DATE). [*S.l.*: *s.n.*], 2014. P. 1–6. DOI: 10.7873/DATE.2014.374.

OPENROAD. **The OpenROAD Project Website**. [*S.l.*: *s.n.*], 2022. urlhttps://theopenroadproject.org/.

PANDEY, M. Machine learning and systems for building the next generation of EDA tools. *In:* 2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC). [*S.l.*: *s.n.*], 2018. P. 411–415.

PANDYA, Rutvija; PANDYA, Jayati. C5.0 algorithm to improved decision tree with feature selection and reduced error pruning. **International Journal of Computer Applications**, Foundation of Computer Science, v. 117, n. 16, p. 18–21, 2015.

PEDREGOSA, Fabian *et al.* Scikit-learn: Machine learning in Python. **the Journal of machine Learning research**, JMLR. org, v. 12, p. 2825–2830, 2011.

QUINE, W. V. A Way to Simplify Truth Functions. **The American Mathematical Monthly**, Taylor & Francis, v. 62, n. 9, p. 627–631, 1955. Disponível em: https://doi.org/10.1080/00029890.1955.11988710.

RAI, Shubham *et al.* Logic Synthesis Meets Machine Learning: Trading Exactness for Generalization. *In:* 2021 Design, Automation Test in Europe Conference Exhibition (DATE). [*S.l.*: *s.n.*], 2021. P. 1026–1031. DOI: 10.23919/DATE51398.2021.9473972.

RIENER, H. *et al.* Scalable Generic Logic Synthesis: One Approach to Rule Them All. *In:* ACM/IEEE Design Automation Conference (DAC). [*S.l.*: *s.n.*], 2019.

ROVINSKI, Austin *et al.* Bridging academic open-source EDA to real-world usability. *In:* IEEE. 2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD). [*S.l.*: *s.n.*], 2020. P. 1–7.

RUDELL, R. L.; SANGIOVANNI-VINCENTELLI, A. Multiple-Valued Minimization for PLA Optimization. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 6, n. 5, p. 727–750, 1987. DOI: 10.1109/TCAD.1987.1270318. Disponível em: https://doi.org/10.1109/TCAD.1987.1270318.

RUPP, Karl. **Microprocessor Trend Data**. [*S.l.*: *s.n.*], 2022. urlhttps://github.com/karlrupp/microprocessor-trend-data.

SCARABOTTOLO, Ilaria *et al.* Approximate Logic Synthesis: A Survey. **Proceedings of the IEEE**, v. 108, n. 12, p. 2195–2213, 2020. DOI: 10.1109/JPROC.2020.3014430.

SENTOVICH, Ellen M *et al.* SIS: A system for sequential circuit synthesis. Citeseer, 1992.

VENKATARAMANI, Swagath; CHAKRADHAR, Srimat T. *et al.* Approximate computing and the quest for computing efficiency. *In:* 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC). [*S.l.*: *s.n.*], 2015. P. 1–6. DOI: 10.1145/2744769.2744904.

VENKATARAMANI, Swagath; KOZHIKKOTTU, Vivek J *et al.* Logic synthesis of approximate circuits. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 39, n. 10, p. 2503–2515, 2019.

WOLF, Clifford; GLASER, Johann; KEPLER, Johannes. Yosys-a free Verilog synthesis suite. *In:* PROCEEDINGS of the 21st Austrian Workshop on Microelectronics (Austrochip). [*S.l.*: *s.n.*], 2013.

A Decision Tree Synthesis Flow for Precise and Approximate Circuits

[1]Isac S. Campos,[1]Augusto Berndt, [2]Brunno A. de Abreu,
[1]Jonata T. Carvalho, [1]Mateus Grellert, and [1]Cristina Meinhardt
[1]Departamento de Informática e Estatistica, PPGCC, Universidade Federal de Santa Catarina - UFSC, Brazil
[2]Instituto de Informática, PGMicro, Universidade Federal do Rio Grande do Sul - UFRGS, Brazil
isac.campos@grad.ufsc.br, augusto.berndt@posgrad.ufsc.br, baabreu@inf.ufrgs.br,
{jonata.tyska, mateus.grellert, cristina.meinhardt}@ufsc.br

2022 IEEE 15th Dallas Circuit And System Conference (DCAS) | 978-1-6654-9885-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/DCAS53974.2022.9845636

***Abstract*—Logic and physical synthesis are getting more integrated, aiming to obtain an optimized circuit in terms of delay, power, and area. However, the current traditional tools used in Boolean logic optimization, in some cases, require a high runtime to afford an efficient minimization. Decision Tree approaches are capable of providing efficient a nd f ast s olutions f or logic optimization. This work presents a logic optimization flow based on Decision Trees for approximate circuits and evaluates its efficiency i n a rea, d elay, a nd a ccuracy c onsidering F PGA and ASIC synthesis. Compared to commercial tools, the proposed flow r educes u p t o 5 8% a rea a nd 3 8% p ower o n A SIC synthesis and up to 77% the resources utilization on FPGA synthesis, with a slight reduction in accuracy on a few outputs.**

***Index Terms*—Logic Synthesis, Logic Minimization, Machine Learning, Decision Trees**

I. Introduction

Logic minimization is a crucial task, as it reduces the hardware required to design a given system. The goal is to lower costs and save area and power consumption by reducing the number of components needed. Some well-known minimization techniques to simplify Boolean functions are the Algebraic method, Karnaugh map [1], Quine-McCluskey [2], and Espresso [3], [4]. However, these traditional methods have some limitations when dealing with circuits containing a high number of inputs [5]. The Quine-McCluskey algorithm, for example, is limited to functions with up to 15 variables.

Automating the whole process is a possible path to reduce the full synthesis time. Machine learning (ML) techniques, such as Decision Trees (DTs), may provide efficient and fast results for logic optimization [6]. DTs are an excellent alternative for learning Boolean functions since tree-based methods are suitable for predictions separated into discrete categories, similar to Truth Tables [6]. Furthermore, they are easily converted to equations, which allows the direct conversion of the solution to a Sum Of Products (SOP). ML techniques can learn a function that maps inputs to outputs by constructing a data-driven model. They are a powerful mechanism for learning how to achieve complex objectives, and Electronic Design Automation (EDA) is an area where these tools can still advance [7], [8].

Moreover, decision trees provide a powerful mechanism to automatize approximate circuit synthesis. Approximate Computing is a trend approach to the power-efficient circuit design. The approximation of the Boolean function can handle adjusting the depth of the decision tree, reducing the number of nodes, and consequently minimizing the number of gates. Also, reducing the decision tree's depth may reduce the critical path by limiting the logical depth. Although, these optimizations can have some inaccuracy in the function outputs. Thus, this optimization is better recommended for error-resilient computationally intensive applications, e.g., big data and image processing. Such applications have the following approximation-enabling characteristics [9]: 1) A golden result does not exist, and a set of outcomes are acceptable, for example, recommendation systems; 2) the best solution is not required or guaranteed where a good-enough result is sufficient, u sers a re c onditioned t o a ccept i mperfect, b ut good enough results; 3) the input data is noisy with iterative-refinement n ature, a nd t he i nexact r esult i s c onsumable by human perception; and 4) Applications frequently use computation patterns such as aggregation or iterative-refinement, which have the property of attenuating or healing the effects of approximations.

In the recent years, approximate arithmetic components, like adders [10] [11] [12] and multipliers [13] [14], have been ex-plored to construct approximate hardware accelerators where the error magnitude depends on the applied inputs. However, it is substantial to consider the approximate error at the early design stage by changing the architecture of the approximate accelerators [15]. There are two main approaches to control error in approximate circuits i) adapting the architecture of the approximate components. For example, an algorithm can control the approximation through error compensation by predicting the error magnitude of specific inputs. Or ii) switching between different implementations of approximate components, changing the approximate design dynamically for different input data, and satisfying a user-defined t arget o utput q uality. M any a pproximate l ogic synthesis techniques have been proposed in the last years [16]. Depending on the abstract level of the approximation, these techniques can be broadly classified i nto t wo approaches: 1) Manipulate the circuit structure/netlist; or 2) Relax the functionality, essentially by altering entries in the Truth table, regardless of the circuit structure.

This work presents a logic optimization flow based on DTs. This flow i s d esigned t o s ynthesize c onventional l ogic and

978-1-6654-9885-2/22/$31.00 ©2022 IEEE

approximate circuits, and it can be integrated with commercial FPGA and ASIC synthesis tools. The possibility to change the way the DTs are built by modifying the tree generation parameters allows to explicitly control the trade-off between area, delay, and the circuits' output quality. This trade-off can be explored for energy-efficient designs, applying the paradigm of approximate computing when the applications are tolerant to loss of quality [17]. The approximation is explored by relaxing the functionality and limiting the decision tree depth during the logic function learning.

II. Background

To better understand the project's scenario, we will briefly explain the technologies used for this work and their main functionalities, starting with one of the most traditional logic synthesis tools, the ABC, adopted as the baseline. Also, we introduce the concepts of And-Inverter Graphs (AIGs) and the DT.

A. ABC

ABC is a software tool for the synthesis and verification of logic circuits appearing in hardware design. It combines several functionalities such as: scalable logic optimization based on And-Inverter Graphs (AIGs), optimal-delay DAG-based technology mapping for look-up tables and standard cells, and innovative algorithms for sequential synthesis and verification [18]. The ABC tool is academic and implemented by Berkeley University. It provides the implementation of algorithms and a programming environment for building similar applications. Aiming to allow users to customize ABC for their needs, future development will focus on improving the algorithms and making most of the packages stand-alone. The ABC project aims to create an open-source implementation of state-of-the-art combinational and sequential synthesis algorithms. Its current version can optimize and map industrial gate-level designs with 100K gates and 10K sequential elements for the optimal delay and heuristically minimized area.

B. AND-Inverter Graph

The AIG is the state-of-the-art data structure representing technology-independent circuits during logic synthesis [19], [20]. Commonly, technology-independent optimizations aim to reduce the size and depth of the data structure representing the circuit regardless of physical characteristics. The AIG is a directed acyclic graph composed of primary inputs, primary outputs, and internal AND nodes. Internal nodes are composed of precisely two inputs and any number of outputs. Direct or inverted edges connect the nodes in the graph. Since the AIG is built with ANDs and NOTs, they can represent any logic circuit.

Classical logic synthesis techniques optimize logic circuits with exact solutions, in contrast to approximate circuits. Such techniques usually apply logic optimizations on logic functions in different formats, such as Boolean networks like AIGs [20], XOR-AND-Inverter Graphs (XAIG) [21], Majority-AND-Inverter Graphs (MIG) [22], lookup-tables (LUTs) [19], among other data structures representing Boolean networks.

C. Decision Trees

Finally, a DT [23] is a decision support tool that uses a tree-like graph to show the predictions that result from a series of feature-based splits. It starts at the root node and traverses the tree based on tests over the input values. The decisions are located at the leaf nodes.

Just like in the regular data structure trees, DTs have a special node called *root node*, which is the starting point for the data analysis. The other leaves are divided according to various root features, and these subgroups are split at each decision node under the root. In addition to the root node, the leaf nodes are the last ones of the flow, not having any subgroups after them.

III. The Proposed Optimization Flow

The Boolean optimization with DT flow starts from a complete or a partial truth table. The information of the logic entries and correspondent outputs is used to build a DT, which will be used to build a logic circuit. Firstly, we pre-process the input truth tables by splitting their primary outputs into separate new circuit benchmarks and expanding the don't care (DC) signals, using the *collapse* command from the ABC tool.

The DT training process for the logic synthesis flow uses the C5.0 classification algorithm [24]. The final tree is converted to the Sum Of Products (SOP) format. Figure 1 shows an example of a decision tree derived from the Full Adder Truth Table and the correspondent SOP generated. After the decision tree learning, the SOP is constructed by visiting all the branches resulting in one output. This decision tree represents the complete Full Adder function, with a depth=4.

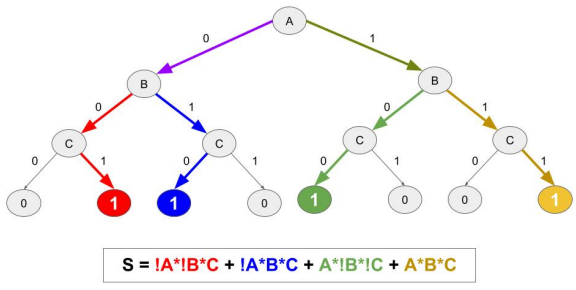


Fig. 1: Full adder sum output logic equation obtained from a Decision Tree representation.

Besides generating precise functions, DTs can also work with approximate models, depending on the tree depth limitation or a partial truth table. Thus, for DT synthesis, the resulting circuit is evaluated considering the accuracy of the solution generated. The tree depth can be set up according to application demands, searching for a good trade-off between power and quality.

The DT output is translated to the equation format (EQN) so that it can be read by the ABC tool [18] to be converted to a Verilog file for each output function. This equation is now read to generate the AIG structure on the ABC tool. ABC can

apply optimization functions available in the tool to further optimizations on the decision tree-based function.

Afterward, a single circuit Verilog top-level entity is created by merging all the previously split outputs.

The flow is automated in Python and can be easily adapted to other FPGA or ASIC synthesis tools. After the FPGA or ASIC synthesis, the final step comprehends the circuit evaluation using area and timing metrics.

IV. Evaluation

In this work, we compare our proposed optimization with two other approaches, exploring three workflows:

1) the proposed Boolean optimization with DT training,
2) the traditional logic optimization with ABC tool [18],
3) the Direct Verilog synthesis (DV).

All approaches partially explore the optimization flow as presented in Figure 2. The initial steps (collapsing and splitting outputs) and the last ones (converting into Verilog, merging them, and performing the synthesis) are the same for all proposed methods.

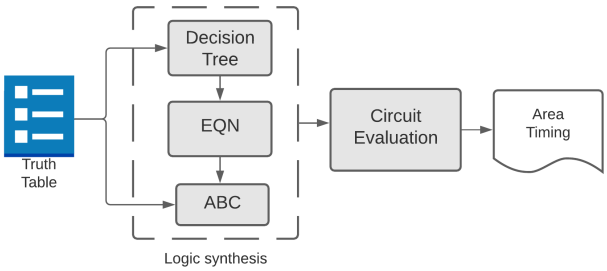


Fig. 2: Equivalent flow diagram for all methods.

The logic optimization with ABC tool reads the Truth Table, represents internally in the AIG strucuture, and, then, it executes the *compress2rs* optimization script [25] with a wide variety of refinement algorithms, such as *balance*, *resub*, *refactor* and *rewrite*.

The Direct Verilog synthesis method analyzes the native logic optimization of commercial tools. In this experiment, we use the Intel Quartus tool for FPGA synthesis and the Genus tool from Cadence for the ASIC synthesis.

The truth table is converted to Verilog by the ABC tool for this approach without any logic optimization. This approach is adopted to evaluate the conversion optimization capability intrinsic to commercial synthesis solutions.

V. Results

To summarize and show the potential of this flow, we adopt two benchmarks from MCNC International Workshop on Logic Synthesis' 93 [26]. The first one is a 4-bit Arithmetic and Logic Unit (ALU4), and the second one is the *PDC* benchmark. The number of inputs and outputs of these benchmarks is presented in Table I.

The evaluation criteria used to compare the performance of the methods were timing and area values. Quartus tool calculated these data, considering the circuits allocated into

TABLE I: Benchmarks Features

Benchmark	Inputs	Outputs
ALU4	14	8
PDC	16	40

FPGA. The FPGA synthesis maps the circuit targeting the Cyclone V family, device model 5CGXFC7C7F23C8. This device contains 150,000 logic elements, 56,480 adaptive logic modules (ALMs), an embedded memory of 6.70 Mbit, 240 I/Os, and an operating supply voltage of 1.1 V. The ASIC synthesis is performed with the Genus tool from Cadence and adopts the STMicroelectronics 65 nm standard cells library targeting a frequency of 500 MHz and input switching frequency of 50%.

Table II shows in the first column the benchmark characteristics of input and output (I/O). Also, each workflow shows the Total Power, Leakage and Dynamic Power, number of cells, area, and NAND2 equivalent area results from ASIC synthesis. The delay and resource utilization for the FPGA synthesis (power data is not computed at this moment on the FPGA synthesis because the circuit is composed only by combinational logic without a switch activity defined).

Some circuits are tolerant to loss of quality, trading off power efficiency at the cost of output accuracy. The two benchmarks are composed of complete truth tables, but the DT synthesis can provide an approximate computing logic, as demonstrated by our analyses. After splitting, the outputs can present different accuracy results with the DT training logic synthesis flow. The accuracy of the analyzed benchmarks outputs can be seen in Figure 3. The DT accuracy is 100% for most outputs on *ALU4*, despite two outputs that are limited to 98.1% and 99.3%. The DT accuracy for the *PDC* benchmark presents only one output with less than 99% of accuracy, and the majority of the 40 outputs reach over 99.5%. The impact of this loss in accuracy is highly dependent on the application.

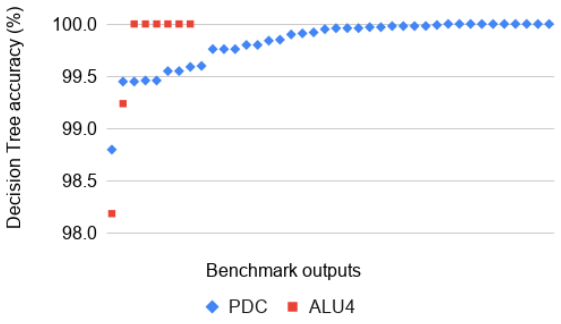


Fig. 3: Benchmarks accuracy for each output

The results obtained for both FPGA and ASIC synthesis indicate that the ABC script optimization may restrict the performance of other tools optimization procedures on commercial synthesis flows, degrading the number of cells and/or resources in the two benchmarks evaluated in this work.

The ASIC syntheses show similar area and power results for DT optimization and the DV method with the ALU4 bench-

TABLE II: Performance Results for the different workflows.

Benchmark	ASIC Synthesis 65 nm @500MHz							FPGA Synthesis	
	Method	Leakage Power (nW)	Dynamic Power (µW)	Total Power (µW)	# Cells	Area (µm2)	NAND2 Eq.	Delay (ns)	Resources (#ALMs)
Alu4	DT	1.25	1517.4	1518.6	675	3665	1762	23.7	275
Inputs: 14	ABC	1.31	1529.8	1531.1	710	3810	1832	25.2	624
Outputs: 8	DV	1.21	1485.4	1486.6	657	3529	1697	23	588
PDC	DT	0.28	325.8	326.1	88	533	256	15.2	26
Inputs: 16	ABC	0.56	565.3	565.9	262	1412	679	19	108
Outputs: 40	DV	0.57	521.6	522.1	223	1277	614	18.2	111

mark. However, with the PDC benchmark, the DT method reduces the number of cells by 61% compared to the DV method, representing a 58% area reduction (observing Total area or NAND2 equivalence) and reaching 38% of power optimization.

When considering the FPGA synthesis, the proposed DT optimization has presented an improvement in almost all metrics analyzed in *ALU4* and *PDC*, only causing a slight increase of 3.2% in the *ALU4* delay, which is compensated by the 53.2% reduction in the logic utilization for the same circuit. Furthermore, the significant decrease of 76.6% in *PDC* resources provided by the DT flow evidences the great capability of this proposal and motivates further research in this segment.

VI. Conclusion

This work evaluates the impact of adopting a DT logic synthesis flow to reduce circuit power, area, and delay. The test environment adopts the FPGA and ASIC synthesis as a case study and uses the ABC optimization script as the baseline. The proposed DT flow can be explored to generate more power-efficient solutions at the cost of a tolerable level of accuracy depending on the application scenario at hand. Indeed, this can be an exciting and fast approach to application designs applying approximate computing.

Acknowledgment

The authors thank CNPq, CAPES Finance code 001, and Propesq/UFSC Brazilian Agencies for financial support.

References

[1] M. Karnaugh. The map method for synthesis of combinational logic circuits. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, 72(5):593–599, 1953.

[2] W. V. Quine. A way to simplify truth functions. *The American Mathematical Monthly*, 62(9):627–631, 1955.

[3] R. L. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for pla optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(5):727–750, 1987.

[4] Robert K. Brayton et al. Logic minimization algorithms for vlsi synthesis. *The Kluwer International Series in Engineering and Computer Science*, 2:1–194, 1984.

[5] Olivier Coudert and Tsutomu Sasao. *Two-Level Logic Minimization*, pages 1–27. Springer US, Boston, MA, 2002.

[6] Brunno A. de Abreu et al. Fast logic optimization using decision trees. In *IEEE ISCAS*, 2021.

[7] P. A. Beerel and M. Pedram. Opportunities for machine learning in electronic design automation. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2018.

[8] M. Pandey. Machine learning and systems for building the next generation of eda tools. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 411–415, 2018.

[9] Swagath Venkataramani, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Approximate computing and the quest for computing efficiency. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2015.

[10] Zhixi Yang, Ajaypat Jain, Jinghang Liang, Jie Han, and Fabrizio Lombardi. Approximate xor/xnor-based adders for inexact computing. In *2013 13th IEEE International Conference on Nanotechnology (IEEE-NANO 2013)*, pages 690–693, 2013.

[11] Vaibhav Gupta, Debabrata Mohapatra, Sang Phill Park, Anand Raghunathan, and Kaushik Roy. Impact: Imprecise adders for low-power approximate computing. In *IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 409–414, 2011.

[12] Marcio Monteiro, Pedro Aquino, Ismael Seidel, Mateus Grellert, Leonardo Soares, José Luis Güntzel, and Cristina Meinhardt. Design of energy-efficient gaussian filters by combining refactoring and approximate adders. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2021.

[13] Mahmoud Masadeh, Osman Hasan, and Sofiene Tahar. Comparative study of approximate multipliers. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, GLSVLSI '18, page 415–418, New York, NY, USA, 2018. Association for Computing Machinery.

[14] Mario Osta, Ali Ibrahim, Hussein Chible, and Maurizio Valle. Approximate multipliers based on inexact adders for energy efficient data processing. In *2017 New Generation of CAS (NGCAS)*, pages 125–128, 2017.

[15] Mahmoud Masadeh, Alain Aoun, Osman Hasan, and Sofiène Tahar. Decision tree-based adaptive approximate accelerators for enhanced quality. In *2020 IEEE International Systems Conference (SysCon)*, pages 1–5, 2020.

[16] Ilaria Scarabottolo, Giovanni Ansaloni, George A. Constantinides, Laura Pozzi, and Sherief Reda. Approximate logic synthesis: A survey. *Proceedings of the IEEE*, 108(12):2195–2213, 2020.

[17] J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *IEEE ETS*, 2013.

[18] Berkeley Logic Synthesis and Verification Group. Abc: A system for sequential synthesis and verification. http://www.eecs.berkeley.edu/ alanmi/abc/.html, 2019.

[19] H. Riener et al. On-the-fly and dag-aware: Rewriting boolean networks with exact synthesis. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1649–1654, 2019.

[20] A. Mishchenko, S. Chatterjee, and R. Brayton. Dag-aware aig rewriting: a fresh look at combinational logic synthesis. In *2006 43rd ACM/IEEE Design Automation Conference*, pages 532–535, 2006.

[21] Ivo Háleček, Petr Fišer, and Jan Schmidt. Towards AND/XOR balanced synthesis: Logic circuits rewriting with XOR. *Microelectronics Reliability*, 81:274–286, 2018.

[22] S. Shirinzadeh, M. Soeken, P. Gaillardon, and R. Drechsler. Fast logic synthesis for rram-based in-memory computing using majority-inverter graphs. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 948–953, March 2016.

[23] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[24] Su-lin Pang and Ji-zhang Gong. C5.0 classification algorithm and application on individual credit evaluation of banks. *Systems Engineering-Theory & Practice*, 29(12):94–104, 2009.

[25] H. Riener et al. Scalable generic logic synthesis: One approach to rule them all. In *ACM/IEEE Design Automation Conference (DAC)*, 2019.

[26] Ken McElvain. Iwls'93 benchmark set: Version 4.0. In *Distributed as part of the MCNC International Workshop on Logic Synthesis' 93 benchmark distribution, May*, 1993.